Linguaggi Formali e Compilatori
Proff. Breveglieri, Crespi Reghizzi, Morzenti
Prova scritta[1]: Domanda relativa alle esercitazioni
06/02/2009

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 semplici macro, sulla falsariga del comando `#define` del preprocessore C, come nell'esempio:

```
define RISPOSTA 42;
define DOMANDA 9;
int x;
read( x );
x = RISPOSTA * x;
write( x );
```

La prima riga dell'esempio definisce la macro `RISPOSTA` come la costante intera `42`. Se tale programma viene compilato e fatto girare, e al prompt in ingresso si scrive "3", verrà stampato in uscita il valore "126" ($126 = 42 \times 3$).

La soluzione deve rispettare le seguenti specifiche:

- Le macro non hanno parametri, e a ogni macro viene assegnato un *valore numerico intero* al momento della sua definizione.
- Si possono definire un numero arbitrario di macro.
- Si può usare una macro *ovunque* è lecito usare una costante intera nella grammatica originale di `Acse`.
- Non è possibile modificare una macro con una istruzione di assegnamento.

Una **soluzione ottimale** non deve generare codice assembly aggiuntivo per un programma `Lance` che usa macro rispetto a uno che specifica le costanti numeriche direttamente nel codice.

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

Si implementi un adeguato comportamento del compilatore nel caso una macro fosse definita una seconda volta: si generi un errore o si sovrascriva la vecchia definizione. Si espliciti ogni eventuale ulteriore assunzione che sia ritenuta necessaria a completare la specifica data.

Si ipotizzi pure di avere a disposizione le seguenti primitive che gestiscono una lista di elementi di tipo `T_DATA` opportuno. In tal caso, si espliciti il tipo `T_DATA` impiegato nella propria soluzione. In alternativa, si possono usare le funzioni messe a disposizione in **collections.h**.

```
void initList( t_list *list );
void addFirst( t_list *list, T_DATA *element );
void addLast( t_list *list, T_DATA *element );
T_DATA *getFirstElement( t_list *list );
T_DATA *getLastElement( t_list *list );
T_DATA *getElementAt( t_list *list, unsigned int position );
```

1. Definire i token (e le relative dichiarazioni in **Acse.lex** e **Acse.y**) necessari per ottenere la funzionalità richiesta. (3 punti)

2. Definire le regole sintattiche (o le modifiche a quelle esistenti) necessarie per ottenere la funzionalità richiesta. (8 punti)

3. Specificare il comportamento del compilatore in caso di definizione doppia e definire le azioni semantiche (o le modifiche a quelle esistenti) necessarie per ottenere la funzionalità richiesta. (13 punti; 19 punti per una soluzione "ottimale")

4. (Bonus) Modificare la soluzione data in maniera tale da rendere possibile definire una macro in funzione di un'altra macro. Esempio:

```
define NUMERO_MAGICO 42;
define RISPOSTA NUMERO_MAGICO;
int x;
x = RISPOSTA;
write( x );
```

   Questo programma stampa "42" quando fatto girare. (5 punti)