Un esempio di compilatore realizzato con `flex` e `bison`

Progetto di Vincenzo Martena
Slide di Daniele Paolo Scarpazza

Tema

- Si costruisca, utilizzando bison e flex, un compilatore per il linguaggio di programmazione Simple.
- Il compilatore prenda in ingresso un file contenete un programma Simple e generi le corrispondenti istruzioni per la macchina virtuale SimpleVM.

Sintassi del linguaggio Simple

program	**-> preamble body**
preamble	**-> DECLARATIONS declarations**
body	**-> BEGIN_PROGRAM cmd_seq END_PROGRAM**
declarations	**-> ε | INTEGER id_seq IDENTIFIER .**
id_seq	**-> ε | id_seq IDENTIFIER ,**
cmd_seq	**-> ε | cmd_seq command ;**
command	**-> ε | SKIP | READ IDENTIFIER | WRITE exp | IDENTIFIER := exp | IF bool_exp THEN cmd_seq ELSE cmd_seq FI | WHILE bool_exp DO cmd_seq OD**
exp	**-> exp + term | exp - term | term**
term	**-> term * factor | term / factor | factor**
factor	**-> factor ^ primary | primary**
primary	**-> NUMBER | IDENT | (exp)**
bool_exp	**-> exp = exp | exp < exp | exp > exp**

Token lessicali

BEGIN_PROGRAM	begin
DECLARATIONS	declarations
DO	do
ELSE	else
END_PROGRAM	end
FI	fi
IDENTIFIER	[a - z][a - z0 - 9]+
IF	if
INTEGER	integer
NUMBER	[0-9]+
OD	od
READ	read
SKIP	skip
THEN	then
WHILE	while
WRITE	write

La macchina virtuale SimpleVM

- Semplice macchina a stack:
  - 2 aree di memoria:
    - area “C”, per il codice
    - area “S”, a pila, per i dati (variabili e risultati intermedi della computazione)
  - 3 registri:
    - stack pointer (SP): punta al top della pila;
    - instruction register (IR): prossima istruzione;
    - program counter (PC): indirizzo della prossima istruzione;

Istruzioni di SimpleVM

`HALT`	termina l'esecuzione
`IN_INT var`	legge un intero da tastiera e memorizza in var
`OUT_INT`	scrive a video il top della pila ed esegue pop
`STORE var`	scrive in var il top della pila ed esegue pop
`JMP_FALSE label`	se SP è falso, copia indirizzo di label in PC
`GOTO label`	copia indirizzo di label in PC
`DATA spazio`	riserva spazio sullo stack
`LD_INT intero`	esegue un push di intero sullo stack
`LD_VAR var`	esegue un push di var sullo stack
`LT / EQ / GT`	pop, pop, confronta i valori, push risultato
`ADD / SUB / MULT / DIV / PWR`	pop, pop, esegue l'operazione, push risultato

(Per maggiori informazioni: `fetch_execute_cycle()`, file `SM.c`)

Traduzione in assembly

- La traduzione segue lo schema riportato nella prossima slide;
- Nella traduzione finale, al nome simbolico delle etichette va sostituito il valore numerico corrispondente all'indirizzo nell'area del codice;

Schema di traduzione

`X := expr`	traduzione di expr `STORE X`
`if C then` `  S1`	traduzione di `C` `JMP_FALSE L1`
`else` `  S2`	traduzione di `S1` `GOTO L2`
`fi`	`L1:` traduzione di `S2` `L2:`
`read X`	`IN_INT X`
`write expr`	traduzione di `expr` `OUT_INT`

Esempi di traduzione

```
declarations          0: data     2
  integer a,b,c.      1: halt     0
begin
  skip;
end
```

Esempi di traduzione

```
declarations
  integer a,b,c.
begin
  a:=3;
  b:=4;
  read c;
  write a+b+c;
end
```

```
 0: data        2
 1: ld_int      3
 2: store       0
 3: ld_int      4
 4: store       1
 5: in_int      2
 6: ld_var      0
 7: ld_var      1
 8: add         0
 9: ld_var      2
10: add         0
11: out_int     0
12: halt        0
```

Esempi di traduzione

```
declarations
  integer a,b,c.
begin
  a:=3;
  b:=4;
  c:=a*b;
  while (c>0) do
     write c;
     c:=c-1;
  od
end
```

```
 0: data       2
 1: ld_int     3
 2: store      0
 3: ld_int     4
 4: store      1
 5: ld_var     0
 6: ld_var     1
 7: mult       0
 8: store      2
 9: ld_var     2
10: ld_int     0
11: gt         0
12: jmp_false 20
13: ld_var     2
14: out_int    0
15: ld_var     2
16: ld_int     1
17: sub        0
18: store      2
19: goto       9
20: halt       0
```

Esempi di traduzione

```
declarations
  integer a,b,c.
begin
  a:=3;
  b:=4;
  read c;
  if (c < 4) then
     write a;
  else
     write b;
  fi;
end
```

```
 0: data       2
 1: ld_int     3
 2: store      0
 3: ld_int     4
 4: store      1
 5: in_int     2
 6: ld_var     2
 7: ld_int     4
 8: lt         0
 9: jmp_false 13
10: ld_var     0
11: out_int    0
12: goto      15
13: ld_var     1
14: out_int    0
15: halt       0
```

