Implementazione di un Compilatore e di una Macchina Virtuale per la Compilazione e l'Esecuzione di un Linguaggio Simple

Nicola Ricci, Francesco Valle

Tema d'anno dell'esame di *Compilatori e Interpreti*
A.A. 2008/2009
Docente: Prof. G. Piscitelli

18 settembre 2009

Indice

Capitolo 1

Introduzione

In questo tema d'anno verrà affrontata una tipologia di problemi leggermente diversa dalla semplice realizzazione di un compilatore di un linguaggio già esistente.

Si è infatti reputato più interessante pensare di chiudere il ciclo della compilazione con l'effettiva esecuzione del codice sorgente mediante una macchina virtuale.

Per far ciò, però, è stato necessario limitare il linguaggio (usare un linguaggio completo come il C avrebbe reso praticamente impossibile l'implementazione della macchina virtuale!) e si è quindi deciso di utilizzare un linguaggio chiamato Simple che riporta (come vedremo in seguito) solo alcuni tratti fondamentali di un linguaggio di programmazione.

1.1 I tools utilizzati

I tools utilizzati per la realizzazione di questo tema d'anno e necessari per la sua esecuzione, sono due dei più usati nell'ambito della realizzazione di compilatori:

- L'analizzatore lessicale *Flex*
- L'analizzatore semantico *Bison*

Va segnalato che per lo studio effettuato ci si è basati su sistemi UNIX-based, e che quindi tutti gli eventuali comandi indicati nel corso di questa documentazione saranno certamente funzionanti solo su sistemi di tale tipologia.

Capitolo 2

Compilatori e Interpreti

Un **compilatore** è un programma che traduce un programma scritto in un linguaggio sorgente in uno logicamente equivalente scritto in un linguaggio target. Il linguaggio sorgente è di solito un linguaggio di alto livello (ad esempio C, Java...), mentre il linguaggio target è il linguaggio macchina (generalmente il linguaggio macchina ospitante). In pratica il compilatore traduce le operazioni scritte rispettando la grammatica del linguaggio sorgente in operazioni computazionali appartenenti alla macchina ospitante.

Il compilatore è composto da diverse fasi ognuna delle quali processa l'input restituito dalla fase precedente e passa il proprio output alla fase successiva.

- L'**analizzatore lessicale** (o *scanner*) raggruppa i caratteri letti dal file sorgente in unità lessicali, dette *token*. L'analizzatore lessicale riceve quindi in input uno *stream di caratteri* e restituisce in output un *stream di token*. Per la definizione dei token riconosciuti dallo scanner si utilizzano solitamente le espressioni regolari. Lo *scanner* è solitamente implementato mediante una macchina a stati finiti.

  Lex e *Flex* sono tools utilizzati per la generazione automatica di scanners;

- L'**analizzatore sintattico** (o *parser*), invece, raggruppa i token in frasi grammaticali. Generalmente le frasi grammaticali del programma sorgente sono rappresentate tramite un albero di derivazione (o *parse tree*). Per definire le regole sintattiche del programma sorgente riconosciuto dal *parser* si utilizzano solitamente grammatiche libere dal contesto (*Context-free grammars*).

  Yacc e *Bison* sono tools per la generazione automatica di parsers;

- L'**analizzatore sematico** controlla se ci sono errori semantici nel programma sorgente e acquisisce l'informazione sui tipi (*Type checking*) che verrà usata nella fase successiva di generazione del codice. L'output generato da questa fase è un *parse tree* annotato; Per descrivere la semantica statica del programma sorgente vengono usate grammatiche ad attributi.

  Questa fase è combinata con il *parser*. Durante il *parsing* informazioni sulle variabili utili per il *context-checking* sono immagazzinate in una tabella chiamata *Symbol Table*.

- Il **generatore di codice**, infine, trasforma il *parse tree* in codice oggetto.

Diversamente da un compilatore, un **interprete** è un programma che riceve in input un programma in un linguaggio sorgente e, di volta in volta, traduce la singola istruzione in linguaggio macchina. Un interprete può essere utilizzato sia per tradurre un linguaggio sorgente, sia per interpretare il codice oggetto per una macchina virtuale.

Molti tipi di traduttori sono spesso utilizzati in congiunzione con il compilatore per facilitare l'esecuzione del programma. L'*Assembler* è un traduttore che effettua un traslitterazione uno a uno del programma sorgente (linguaggio *Assembly*) in linguaggio macchina. Tuttavia, alcuni compilatori generano codice *Assembly* intermedio che viene poi in seguito tradotto in effettivo codice macchina dall'Assembler.

