11 Gennaio 2012

Linguaggi di Programmazione

Cognome e nome	
N° fogli consegnati	

1. Specificare la definizione regolare relativa ad una tabella (non vuota), in cui ogni riga rappresenta nome, cognome e codice fiscale di una persona, come nel seguente esempio,

```
Luigi Rossi ROSLGI84L14M634U
Anna Bianchi BCIANN92A07H584X
Enrico Verdi VRDERC67M13S124Y
```

sulla base dei seguenti vincoli lessicali:

- Nome e cognome iniziano con una lettera maiuscola, seguita da una o più lettere minuscole;
- Il codice fiscale, composto da lettere maiuscole e cifre, è suddiviso in quattro parti: tre lettere per il cognome, tre lettere per il nome, la data di nascita (due cifre per l'anno, una lettera per il mese, giorno del mese compreso tra 01 e 31) ed una stringa alfanumerica di cui il primo carattere e l'ultimo sono lettere mentre i tre caratteri intermedi sono cifre;
- Nome e cognome sono seguiti da uno spazio;
- Ogni riga, ad eccezione dell'ultima, è seguita da un newline.

2. Specificare la grammatica BNF di un generatore *Haskell*-like di liste, come nel seguente esempio:

```
[ a+(b*c) | (a,b) <- alfa, (c,d,e) <- beta, a > b-2, b == d*(c/e) ]
```

La parte sinistra (target) del generatore è una espressione aritmetica. La parte destra del generatore è suddivisa in due sezioni. La prima sezione è una lista (non vuota) di condizioni di appartenenza espresse mediante il pattern tupla. La seconda sezione è una lista (non vuota) di semplici confronti fra espressioni aritmetiche (mediante gli operatori di confronto `==`, `!=`, `>` , `>=`, `<`, `<=`).

3. Specificare la semantica operazionale dell'operatore di contenimento (sottoinsieme) stretto tra tabelle (senza duplicati, per assunzione) mediante una notazione imperativa:

```
S ⊂ T
```

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

- `schema(X)`: lista di coppie (`nome`, `tipo`) che definiscono lo schema della tabella `X`;
- `istanza(X)`: lista di tuple che definiscono l'istanza della tabella `X`;
- `length(L)`: lunghezza della lista `L`.

Nel linguaggio di specifica è possibile esprimere l'operazione di appartenenza mediante il simbolo `∈` . Si richiede che le tabelle operando siano compatibili per struttura. Nel caso di errore semantico, il valore della espressione è `error`.

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

frammento → **id = [** *num-list* **] ; num in id.**
num-list → **num ,** *num-list* | **num**

esempio di frase

```
S = [1,3,6,9,24];
4 in S.
```

Il frammento è costituito da due operazioni: istanziazione di una lista di interi e appartenenza di un intero a tale lista. La lista non deve contenere duplicati e il nome della lista nella espressione di appartenenza deve coincidere con il nome della lista istanziata precedentemente. In particolare, si richiede la specifica delle seguenti funzioni semantiche:

- M_f(*frammento*): restituisce il risultato della operazione di appartenenza in *frammento*, (eventualmente `error`);
- M_{nl}(*num-list*): restituisce la lista di interi relativa a *num-list*, (eventualmente `error`).

Nel linguaggio di specifica denotazionale, è possibile esprimere l'operazione di appartenenza mediante il simbolo **∈**. Sono inoltre disponibili le seguenti funzioni ausiliarie (di cui non è richiesta la specifica):

- `name(`**id**`)`: restituisce il valore lessicale (nome) di **id**;
- `value(`**num**`)`: restituisce il valore lessicale (valore) di **num.**

5. Definire nel linguaggio *Scheme* la funzione `sumfun`, avente in ingresso un numero naturale `n` ed una una funzione unaria `f` (che computa un valore intero). La funzione `sumfun` computa `f(0) + f(1) + ... + f(n)`.

6. Specificare nel linguaggio *Haskell* la classe di tipi `AddNeg`, nella quale sono definite due funzioni:

- `add`: somma di due elementi dello stesso tipo;
- `neg`: negazione di un elemento.

Quindi, istanziare tale classe mediante i seguenti tipi:

- `Int`: in cui `add` è la somma aritmetica, mentre `neg` è il cambiamento di segno.
- `Bool`: in cui `add` è `False` se e solo se entrambi gli operandi sono `False` (altrimenti è `True`), mentre `neg` è la negazione logica.
- `[Int]`: in cui `add` genera la lista (di lunghezza minima tra le due) in cui ogni elemento è la somma dei rispettivi elementi nelle due liste, mentre `neg` è la lista in cui tutti i numeri (rispetto alla lista operando) sono cambiati di segno.

7. È data una base di fatti *Prolog* che specifica un grafo aciclico diretto, come nel seguente esempio:

```
nodi([0, 4, 5, 8, 9, 10]).

archi([arco(0, 2, 10),
       arco(0, 3, 4),
       arco(10, 3, 5),
       arco(4, 6, 5),
       arco(5, -1, 8),
       arco(5, 4, 9)]).
```

Si assume implicitamente che il nodo iniziale sia il primo della lista di `nodi`. Ogni nodo è identificato da un numero intero. Ogni arco tra due nodi è marcato da un numero intero (ad esempio, `arco(10, 3, 5)` indica un arco dal nodo `10` al nodo `5`, marcato dal numero `3`). Si chiede di specificare il predicato `bilanciato(X)`, che risulta vero se e solo se esiste un cammino nel grafo che, partendo dal nodo iniziale, raggiunge il nodo `X` in modo tale che la somma dei numeri che marcano gli archi di tale cammino coincida con `X` (nel caso di cammino nullo, tale somma vale zero per definizione).

8. Illustrare, mediante l'ausilio di semplici esempi, la differenza che sussiste nel linguaggio *Haskell* tra polimorfismo e overloading di funzioni.