Ingresso per flex

```
DIGIT               [0-9]
ID                  [a-z][a-z0-9]*
%%
":="                { return(ASSGNOP);                  }
{DIGIT}+            { yylval.intval = atoi( yytext );
                      return(NUMBER);                   }
do                  { return(DO);                       }
od                  { return(OD);                       }
else                { return(ELSE);                     }
end                 { return(END_PROGRAM);              }
fi                  { return(FI);                       }
if                  { return(IF);                       }
begin               { return(BEGIN_PROGRAM);            }
integer             { return(INTEGER);                  }
declarations        { return(DECLARATIONS);             }
read                { return(READ);                     }
skip                { return(SKIP);                     }
then                { return(THEN);                     }
while               { return(WHILE);                    }
write               { return(WRITE);                    }
{ID}                { yylval.id = (char *) strdup(yytext);
                      return(IDENTIFIER);               }
[ \t\n]+            /* consuma i separatori */
.                   { return(yytext[0]);                }
```

Ingresso per bison (token)

```
%union semrec{                  /* I record semantici           */
 int      intval;               /* Valori Interi                */
 char     *id;                  /* Identificatori               */
 lbs      *lbls;                /* Per backpatching             */
}

%start program
%token <intval>  NUMBER
                 /* Intero in Simple          */
%token <id>      IDENTIFIER
                 /* Identificatore in Simple */
%token <lbls>    IF WHILE
                 /* For backpatching labels  */
%token SKIP THEN ELSE FI DO OD END_PROGRAM
%token INTEGER READ WRITE DECLARATIONS BEGIN_PROGRAM
%token ASSGNOP

%left '-' '+'
%left '*' '/'
%right '^'
```

Ingresso per bison (sintassi)

```
program   : DECLARATIONS
              declarations
            BEGIN_PROGRAM    { gen_code( DATA,
                                reserve_data_location()-1); }
              commands
            END_PROGRAM      { gen_code(HALT,0); YYACCEPT;  }
          ;

declarations : /* empty */
          | INTEGER id_seq IDENTIFIER '.' { insert( $3 ); }
          ;

id_seq    : /* empty */
          | id_seq IDENTIFIER ','         { insert( $2 ); }
          ;

commands  : /* empty */
          | commands command ';'
          ;
```

Ingresso per bison (sintassi)

```
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 = (lbs *) newlblrec();
                           $1->for_jmp_false=reserve_code_loc(); }
      THEN commands      { $1->for_goto = reserve_code_loc();     }
      ELSE               { gen_back_code( $1->for_jmp_false,
                               JMP_FALSE, current_code_loc() );  }
           commands
      FI                 { gen_back_code( $1->for_goto, GOTO,
                               current_code_loc() );             }
    | WHILE              { $1 = (lbs *) newlblrec();
                           $1->for_goto = current_code_loc();    }
         exp             { $1->for_jmp_false=reserve_code_loc(); }
      DO
         commands
      OD                 { gen_code( GOTO, $1->for_goto );
                           gen_back_code( $1->for_jmp_false,
                               JMP_FALSE, current_code_loc() );  }
    ;
```

Ingresso per bison (sintassi)

```
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 = (lbs *) newlblrec();
                           $1->for_jmp_false=reserve_code_loc(); }
      THEN commands      { $1->for_goto = reserve_code_loc();     }
      ELSE               { gen_back_code( $1->for_jmp_false,
                               JMP_FALSE, current_code_loc() );  }
           commands
      FI                 { gen_back_code( $1->for_goto, GOTO,
                               current_code_loc() );             }
    | WHILE              { $1 = (lbs *) newlblrec();
                           $1->for_goto = current_code_loc();    }
         exp             { $1->for_jmp_false=reserve_code_loc(); }
      DO
         commands
      OD                 { gen_code( GOTO, $1->for_goto );
                           gen_back_code( $1->for_jmp_false,
                               JMP_FALSE, current_code_loc() );  }
    ;
```

Backpatching

```
 typedef struct{          /* Etichette per i dati, if e while */
  int for_goto;
  int for_jmp_false;
 } lbs;

int current_code_loc(){
   return code_offset;
}

int reserve_code_loc(){
   return code_offset++;
}
```

Symbol table

```
symrec *identifier; /* Elemento della tabella dei simboli */

symrec *sym_table = NULL; /* puntatore a tabella dei simboli */

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 = reserve_data_location();
  ptr->next    = (symrec *)sym_table;
  sym_table    = ptr;
  return ptr;
}

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

Code generator

```
char *op_name[17]={"halt", "store", "jmp_false", "goto", "data", ...};
int data_offset = 0, code_offset = 0;
int reserve_data_location()  {  return data_offset++; }
int current_code_loc()       {  return code_offset;   }
int reserve_code_loc()       {  return code_offset++; }

void gen_code( code_ops operation, int arg ){
  setOpInstruction(code_offset,operation);
  setArgInstruction(code_offset++,arg);
}

void gen_back_code( int addr,  code_ops operation, int arg  ){
  setOpInstruction(addr,operation);
  setArgInstruction(addr,arg);
}

void print_code(){
   int i;
   for (i=0; i < code_offset; i++)
      printf("%3ld: %-10s%4ld\n", i,
        op_name[(int) getOpInstruction(i)], getArgInstruction(i));
}
```