In questo tema d'anno realizzeremo un compilatore per un semplice linguaggio chiamato *Simple*.

La grammatica context-free utilizzata per tale linguaggio è la seguente:

```
program ::= VARIABLES [ declarations ] START command sequence END
declarations ::= INTEGER [ id seq ] IDENTIFIER .
id seq ::= id seq... IDENTIFIER ,
command sequence ::= command... command
command ::= SKIP ;
          | IDENTIFIER := expression ;
          | IF exp THEN command sequence ELSE command sequence FI ;
          | WHILE exp DO command sequence END ;
          | READ IDENTIFIER ;
          | WRITE expression ;
          | FOR (IDENTIFIER ::= expression; expression;
              IDENTIFIER ::= expression) command sequence END;
          | DO command sequence WHILE (expression) END;
expression ::= NUMBER | IDENTIFIER | '(' expression ')'
```

```
| expression + expression | expression - expression
| expression * expression | expression / expression
| expression = expression
| expression < expression
| expression > expression
```

I simboli, terminali e non terminali, sono annotati rispettivamente in lettere maiuscole e minuscole.

Capitolo 3

Lo Scanner

Come detto nel capitolo precedente, il compito di un analizzatore lessicale è, data una sequenza di caratteri di un alfabeto, verificare se la sequenza può essere raggruppata in token, tale che ogni token appartenga al lessico. In poche parole, uno scanner è un programma capace di riconoscere dei pattern in un testo; i pattern, usualmente, sono descritti sotto forma di espressioni regolari.

Per la generazione automatica di scanner in ambiente Unix è molto diffuso il tool *Flex*. *Flex* riceve in input un file di testo, la cui struttura approfondiremo più avanti, e restituisce come output un programma sorgente scritto in linguaggio C che realizza il *pattern matching* dei pattern specificati nel file di input.

Di seguito si riporta la struttura di un file di input per *Flex*:

```
Definizioni
%%
Regole
%%
Procedure di supporto
```

Nella prima sezione vanno inserite racchiuse tra i caratteri '%{' e '%}' le definizioni di costanti e/o macro personalizzate del programma C che verrà realizzato; questa parte di testo verrà letteralmente copiata nel programma C generato. Proprio qui è necessario inserire il file `Simple.tab.h`, generato da *Bison*, e contenente la definizione dei tokens generabili in funzione della grammatica. La prima sezione contiene anche le definizioni usate nelle espressioni regolari, nel nostro caso abbiamo usato `DIGIT`, che così come lo abbiamo definito, rappresenta un simbolo tra 0 e 9, e `ID` per rappresentare le variabili.

```
%{
```

```
#include "Simple.tab.h" /* The tokens */
%}
DIGIT     [0-9]
ID        [a-z][a-z0-9]*
%%
Token definitions and actions
%%
C subroutines
```

Dopo la sequenza di caratteri %% inizia la sezione di definizione dei pattern con le relative azioni da intraprendere una volta riconosciuti. Notiamo che se viene riconosciuto il pattern degli spazi bianchi, caratteri di tabulazione o caratteri *newline* non viene intrapresa nessuna azione; tutti i singoli caratteri sono semplicemente ritornati così come sono al parser.

```
Dichiarazioni
%%
":="     { return(ASSGNOP); }
{DIGIT}+ { yylval.intval = atoi( yytext );
           return(NUMBER);   }
do       { return(DO);       }
else     { return(ELSE);     }
end      { return(END);      }
fi       { return(FI);       }
if       { return(IF);       }
start    { return(START);    }
integer  { return(INTEGER);  }
variables{ return(VARIABLES);}
read     { return(READ);     }
skip     { return(SKIP);     }
then     { return(THEN);     }
while    { return(WHILE);    }
write    { return(WRITE);    }
for      { return(FOR);      }
{ID}     { yylval.id = (char *) strdup(yytext);
           return(IDENTIFIER);     }
\n        ++num_lines;
[ \t]+   /* eat up whitespace */
.        { return(yytext[0]);}
"//"[^\n]*{}   /* do nothing */
%%
Subroutine di supporto
```

Il valore associato ai token è un intero che lo scanner passa al parser ogni volta che riconosce un token. La variabile `yylval` è accessibile sia al parser che allo scanner e può contenere informazioni aggiuntive sul rispettivo token.

