Progetto "Interprete di linguaggio LISP-like"

Armando Tacchella

Obiettivo

Realizzare un inteprete di programmi per un linguaggio LISP-like contenente definizioni di variabili, istruzioni di input/output, scelte condizionali e cicli, limitato alle sole operazioni tra variabili di tipo intero. L'interprete deve eseguire le sequenti operazioni:

- Leggere un file, chiamato *file sorgente* nel seguito, in cui è contenuto il programma da interpretare; la sintassi del programma è definita da una grammatica context free non ambigua (riportata di seguito).
- Eseguire il programma contenuto nel file sorgente.
- Richiedere a console l'eventuale inserimento di dati previsto dalle istruzioni di input e visualizzare su console il risultato di espressioni previsto dalle istruzioni di output.

Formato di input

Nella seguente descrizione di grammatica libera da contesto scriviamo la produzione $N \rightarrow \alpha \mid \beta$ come abbreviazione delle produzioni

$$\begin{array}{l} N \rightarrow \alpha \\ N \rightarrow \beta \end{array} \tag{1}$$

Utilizziamo i caratteri corsivi per definire simboli non terminali, ed i caratteri normali per definire simboli terminali. Il simbolo ϵ denota la stringa vuota. Le produzioni della grammatica sono riportate in Figura 1.

Le *parole chiave* di questo semplice linguaggio sono SET, PRINT, INPUT, IF, WHILE per le istruzioni (statement); ADD, SUB, MUL, DIV per gli operatori aritmetici; LT, GT, EQ per gli operatori relazionali; AND, OR, NOT per gli operatori booleani, TRUE e FALSE per le relative costanti. Ad esempio, un possibile programma in input è il seguente:

```
(BLOCK
  (INPUT n)
  (SET result 1)
  (WHILE (GT n 0)
    (BLOCK
      (SET result (MUL result n))
      (SET n (SUB n 1)))
  )
  (PRINT result))
```

Non vi sono assunzioni circa il numero di istruzioni contenute in una particolare riga, mentre spazi, tabulazioni e righe vuote non fanno parte della sintassi del programma, ma devono essere considerati come "spazio vuoto" e ignorati. Il programma precedente potrebbe anche essere scritto correttamente in questo modo:

$$
\begin{aligned}
\textit{program} &\rightarrow \textit{stmt\_block} \\
\textit{stmt\_block} &\rightarrow \textit{statement} \mid (\ \text{BLOCK}\ \textit{statement\_list}\ ) \\
\textit{statement\_list} &\rightarrow \textit{statement}\ \textit{statement\_list} \mid \textit{statement} \\
\\
\textit{statement} &\rightarrow \textit{variable\_def} \mid \\
&\qquad \textit{io\_stmt} \mid \\
&\qquad \textit{cond\_stmt} \mid \\
&\qquad \textit{loop\_stmt} \\
\\
\textit{variable\_def} &\rightarrow (\ \text{SET}\ \textit{variable\_id}\ \textit{num\_expr}\ ) \\
\textit{io\_stmt} &\rightarrow (\ \text{PRINT}\ \textit{num\_expr}\ ) \mid (\ \text{INPUT}\ \textit{variable\_id}\ ) \\
\textit{cond\_stmt} &\rightarrow (\ \text{IF}\ \textit{bool\_expr}\ \textit{stmt\_block}\ \textit{stmt\_block}\ ) \\
\textit{loop\_stmt} &\rightarrow (\ \text{WHILE}\ \textit{bool\_expr}\ \textit{stmt\_block}\ ) \\
\\
\textit{num\_expr} &\rightarrow (\ \text{ADD}\ \textit{num\_expr}\ \textit{num\_expr}\ ) \\
&\quad \mid (\ \text{SUB}\ \textit{num\_expr}\ \textit{num\_expr}\ ) \\
&\quad \mid (\ \text{MUL}\ \textit{num\_expr}\ \textit{num\_expr}\ ) \\
&\quad \mid (\ \text{DIV}\ \textit{num\_expr}\ \textit{num\_expr}\ ) \\
&\quad \mid \textit{number} \\
&\quad \mid \textit{variable\_id} \\
\\
\textit{bool\_expr} &\mid (\text{LT}\ \textit{num\_expr}\ \textit{num\_expr}) \\
&\quad \mid (\text{GT}\ \textit{num\_expr}\ \textit{num\_expr}) \\
&\quad \mid (\text{EQ}\ \textit{num\_expr}\ \textit{num\_expr}) \\
&\quad \mid (\text{AND}\ \textit{bool\_expr}\ \textit{bool\_expr}) \\
&\quad \mid (\text{OR}\ \textit{bool\_expr}\ \textit{bool\_expr}) \\
&\quad \mid (\text{NOT}\ \textit{bool\_expr}) \\
&\quad \mid \text{TRUE} \mid \text{FALSE} \\
\\
\textit{variable\_id} &\rightarrow \textit{alpha\_list} \\
\textit{alpha\_list} &\rightarrow \textit{alpha}\ \textit{alpha\_list} \mid \textit{alpha} \\
\textit{alpha} &\rightarrow \text{a} \mid \text{b} \mid \text{c} \mid \ldots \mid \text{z} \mid \text{A} \mid \text{B} \mid \text{C} \mid \ldots \mid \text{Z} \\
\\
\textit{number} &\rightarrow \text{-}\ \textit{posnumber} \mid \textit{posnumber} \\
\textit{posnumber} &\rightarrow 0 \mid \textit{sigdigit}\ \textit{rest} \\
\textit{sigdigit} &\rightarrow 1 \mid \ldots \mid 9 \\
\textit{rest} &\rightarrow \textit{digit}\ \textit{rest} \mid \epsilon \\
\textit{digit} &\rightarrow 0 \mid \textit{sigdigit}
\end{aligned}
$$

