Algoritmi e strutture dati - Progetto didattico 2008/2009

Nicoletta De Francesco, Luca Martini

17 aprile 2009

Sommario

Il progetto didattico opzionale di Fondamenti II per l'anno accademico 2008/2009 consiste nel progetto e nell'implementazione di un interprete per un semplice linguaggio denominato Rmm (Ruby meno meno). Il linguaggio Rmm è un sottoinsieme molto ristretto del linguaggio Ruby. L'interprete dovrà essere in grado di eseguire un qualsiasi programma scritto in Rmm. Gli obiettivi del progetto sono la progettazione di un applicazione usando i costrutti object-oriented del linguaggio C++, l'implementazione di alcuni algoritmi ricorsivi e l'utilizzo di strutture dati appropriate.

Indice

1 Il linguaggio Rmm

Il linguaggio Rmm è ispirato ad un minuscolo sottoinsieme del linguaggio Ruby. Ruby è un linguaggio dalla sintassi molto ricca, mentre nella definizione di Rmm faremo molte semplificazioni. Descriviamo adesso le caratteristiche salienti di Rmm, per dare poi una specifica più formale.

Il linguaggio Rmm è un linguaggio *interpretato*. Ciò significa che i programmi Rmm non devono essere compilati in linguaggio macchina. Al contrario, vengono direttamente eseguiti una volta che siano stati messi come input ad uno speciale programma denominato *interprete*.

Il linguaggio Rmm, a differenza del C++, non è un linguaggio fortemente tipato. Ciò significa che in Rmm le variabili hanno un tipo, ma questo può cambiare all'interno del programma. In Rmm è perfettamente legale il seguente pezzo di codice:

```
a = "hello";
a = 9;
```

laddove in C++ il tipo della variabile `a` sarebbe rimasto lo stesso in tutto il programma. Per questa ragione, in Rmm non sono presenti dichiarazioni di tipo. Questo non significa che in Rmm non esistano i tipi, ma solo che i tipi sono associati ai *valori* e non alle *variabili*. I tipi possibili in Rmm sono cinque: `Number`, per i numeri interi, `Bool` per valori booleani, `String` per le sequenze di caratteri, `Array` per i vettori, e `Hash` per gli array associativi.

Attenzione I tipi `Array` e `Hash` non fanno parte delle specifiche minime, ma vengono presentati per completezza e come possibile estensione del progetto.

I numeri interi corrispondono al tipo `int` del linguaggio C++. Sui numeri sono definite le operazioni aritmetiche di addizione, sottrazione, moltiplicazione e divisione, tramite gli operatori `+`, `-`, `*`, `/`.

```
a = 5;
b = a+1;      # b vale 5
c = a*b;      # c vale 30
d = (c+1)/2; # d vale 15
```

Da questo semplice esempio, abbiamo anche visto come si specificano i commenti singola linea in Rmm. Infatti i caratteri compresi fra `#` e la fine della riga vengono ignorati dall'interprete.

Un valore di tipo `Bool` è una delle due costanti letterali `true`, `false`, con l'ovvio significato. Valori di tipo booleano possono essere manipolati tramite gli operatori logici `and` e `or`.

```
a = true;
b = false;
c = a or b;   # c vale true
d = a and b;  # d vale false
```

Le stringhe sono sequenze di caratteri. Un letterale stringa è una sequenza di caratteri compresi fra doppie virgolette. Un operatore definito fra stringhe è la concatenazione, tramite l'operatore `+`.

```
a = "ciao";
b = "mondo";
c = a+b;      # c vale "ciaomondo"
```

Esiste poi un valore particolare: `nil`, che indica un valore nullo.

