Linguaggi Formali e Compilatori
Prof. Crespi Reghizzi, Prof.ssa Sbattella
Prova scritta – Parte di Laboratorio[1]
09/09/2005

COGNOME e NOME:..... Matricola:.....
Iscritto a: ○ Laurea Specialistica ○ Vecchio Ordinamento ○ Altro:.....

Data la macchina SimpleVM e il compilatore per il linguaggio Simple allegati al tema d'esame, si consideri il seguente codice SimpleAssembler, parzialmente commentato, nel quale sono presenti chiamate a funzione.

```
    jmp    main

                    /* Inizio codice funzione */
X:  ldvar    -2   /* -2 è il "nome" del secondo parametro in
ingresso */
    ldint     0
    gt
    jmp_false Y
    ld_var   -3
    ld_int    0   /* Riserva spazio valore di ritorno */
    ld_var   -3   /* -3 è il "nome" del primo parametro di ingresso */
    ld_var   -2
    ld_int    1
    sub
    call      X   /* Chiamata Ricorsiva */
    data     -2   /* Distrugge i primi due elementi dello stack */
    add
    store    -4   /* -4 = valore di ritorno */
    ret
Y:  ld_var   -3
    store    -4
    ret           /* Fine Codice Funzione */

main: data    1   /* Codice Main */
    readint   0
    readint   1
```

[1]Tempo 30'. Libri e appunti personali possono essere consultati. È consentito scrivere a matita. Scrivere il proprio nome sugli eventuali fogli aggiuntivi. *Rispondere punto per punto alle domande!*

```
ld_int    0   /* Riserva spazio per valore di ritorno */
ld_var    0   /* Primo parametro della funzione *
ld_var    1   /* Secondo parametro della funzione */
call      X   /* Chiamata a funzione */
data     -2   /* Distrugge i primi due elementi dello stack */
writeint
halt
```

La semantica delle istruzioni `call` e `ret` è riportata qui di seguito.

call $< to >$ Salva sul top dello stack l'indirizzo di ritorno e il registro AR relativo all'*Activation Record* del *chiamante* (prima PC, poi AR). Quindi setta il PC a $< to >$ e setta AR al nuovo top dello stack.

ret AR=stack[top]. PC=stack[top-1]. top=top-2.

Noti questi elementi, si svolgano i seguenti punti:

1. Introdurre un costrutto sintattico del linguaggi sorgente per la dichiarazione e chiamata a funzione, che sia compatibile con il "byte-code" proposto.

   Dichiarazione:

   ```
   def function_name ( integer par1, par2, ... parN ) : integer
       declarations
       integer local_var1, ... local_varN.
       begin
           ...
       end
   ```

   Invocazione:

   ```
   a := function_name ( par1, par2, ... parN )
   ```

   Ritorno:

   ```
   return <espressione>
   ```

2. Scrivere il codice sorgente corrispondente al programma bytecode fornito.

   ```
   def X ( integer a, b) : integer
       declarations
       begin
           if b>0 then
               return a + X(a,b-1);
           else
               return a;
           fi
       end
   ```

```
def main () : integer
    declarations
        integer a,b.
    begin
        read a;
        read b;
        write X(a,b);
    end
```

3. Modificare le specifiche Flex e Bison del compilatore Simple con le regole grammaticali necessarie per gestire i nuovi costrutti.

   Simple.lex :

```
return      { return(RETURN); } def       { return(DEF);    }
```

   Simple.y :

```
%token RETURN DEF

...

program: /* empty */
        | program function_decl

function_decl :
        DEF IDENTIFIER '(' formal_pars ')' ':' INTEGER DECLARATIONS declarations BEGI

formal_pars : /* empty */
    | INTEGER par_seq IDENTIFIER

par_seq : /* empty */
    | par_seq IDENTIFIER ','

commands :
    ...
    | RETURN exp

exp :
    ...
    | IDENTIFIER '(' actual_pars ')'

actual_pars : /* empty */
    | actual_par_seq exp

actual_par_seq : /* empty */
    | actual_par_seq exp ','
```

4. Proporre uno schema di traduzione e implementarlo con azioni semantiche in Bison e Flex.

   É necessario disporre nella tabella dei simboli delle entry speciali per le funzioni. Vengono aggiunti i seguenti campi:

```
enum SYMBOL_TYPE = { VAR, FUN, PAR }; typedef struct _symrec {
  char * name;    /* nome del simbolo */
    int    offset;  /* offset della variabile o indirizzo della funzione */
```

```
    enum SYMBOL_TYPE    type;    /* 0: variable; 1: function; 2: formal parameter */
    struct _symrec * locals; /* lista di variabili locali;
                        inutilizzato se il simbolo e' un nome di variabile */
    struct _symrec * parameters; /* lista di parametri formali;
                        inutilizzato se il simbolo e' un nome di variabile */
    int    pars;    /* numero di parametri */
    struct _symrec * next; /* prossimo simbolo */
} symrec;
```

É necessario modificare la insert aggiungendo il parametro relativo al tipo del simbolo. Inoltre, la insert e la putsym non allocano (ovviamente) spazio in memoria per le funzioni e i parametri. Non e' richiesta l'implementazione di queste funzioni.

Serve anche sapere in ogni momento in quale funzione ci si trova. Per questo, e' possibile usare una variabile globale, `char *curr_fun;`. Questa variabile globale viene usata da insert e putsym per inserire i parametri e le variabili locali nella lista corretta.

Simple.y :

```
function_decl :
        DEF IDENTIFIER { insert($2,FUN); }
        '(' formal_pars ')' ':' INTEGER DECLARATIONS declarations BEGIN commands END

formal_pars : /* empty */
    | INTEGER par_seq IDENTIFIER { insert($3, PAR); }

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

commands :
    ...
    | RETURN exp { gen_code(STORE,-(get_sym(curr_fun)->pars+2));
                   gen_code(RET,0); }

exp :
    ...
    | IDENTIFIER '('  { context_check($1);
                        gen_code(LD_INT,0); /* spazio per valore di ritorno */ }
      actual_pars ')' { gen_code(CALL,get_sym($1)->offset);
                          gen_code(DATA,-(get_sym($1)->pars)); }

actual_pars : /* empty */
    | actual_par_seq exp

actual_par_seq : /* empty */
```

```
| actual_par_seq exp ','
```

5. Modificare l'interprete per gestire correttamente `call`, `ret`.
   SM.c: fetch_execute_cycle()

```
case CALL : stack[++top]=pc; stack[++top]=ar; pc=ir.arg; ar=top; break;
case RET  : ar=stack[top]; pc=stack[top-1]; top=top-2; break;
```

6. **bonus**: aggiungere almeno un controllo semantico durante la compilazione (es: numero di parametri formali = numero di parametri attuali).

É sufficiente modificare le regole formal_pars, par_seq, actual_pars, e actual_pars_seq in modo che restituiscano un parametro sintetizzato:

```
%type <intval> formal_pars par_seq actual_pars actual_pars_seq
```

Nelle corrispondenti azioni semantiche, si avrà:

```
{ $$=$1 + 1; } /* nelle regole non vuote */ { $$=0;      } /* nelle
regole vuote     */
```

A questo punto, nella regola di invocazione, é sufficiente eseguire il controllo:

```
{ if (get_sym($1)->pars!=$4)
   fprintf(stderr, "Errore: il numero di parametri non corrisponde!\n");
    exit(0);
}
```