La terza sezione è vuota nel nostro caso, ma può contere del codice C associato alle azioni.

Capitolo 4

Il Parser

Come abbiamo già visto nel Capitolo 2, il parser ottiene una stringa di token dall'analizzatore lessicale e tenta di costruire l'albero di derivazione o *parse tree* della stringa di token in accordo con la grammatica del linguaggio (grammatica libera dal contesto). Possiamo utilizzare software per la generazione automatica di parser per costruire parser per un ampio range di linguaggi di programmazione.

Bison è un generatore automatico di parser che, ricevendo in input una grammatica context-free, costruisce un programma in C in grado di riconoscere frasi appartenenti al linguaggio generato dalla grammatica data. Un file di input in *Bison* ha la seguente forma:

```
Dichiarazioni
%%
Regole
%%
Sezione routines ausilarie
```

Nella sezione dichiarazione (o definizioni) si definiscono alcune informazioni globali da fondamentali per interpretare la grammatica. Questa sezione può contenere, inoltre, direttive sulla precedenza e l'associatività degli operatori.

```
%start program
%token NUMBER
%token IDENTIFIER
%token IF WHILE DO
%token SKIP THEN ELSE FI END
%token INTEGER READ WRITE LET IN VARIABLES START
%token ASSGNOP
%left '-' '+'
%left '*' '/'
%%
```

```
Regole
%%
Sezione routines
```

La sezione regole è composta da una o più produzioni espresse nella forma:

```
A : BODY;
```

dove `A` rappresenta un simbolo non terminale e `BODY` rappresenta una sequenza di uno o più simboli sia terminali che non terminali. Le produzioni sono separate dal ';', il simbolo '::=' usato dalla BNF è sostituito dal simbolo ':'. I simboli non terminali sono scritti in caratteri minuscoli, mentre i simboli terminali in caratteri maiuscoli. Grazie alla definizione della precedenza tra gli operatori è stato possibile semplificare di molto la grammatica.

```
C and parser declarations
%%
program : VARIABLES declarations START commands END ;
declarations : /* empty */
    | INTEGER id seq IDENTIFIER '.'
;
id seq : /* empty */
    | id seq IDENTIFIER ','
;
commands : /* empty */
    | commands command ';'
;
command : SKIP
    | READ IDENTIFIER
    | WRITE exp
    | IDENTIFIER ASSGNOP exp
    | IF exp THEN commands ELSE commands FI
    | WHILE exp DO commands END
    | FOR '(' IDENTIFIER ASSGNOP exp ';' exp ';'
        IDENTIFIER ASSGNOP exp ')' commands END
    | DO commands WHILE '(' exp ')' END
;
exp : NUMBER
    | IDENTIFIER
    | exp '<' exp
    | exp '=' exp
    | exp '>' exp
    | exp '+' exp
    | exp '-' exp
    | exp '*' exp
```

```
      | exp '/' exp
      | '(' exp ')'
;
%%
C subroutines
```

La terza sezione del file di input *Bison* è usata per definire delle routines C. In questa sezione deve esserci una funzione `main()` che chiami a sua volta la funzione `yyparse()`. `yyparse()` è la funzione principale generata da *Bison* che si serve della funzione *yylex* generata da *Flex* ogni volta che ha bisogno di un nuovo token. La funzione `yyerror()` è invece usata dal parser quando deve segnalare errori nel programma sorgente. Un esempio delle funzioni `main()` e `yyerror()` è il seguente:

```
C and parser declarations
%%
Grammar rules and actions
%%
main( int argc, char *argv[] )
{ extern FILE *yyin;
   ++argv; --argc;
   yyin = fopen( argv[0], "r" );
   yydebug = 1;
   errors = 0;
   yyparse ();
}
yyerror (char *s) /* Called by yyparse on error */
{
   printf ("%s\n", s);
}
```

Così come è scritto, il parser non ha nessun output, ma, quando viene mandato in esecuzione, cerca comunque di costruire l'albero di derivazione rispetto alla grammatica del programma sorgente. L'analizzatore sintattico generato da *Bison* è uno **shift-reduce parser** e funziona nel seguente modo: mette nello stack uno o più simboli di input, prelevandoli da un buffer di input, fino a quando non riconosce una parte destra di una produzione appartenente alla grammatica. A questo punto riduce la stringa riconosciuta eliminandola dallo stack e mettendo al suo posto (in testa allo stack) il simbolo non terminale a sinistra della produzione riconosciuta. Questo processo viene iterato fino a che non viene incontrato un errore (sintattico) oppure si raggiunge la configurazione di accettazione: lo stack ed il buffer di input contengono il solo simbolo di fine stringa $.