Figura 1: Grammatica di input dell'interprete.

```
(BLOCK(INPUT n)(SET result 1)(WHILE(GT n 0)(BLOCK(SET result(MUL result n))(SET n (SUB n 1))))
(PRINT result))
```

ossia con il numero minimo di spazi per rendere il programma sintatticamente corretto, oppure in questo modo:

```
(BLOCK

  (INPUT n)
  (SET result 1)
  (WHILE (GT n 0)

    (BLOCK
      (SET result (MUL result n))
      (SET n (SUB n 1)))
  )

  (PRINT result)

)
```

ossia con spazi e tabulazioni aggiuntive che vanno ignorate. Non è invece corretto il programma:

```
(PRINT (ADD (MUL14 10)) 25)
```

in quanto `MUL14` non è un nome di operatore corretto (MUL), ma nemmeno il nome di una variabile (gli identificativi non ammettono una parte nuerica).

È inoltre utile fare le seguenti precisazioni:

- La grammatica **non permette** la definizione di variabili che non siano di tipo numerico: SET richiede una *num_expr*.
- Nel caso di INPUT viene imposto il vincolo che il valore letto sia un intero.
- Esiste un unico ambito globale per le variabili.
- Le variabili sono definite dinamicamente (non è necessario dichiararle prima di utilizzarle per la prima volta come in C/C++/Java).

Come ulteriore semplificazione, si assuma che gli interi forniti in ingresso siano sempre rappresentabili in registri a 64 bit (il tipo `long` di Java), ossia per qualsiasi numero n utilizzato si ha che $-2^{63} \leq n < 2^{63}$.

Formato di output

Istruzione SET. Il risultato corrispondente ad una istruzione

(SET *variable_id num_expr*)

deve essere:

- la **creazione** della variable *variable_id*, se non era stata già definita in precedenza, e
- l'**assegnazione** alla variable *variable_id* del **valore** dell'espressione *num_expr*.

Possono verificarsi i seguenti errori:

- la dichiarazione non è sintatticamente corretta,
- *variable_id* è una parola chiave, oppure
- *num_expr* non è sintatticamente corretta, oppure contiene un errore semantico (ad e.s., divisione per zero).

Viene ovviamente generato un errore anche qualora l'espressione *num_expr* contenga la stessa variabile di *variable_expr* e quest'ultima non sia già stata definita in precedenza.

Istruzione INPUT. Il risultato corrispondente ad una istruzione

(INPUT *variable_id*)

deve essere:

- la **creazione** della variable *variable_id*, se non era stata già definita in precedenza, e
- l'**assegnazione** alla variable *variable_id* del valore **letto da console**.

Possono verificarsi i seguenti errori:

- l'istruzione non è sintatticamente corretta,
- *variable_id* è una parola chiave, oppure
- viene inserito da console un valore illecito (sono consentiti **solo** numeri interi positivi e negativi).

