Linguaggi Formali e Compilatori
Proff. Breveglieri, Crespi Reghizzi, Morzenti
Prova scritta [1]: Domanda relativa alle esercitazioni
02/09/2010

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 le operazioni di somma e sottrazione tra vettori.

```
1 int a[10];
2 int b[10];
3 int c[10];
4
5 vec_add(c, a, b);
```

(a) Somma

```
1 int a[10];
2 int b[10];
3 int c[10];
4
5 vec_sub(c, a, b);
```

(b) Sottrazione

Figura 1: Operazioni vettoriali. Figura 1(a) computa la somma vettoriale $\overline{c} = \overline{a} + \overline{b}$ utilizzando `vec_add`. In maniera analoga, Figura 1(b) calcola la differenza vettoriale $\overline{c} = \overline{a} - \overline{b}$ sfruttando `vec_sub`.

Le operazioni vettoriali sono identificate dalle keyword `vec_add` e `vec_sub`:

Somma vettoriale `vec_add(c, a, b)` calcola la somma elemento per elemento tra `a` e `b` salvandone il risultato in `c`. Il generico elemento `c[i]` sarà quindi pari ad `a[i] + b[i]`

Differenza vettoriale `vec_sub(c, a, b)` calcola la differenza elemento per elemento tra `a` e `b` salvandone il risultato in `c`. Il generico elemento `c[i]` sarà quindi pari ad `a[i] - b[i]`

Le nuove operazioni agiscono su vettori aventi la stessa lunghezza. Negli esempi di Figura 1 i tre operandi delle operazioni vettoriali contengono tutti 10 elementi. Se i vettori non hanno la stessa lunghezza la traduzione non può essere eseguita ed è necessario generare un errore a compile-time.

Si espliciti ogni eventuale ulteriore assunzione che sia ritenuta necessaria a completare la specifica data.

[1] Tempo 45'. 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 relative dichiarazioni in `Acse.lex` e `Acse.y`) necessari per ottenere la funzionalità richiesta. (3 punti)

   La soluzione è riportata nella patch allegata.

2. Definire le regole sintattiche (o le modifiche a quelle esistenti) necessarie per ottenere la funzionalità richiesta. (4 punti)

   La soluzione è riportata nella patch allegata.

3. Definire le azioni semantiche (o le modifiche a quelle esistenti) necessarie per ottenere la funzionalità richiesta. (18 punti)

   La soluzione è riportata nella patch allegata.

4. Dato il codice di Figura 2:

```
1 if(a == 4)
2   a = 7;
```

Figura 2: Costrutto `if`.

Scrivere l'albero sintattico relativo partendo dalla grammatica Bison definita in `Acse.y` *iniziando dal non-terminale* `statements`. (5 punti)

La soluzione è riportata in Figura 3.