Capitolo 5

Il Contesto

I file di input di *Flex* e *Bison* possono essere estesi in modo da gestire le informazioni *context sensitive*. Ad esempio, se desiderassimo che nel nostro linguaggio Simple, le variabili prima di essere referenziate debbano essere dichiarate, dovremmo permettere al parser di confrontare le variabili dichiarate con le variabili referenziate.

Un metodo per soddisfare questo requisito è costuire una lista di variabili mentre il parser esegue l'analisi sintattica sulla sezione delle dichiarazioni, ed in seguito confrontare le variabili referenziate con quelle nella lista. La lista è denominata *Symbol Table*; essa può essere implementata mediante un vettore, una lista, un albero, una tabella hash, ecc...

5.1 La Tabella dei Simboli

Per mantenere le informazioni richieste da un **grammatica ad attributi** abbiamo costruito una tabella dei simboli. La tabella dei simboli è una struttura dati che contiene un record per ogni identificatore, in cui i campi del record sono gli attributi dello stesso. Questi attributi possono essere per esempio il tipo di identificatore, il suo scope, il suo nome testuale, la locazione di memoria in cui è contenuto, ecc...

Abbiamo sviluppato la tabella dei simboli in un modulo a parte, incluso nel file di input di *Bison*.

Dal punto di vista prettamente implementativo, la tabella dei simboli consiste in una lista linkata di identificatori inizialmente vuota. Qui di seguito è riportato il codice C per la definizione della struttura del nodo e per la dichiarazione di due operazioni: `putsym` che inserisce un simbolo nella tabella dei simboli, e `getsym` che restituisce un puntatore alla tabella dei simboli corrispondente all'identificatore cercato.

```
struct symrec
{
char *name; /* name of symbol */
struct symrec *next; /* link field */
};
typedef struct symrec symrec;
symrec *sym table = (symrec *)0;
symrec *putsym ();
symrec *getsym ();
symrec *

putsym ( char *sym name )
{
symrec *ptr;
ptr = (symrec *) malloc (sizeof(symrec));
ptr->name = (char *) malloc (strlen(sym name)+1);
strcpy (ptr->name,sym name);
ptr->next = (struct symrec *)sym table;
sym table = ptr;
return ptr;
}

symrec *
getsym ( char *sym name )
{
symrec *ptr;
for (ptr = sym table; ptr != (symrec *) 0;
ptr = (symrec *)ptr->next)
if (strcmp (ptr->name,sym name) == 0)
return ptr;
return 0;
}
```

5.2 Modifiche al Parser

Per includere la tabella dei simboli con le sue routines e per eseguire l'installazione degli identificatori nella stessa tabella ed il *context checking*, risulta necessario modificare il file di input di *Yacc* come segue:

```
 %{
#include <stdlib.h> /* For malloc in symbol table */
#include <string.h> /* For strcmp in symbol table */
#include <stdio.h> /* For error messages */
```

```
#include "ST.h" /* The Symbol Table Module */
#define YYDEBUG 1 /* For debugging */
install ( char *sym name )
{ symrec *s;
s = getsym (sym name);
if (s == 0)
s = putsym (sym name);
else { errors++;
printf( "%s is already defined\n", sym name );
}
}
context check( char *sym name )
{ if ( getsym( sym name ) == 0 )
printf( "%s is an undeclared identifier\n", sym name );
}
%}
Parser declarations
%%
Grammar rules and actions
%%
C subroutines
```

Poichè lo scanner restituità degli identificatori, dobbiamo associare alla classe di token `IDENTIFIER` un record sematico per contenere assieme al tipo della varibile il suo nome:

```
C declarations
%union { /* SEMANTIC RECORD */
char *id; /* For returning identifiers */
}
%token INT SKIP IF THEN ELSE FI WHILE DO END
%token <id> IDENTIFIER /* Simple identifier */
%left '-' '+'
%left '*' '/'
%%
Grammar rules and actions
%%
C subroutines
```

La grammatica context-free, inoltre, richiede delle modifiche in modo da eseguire le operazioni di context checking quando richiesto:

```
C and parser declarations
%%
...
```

```
declarations : /* empty */
| INTEGER id seq IDENTIFIER '.' { install( $3 ); }
;
id seq : /* empty */
| id seq IDENTIFIER ',' { install( $2 ); }
;
command : SKIP
| READ IDENTIFIER { context check( $2 ); }
| IDENTIFIER ASSGNOP exp { context check( $1 ); }
...
exp : INT
| IDENTIFIER { context check( $1 ); }
...
%%
C subroutines
```

Così facendo abbiamo ottenuto implicitamente un parse-tree annotato; le informazioni annotate ci dicono se una variabile è stata dichiarata prima di essere referenziata.