Istruzione PRINT. Il risultato corrispondente ad una istruzione

(PRINT *num_expr*)

deve essere la **visualizzazione in console** del **valore** dell'espressione *num_expr*.

Possono verificarsi i seguenti errori:

- l'istruzione non è sintatticamente corretta,
- *num_expr* non è sintatticamente corretta, oppure contiene variabili non definite in precedenza con SET o INPUT, oppure contiene un errore semantico (ad e.s., divisione per zero).

Istruzione IF. Il risultato corrispondente ad una istruzione

(IF *bool_expr* $stmt\_block_1$ $stmt\_block_2$)

deve essere:

- l'esecuzione di $stmt\_block_1$ nel caso in cui *bool_expr* valuti a vero, oppure
- l'esecuzione di $stmt\_block_2$ nel caso in cui *bool_expr* valuti a falso.

Possono verificarsi i seguenti errori:

- la dichiarazione non è sintatticamente corretta,
- *bool_expr* non è sintatticamente corretta o contiene variabili non definite in precedenza, oppure
- uno dei due *stmt_block* non è sintatticamente corretto, oppure, se eseguito, contiene un errore semantico (ad e.s., divisione per zero).

Istruzione WHILE. Il risultato corrispondente ad una istruzione

(WHILE *bool_expr stmt_block*)

deve essere l'esecuzione di *stmt_block* fintanto che l'espressione *bool_expr* valuta a vero; l'esecuzione non avviene se *bool_expr* è falsa in partenza. Possono verificarsi i seguenti errori:

- la dichiarazione non è sintatticamente corretta,
- *bool_expr* non è sintatticamente corretta o contiene variabili non definite in precedenza, oppure
- lo *stmt_block* non è sintatticamente corretto, oppure, se eseguito, contiene un errore semantico (ad e.s., divisione per zero).

Istruzione BLOCK. Il risultato corrispondente ad una istruzione

(BLOCK $statement_1 \ldots statement_n$)

deve essere l'esecuzione **in sequenza** dei diversi statement. Possono verificarsi i seguenti errori:

- la dichiarazione non è sintatticamente corretta,
- per qualche i, $statment_i$ non è sintatticamente corretto, contiene variabili non definite in precedenza, oppure contiene un errore semantico (ad e.s., divisione per zero).

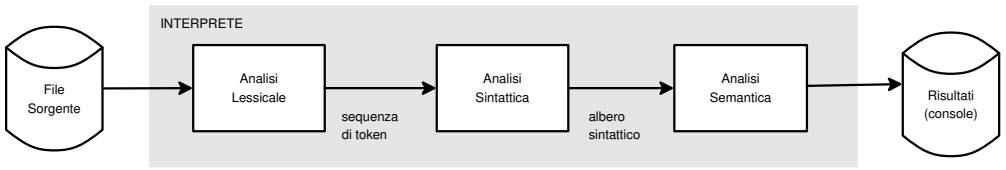

Figura 2: Architettura funzionale dell'interprete.

Espressioni. Le espressioni numeriche vengono interpretate con l'ovvia semantica per valori ed operatori: somma per ADD, sottrazione per SUB, ecc. Nelle espressioni booleane, gli operatori relazionali hanno la seguente semantica:

- GT sta per "greater than", quindi (GT a b) è vero se il valore di a è strettamente maggiore di quello di b;
- Analogamente LT sta per "lesser than" e EQ sta per "equal" con il relativo significato inteso.

Gli operatori booleani AND, OR, NOT hanno anch'essi l'ovvia semantica; AND e OR sono **cortocircuitati**, ossia

- (a AND b) nel caso in cui a valuti a falso è immediatamente valutata a falso (b non viene valutato);
- (a OR b) nel caso in cui a valuti a vero è immediatamente valutata a vero (b non viene valutato).

Ad esempio, in corrispondenza del programma sopra mostrato, l'output sarebbe il seguente:

```
3628800
```

Gli errori presenti nel file devono essere trattati sollevando eccezioni che vanno gestite opportunamente con l'uso di `try ... catch` (*i.e.*, il main non deve propagare eccezioni).

