Linguaggi Formali e Compilatori
Proff. Breveglieri, Crespi Reghizzi, Morzenti
Prova scritta[1]: Domanda relativa alle esercitazioni
05/02/2008

COGNOME: .....
NOME: ..... Matricola: .....
Iscritto a: ○ Laurea Specialistica ○ V. O. ○ Laurea Triennale ○ Altro:.....
Sezione: ○ Prof. Breveglieri ○ Prof. Crespi ○ Prof.Morzenti

Per la risoluzione della domanda relativa alle esercitazioni si deve utilizzare l'implementazione del compilatore `Acse` che viene fornita insieme al compito.

Si richiede di modificare la specifica dell'analizzatore lessicale da fornire a `flex`, quella dell'analizzatore sintattico da fornire a `bison` ed i file sorgenti per cui si ritengono necessarie delle modifiche in modo da estendere il compilatore `Acse` con la possibilità di gestire un costrutto iterativo di tipo *per ogni*, del quale viene qui fornito un esempio.

```
int a[100];
int x;
int y;
...
for(x:a){
   y = y + x;
}
write( y );
...
```

Questo tipo di costrutto `for` non eplicita l'indice del vettore. Alla prima iterazione la variabile `x` vale `a[0]`, alla seconda `a[1]` e così via per tutti gli elementi di `a`. Nell'esempio sopra, il frammento calcola e stampa la somma di tutti gli elementi di `a`.

Le modifiche devono mettere il compilatore `Acse` in condizione di analizzare la correttezza sintattica dei costrutti sopra descritti e di generare una traduzione corretta nel linguaggio assembler della macchina `Mace`.

Per risolvere il problema si consiglia di utilizzare la funzione `getNewRegister` di `axe_engine.[h,c]`, la quale restituisce il numero di un registro (del banco di registri infinito) non ancora assegnato.

```
/* get a register still not used. This function returns
```

[1]Tempo 45'. Libri e appunti personali possono essere consultati.
È consentito scrivere a matita. Scrivere il proprio nome sugli eventuali fogli aggiuntivi.

```
 * the ID of the register found */
int getNewRegister(t_program_infos *program);
```

1. Definire i token (e le relative dichiarazioni in Acse.lex e Acse.y) necessari per ottenere la funzionalità richiesta.

   Il token `for`, l'unico nuovo rispetto al linguaggio, è già definito sia in Acse.lex che Acse.y.

2. Definire le regole sintattiche necessarie per ottenere la funzionalità richiesta.

   Si definisce una regola per il nuovo costrutto

   foreach_statement : FOR LPAR IDENTIFIER COLON IDENTIFIER RPAR *code_block* ;

   che va inserito come alternativa per *control_statement*:

   control_statement : ...
   | *foreach_statement*
   ;

3. Definire le modifiche alle strutture dati (se necessarie) per supportate la funzionalità richiesta.

   In **axe_struct.h**:

```
typedef struct {
       t_axe_label *label;     /* Label at the beginning
                                  of the loop */
       int index;              /* Register for the index */
       int length;             /* Array length */
} t_foreach_statement;
```

   In **Acse.y**:

```
%token <foreach_stmt> FOR
%union {
    ...
   t_foreach_statement foreach_stmt;
}
```

4. Definire le azioni semantiche necessarie per ottenere la funzionalità richiesta.

foreach_statement : FOR LPAR IDENTIFIER COLON IDENTIFIER RPAR

```
{
  int loopvar, tempreg;  /* Registers */
  t_axe_variable *looparray;
  t_axe_expression expr;

  loopvar = get_symbol_location( program, $3, 0 );
  looparray = getVariable( program, $5 );
  if (! looparray->isArray) {
     fprintf( stderr, "Variable %s is not an array\n",
              looparray->ID );
     notifyError( -1 );
  }
  $1.index = getNewRegister( program );
  $1.length = looparray->arraySize;
  /* index = 0; */
  gen_add_instruction( program, $1.index, REG_0, REG_0,
                       CG_DIRECT_ALL );
  /* var = array[index]; */
  $1.label = assignNewLabel( program );
  expr = create_expression ( $1.index, REGISTER);
  tempreg = loadArrayElement( program, $5, expr );
  gen_add_instruction( program, loopvar, REG_0, tempreg,
                       CG_DIRECT_ALL );
  free( $3 );
  free( $5 );
}
```

code_block

```
{
  int reg;

  /* index = index + 1; */
  gen_addi_instruction( program, $1.index, $1.index, 1 );
  /* reg = index - array_length; */
  reg = getNewRegister( program );
  gen_subi_instruction( program, reg, $1.index, $1.length );
  /* BLT beginning; */
  gen_blt_instruction( program, $1.label, 0 );
}
```

;

Il codice generato crea una variabile temporanea in un registro (campo `index` della struttura) che viene fatto variare su tutti i valori legatli per l'indice dell'array. Il codice

```
for (var : array) {
  /* body */
}
```

viene tradotto in (usando un misto di pseudo-codice e assembly):

```
   index := 0
label:
   var := array[index]
   /* body */
   index := index + 1
   index - array_length
   BLT label
```

dove `index` è la variabile temporanea descritta prima, e `array_length` è la lunghezza dell'array. Si noti che il risultato della sottrazione viene in realtà assegnato a un registro temporaneo.

5. **Bonus** (da svolgersi solo a termine dei punti precedenti). Si vieti qualsiasi assegnamento alla variabile di induzione (`x`, nell'esempio) all'interno del corpo del ciclo.

   Viene brevemente descritto lo schema di una possibile soluzione.

   Si crea una lista globale, inizialmente vuota, i cui nodi contengono ognuno un identificatore (una stringa). Prima del corpo del ciclo *for*, cioè nella prima azione semantica al punto precedente, si aggiunge un nodo alla lista contente il nome della variabile di ciclo (`$3`); mentre alla fine di ogni ciclo, nella seconda azione semantica, si elimina tale nodo. Questa lista permette di avere disponibili ovunque le informazioni sulle variabili di ciclo a un dato passo della traduzione.

   Va quindi aggiunto un controllo nelle azioni semantiche della regole già esistenti

   assign_statement : IDENTIFIER ASSIGN *exp* ;
   read_statement : READ LPAR IDENTIFIER RPAR ;

   che verifichi che `IDENTIFIER` non sia presente nella lista, e generi un errore in caso contrario. Un analogo controllo va fatto nella prima azione semantica del ciclo *for*, per evitare che due cicli *for* innestati utilizzino la stessa variabile di ciclo.