5.3 Modifiche allo Scanner

Anche lo scanner, con l'inserimento della tabella dei simboli, necessita di alcune modifiche. Risulta necessario, infatti, fargli restituire alla lettera la stringa associata ad ogni identificatore (il valore semantico del token). Il valore semantico è passato dallo scanner al parser tramite la varibile `yylval`. Poichè `yylval` è un `union` dichiarato nel parser, dobbiamo memorizzare il valore semantico nel campo appropriato dell'unione: `yylval.id`.

Il valore semantico è quindi copiato dalla variabile globale `yytext` alla variabile `yylval.id`. Per poter utilizzare la funzione `strdup()` dobbiamo includere nel nostro file *Yacc* la libreria `string.h`.

```
%{
#include<string.h>
#include."y.tab.h"
%}
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
%%
":="     { return(ASSGNOP); }
{DIGIT}+ { yylval.intval = atoi( yytext );
           return(NUMBER);   }
do       { return(DO);       }
```

```
else     { return(ELSE);     }
end      { return(END);      }
fi       { return(FI);       }
if       { return(IF);       }
start    { return(START);    }
integer  { return(INTEGER);  }
variables{ return(VARIABLES);}
read     { return(READ);     }
skip     { return(SKIP);     }
then     { return(THEN);     }
while    { return(WHILE);    }
write    { return(WRITE);    }
for      { return(FOR);      }
{ID}     { yylval.id = (char *) strdup(yytext);
           return(IDENTIFIER);     }
\n       /* do nothing */
[ \t]+   /* eat up whitespace */
.        { return(yytext[0]);}
%%
Subroutine di supporto
```

Capitolo 6

La Macchina Virtuale

La macchina virtuale che utilizzata per questo progetto è una macchina virtuale a stack (*stack machine*) che d'ora in poi chiameremo **S-machine**.

Si riportano di seguito alcune delle caratteristiche più importanti di tale macchina:

- **Struttura della Macchina**: La *S-machine* è composta da due diverse aree di memorizzazione; la prima, **C**, dedicata al salvataggio delle istruzioni del programma sorgente (strutturata come un array e di sola lettura) e la seconda, **S**, (organizzata come uno stack) dedicata al salvataggio dei dati temporanei. All'interno di questa macchina ci sono inoltre tre registri:
  - **IR**: contiene l'istruzione corrente che sta per essere interpretata;
  - **PC**: contiene l'indirizzo della prossima istruzione;
  - **T**: contiene l'indirizzo dell'elemento in cima allo stack.
- **Set di istruzioni**: Ogni istruzione della *S-machine* è composta da 2 campi. Il primo campo è il *codice operativo* che rappresenta l'istruzione, il secondo campo rappresenta, invece, l'*operando* dell'istruzione stessa. Non tutte le istruzioni usufruiscono dell'operando, ad esempio istruzioni come `ADD` e `SUB` eseguono implicitamente le loro operazioni sui primi due valori in cima allo stack. Il tipo di dato che rappresenta un operando dipende dal codice operativo dell'istruzione.
- **Ciclo di esecuzione**: i registri e lo stack della S-machine sono inizializzati come segue:

```
PC = 0. {Program Counter}
T = 0;  {Stack Top}
```

```
S[0] := 0;
```

La macchina esegue iterativamente il fetching dell'istruzione all'indirizzo indicato dal registro **PC**. Le istruzioni sono eseguite finchè il registro **PC** non contiene l'indirizzo zero.

```
execution-loop : IR:= C(PC);
                 PC:= PC+1;
                 interpret(IR);
                 if { PC <= 0 -> halt
                   & PC > 0 -> execution-loop }
```