Suggerimenti per il progetto e l'implementazione

Per semplificare la soluzione conviene suddividere l'implementazione in tre parti distinte, utilizzate in sequenza: analisi lessicale, analisi sintattica (parsing e costruzione dell'albero sintattico) e analisi semantica (valutazione). La struttura corrispondente del programma a livello funzionale è mostrata in Figura 2.

Analisi lessicale. Ha il compito di leggere il file sorgente contenente il programma e iterarlo fornendo in uscita la sequenza degli *elementi lessicali* (*parole* o *token*) del programma. Formalmente, per la grammatica dei file sorgente, un elemento lessicale corrisponde ad uno dei seguenti gruppi:

- una parola chiave: SET, PRINT, INPUT, IF, WHILE, ADD, SUB, MUL, DIV, GT, LT, EQ, AND, OR, NOT, TRUE, FALSE;
- una parentesi aperta o chiusa;
- un numero (definito con le regole *number*);
- un variabile (definita con le regole *variable_id*);

L'utilizzo dei token corrisponde a utilizzare per la fase successiva di analisi sintattica (parsing) le regole mostrate in Figura 3 in cui i token sono stati indicati utilizzando il grassetto. Nel caso in cui l'analizzatore lessicale incontri un token non previsto, restituisce un messaggio di errore.

$$
\begin{aligned}
\textit{program} &\rightarrow \textit{stmt\_block} \\
\textit{stmt\_block} &\rightarrow \textit{statement} \mid \textbf{(}\ \textbf{BLOCK}\ \textit{statement\_list}\ \textbf{)} \\
\textit{statement\_list} &\rightarrow \textit{statement}\ \textit{statement\_list} \mid \textit{statement} \\
\\
\textit{statement} &\rightarrow \textit{variable\_def} \mid \\
&\quad\ \textit{io\_stmt} \mid \\
&\quad\ \textit{cond\_stmt} \mid \\
&\quad\ \textit{loop\_stmt} \\
\\
\textit{variable\_def} &\rightarrow \textbf{(}\ \textbf{SET}\ \textit{variable\_id}\ \textit{num\_expr}\ \textbf{)} \\
\textit{io\_stmt} &\rightarrow \textbf{(}\ \textbf{PRINT}\ \textit{num\_expr}\ \textbf{)} \mid \textbf{(}\ \textbf{INPUT}\ \textit{variable\_id}\ \textbf{)} \\
\textit{cond\_stmt} &\rightarrow \textbf{(}\ \textbf{IF}\ \textit{bool\_expr}\ \textit{stmt\_block}\ \textit{stmt\_block}\ \ \textbf{)} \\
\textit{loop\_stmt} &\rightarrow \textbf{(}\ \textbf{WHILE}\ \textit{bool\_expr}\ \textit{stmt\_block}\ \textbf{)} \\
\\
\textit{num\_expr} &\rightarrow \textbf{(}\ \textbf{ADD}\ \textit{num\_expr}\ \textit{num\_expr}\ \textbf{)} \\
&\quad \mid \textbf{(}\ \textbf{SUB}\ \textit{num\_expr}\ \textit{num\_expr}\ \textbf{)} \\
&\quad \mid \textbf{(}\ \textbf{MUL}\ \textit{num\_expr}\ \textit{num\_expr}\ \textbf{)} \\
&\quad \mid \textbf{(}\ \textbf{DIV}\ \textit{num\_expr}\ \textit{num\_expr}\ \textbf{)} \\
&\quad \mid \textbf{\textit{number}} \\
&\quad \mid \textbf{\textit{variable\_id}} \\
\\
\textit{bool\_expr} &\mid \textbf{(LT}\ \textit{num\_expr}\ \textit{num\_expr}\ \textbf{)} \\
&\quad \mid \textbf{(GT}\ \textit{num\_expr}\ \textit{num\_expr}\ \textbf{)} \\
&\quad \mid \textbf{(EQ}\ \textit{num\_expr}\ \textit{num\_expr}\ \textbf{)} \\
&\quad \mid \textbf{(AND}\ \textit{bool\_expr}\ \textit{bool\_expr}\ \textbf{)} \\
&\quad \mid \textbf{(OR}\ \textit{bool\_expr}\ \textit{bool\_expr}\ \textbf{)} \\
&\quad \mid \textbf{(NOT}\ \textit{bool\_expr}\ \textbf{)} \\
&\quad \mid \textbf{TRUE} \mid \textbf{FALSE}
\end{aligned}
$$

Figura 3: Grammatica astratta per l'analisi sintattica.