Gli array sono sequenze di valori. Vengono indicati tramite una sequenza (racchiusa fra parentesi quadre) di espressioni separate da virgola. I singoli elementi dell'array sono a loro volta valori di Rmm, che non hanno necessariamente lo stesso tipo. Gli elementi sono numerati a partire da 0, come in C++. Le parentesi quadre vengono usate per accedere i singoli elementi dell'array.

```
a = [5, "ciao", true];  # a e' un array di tre elementi
b = a[0];               # b vale 6
a[1] = "mondo";         # a vale [5, "mondo", true]
b = a[7];               # b vale nil
```

La lunghezza degli array non è fissata a priori come in C++. In Rmm gli array si possono espandere dinamicamente senza che il programmatore debba manualmente allocare memoria. Le locazioni aggiunte e non ancora inizializzate vengono poste al valore `nil`.

```
a = [5, "ciao", true];  # a e' un array di tre elementi
a[4] = 6;               # a adesso vale [5, "ciao", true, nil, 6]
```

Un valore di tipo `Hash` è un cosidetto array associativo. Gli array associativi sono sequenze di coppie chiave-valore. Non possono esistere nello stesso `Hash` due coppie con la stessa chiave. Con un esempio possiamo vedere la sintassi per l'utilizzo delle hash.

```
h = { "Inter" => "Milano", "Juve" => "Torino", "Milan" => "Milano"};
b = h["Juve"]              # b vale "Torino"
h["Atalanta"] = "Bergamo"  # aggiunta la coppia "Atalanta"=>"Bergamo"
p = h["Pisa"]              # p vale nil, chiave non presente
```

$+ \; - \; * \; /$	operazioni aritmetiche
$< \; > \; == \; <= \; >= \; != $	confronti fra stringhe
$< \; > \; == \; <= \; >= \; != $	confronti fra numeri
$== \; != $	confronti fra array
$== \; != $	confronti fra hash
`and`,`or`	operatori logici

Figura 1: Operatori binari su numeri, valori booleani e stringhe

Descriviamo adesso formalmente la grammatica di un'espressione in Rmm.

$$
\begin{array}{rcl}
\mathit{Expr} & ::= & \mathit{Literal} \mid I \mid I[\mathit{Expr}] \mid \\
& & \mathit{Expr}\ \mathit{BinOp}\ \mathit{Expr} \mid \\
& & [\mathit{Expr}, \ldots, \mathit{Expr}] \mid \\
& & \{\mathit{Expr} => \mathit{Expr}, \ldots, \mathit{Expr} => \mathit{Expr}\} \mid \\
& & (\mathit{Expr}) \mid \\
& & I(\mathit{Expr}, \ldots, \mathit{Expr})
\end{array}
$$

Un valore *Literal* rappresenta una costante letterale (una stringa, un numero, oppure uno dei due valori letterali booleani `true` e `false`). Un'espressione può essere data da una variabile, da un accesso ad un'array o ad uno hash (il cui indice è specificato tramite una sottoespressione), da un operatore binario applicato a due sottoespressioni. Inoltre, un'espressione può essere un array od uno hash, una sottoespressione racchiusa tra parentesi, oppure l'applicazione di una funzione. La sintassi $I(\mathit{Expr} \ldots \mathit{Expr})$ indica la chiamata di funzione, dove le sottoespressioni all'interno delle parentesi rappresentano i parametri attuali con cui viene chiamata la funzione. Come in C++, i parametri sono separati da virgole. Gli operatori utilizzabili in Rmm sono riportati nella Figura 1. Gli operatori di confronto possono essere utilizzati per confrontare numeri o stringhe (in questo caso vale l'ordine lessicografico). Gli operatori di uguaglianza e diseguaglianza possono essere usati anche su array e hash con ovvio significato. Ogni operatore di confronto restituisce un valore booleano.

Un programma in Rmm è un'insieme di funzioni. Una di esse è chiamata `main`. Ecco la grammatica che descrive un programma Rmm.

$$
\begin{array}{rcl}
\mathit{Program} & ::= & \mathit{Fun} \ldots \mathit{Fun} \\
\mathit{Fun} & ::= & \texttt{def}\ I(I, \ldots, I)\ \mathit{Command}\ \texttt{end} \\
\mathit{Command} & ::= & I = \mathit{Expr}; \mid I[\mathit{Expr}] = \mathit{Expr}; \mid \texttt{return}\ \mathit{Expr}; \mid \\
& & \texttt{if}\ \mathit{Expr}\ \mathit{Command}\ \texttt{else}\ \mathit{Command}\ \texttt{end} \mid \\
& & \texttt{while}\ \mathit{Expr}\ \mathit{Command}\ \texttt{end} \mid \\
& & \mathit{Command}\ \mathit{Command}
\end{array}
$$

Ogni funzione *Fun* ha un nome (che è un identificatore), una lista di parametri formali, e un corpo. Il corpo è una sequenza di istruzioni, terminata da `end`. Gli operatori binari *BinOp* supportati dal linguaggio sono specificati in Figura 1. Per semplicità si considerino tutti alla stessa priorità. Ci sono quattro tipi di comandi: assegnamenti a variabili locali o ad array o hash (tramite l'operatore =), un comando condizionale (`if` con `else`), un comando ripetitivo (`while`). La semantica dei comandi e delle espressioni nel linguaggio Rmm è la stessa dei rispettivi comandi e espressioni nel linguaggio C.

