rpn-lang

- Enrico Bacis, Daniele E. Ciriello -

Relazione del progetto rpn-lang di Enrico Bacis e Daniele E. Ciriello del corso di Linguaggi e Compilatori tenuto dal Professor Giuseppe Psaila presso l'Università degli Studi di Bergamo nell'anno accademico 2012/2013.

Versione del Documento: 1.0 - 20130320

Indice

1 Introduzione

L'obiettivo del progetto è la creazione di un semplice linguaggio di programmazione, ci si è però concentrati non tanto sulla specificità del linguaggio ma sulla sua generalizzazione e quindi sull'usabilità di esso. Infatti esistono due diversi tipi di linguaggi che si possono progettare:

1. Linguaggio *single-purpose* che risolve un problema specifico
2. Linguaggio *general-purpose* che si applica a diverse problematiche

Ci si è dunque concentrati sulla seconda alternativa focalizzando l'attenzione sulla creazione di un linguaggio Turing-completo[1]. In questo modo è possibile esprimere qualsiasi tipo di problema e sviluppare così un linguaggio che più si avvicina al tipo di linguaggi di programmazione a cui si è abituati piuttosto che a un problema che risulta essere più algoritmico e relativo ad una tematica specifica (rischiando così di spendere più tempo nel risolvere i problemi della tematica piuttosto che concentrarsi sul linguaggio).

2 Il linguaggio

Seguendo le aspettative, si è realizzato un linguaggio con:

- Valutazione di espressioni in notazione prefissa, da cui il nome rpn-lang[2] con l'utilizzo di operatori unari e binari.
- Gestione degli errori e dei warning semantici.
- Assegnamento di valori di tipo **double** a variabili senza la loro preventiva dichiarazione.
- Definizione di funzioni a qualsiasi livello dello stack (*higher-order functions*).
- Gestione dei *record di attivazione* e ricerca delle variabili e delle funzioni in tutto lo stack delle chiamate.
- Definizione dei costrutti **if** e **while** e conseguente valutazione delle espressioni booleane.
- Gestione dell'input e dell'output delle variabili.

2.1 Grammatica

La grammatica è stata prima scritta in notazione BNF seguendo i precetti per la costruzione di grammatiche formali studiati durante il corso, e successivamente, utilizzando il tool ANTLR, trasformata in EBNF per poter meglio rappresentare quali siano gli aspetti strutturali del linguaggio. In questo documento verrà presentata solamente la grammatica EBNF.

[1] Un linguaggio si dice *Turing-completo* quando permette la descrizione di qualsiasi algoritmo. Un linguaggio imperativo è Turing-completo se offre un costrutto condizionale (come *if* e la possibilità di cambiare locazioni di memoria (ad esempio permettendo il salvataggio di un numero arbitrario di variabili).