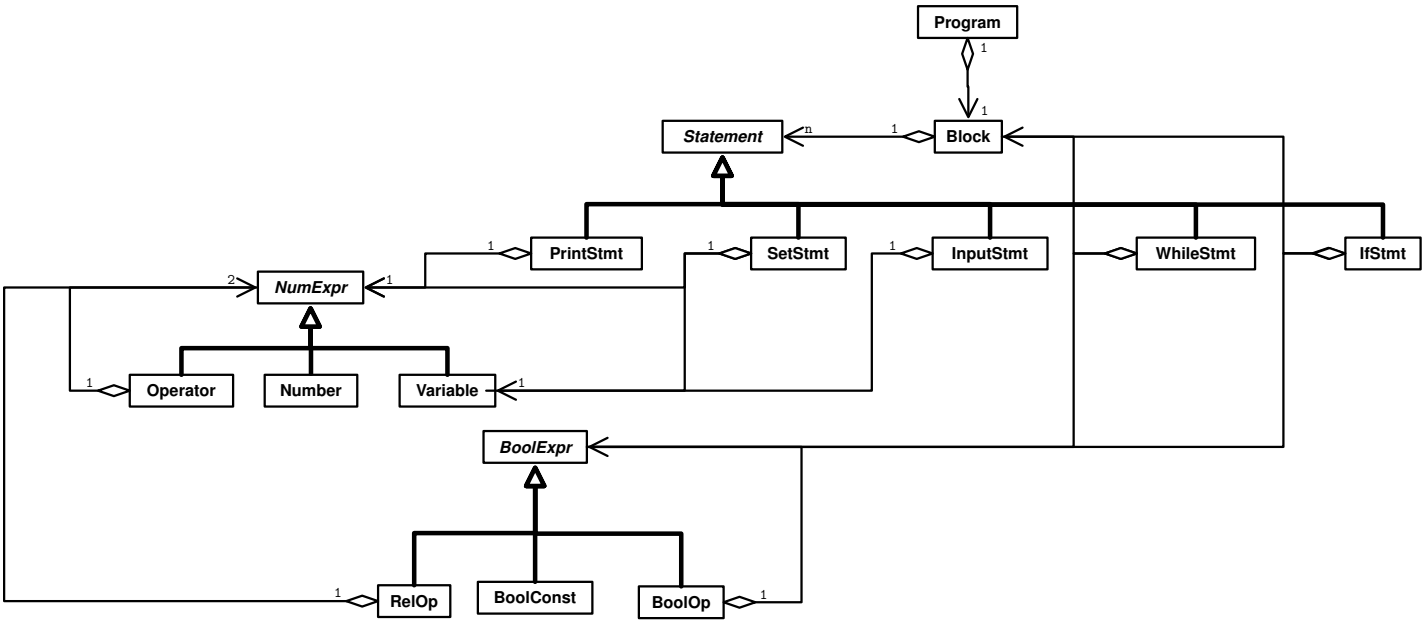

Figura 4: Gerarchia delle classi per la costruzione dell'albero sintattico (pattern Interpreter).

Analisi sintattica. Ha il compito di leggere la sequenza di token fornita dall'analizzatore lessicale e di controllare la correttezza sintattica dei costrutti. Fornisce in uscita una descrizione sotto forma di albero del programma in ingresso (c.d. *albero sintattico*) L'albero sintattico può essere costruito utilizzando il pattern

"Interpreter" sulla base della gerarchia delle classi mostrata in Figura 4. In caso di errori sintattici, si restituisce un messaggio di errore al primo incontrato.

Analisi semantica. Ha il compito di eseguire/valutare gli statement tenendo conto degli assegnamenti, delle istruzioni di input/output delle scelte condizionali e dei cicli. Costruisce la tabella dei simboli e aggiunge elementi man mano che vengono incontrate le definizioni di variabili e vengono assegnati i relativi valori. Identifica gli errori semantici quali le divisioni per zero (overflow e underflow non vegono considerati). Opera comunque una visita dell'albero sintattico e produce direttamente in output quanto richiesto dalle istruzioni PRINT. Per eseguire la valutazione delle espressioni si può utilizzare un oggetto "Visitor" con opportuni metodi che consentano l'analisi dei vari elementi degli oggetti "Interpreter".