Linguaggi Formali e Compilatori
Proff. Breveglieri, Crespi Reghizzi, Sbattella
Prova scritta[1]: Domanda relativa alle esercitazioni
07/07/2006

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

Per la risoluzione della domanda relativa alle esercitazioni si deve utilizzare l'implementazione del compilatore `Simple` 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 linguaggio `Simple` con la possibilità di usare assegnamenti all'interno delle espressioni come nel linguaggio `C`:

```
declarations
 integer a, b, c.
 ...
begin
 ...
 if (b<a:=c+4)
    a := b := c + 3;
 ...
end
```

dove un'espressione di assegnamento ha come valore il valore assegnato. L'operatore di assegnamento ha precedenza minore rispetto agli operatori aritmetici ma maggiore di quelli di confronto.

Le modifiche devono mettere il compilatore `Simple` in condizione di analizzare la correttezza sintattica dei costrutti sopra descritti e di generare una traduzione corretta nel linguaggio assembler della macchina `SimpleVM`. Sebbene non sia strettamente necessario, è consentito estendere la VM con il bytecode DUP, che duplica il valore in cima allo stack. Se il bytecode DUP viene usato, deve essere fornita l'implementazione nell'interprete.

[1] Tempo 30'. Libri e appunti personali possono essere consultati.
È consentito scrivere a matita. Scrivere il proprio nome sugli eventuali fogli aggiuntivi.

1. Definire i token e le regole sintattiche necessari per ottenere la funzionalità richiesta.

   Bisogna rimuovere la definizione del token `ASSGNOP` preesistente, che non definisce precedenza e associatività,

   %token ASSGNOP

   e inserire le definizioni della priorità e dell'associatività degli operatori di confronto e assegnamento prima di quelle degli operatori aritmetici già esistenti:

   ```
   /*============================
     PRECEDENZE TRA GLI OPERATORI
     ============================*/
   %nonassoc '<' '>' '='
   %right ASSGNOP
   %left '-' '+'
   %left '*' '/'
   %right '^'
   ```

   Si è scelto di non avere operatori di confronto associativi, perciò espressioni come

   3 < a < 10

   sono considerate errori di sintassi. Questa scelta dipende dal fatto che tali espressioni se fossero valide non verrebbero interpretate dalla macchina `Simple` secondo la naturale interpretazione matematica. Per esempio, l'espressione sopra verrebbe considerata sempre vera se l'operatore `<` fosse associativo a sinistra, sempre falsa se fosse associativo a destra, indipendentemente dal valore di `a`.

   Bisogna quindi permettere l'uso di un assegnamento in un'espressione aggiungendo un'alternativa all'espansione del nonterminale `exp`:

   ```
   exp:
     ...
     | IDENTIFIER ASSGNOP exp
   ```

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

   Bisogna inserire l'azione semantica per la regola sintattica aggiunta nel punto precedente:

   ```
   exp:
     ...
     | IDENTIFIER ASSGNOP exp
   { context_check( STORE, $1 );
     context_check( LD_VAR, $1 ); }
   ```

   Questa soluzione scrive il valore della sottoespressione `exp` di destra nella variabile e poi lo ricarica sulla pila per renderlo disponibile come valore dell'intera espressione di assegnamento.

   In alternativa, si può usare l'istruzione DUP:

   ```
   exp:
     ...
     | IDENTIFIER ASSGNOP exp
   { gen_code( DUP, 0 );
     context_check(STORE, $1 ); }
   ```

3. Implementazione, se necessaria, dell'opcode DUP.

   Bisogna aggiungere un'elemento all'enumeratore `code_ops` in *SM.h*:

   ```
   ...
   PWR,
   DUP } code_ops;
   ```

   e un elemento all'array `op_name` in *CG.c*:

   ```
   char *op_name[18] = {"halt",
     ...
     "pwr",
     "dup" };
   ```

   Quindi si inserisce il codice che interpreta l'istruzione nella funzione `fetch_execute_cycle` in *SM.c*:

   ```
   case DUP:
     stack[top+1] = stack[top]; top++;
     break;
   ```