[2] Reverse Polish Notation, Notazione Polacca Inversa

```
grammar Rpn;

options {
    language = Java;
    k = 1;
}

@header {...}

@lexer::header {...}

@members {
private Environment env;
private Semantic sem;

void init () {
    env = new Environment();
    sem = new Semantic(env, this);
}

...
}

@lexer::members {}

start
@init { init (); }
    :   ( statement[true] )* EOF
    ;

statement[boolean execute]
    :   assign[execute]
    |   output[execute]
    |   input[execute]
    |   ifstat[execute]
    |   whilestat[execute]
    |   def[execute]
    ;

assign[boolean execute]
    :   ID '=' expr[execute] ';'
        { if (execute) sem.assign($ID, $expr.value); }
    ;

output[boolean execute]
    :   '<<' expr[execute] ';'
        { if (execute) sem.output($expr.text + " = " + $expr.value); }
    ;
```

```
input[boolean execute]
    :   '>>' ID ';'
        { if (execute) sem.input($ID); }
    ;

ifstat[boolean execute]
    :   'if' cond[false] il=slist[false] ( 'else' el=slist[false] )?
        { if (execute) sem.ifstat($cond.start, $il.start, $el.start); }
    ;

whilestat[boolean execute]
    :   'while' cond[false] wl=slist[false]
        { if (execute) sem.whilestat($cond.start, $wl.start); }
    ;

cond[boolean execute] returns [boolean satisfied]
    :   '(' tk=CONOP l=expr[execute] r=expr[execute] ')'
        { if (execute) $satisfied = sem.cond($l.value, $r.value, $tk); }
    ;

def[boolean execute]
@init { List<Token> args = new ArrayList<Token>(); }
    :   'def' name=ID '(' ( arg=ID { args.add($arg); }
        ( ',' arg=ID { args.add($arg); } )* )? ')' deflist[false]
        { if (execute) sem.def($name.text, args, $deflist.start); }
    ;

deflist[boolean execute] returns [double value]
    :   '{' ( statement[execute] )* ret[execute] '}'
        { if (execute) $value = $ret.value; }
    ;

slist[boolean execute]
    :   '{' ( statement[execute] )* '}'
    ;

ret[boolean execute] returns [double value]
    :   'return' expr[execute] ';'
        { if (execute) $value = $expr.value; }
    ;

expr[boolean execute] returns [double value]
options { k = 2; }
    :   tk=BINOP a=expr[execute] b=expr[execute]
        { if (execute) $value = sem.binop($a.value, $b.value, $tk); }
    |   tk=UNAOP u=expr[execute]
        { if (execute) $value = sem.unaop($u.value, $tk); }
    |   NUM
        { if (execute) $value = sem.getNumber($NUM); }
```

```
    |   ID
        { if (execute) $value = sem.getVar($ID); }
    |   fun=call[execute]
        { if (execute) $value = $fun.value; }
    ;

call[boolean execute] returns [double value]
@init { List<Double> args = new ArrayList<Double>(); }
    :   name=ID '('
        ( arg=expr[execute] { args.add($arg.value); }
          ( ',' arg=expr[execute] { args.add($arg.value); } )*
        )? ')'
        { if (execute) $value = sem.call($name, args); }
    ;

UNAOP
    :   ('~' | '!')
    ;

BINOP
    :   ('+' | '-' | '*' | '/')
    ;

CONOP
    :   ('<' | '<=' | '==' | '>=' | '>')
    ;

ID
    :   ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*
    ;

NUM
    :   ( ('1'..'9') ('0'..'9')* | '0' ) ( '.' ('0'..'9')+ )?
    ;

WS
    :   ('\t' | '\r' | '\n' | ' ')+ { $channel=HIDDEN; }
    ;

SL_COMMENT
    :   '#' ~('\r'|'\n')* { $channel=HIDDEN; }
    ;

SCAN_ERROR
    :   .
    ;
```

2.2 Proprietà della Grammatica

La grammatica è non ambigua, e può essere processata con un parser di tipo **LL(1)** con l'unica eccezione della regola *expr* che ha bisogno di una prospezione di 2 tokens. Questo è un problema di tutte le grammatica che permette di invocare delle funzioni, infatti non si può distinguere una variabile dall'invocazione di una funzione se sia le variabili che i nomi delle funzioni appartengono allo stesso insieme di identificatori. Le soluzioni analizzate sono tre:

1. Utilizzo di un token non presente nella grammatica, prima della chiamata della funzione (ad esempio chiamare le funzioni con *:f()* invece che con *f()*.
2. Separare lo spazio di identificatori delle variabili da quello delle funzioni (ad esempio riservare gli identificatori che iniziano con *f_* per le funzioni.
3. Aumentare a due i token di prospezione per la regola *expr* in modo da poter distinguere i nomi delle variabili dalle chiamate di funzioni, che sono seguite da *'('*.

Si è optato per la terza alternativa poiché quella più naturale e non computazionalmente complessa per il parser, a patto di mantenere la prospezione di un solo token altrove.

2.3 Attributi

È stata utilizzata una grammatica ad attributi per risolvere l'albero semantico generato dal parser. In particolare si sono utilizzati i seguenti attributi principali:

value Attributo *sintetizzato* che contiene il valore dell'espressione calcolata fino a quel momento. Risale l'albero fino alla regola *expr*.

execute Attributo *ereditato* che informa la semantica se si rende necessario eseguire l'azione legata alla regola. Infatti quando ad esempio vengono definite le funzioni, oppure durante il parsing dei blocchi *if-else* e *while*, non bisogna eseguire le azioni ma fare il parsing di tutto il blocco e poi successivamente eseguirlo in base alla condizione (nel caso di if-else e while) oppure al momento dell'invocazione per il caso delle funzioni.