6.1 Il Modulo Stack Machine

Il set di istruzioni e la struttura di una generica istruzione sono così definite:

```
/* OPERATIONS: Internal Representation */
enum code_ops { HALT, STORE, JMP_FALSE, GOTO,
                DATA, LD_INT, LD_VAR,
                READ_INT, WRITE_INT,
                LT, EQ, GT, ADD, SUB, MULT, DIV };
/* OPERATIONS: External Representation */
char *op_name[] = { "halt", "store", "jmp_false", "goto",
                    "data", "ld_int", "ld_var",
                    "read", "write",
                    "lt", "eq", "gt", "add", "sub",
                    "mult", "div" };
struct instruction
{
    enum code_ops op;
    int arg;
};
```

Definiamo le due aree di memoria della *S-machine* come due vettori:

```
struct instruction code[999];
int stack[999];
```

La definizione dei registri risulta molto semplicemente:

```
int                 pc  = 0; /*Program Counter*/
struct instruction ir;      /*istruction register*/
int                 top = 0; /* top Stack pointer*/
```

Il codice associato alla fase fetch della *S-machine* è il seguente:

```
void fetch_execute_cycle()
 {   do { /* Fetch          */
       ir = code[pc];
       printf( "PC = %3d IR.op = %12s IR.arg = %8d Top = %3d, %8d\n",
           pc, op_name[ir.op], ir.arg, top, stack[top]);
       pc++;
         /* Execute       */
         switch (ir.op) {
           case HALT     : printf( "halt\n" );
                           pc = 0;
                           break;
           case READ_INT : printf( "Input: " );
                           scanf( "%d", &stack[ir.arg] );
                           break;
           case WRITE_INT: printf( "Output: %d\n", stack[top--] );
                           break;
           case STORE    : stack[ir.arg] = stack[top--];
                           break;
           case JMP_FALSE: if ( stack[top--] == 0 )
                             pc = ir.arg;
                           break;
           case GOTO     : pc = ir.arg;
                           break;
           case DATA     : top = top + ir.arg;
                           break;
           case LD_INT   : stack[++top] = ir.arg;
                           break;
           case LD_VAR   : stack[++top] = stack[ir.arg];
                           break;
           case LT       : if ( stack[top-1] < stack[top] )
                               stack[--top] = 1;
                           else stack[--top] = 0;
                           break;
           case EQ       : if ( stack[top-1] == stack[top] )
                               stack[--top] = 1;
                           else stack[--top] = 0;
                           break;
           case GT       : if ( stack[top-1] > stack[top] )
                               stack[--top] = 1;
                           else stack[--top] = 0;
                           break;
           case ADD      : stack[top-1] = stack[top-1] + stack[top];
                           top--;
                           break;
           case SUB      : stack[top-1] = stack[top-1] - stack[top];
                           top--;
                           break;
           case MULT     : stack[top-1] = stack[top-1] * stack[top];
                           top--;
                           break;
           case DIV      : stack[top-1] = stack[top-1] / stack[top];
                           top--;
                           break;
           default    : printf( "Internal Error: Memory Dump\n" );
```

```
                        break;
            }
        }
      while (pc != 0);
    }
```

Capitolo 7

Il Generatore di Codice

L'ultima fase di un compilatore è quella della generazione del codice target (usualmente della macchina ospite) che normalmente è un codice macchina rilocabile. Celle di memoria in uno spazio logico vengono assegnate ad ogni variabile, ed il programma sorgente viene tradotto nel linguaggio macchina ospite.

Nel prossimo paragrafo mostreremo come saranno tradotte in codice like-assembly alcune delle istruzioni appartenenti al linguaggio Simple.

7.1 Traduzione della dichiarazione di variabili

Per riservare spazio alle varibili dichiarate usiamo il comando `DATA`:

```
variables x,y,z.          DATA 2
```

7.2 Traduzione degli Statement

Gli statements più rappresentativi della grammatica sono tradotti come segue:

```
X := expr    traduzione di expr
             STORE X

if C then    traduzione di C
   S1        JMP_FALSE L1
             traduzione di S1
   else
   S2        GOTO L2
```

```
              L1: traduzione di S2
   fi         L2:

read X        IN_INT X

for (INIT; COND; INCR)    traduzione INIT
                          L1: traduzione COND
                                JMP_FALSE L4
                                GOTO L3
                          L2: traduzione INCR
                                GOTO L1
   S1                     L3: traduzione S1
                                GOTO L2
   end                    L4:

write EXP  traduzione di EXP
            WRITE
```