Un esempio di un programma nel linguaggio Rmm è il seguente:

```
def pari_o_dispari(n) # calcola la parita' di un numero positivo
  while n > 1
    n = n-2;
  end
  if n == 0
    return "pari"
  else
    return "dispari"
  end
end
```

```
def main()
  numero = 16373;
  risposta = pari_o_dispari(numero); # risposta vale "dispari"
end
```

E' importante notare che il passaggio dei parametri in Rmm avviene sempre per valore. Ad esempio, nell'esempio precedente, dopo la chiamata alla funzione `pari_o_dispari` il valore della variabile numero è inalterato. Infatti, nella variabile formale `n` viene fatta una copia del contenuto della variabile `numero` che quindi non viene alterata dalla chiamata di funzione.

2 Un interprete

Un interprete è un programma che esegue programmi scritti in un certo linguaggio di programmazione. Al contrario, un compilatore traduce programmi scritti in un linguaggio sorgente in altri programmi scritti in un diverso linguaggio (il linguaggio oggetto). Tuttavia, molte delle operazioni compiute da un'interprete vengono svolte anche da un compilatore. Infatti, molti interpreti traducono il programma da eseguire in una forma particolare, denominata *albero sintattico*, la cui esecuzione risulta semplificata. Semplificando, il processo di interpretazione può essere diviso nelle seguenti fasi:

Analisi lessicale Il testo in ingresso viene suddiviso in *token* (gettoni). Ogni token rappresenta una unità del linguaggio considerato (come ad esempio una parola chiave, un identificatore, etc). Solitamente il riconoscimento dei token viene effettuato tramite un automa a stati finiti generato da espressioni regolari. Esistono programmi, come Flex [1], che da una specifica della sintassi di un linguaggio generano in maniera automatica degli analizzatori lessicali (o *scanner*).

Analisi sintattica In questa fase la sequenza di token viene analizzata e, in accordo alla grammatica del linguaggio, viene costruito un albero sintattico che descrive il programma sorgente. Anche per l'analisi sintattica esistono programmi, come Bison [2], che da una specifica della grammatica di un linguaggio generano in maniera automatica degli analizzatori sintattici (o *parser*).

Analisi semantica Una volta costruito l'albero sintattico lo si può visitare al fine di controllare eventuali errori semantici (ad esempio chiamata di funzioni non dichiarate) e per fare eventuali controlli di tipo.

Esecuzione In quest'ultima fase l'albero sintattico (eventualmente modificato durante l'analisi semantica) viene visitato. La visita di ogni nodo rappresenta l'esecuzione della corrispondente unità sintattica del programma sorgente.

Altre fasi sono normalmente presenti (come ad esempio ottimizzazioni varie) ma su queste sorvoliamo.

3 Analisi lessicale

Per il progetto didattico non è necessario sviluppare un analizzatore lessicale per il linguaggio Rmm. Infatti questo è fornito agli studenti sotto forma di alcune classi C++ che ora andremo a descrivere. L'analizzatore lessicale è realizzato tramite una classe `Analyzer` dalla seguente dichiarazione:

```
class Analyzer : public yyFlexLexer {
  TokenList tl; // lista dei token
public:
  Analyzer(istream* arg_yyin = 0, ostream* arg_yyout = 0);
  virtual int yylex(); // metodo che effettua l'analisi
  TokenList& getTokenList() { return tl; };
  ~Analyzer();
};
```