La grammatica così partizionata è adatta a un valutatore **one-sweep**.

3 Costrutti

Vengono di seguito esposti i principali costrutti del linguaggio Rpn-Lang.

3.1 Espressioni

Un'espressione *expr* rappresenta una qualsiasi espressione aritmetica in notazione polacca antefissa, ossia nel formato *operatore operando operando"*, se ad esempio si vuole rappresentare l'espressione $a + b$ in RPN, tale espressione diventa $+\,a\,b$. L'inserimento di un'espressione in una notazione diversa dalla notazione polacca inversa all'interno di codice Rpn-Lang causa la segnalazione di un'errore sintattico.

Sono stati definiti 4 operatori binari e 2 operatori unari:

$+\,a\,b$	somma $(a + b)$
$-\,a\,b$	differenza $(a - b)$
$*\,a\,b$	moltiplicazione $(a * b)$
$/\,a\,b$	divisione (a/b)
$\sim\ a$	inversione di segno $(-a)$
$!\,a$	fattoriale $(a!)$

3.2 Gestione degli errori

- La divisione per 0 genera un errore semantico. L'applicazione del fattoriale su un numero non intero oppure su un numero negativo genera un warning (nel primo caso il numero viene arrotondato all'intero inferiore, mentre nel secondo caso il numero negativo viene portato a 0 per poter applicare il fattoriale).
- Quando viene invocata una funzione oppure si cerca di accedere al valore di una variabile che non è stata definita, viene generato un errore semantico, tuttavia il compilatore prosegue sempre, infatti la comodità dell'utilizzare il tipo di dato Double di Java è che può assumere il valore *Not a Number* che è perfetto per i nostri scopi.

3.3 Assegnamenti

Gli assegnamenti sono rappresentati dalla grammatica come una sequenza data da *"var = expr"*, in tal caso il compilatore assegna il risultato dell'espressione *expr* alla variabile *var*.

Come detto in precedenza, Rpn-Lang utilizza uno stack di record di attivazione per salvare le variabili e le funzioni definite all'interno di funzioni o di blocchi *if-else* e *while*. Ogni blocco può accedere anche a tutte le variabili e tutte le funzioni definite negli altri blocchi dello stack, a patto che gli identificativi non siano stati oscurati con la definizione di nuove variabili o funzioni aventi lo stesso nome all'interno del blocco. Le variabili e le funzioni definite all'interno dei blocchi esistono sono all'interno del blocco e vengono deallocate successivamente.

3.4 Funzioni

È possibile definire funzioni tramite la sintassi

$$def\ id\_funzione\,(\,lista\_parametri\,)\,\{\,lista\_istruzioni\,\}$$

Per poter invece invocare la funzione si usa invece la sintassi

$$id\_funzione\,(\,lista\_parametri\,)$$

Non avendo una struttura intermedia di supporto nella quale il codice sorgente viene compilato (di fatto ciò che sviluppiamo con ANTLR è un interprete e non un compilatore vero e proprio, visto che non viene prodotto alcun compilato), si è cercato un modo per poter richiamare dei blocchi quando richiesti. Seguendo il suggerimento di Terrence Parr[3] nel suo libro *Language Implementation Patterns* abbiamo seguito la seguente procedura:

1. Avere un attributo ereditato che dice al parser se deve eseguire l'istruzione letta o no.
2. Salvare la posizione del primo Token della regola che contiene il blocco.
3. Quando il blocco deve essere richiamato si salva la posizione corrente del cursore dei Token[4] e successivamente lo si fa puntare all'indice che avevamo salvato, poi si invoca la regola del parser relativa al blocco da eseguire.
4. Una volta terminato il blocco, che può anche terminare con la sintetizzazione di un attributo, si ripristina il cursore dei Token alla posizione che aveva prima di invocare il blocco.

Quando le funzioni vengono invocate, la lista dei parametri formali e quella dei parametri attuali viene confrontata, e se hanno la stessa dimensione, allora si creano le variabili relative ai parametri nel record di attivazione della funzione.