7.3 Il Modulo Generatore di Codice

La macchina virtuale per il linguaggio Simple è composta da tre segmenti: Un segmento per i dati, un segmento per il codice ed un terzo segmento per le contenere i risultati temporanei (il primo e l'ultimo segmento risiedono entrambi nello stack). Inizialmente il segmento dati ha un offset nullo; per riservare una locazione di memoria nel segmento dati usiamo la funzione `data_location`.

```
int data_offset = 0;
int data_location() { return data_offset++; }
```

Anche il segmento per il codice inizialmente ha offset nullo e, per riservare una locazione di memoria nell'area riservata al codice, utilizziamo la funzione `reserve_loc`, mentre per ottenere il valore attuale del code offset `gen_label`.

```
int code_offset = 0;
int reserve_loc()
{
   return code_offset++;
}
int gen_label()
{
   return code_offset;
}
```

Le funzioni `reserve_location` e `gen_label` sono usate per il *backpatching*.

Le funzioni `gen_code` e `back_patch` servono per memorizzare il codice rispettivamente in un area di memoria indicizzata dall'offset corrente o in un area di memoria precedentemente riservata.

```
void gen_code( enum code_ops operation, int arg )
{ code[code_offset].op    = operation;
  code[code_offset++].arg = arg;
}
void back_patch( int addr, enum code_ops operation, int arg
{
  code[addr].op = operation;
  code[addr].arg = arg;
}
```

7.4 Modifiche alla Tabella dei simboli

Dobbiamo modificare la tabella dei simboli per contenere l'informazione riguardo l'area di memoria in cui una variabile è salvata. Inoltre, aggiungiamo la variabile `isused` per verificare che una variabile dichiarata venga effettivamente utilizzata.

```
struct symrec
{
   char *name;                    /* name of symbol
   enum bool isused;
   int offset;                    /* data offset
   struct symrec *next;           /* link field
};
...
symrec * putsym (char *sym_name)
{
   symrec *ptr;
   ptr = (symrec *) malloc (sizeof(symrec));
   ptr->name = (char *) malloc (strlen(sym_name)+1);
   strcpy (ptr->name,sym_name);
   ptr->offset = data_location();
   ptr->next = (struct symrec *)sym_table;
   sym_table = ptr;
   return ptr;
}
...
symrec * function_is_used()
```

```
    {
       symrec *ptr;
       for (ptr = sym_table;
              ptr != (symrec *) 0;
              ptr = (symrec *)ptr->next)
       {
          if(ptr->isused==false)
          {
             ptr->isused=true;
             return ptr;
          }
       }
       return (symrec *) 0;
    }
    ...
```

7.5 Modifiche al Parser

Per supportare il modulo di generazione del codice abbiamo apportato delle modifiche ai file di input per *Bison* e *Flex*. Prima di tutto dobbiamo modificare la definizione del record semantico nel file di input *Bison* per:

- permettere ai token appartenenti alla classe `NUMBER` di memorizzare il loro valore;
- memorizzare due valori di offset utili per realizzare il backpatching nelle istruzioni `if`, `while` e `for`.

Le funzioni `newlblrec` e `newforlblrec` generano dinamicamente lo spazio necessario per memorizzare gli offset utili per il *backpatching*.

```
%{#include <stdio.h>       /* For I/O */
#include <stdlib.h> /* For malloc here and in symbol table */
#include <string.h>        /* For strcmp in symbol table */
#include "ST.h"            /* Symbol Table */
#include "SM.h"            /* Stack Machine */
#include "CG.h"            /* Code Generator */
#define YYDEBUG 1          /* For Debugging */
int errors;         /* Error Count-incremented in CG, ckd here */
struct lbs               /* For labels: if and while */
{
   int for_goto;
   int for_jmp_false;
```

```
};
struct forlbs               /* For labels: for */
{
   int for_goto_start;
   int for_goto_body;
   int for_goto_increment;
   int for_jmp_false;
};
struct lbs * newforlblrec() /* Allocate space for the labels */
{
    return (struct forlbs *) malloc(sizeof(struct forlbs));
}
struct lbs * newlblrec() /* Allocate space for the labels */
{
    return (struct lbs *) malloc(sizeof(struct lbs));
}
```