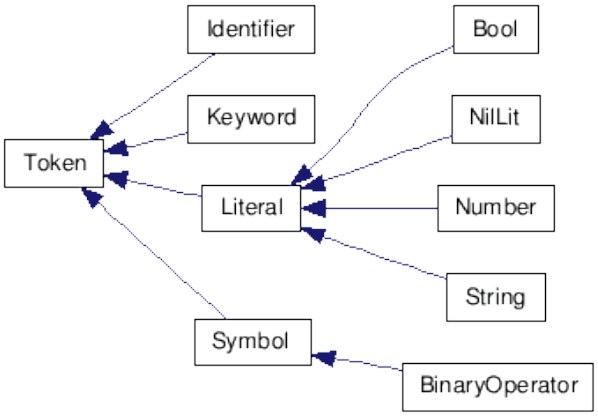

Figura 2: La gerarchia delle classi dei token nell'analizzatore lessicale

La classe `Analyzer` è derivata dalla classe `yyFlexLexer` che possiamo qui ignorare. Il costruttore prende come argomento un puntatore a stream di ingresso (lo stream dal quale si legge il programma da analizzare) e un puntatore a stream di uscita (lo stream sul quale si scrivono eventuali messaggi di errore). Il metodo `yylex` è il metodo che effettua l'analisi vera e propria: legge dallo stream un token e lo inserisce nella lista `tl` che è una lista di puntatori a oggetti di tipo `Token`, ovvero `std::list<Token*>`. Quando la funzione `yylex` restituisce zero si è raggiunta la fine del file. Il metodo `yylex` può sollevare delle eccezioni di tipo `ParserError` se l'analizzare lessicale incontra delle stringhe che non riesce ad interpretare come token validi. Un esempio di utilizzo della classe `Analyzer` è il seguente:

```
Analyzer* lexer = new Analyzer(&cin,&cerr);
try {
  while(lexer->yylex() != 0);
} catch (ParserError e) {
  cerr << e << endl;
}
// processa lexer->getTokenList()
```

La classe virtuale astratta `Token` rappresenta un token generico: le sue classi derivate sono invece token più specifici, come mostrato in Figura 2.

4 Analisi sintattica

Una volta completata la traduzione da codice sorgente a lista di token occorre fare un'analisi sintattica, ovvero controllare se la sintassi del linguaggio è rispettata. In questa fase si costruisce l'*albero sintattico*. L'albero sintattico è una rappresentazione del programma sotto forma di albero: ogni nodo rappresenta un'espressione od un comando presente nel programma. I nodi interni dell'albero saranno simboli non-terminali della grammatica, mentre le foglie saranno simboli terminali. L'albero può essere perciò costruito con la seguente regola induttiva:

$\mathbf{A}(\mathit{Literal})$ Albero con un unico nodo etichettato con il valore letterale

$\mathbf{A}(I)$ Albero con un unico nodo etichettato con un identificatore

$\mathbf{A}(I[\mathit{Expr}_1])$ Albero con radice l'identificatore I, e con sottoalbero A(Expr)

$\mathbf{A}(\mathit{Expr}_1\ \mathit{BinOp}\ \mathit{Expr}_2)$ Albero con radice l'operatore binario BinOp e con due sottoalberi: A(Expr_1) e A(Expr_2)

$\mathbf{A}(I = \mathit{Expr};)$ Albero con radice etichettata con "Assegnamento" e avente come sottoalberi A(I) e A(Expr)

$\mathbf{A}(I[\mathit{Expr}_1] = \mathit{Expr}_2;)$ Albero con radice etichettata con "Assegnamento" e avente come sottoalberi A(I), A(Expr_1) e A(Expr_2)

A($I(Expr_1, \ldots, Expr_n)$**)** Albero con radice etichettata come "Chiamata di Funzione" e $n+1$ sottoalberi: il primo contiene I, gli altri n sono i sottoalberi sintattici delle espressioni $Expr_i$ con $i \in [1, \ldots, n]$

A(`if` *Expr* *Command*$_1$ `else` *Command*$_2$ `end`**)** Albero con radice etichettata con "If" e avente come sottoalberi A(*Expr*), A(*Command*$_1$) e A(*Command*$_2$)

A(`while` (*Expr*) *Command*`)` Albero con radice etichettata con "While" e avente come sottoalberi A(*Expr*) e A(*Command*)

La costruzione dell'albero sintattico avviene tramite un algoritmo denominato *a discesa ricorsiva* che in base ai token prodotti dall'analisi lessicale riconosce i vari idiomi del linguaggio. L'algoritmo segue la grammatica del linguaggio e mira a riconoscere ogni costrutto in base alle alternative che pone la grammatica. L'algoritmo è ricorsivo in quanto la grammatica stessa è ricorsiva (ad esempio, per riconoscere una espressione composta da un operatore binario e due sottoespressioni dovremo invocare ricorsivamente la funzione di riconoscimento di una espressione su ciascuna delle due sottoespressioni).