Tutte le funzioni in Rpn-Lang devono finire con lo statement **return**, e questo statement può essere usato solo in questa occasione. Quindi le funzioni hanno un solo punto di uscita, questo semplifica di molto la lettura delle funzioni. In questo modo poi non è quindi possibile programmare delle procedure, ma solo delle funzioni (in uno stile che ricorda le funzioni programmabili sulle calcolatrici HP, a cui il nostro linguaggio si ispira).

In Rpn-Lang è possibile definire delle **Higher-order Functions**, ossia si possono definire funzioni all'interno di altri blocchi (come altre funzioni, oppure come blocchi *if-else* o *while*) visibili solamente all'interno di questo blocco, il cui ciclo di vita è intrinsecamente legato a quello del blocco che le contiene. Queste funzioni sono visibili solamente all'interno del blocco in cui sono state definite.

Il metodo utilizzato da Rpn-Lang per cercare le funzioni è analogo a quello descritto per le variabili, ossia le funzioni vengono ricercate in tutto lo stack.

[3] Ideatore di ANTLR.

[4] Il cursore dei Token è l'analogo del *Program Counter* nelle CPU.

3.5 Istruzioni if

Rpn-Lang permette la definizione di istruzioni condizionali attraverso il costrutto

$$if\ condizione\ \{\ statements\_vero\ \}\ [\ else\ \{\ statements\_falso\ \}\ ]$$

Una condizione è rappresentata dalla sintassi

$$(operatore\ operando\ operando)$$

Quindi la sintassi è analoga a quella delle espressioni, utilizzando però gli operatori di confronto. La particolarità rispetto agli altri linguaggi di programmazione è che anche le condizioni sono espresse in logica prefissa. Il funzionamento interno è analogo a quanto esposto per le funzioni.

3.6 Cicli While

Rpn-Lang permette l'utilizzo di cicli while mediante la classica sintassi

$$while\ (\ condizione\ )\ \{\ lista\_istruzioni\ \}$$

Come nel classico while-loop ad ogni iterazione viene controllata la condizione *cond*, se essa è verificata vengono eseguite le istruzioni nella lista istruzioni. Anche qui il funzionamento interno è quello esposto per le funzioni, ossia il cursore dei Token viene spostato indietro e avanti per verificare la condizione ed eseguire (se la condizione viene verificata) il blocco delle istruzioni.

3.7 Input e Output

Rpn-Lang permette anche l'interazione con l'utente, infatti è possibile richiedere l'inserimento del valore di una variabile all'utente attraverso la sintassi

$$\gg VAR;$$

La richiesta di inserimento della variabile viene mostrata a schermo e viene ripetuta finché l'utente non inserisce un numero decimale valido. Per mostrare invece a video il valore di una variabile o di una espressione si utilizza la sintassi

$$\ll VAR\_O\_EXPR;$$

In questo modo è possibile descrivere algoritmi generici che prendono come input un dato dell'utente, lo elaborano e ne mostrano il risultato della computazione.

4 Conclusioni

Ovviamente avendo la libertà di poter scrivere un proprio linguaggio general-purpose il rischio è quello di poter descrivere talmente tanti costrutti che si potrebbe continuare ad implementare miglioramenti al compilatore per moltissimo tempo. Infatti anche nel piccolo di questo progetto, siamo partiti dal voler realizzare semplicemente un linguaggio che operasse su variabili ed espressioni scritte in notazione polacca inversa e siamo giunti ad aggiungere le funzioni, le higher-order functions, i costrutti if-else e while, l'input e l'output.

Avremmo voluto aggiungere molte altre cose ma a un certo punto abbiamo dovuto decidere di concludere. Tuttavia abbiamo avuto modo di sperimentare come avviene la costruzione di un linguaggio abbastanza completo nei suoi costrutti, trovando il processo anche divertente. Crediamo che il progetto sia stato interessante, anche perché ci ha permesso di toccare con mano le problematiche dei linguaggi che fino ad oggi avevamo semplicemente utilizzato, ma senza mai essere stati dalla parte degli ideatori del linguaggio.