Aggiungiamo le azioni associate alla generazione del codice per la macchina virtuale nel file di input per *Bison* nella sezione dedicata alle regole.

```
%%
program : VARIABLES
            declarations
          START          { gen_code( DATA, data_location() - 1 );         }
            commands
          END          { gen_code( HALT, 0 ); YYACCEPT;                 }
;
declarations : /* empty */
   | INTEGER id_seq IDENTIFIER '.' { install( $3 ); }
| INTEGER id_seq IDENTIFIER  { errors++;
                                printf("missing '.'\n");}
;
id_seq : /* empty */
   | id_seq IDENTIFIER ',' { install( $2 );                             }
;
commands : /* empty */
   | commands command ';'
   | commands command {errors++;
       printf("missing ';'\n");}
;
command : SKIP
   | READ IDENTIFIER   { context_check( READ_INT, $2 );                }
   | WRITE exp         { gen_code( WRITE_INT, 0 );                     }
   | IDENTIFIER ASSGNOP exp { context_check( STORE, $1 );              }
   | IF exp            { $1 = (struct lbs *) newlblrec();
                         $1->for_jmp_false = reserve_loc();            }
     THEN commands     { $1->for_goto = reserve_loc();                 }
     ELSE              { back_patch( $1->for_jmp_false,
                                     JMP_FALSE,
                                     gen_label() );                    }
          commands
     FI                { back_patch( $1->for_goto, GOTO, gen_label() );  }
   | WHILE             { $1 = (struct lbs *) newlblrec();
                         $1->for_goto = gen_label();                   }
```

```
       exp           { $1->for_jmp_false = reserve_loc();            }
     DO
       commands
     END             { gen_code( GOTO, $1->for_goto );
                       back_patch( $1->for_jmp_false,
                                   JMP_FALSE,
                                   gen_label() );                    }
   |FOR '('IDENTIFIER ASSGNOP exp';'  { $1 = (struct forlbs *) newforlblrec();
                                        context_check( STORE, $3 );}
                                      { $1->for_goto_start = gen_label();   }
          exp';'
                                      { $1->for_jmp_false = reserve_loc();
                                        $1->for_goto_body = reserve_loc();
                                        $1->for_goto_increment = gen_label();}
          IDENTIFIER ASSGNOP exp')'  { context_check( STORE, $3 );
                                        gen_code( GOTO, $1->for_goto_start );}

                              { back_patch( $1->for_goto_body, GOTO, gen_label() );}
          commands

    END     { gen_code( GOTO, $1->for_goto_increment );
              back_patch( $1->for_jmp_false,JMP_FALSE,gen_label() ); }
;
exp : NUMBER          { gen_code( LD_INT, $1 );                       }
   | IDENTIFIER       { context_check( LD_VAR, $1 );                  }
   | exp '<' exp      { gen_code( LT,    0 );                         }
   | exp '=' exp      { gen_code( EQ,    0 );                         }
   | exp '>' exp      { gen_code( GT,    0 );                         }
   | exp '+' exp      { gen_code( ADD, 0 );                           }
   | exp '-' exp      { gen_code( SUB, 0 );                           }
   | exp '*' exp      { gen_code( MULT, 0 );                          }
   | exp '/' exp      { gen_code( DIV, 0 );                           }
   | exp '^' exp      { gen_code( PWR, 0 );                           }
   | '(' exp ')'
;
%%
```

7.6 Modifiche allo Scanner

Estendiamo, infine, il file *Flex* per restituire il valore degli interi nel record semantico.

```
%{
#include <string.h>     /* for strdup                          */
#include "simple.tab.h" /* for token definitions and yylval */
%}
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
%%
{DIGIT}+ { yylval.intval = atoi( yytext );
           return(INT);      }
...
{ID}     { yylval.id = (char *) strdup(yytext);
```

```
           return(IDENT);    }
[ \t\n]+ /* eat up whitespace */
.        { return(yytext[0]);}
%%
```

Bibliografia

[1] G. Piscitelli. *Dispense del corso di Compilatori e Interpreti*. Politecnico di Bari, 2008.

[2] G. Piscitelli. *Dispense del corso di Teoria dei Linguaggi*. Politecnico di Bari, 2008.

[3] J. Levine, T. Mason, D. Brown. *Lex & Yacc*. Ed. O'Really, 1992.

[4] T. Niemann. *A Compact Guide to Lex & Yacc*. Ed. ePaperPress.

[5] V. Martena, D. P. Scarpazza. *Un esempio di compilatore realizzato con Flex e Bison*. Politecnico di Milano.

[6] L. Tesei. *Dispense del corso di Linguaggi di Programmazione e Compilatori*. Università di Camerino, 2003.