Si suggerisce la creazione di una classe C++ per ogni costrutto del linguaggio. Una tale organizzazione potrà poi risultare utile nella fase successiva.

A titolo di esempio mostriamo alcuni spezzoni in pseudo-codice della funzione che riconosce le espressioni (Figura 3), in una versione che non riconosce le chiamate di funzione, e della funzione che riconosce i comandi (Figura 4). Inoltre, suggeriamo una possibile gerarchia di classi per i costrutti del linguaggio (Figura 5). In questa bozza di gerarchia si definisce una classe base astratta `Expression`, le cui classi derivate sono le varie tipologie di espressione. La struttura ad albero deriva direttamente dal fatto che alcune di queste classi contengono dei puntatori a oggetti di tipo `Expression`. Il tipo dei puntatori è quello della classe base perché non sappiamo di volta in volta a che tipo di espressioni concrete punteranno.

In realtà dovranno essere costruiti tanti alberi sintattici, uno per ogni funzione definita nel programma.

5 Interpretazione del codice

Per capire come dobbiamo intepretare i programmi Rmm, dobbiamo approfondire il modello di esecuzione di questo linguaggio. È importante notare come il modello di esecuzione sia sostanzialmente diverso da quello dei programmi in C++. In ogni momento dell'esecuzione di un programma Rmm, l'inteprete ha un *ambiente attivo*. Un ambiente è un'associazione fra i nomi delle variabili e il loro valore. Inizialmente l'ambiente è vuoto. Quando una variabile viene assegnata (si ricordi che in Rmm non esistono dichiarazioni) si crea una nuova associazione nell'ambiente. Quando invece una variabile viene acceduta in lettura, l'inteprete dovrà verificare che esista la variabile nell'ambiente attivo e ricavarne il valore. Quando si incontra una chiamata di funzione, si crea un nuovo ambiente che diventa il nuovo ambiente attivo. Il nuovo ambiente contiene solamente le associazioni per i parametri formali della funzione. I parametri formali sono associati ai parametri attuali. Il passaggio dei parametri viene fatto esclusivamente per copia. Le variabili presenti nell'ambiente del chiamante non sono accessibili. Quando una funzione termina, il risultato della funzione viene passato all'interprete, l'ambiente della funziona terminata viene distrutto, e l'ambiente del chiamante diviene il nuovo ambiente attivo. Supponiamo che fra le funzioni definite nel programma ce ne sia una denominata `main`, che non ha parametri formali. Quando si esegue un programma Rmm, viene eseguita la funzione `main`, partendo da un ambiente vuoto. A titolo di esempio, mostriamo lo stato dell'inteprete durante l'interpretazione di un semplice programma Rmm.

```
1  def incrementa(n)
2    n = n+1;
3    return n;
4  end
5
6  def main()
7    numero = 16;
8    b = incrementa(numero);
9  end
```

All'inizio, viene eseguita la prima istruzione della funzione `main`, che crea una nuova associazione per la variabile `numero`.

```
Espressione* riconosciE() {
  Espressione *e1, *e2;
  if (/* il prossimo token */ ==  '(' ) {
    e1 = riconosciE();
    if ((e1 == NULL) ||
    (/* il prossimo token */ != ')' ))
         Errore;
    // consuma un token
    return e1;
  } else if (/* il prossimo token e' letterale */) {
           e1 = new Espressione(/* prossimo token */);
   } else if (/* il prossimo token e' identificatore */) {
     // consuma un token e salvando l'identificatore I
     if (/* il token successivo e' una parentesi [*/) {
       // consuma un token
       Espressione *indice = riconosciE();
       if ((indice == NULL) || (/* il prossimo token */ != ']' ))
         Errore;
       // consuma un token
       e1 = new Espressione(I,indice);
     } else
     e1 = new Espressione(I);
     }
   if (/* il prossimo token e' operatore binario */) {
      BinOp b = /*prossimo token*/;
      // consuma un token
      e2 = riconosciE();
      if (e2 == NULL)
         Errore;
      else
         return new Espressione(b,e1,e2);
   }
   else return e1;
 }
}
```

