15 dicembre 2008

Linguaggi di Programmazione

Nome e Cognome	
Corso di laurea	

1. Specificare la grammatica BNF del sotto-linguaggio di *Haskell* relativo alla specifica di classi di tipi. Ogni frase è composta da una lista non vuota di classi, come nel seguente esempio:

```
class Visible a where
  tostring :: a -> String
  size     :: a -> Int

class Eq a where
  (==), (/=) :: a -> a -> Bool

class Eq b => Ord b where
  (<), (<=), (>), (>=) :: b -> b -> Bool
  max, min             :: b -> b -> b
```

Ogni classe di tipo contiene almeno una funzione. Ogni funzione ha almeno un operando. Per semplicità, oltre alle variabili di tipo, si assumono unicamente i tipi `Int`, `Bool` e `String` (senza costruttori). Gli operatori binari (definiti tra parentesi) sono rappresentati dal simbolo terminale **operator** (che non include le parentesi). Si assume ereditarietà singola (se presente). Le variabili di tipo sono rappresentate dal terminale **var**.

2. Specificare la semantica denotazionale di una espressione relazionale di intersezione definita dalla seguente BNF:

intexpr → **(** *intexpr* ∩ *intexpr* **)** | **table**

in cui *intexpr* rappresenta l'operazione insiemistica di intersezione e **table** il nome di una tabella. Si assume che l'istanza (anche vuota) di ogni tabella sia rappresentata da una lista (anche vuota) di tuple senza duplicati.

In particolare, si chiede di specificare la funzione semantica $\mathbf{M}_{int}$(*intexpr*) assumendo di avere a disposizione la funzione ausiliaria $\mathbf{M}_{id}$(**table**), di cui non è richiesta la specifica, che restituisce la lista di tuple della tabella di nome **table** (se è definita) oppure **errore** (se non è definita). Si richiede che la specifica denotazionale sia definita mediante la notazione di pattern-matching. In particolare, è possibile utilizzare il pattern `(testa:coda)` per una lista non vuota, ed il pattern `[]` per una lista vuota. Il linguaggio di specifica denotazionale non contiene operatori insiemistici, ad eccezione dell'appartenenza, ∈. Non è richiesto il controllo di compatibilità dei due operandi.

3. Definire nel linguaggio *Scheme* la funzione **`inserisci`**, la quale, ricevendo in ingresso un intero **`n`** ed una **`lista`** ordinata (in modo ascendente) di numeri, genera la lista ordinata ottenuta inserendo **`n`** in lista, come nei seguenti esempi:

`n`	`lista`	`(inserisci n lista)`
4	(2 3 6 7 9)	(2 3 4 6 7 9)
5	(2 3 5 6 7)	(2 3 5 5 6 7)
6	()	(6)
8	(2 3 5 6 7)	(2 3 5 6 7 8)
1	(2 3 5 6 7)	(1 2 3 5 6 7)

Quindi, definire la funzione **`ordina`** che, ricevendo una **`lista`** (anche vuota) di interi, genera la lista ordinata.

4. Definire nel linguaggio *Haskell* la funzione **`discendenti`** (protocollo incluso) che, avente in ingresso una **`genealogia`** rappresentata da una lista di genitori associati alla rispettiva lista di figli ed una **`persona`**, computa la lista dei discendenti di **`persona`**, come nel seguente esempio:

Genitore	*Figli*
guido	luisa, franco, elena
luisa	andrea, dario, gino
franco	giovanni, paola, letizia, sofia
elena	emma, zeno
andrea	rino, anna

`genealogia`

`discendenti genealogia "luisa" = ["andrea","dario","gino","rino","anna"].`

5. Specificare nel linguaggio *Prolog* il predicato **`ins(N,L1,L2)`**, che risulta vero qualora **`N`** sia un numero intero, **`L1`** una lista ordinata (in modo ascendente, eventualmente vuota) di numeri interi, ed **`L2`** la lista ordinata ottenuta inserendo **`N`** in **`L1`**.

6. Illustrare, sulla base di alcuni semplici esempi, le differenze semantiche tra i seguenti tre diversi predicati *Prolog*:

- `X = Y`
- `X == Y`
- `X is Y`