11 Aprile 2012

Linguaggi di Programmazione

Cognome e nome	
Corso di laurea	

1. Specificare la definizione regolare relativa ad una lista (anche vuota) di tuple, in cui ogni elemento di una tupla può essere una stringa, un numero intero, un numero reale o una costante booleana, come nel seguente esempio,

```
[("alfa",+10,12.34,true)("\"beta\"",false)("Gamma 25",-0.12,false)]
```

sulla base dei seguenti vincoli lessicali:

- Ogni tupla (racchiusa tra parentesi) contiene almeno due elementi, separati tra loro da una virgola.
- Le tuple non hanno separatori tra loro.
- Una stringa (anche vuota) è delimitata da due doppi apici e può contenere caratteri alfanumerici, spazi e doppi apici (per evitare ambiguità, questi ultimi devono essere preceduti dal backslash).
- Un numero intero diverso da zero non può iniziare con la cifra 0.
- Un numero reale ha una parte intera ed una parte decimale, separate da un punto.
- La parte intera diversa da zero di un numero reale non può iniziare con la cifra 0.
- I numeri possono essere preceduti dal segno.

2. Specificare la grammatica BNF di un linguaggio per definire ed assegnare tabelle, come nel seguente esempio:

```
table (int a, string b, table(int d, real e) c) tab1;

table (int lung, table (table (int delta) t) r) tab2;

tab1 = [(3, "sole", [(10, 23.12)(12, 1.44)])
        (4, "mare", [(16, 3.56)])
        (5, "stella", [])];

tab2 = [(1, [([(12)(33)(37)])])
        (3, [([(256)(1)])])
        (24, [([])])
        (46, [])];

table (int x, int y) Zeta;

Zeta = [];
```

Ogni frase contiene almeno una istruzione. Le istruzioni di definizione ed assegnamento delle tabelle possono essere specificate in qualsiasi ordine. Gli attributi atomici sono **`int`**, **`real`** e **`string`**. Non esiste limite di innestamento delle tabelle. Una tabella (o un attributo di tipo tabella) può essere assegnato con la tabella vuota **[ ]**.

3. Specificare la semantica operazionale dell'operatore di appartenenza di una tupla ad una tabella (senza duplicati, per assunzione) mediante una notazione imperativa:

```
t ∈ T
```

Si assumono le seguenti funzioni ausiliarie (di cui non è richiesta la specifica):

- `schema(X)`: lista di coppie (`nome`, `tipo`) che definisce lo schema della tupla o tabella `X`;
- `tupinst(t)`: record che definisce l'istanza della tupla `t`;
- `tabinst(T)`: lista di record che definisce l'istanza della tabella `T`;
- `length(L)`: lunghezza della lista `L`.

Si richiede che gli operandi siano compatibili per struttura. È possibile confrontare due tuple con l'operatore di uguaglianza. Nel caso di errore semantico, il valore della espressione è `errore`.

4. Specificare la semantica denotazionale di un frammento di codice definito dalla seguente BNF:

codice → **id = [** *string-list* **] ; id [num].**
string-list → **string ,** *string-list* | **string**

esempio di frase

```
v = ["alfa","beta","gamma","delta","epsilon];
v[3].
```

Il frammento è costituito da due operazioni: assegnamento di un vettore di stringhe e indicizzazione di tale vettore. Il nome del vettore nella espressione di indicizzazione deve coincidere con il nome del vettore assegnato precedentemente. Inoltre, il valore dell'indice non deve uscire dai limiti di indicizzazione [0 .. *n*-1], in cui *n* è la dimensione del vettore. Si richiede la specifica delle seguenti funzioni semantiche:

- M_c(*codice*): restituisce il risultato della operazione di indicizzazione in *codice*, (eventualmente `errore`);
- M_{sl}(*string-list*): restituisce la lista di stringhe relativa a *string-list*.

Nel linguaggio di specifica denotazionale sono disponibili le seguenti funzioni ausiliarie (di cui non è richiesta la specifica):

- `name(`**id**`)`: restituisce il valore lessicale (nome) di **id**;
- `ivalue(`**num**`)`: restituisce il valore lessicale (valore) di **num.**
- `svalue(`**string**`)`: restituisce il valore lessicale (valore) di **string.**
- `length(L)`: restituisce la lunghezza della lista `L`**.**
- `elem(L,i)`: restituisce l'elemento `i`-esimo della lista `L`**.**
- `cons(e,L)`: restituisce la lista `(e:L)`**.**

5. Definire nel linguaggio *Scheme* la funzione `combina`, avente in ingresso una `funzione` binaria, una `lista1` e una `lista2`, che genera la lista i cui elementi sono il risultato dell'applicazione di `funzione` agli elementi di `lista1` e `lista2` nella stessa posizione. La lunghezza della lista generata coincide con la lunghezza della lista più corta. Ecco alcuni esempi:

funzione	**lista1**	**lista2**	**(combina funzione lista1 lista2)**
+	(1 2 3 4)	(10 20 30 40 50)	(11 22 33 44)
*	(1 2 3 4)	(10 20 30 40 50)	(10 40 90 160)
append	((1 2)(3 4 5))	((a b)(g d e)(z)))	((1 2 a b) (3 4 5 g d e))

6. È data la seguente dichiarazione nel linguaggio *Haskell*, per la rappresentazione di alberi:

```
data Tree = Node Int [Tree]
```

Si chiede di codificare la funzione `somma` che, ricevendo in ingresso un albero, computa la somma dei numeri memorizzati nell'albero.

7. Definire nel linguaggio Prolog il predicato **`cat(L, C)`**, in cui **`L`** è una lista di liste e **`C`** una lista, il quale risulta vero qualora **`C`** sia la concatenazione di tutte le liste di **`L`**.

8. Sulla base dei seguenti due esempi, illustrare la differenza tra i costrutti `let` e `let*` nel linguaggio *Scheme*:

```
(define n 10)
(let ((n 2) (m (+ n 5))) m)
```

```
(define n 10)
(let* ((n 2) (m (+ n 5))) m)
```