Figura 3: Pseudo codice per l'analisi sintattica delle espressioni

```
Comando* riconosciComando() {
   switch(/*prossimo token*/) {
        case Identifier I:
             salva I;
             riconosci '=';
             riconosci ';';
             return new ComandoAssegnamento(I,e);
        case "if":
             Espressione* e = riconosciEspressione();
             while (/* prossimo token != else */)
                  riconosciComando();
             riconosci "else"
             while (/* prossimo token != end */)
                  riconosciComando();
             riconosci "end"
             return new IF(e,...);
        case "while": ...
        case "return": ...
   }
}
```

Figura 4: Pseudo codice per l'analisi sintattica dei comandi

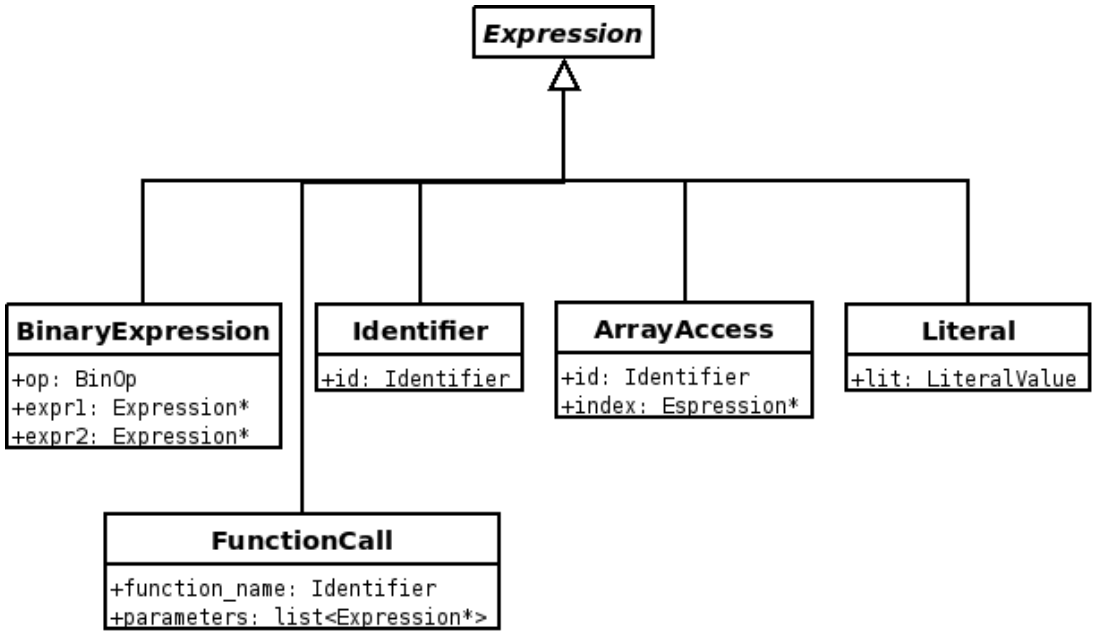

Figura 5: Una bozza di gerarchia per la costruzione dell'albero sintattico. In questa figura viene mostrata parzialmente la gerarchia per le espressioni. Seguendo la grammatica, una gerarchia equivalente dovrà essere progettata per i comandi.

Ambienti	Chiamate
numero→16	main

Quando viene valutata la chiamata di funzione alla linea 8, viene creato un nuovo ambiente:

Ambienti	Chiamate
n→16	incrementa
numero→16	main

Quando viene eseguita la linea 2, viene modificato il valore di `n`, ma non quello di `numero` (che infatti non sta nell'ambiente attivo), e si arriva alla seguente situazione.

Ambienti	Chiamate
n→17	incrementa
numero→16	main

Dopo la terminazione della funzione `incrementa`, l'ambiente corrispondente viene distrutto e l'ambiente attivo diventa quello della chiamata alla funzione `main` (Suggerimento: la struttura dati *stack* può essere utilizzata per gestire le chiamate di funzione). Prima della terminazione del programma ci troviamo perciò nella seguente situazione.

