Soluzioni esame scritto di Linguaggi di Programmazione del 11/6/18, prova A.

Domande di teoria.

A titolo esemplificativo, al fine di illustrare lo stile atteso nelle risposte, vengono presentate le risposte delle prime 6 domande.

1. Il comando GOTO permette, e in qualche modo incoraggia, la scrittura di codice poco strutturato, di difficili comprensione e dove eventuali errori sono difficilmente diagnosticabili.
2. Nelle definizione di type class viene definita la lista delle funzioni, con relativi tipi, che è necessario fornire per inserire un tipo in una type class. Opzionalmente, vengono definite sovra-classi e funzioni derivate.
3. Nella comunicazione sincrona, send e receive devono avvenire quasi contemporaneamente, nella asincrona no. Come conseguenza, la send sincrona è bloccante, e la send asincrona deve essere supportata da un buffer di dimensione arbitraria.
4. La pragmatica descrive come un particolare linguaggio di programmazione viene usato; consiste in una serie di esempi, norme e convenzioni su come vada scritto del buon codice.
5. Durante l'analisi semantica vengono svolti tutti quei controlli sulla struttura del codice che non possono essere espressi con una grammatica libera dal contesto. Principalmente controllo di tipi, ma anche controlli per esempio numero degli argomenti usati per una funzione.
6. Le funzioni di ordine superiore sono funzioni che hanno come argomenti, o che restituiscono come risultato, altre funzioni.

Grammatica

La grammatica vuol rappresentare un linguaggio di stringhe di parentesi bilanciate, dove ogni parentesi aperta a vale doppio, ossia corrisponde a due parentesi

chiuse c.

- Le parole di lunghezza minore di 11 sono
  - ε,
  - acc,
  - aacccc, acaccc, accacc
  - aaaccccccc, aacaccccc, aaccacccc, aacccaccc, aaccccacc, acaaccccc, acacacccc, acaccaccc, acacccacc, accaaccc, accacaccc, accaccacc.
- La grammatica è ambigua, infatti la stringa ε può essere ottenuta dalla due derivazioni
  - $\mathrm{L} \to \varepsilon$
  - $\mathrm{L} \to \mathrm{LL} \to \varepsilon\mathrm{L} \to \varepsilon$
- Grammatiche non ambigue equivalenti sono
  - $\mathrm{L} \to \varepsilon \mid \mathrm{aLcLcL}$
  - $\mathrm{L} \to \varepsilon \mid \mathrm{LaLcLc}$
- La grammatica genera parole p contenenti le lettere a e c e tali che:
  - il numero delle c è esattamente il doppio del numero delle a
  - per ogni sottostringa iniziale di p, in numero delle c è minore o uguale al doppio del numero delle a.
- Il linguaggio non è regolare, infatti per poter verificare le condizioni del punto precedente devo poter contare un numero arbitrariamente alto di simboli a e questo non può essere svolto con automi a stati finiti (riconoscitori dei linguaggi regolari)
- Non esiste una corrispondete espressione regolare.
- Diverse soluzioni possibili per la definizione del parser in Happy
  - Usando la grammatica non ambigua: $\mathrm{L} \to \varepsilon \mid \mathrm{aLcLcL}$

    definisco tipo di dato per la rappresentazione dell'albero

    ```
    data Par = Nul | Comb Par Par Par
    ```

    e regole

    ```
    Exp   :                              { Null}
        | 'a' Exp 'c' Exp 'c'  Exp       { Comb $2 $4 $6 }
    ```

  - Usano la grammatica ambigua originale, in questo caso la compilazione Happy avvertirà della presenza di conflitti.

    definisco tipo di dato per la rappresentazione dell'albero

    ```
    data Par = Nul | Comb Par Par  | Conc Par Par
    ```

e regole

```
Exp   :  'a' Exp 'c' Exp 'c'      { Comb $2 $4 }
  |                               { Null}
  |  Exp Exp                      { Conc $1 $2 }
```

Scoping

Allo scopo di illustrare meglio la soluzione, viene presentato lo stato dello stack di attivazione in 5 istanti diversi. Nel compito d'esame, due o tre dovrebbero essere sufficienti.

Stack di attivazione immediatamente dopo la chiamata `write ( H(v[y]) )`:

```
Main:
y -> 1
v -> [2,3,4]
------------
H:
Link Statico -> Main
x  -> (`v[y]`, Main)
```

Stack di attivazione e output immediatamente prima della terminazione della chiamata `write(H(v[y]))`:

```
Main:
y -> 2
v -> [2,6,4]
------------
H:
Link Statico -> Main
x  -> (`v[y]`, Main)

output: 2 6 4
```

Stack di attivazione e output immediatamente dopo la chiamata `write(G(x + y--))`:

```
Main:
y -> 2
v -> [2, 7, 4]
------------
F:
Link Statico -> Main
G -> H
x -> v[1]
y -> 6
```

```
------------
H:
Link Statico -> Main
x -> (`x + y--`, F)

output: 2 6 4  4
```

Stack di attivazione e output immediatamente prima della terminazione della chiamata `write(G(x + y--))`:

```
Main:
y -> 3
v -> [16, 7, 4]
------------
F:
Link Statico -> Main
G -> H
x -> v[1]
y -> 3
------------
H:
Link Statico -> Main
x -> (`x + y--`, F)

output: 2 6 4  4  16 7 4
```

Stack di attivazione e output finale:

```
Main:
y -> 3
v -> [16, 7, 4]

output: 2 6 4  4  16 7 4  11  3
```

Haskell

```
checkPermutation :: Eq a => [a] -> [a] -> Bool

checkPermutation [] [] = True
checkPermutation [] _  = False
checkPermutation (x:xs) ys = let (test, zs) = checkRemove x ys in
  test && checkPermutation xs zs
  where
    checkRemove x [] = (False, [])
    checkRemove x (y : ys) = if x == y
      then (True, ys)
```

```
      else let (test, zs) = checkRemove x ys in (test, y:zs)
-- checkRemove :: Eq a => a -> [a] -> (Bool, [a])
-- checkRemove x ys  controlla se x appare in ys e restituisce
-- il valore del test e la stringa ys con l'eventuale valore x rimosso
```

Sistemi di assegnazione di tipi

Una derivazione di tipo, non completa ma sufficiente per l'esercizio, è la seguente.

```
...                                                                  ...
-------------------------------------                                ------------------------------
x:(Ref Nat), \y:Unit |- deref x : Nat     ...|-x:(Ref Nat)   x:(Ref Nat)|-(deref x)+1 : Nat
-------------------------------------   ------------------------------------------------
... |- (\y:Unit->(deref x)) Unit -> Nat   ...|- (x:= (deref x)+1) : Unit
-------------------------------------------------------------------------
x:(Ref Nat) |- (\y:Unit->(deref x))(x:=((deref x)+1))) : Nat              ...
-------------------------------------------------------------------------   ------------------
|-(\x:(Ref Nat)->(\y:Unit->(deref x))(x:=((deref x)+1))):(Ref Nat)->Nat   |-(ref 3):(Ref Nat
-------------------------------------------------------------------------------------------------
|-(\x :(Ref Nat) -> (\y : Unit -> (deref x) )(x := ((deref x) + 1))) (ref 3) : Nat
```

- La valutazione eager restituisce 4.
- La valutazione lazy non forza la valutazione del comando

  `x := (deref x) + 1`, e pertanto restituisce il valore 3.