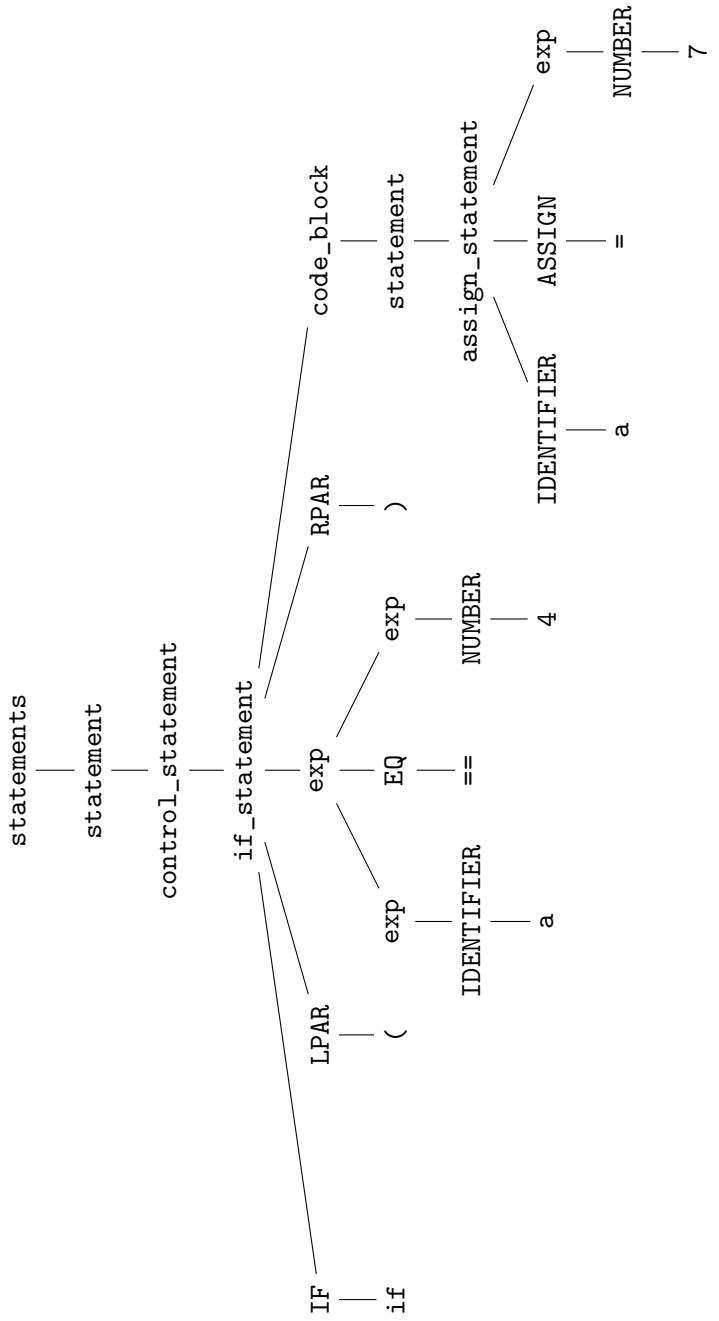

Figura 3: Albero sintattico di uno statement `if`.

5. (Bonus) Si supponga che il linguaggio targettato dal compilatore `Acse` sia stato esteso in modo da supportare istruzioni che operano con vettori di lunghezza 4 elementi. Di conseguenza sono disponibili dei registri vettoriali `vect_t` e le relative funzioni per manipolarli, riportate in Figura 4.

```
1  vect_t
2  getNewVectRegister(
3    t_program_infos* program);
4
5  vect_t
6  loadArrayChunk(
7    t_program_infos* program,
8    char* ID,
9    t_axe_expression index);
10
11 void
12 storeArrayChunk(
13   t_program_infos* program,
14   char* ID,
15   t_axe_expression index,
16   vect_t r_source);
17
18 t_axe_instruction*
19 gen_addv_instruction(
20   t_program_infos* program,
21   vect_t r_dest,
22   vect_t r_source1,
23   vect_t r_source2);
```

Figura 4: Assembly vettoriale. Il comportamento delle tre funzioni è analogo a quelle già disponibili in `Acse`, ma invece che operare con registri scalari operano con registri vettoriali.

È possibile ottenere nuovi registri vettoriali tramite la funzione `getNewVectRegister`. La funzione `loadArrayChunk` permette di caricare 4 elementi di un array, partendo dall'indice `index`, in un registro vettoriale; essa ritorna il registro destinazione. La sua duale, `storeArrayChunk` permette di salvare il contenuto di un registro vettoriale in un array, sovrascrivendo 4 elementi partendo dall'indice `index`. Infine la funzione `gen_addv_instruction` genera il codice necessario per sommare i registri vettoriali `r_source1` e `r_source2` e salvare il risultato in `r_dest`.

Avendo a disposizione queste primitive, come le si può sfruttare per implementare le operazioni vettoriali?

La soluzione proposta nei punti precedenti spezza la somma vettoriale in tante piccole somme scalari tra i corrispondenti elementi degli array operandi.

Le nuove istruzioni possono essere sfruttate in modo da spezzare gli operandi in blocchi di 4 elementi ciascuno, che poi verranno processati dalle istruzioni vettoriali.

Ovviamente bisogna tenere conto del fatto che non tutti gli array posso essere spezzati in blocchi di 4 elementi. Per questi casi patologici bisogna ancora ricorrere alla soluzione precedente.

Ad esempio, un array contenete 11 elementi puo essere spezzato in 3 blocchi. I primi 2, contenenti gli elementi con indice compreso tra 0 e 7, possono essere trattati con le nuove istruzioni vettoriali. I restanti 3 elementi, quelli con indice compreso tra 8 e 10, devono essere trattati con istruzioni scalari.