Ambienti	Chiamate
numero→16 b→17	main

Per poter eseguire il singolo comando presente all'interno del corpo di una funzione è necessario valutare le espressioni ed eseguire i sottocomandi presenti nell'istruzione stessa.

5.1 Valutazione delle espressioni

Data una variabile x indicheremo con $\mathcal{E}(x)$ il suo valore nell'ambiente corrente. La funzione di valutazione $\mathcal{I}_{\text{exp}}$ si occupa di valutare un'espressione. Questa funzione ha come argomenti l'espressione da tradurre e l'ambiente corrente e produce un valore. Vediamo ora la valutazione di alcune espressioni. Il primo caso nella definizione di $\mathcal{I}_{\text{exp}}$ riguarda la valutazione di espressioni letterali (numeri o stringhe). Questo caso è molto semplice in quanto il valore considerato è scritto nel letterale stesso

$$\mathcal{I}_{\text{exp}}(\texttt{k}, \mathcal{E}) \rightarrow \texttt{k}$$

La valutazione di un'espressione costituita da un identificatore, consiste nell'accedere all'ambiente corrente per ottenere il valore della variabile corrente. Se l'ambiente corrente non contiene alcuna variabile con tale nome, l'interprete dovrà segnalare un errore.

$$\mathcal{I}_{\text{exp}}(\texttt{x}, \mathcal{E}) \rightarrow \mathcal{E}(x)$$

Per quanto riguarda le espressioni aritmetiche del tipo E_1 *BinOp* E_2, la valutazione consiste nel: valutare E_1, valutare E_2, e applicare l'operazione corrispondente ad *BinOp* ai due valori ottenuti. Riportiamo, come esempio, la traduzione di $E_1 + E_2$.

$$\mathcal{I}_{\text{exp}}(E_1\texttt{+}E_2, \mathcal{E}) \rightarrow \mathcal{I}_{\text{exp}}(E_1, \mathcal{E}) + \mathcal{I}_{\text{exp}}(E_2, \mathcal{E})$$

La valutazione dell'espressione $I(E_1, \ldots, E_n)$ si ottiene dal risultato del corpo della funzione I applicato ad un ambiente $\mathcal{E}'$, che contiene solamente i parametri formali della funzione I, in cui al parametro formale x_i viene assegnato il valore $\mathcal{I}_{\text{exp}}(E_i, \mathcal{E})$.

La definizione di $\mathcal{I}_{\text{exp}}$ per gli altri tipi di espressioni è lasciata allo studente.

5.2 Esecuzione dei comandi

L'esecuzione dei comandi segue lo stesso schema. I comandi, a differenza delle espressioni, modificano l'ambiente attivo. L'esecuzione del comando di assegnamento consiste nella valutazione dell'espressione da assegnare e nella modifica dell'ambiente attivo. Esemplifichiamo l'azione di un comando come una funzione $\mathcal{I}_{\text{com}}$ da ambiente a ambiente, e indichiamo con $\mathcal{E}\,[x \to k]$ l'ambiente $\mathcal{E}'$ ottenuto da $\mathcal{E}$ assegnando alla variabile x il valore k.

$$\mathcal{I}_{\text{com}}(\texttt{x=}E, \mathcal{E}) = \mathcal{E}\,[x \to \mathcal{I}_{\text{exp}}(E, \mathcal{E})]$$

L'esecuzione del comando condizionale è ottenuta valutando l'espressione della condizione dell'`if`. Se vera, si procede ad eseguire i comandi immediatamente successivi all'if, se falsa si eseguono i comandi dopo l'`else`.

L'esecuzione del comando ripetitivo è ottenuta dalla valutazione della guardia. In caso l'espressione valutata sia vera, si procede ad eseguire il corpo del `while` e poi si riesegue l'istruzione ripetitiva. Nel caso sia falsa, si procede con l'istruzione successiva al `while`.

L'intepretazione degli altri tipi di comando è lasciata allo studente.

5.3 Funzioni native

Per rendere il linguaggio un po' più interessante, si possono aggiungere delle funzioni *native*, ovvero funzioni la cui semantica sia specificata nell'interprete stesso. Ne suggeriamo alcune:

length Funzione ad un parametro. Applicata ad un'array o ad una stringa restituisce la sua lunghezza. Ad esempio:

```
a = "ciao";
b = length(a);       # b vale 4
a = [6,9, "pippo"]
b = length(a);       # b vale 3
```

to_s Converte una variabile intera in una stringa. Ad esempio:

```
a = 67;
b = to_s(a);         # b vale "67"
```

to_i Converte una stringa in un numero. Ad esempio:

```
a = "897";
b = to_s(a);         # b vale 897
```

funzioni di I/O Si possono aggiungere funzioni di I/O. Suggeriamo una funzione `gets` (equivalente al `cin` del C++) ed una funzione `puts` (equivalente al `cout` del C++)

6 Suggerimenti vari

- Prima di procedere alla realizzazione di una funzionalità dell'applicazione, testare intensivamente le funzionalità già implementate. Ad esempio, prima si può pensare che non ci siano chiamate di funzione, per poi aggiungere il supporto in un secondo momento.
- Può essere una buona idea creare una gerarchia di classi per gli elementi sintattici (espressioni e comandi), in maniera simile a quanto è stato fatto nell'analizzatore lessicale con i token. In questo modo l'interpretazione può essere realizzata attraverso la scrittura di una funzione virtuale `valuta` o `esegui` che sarà implementata in maniera diversa in ogni categoria sintattica
- Scrivere dei programmi di test in Rmm che devono essere provati ad ogni estensione/modifica dell'applicazione

- Il codice dell'applicazione deve essere adeguatamente commentato e unitamente al progetto deve essere consegnata una **breve** documentazione che illustri le scelte progettuali adottate e le funzionalità realizzate dall'applicazione. A tal proposito si consiglia l'uso di Doxygen [3] per poter generare automaticamente la documentazione a partire dai commenti presenti nei file sorgente.
- Una volta che il progetto è funzionante si può tentare di ottimizzare quelle parti di codice che sono eseguite più di frequente.

7 Valutazione e regole generali

Il progetto deve essere consegnato entro l'ultimo appello di luglio (2009). Il progetto potrà essere presentato secondo le date comunicate sul sito del corso[1]. Aver consegnato un progetto valido esonera da una parte della prova scritta. L'esenzione vale fino all'ultimo appello dell'anno accademico (quindi, febbraio 2010). Il progetto deve essere svolto a gruppi di due persone. Il progetto verrà valutato fino ad un massimo di 12 (dodici) punti che si andranno ad aggiungere al risultato della prova scritta (che dovrà comunque essere sufficiente). Ulteriori 4 (quattro) punti potranno essere accordati a progetti che realizzino una delle estensioni proposte di seguito (Sezione 7.1).

I membri di un gruppo possono dividersi equamente i compiti ma devono essere in grado di conoscere a fondo ogni aspetto del progetto. Qualora il docente ritenga, a giudizio insindacabile, che i membri del gruppo non siano i reali autori del progetto, il progetto verrà ritenuto nullo e agli studenti interessati verrà comminata un'adeguata sanzione.

7.1 Possibili estensioni

- Supporto per gli array e le tabelle hash
- Aggiunta di opportune funzioni di libreria
- Aggiunta di altri operatori o comandi
- Implementazione della priorità degli operatori
- Aggiunta del concetto di oggetti (contattare il docente)

7.2 Criteri di valutazione

Saranno valutati i seguenti aspetti:

Documentazione Documentazione del codice sorgente, Accuratezza nella descrizione delle scelte progettuali

Progettazione Suddivisione oculata dell'applicazione in moduli funzionali, Progettazione accurata delle interfacce di comunicazione fra i moduli

Algoritmi e strutture dati Scelta motivata delle strutture dati utilizzate, Ottimizzazione delle procedure più usate, Utilizzo opportuno dei costrutti object-oriented del C++

Interfaccia Utente Chiarezza nei messaggi di errore dell'interprete.

Riferimenti bibliografici

[1] The Flex Project. flex: The Fast Lexical analyzer. 2008. http://flex.sourceforge.net/.

[2] The Free Software Foundation. Bison - GNU parser generator. 2007. http://www.gnu.org/software/bison/.

[3] Dimitri van Heesch. Doxygen - source code documentation generator tool. 2008. http://www.doxygen.org.

[1] http://info.iet.unipi.it/~fondii/