Prova di esame del 30 settembre 2019

Esercizio 1) [10 punti]
Marcare le affermazioni che si ritengono vere. Ogni domanda può avere un qualunque numero naturale di affermazioni vere. Vengono **assegnati** 0.5 punti sia per ogni affermazione *vera che viene marcata* che per ogni affermazione *falsa che viene lasciata non marcata*. Analogamente vengono **sottratti** 0.5 punti sia per ogni affermazione *falsa che viene marcata* che per ogni affermazione *vera che viene lasciata non marcata*.

1. …
   a. La feature **.twin** crea un duplicato dell'oggetto duplicando ricorsivamente tutti gli oggetti collegati all'oggetto originale
   b. La feature **.deep_twin** crea un duplicato dell'oggetto copiando solo il valore dei campi dell'oggetto originale
   c. La feature **.copy** copia solo i valori dei campi dell'oggetto originale senza copiare ricorsivamente tutti i valori dei campi degli oggetti collegati all'oggetto originale
   d. Dopo l'istruzione *a := b.twin* l'espressione *a = b* vale **False**

2. Nell'istruzione **inspect** …
   a. È obbligatoria almeno una clausola *when*
   b. È obbligatoria la clausola *else*
   c. I valori testati dalle clausole *when* devono essere tra loro disgiunti
   d. La variabile testata da inspect deve essere di tipo *BOOLEAN*

3. …
   a. Se la clausola *until* del **loop** è vuota il programma compila correttamente
   b. L'invariante del **loop** deve essere vero anche immediatamente prima della prima iterazione
   c. L'invariante del **loop** può essere falso immediatamente dopo l'ultima iterazione
   d. Il variante del **loop** deve essere sempre un intero positivo

4. La ridefinizione di una *feature* che è una funzione può…
   a. Cambiare arbitrariamente l'implementazione purché vengano rispettati i contratti ereditati
   b. Cambiare arbitrariamente la lista dei tipi degli argomenti
   c. Cambiare il contratto
   d. Cambiare arbitrariamente il tipo del risultato

5. …
   a. Una classe *deferred* non può ereditare da una classe *effective*
   b. Una classe *C* non può ereditare da due classi differenti *B1* e *B2*, se sia *B1* che *B2* hanno una stessa classe *A* come antenato comune
   c. Ad un entità di tipo statico *C* si può assegnare un oggetto di un tipo che è un antenato di *C*
   d. In una classe *C* una feature *f* ereditata dalla classe *A* può essere ridefinita soltanto se *f* è *deferred* in A

SOLUZIONE:

1. …
   a. La feature **.twin** crea un duplicato dell'oggetto duplicando ricorsivamente tutti gli oggetti collegati all'oggetto originale
   b. La feature **.deep_twin** crea un duplicato dell'oggetto copiando solo il valore dei campi dell'oggetto originale
   c. La feature **.copy** copia solo i valori dei campi dell'oggetto originale senza copiare ricorsivamente tutti i valori dei campi degli oggetti collegati all'oggetto originale

d. Dopo l'istruzione *a := b.twin* l'espressione *a = b* vale **False**

2. Nell'istruzione **inspect** …
   a. È obbligatoria almeno una clausola *when*
   b. È obbligatoria la clausola *else*
   c. I valori testati dalle clausole *when* devono essere tra loro disgiunti
   d. La variabile testata da inspect deve essere di tipo *BOOLEAN*

3. …
   a. Se la clausola *until* del **loop** è vuota il programma compila correttamente
   b. L'invariante del **loop** deve essere vero anche immediatamente prima della prima iterazione
   c. L'invariante del **loop** può essere falso immediatamente dopo l'ultima iterazione
   d. Il variante del **loop** deve essere sempre un intero positivo

4. La ridefinizione di una *feature* che è una funzione può…
   a. Cambiare arbitrariamente l'implementazione purché vengano rispettati i contratti ereditati
   b. Cambiare arbitrariamente la lista dei tipi degli argomenti
   c. Cambiare il contratto
   d. Cambiare arbitrariamente il tipo del risultato

5. …
   a. Una classe *deferred* non può ereditare da una classe *effective*
   b. Una classe *C* non può ereditare da due classi differenti *B1* e *B2*, se sia *B1* che *B2* hanno una stessa classe *A* come antenato comune
   c. Ad un entità di tipo statico *C* si può assegnare un oggetto di un tipo che è un antenato di *C*
   d. In una classe *C* una feature *f* ereditata dalla classe *A* può essere ridefinita soltanto se *f* è *deferred* in A

Esercizio 2) [10 punti]
Siamo in un contesto ***void safe***. La classe *INT_LINKABLE* modella il generico elemento di una lista di valori interi. La sua implementazione è la seguente:

```
class
   INT_LINKABLE
create
   set_value

feature -- accesso
   value : INTEGER
      -- L'intero memorizzato in questo elemento

   next : detachable INT_LINKABLE
      -- Il successivo elemento della lista

feature {NONE} -- assegnazione
   set_value (a_value : INTEGER)
      -- assegna l'intero memorizzato in questo elemento
      do
         value := a_value
      ensure
         value = a_value
      end

feature {NONE} -- manipolazione
   link_to (other: detachable INT_LINKABLE)
      -- collega questo elemento con `other'
      do
         next := other
      ensure
         next = other
      end

   link_after (other: INT_LINKABLE)
      -- inserisce questo elemento dopo `other' conservando quello che c'era dopo
      do
         link_to (other.next)
         other.link_to (Current)
      ensure
         next = old other.next
         other.next = Current
      end

end
```

Sempre in un contesto ***void safe***, la classe *INT_LINKED_LIST* modella una lista di interi. La sua attuale implementazione è <u>solo</u> quella fornita qua sotto:

```
class
   INT_LINKED_LIST

feature -- accesso
   first_element: detachable INT_LINKABLE
        -- Il primo elemento della lista

   last_element: detachable INT_LINKABLE
        -- L'ultimo elemento della lista
```

```
  active_element: detachable INT_LINKABLE
      -- L'elemento corrente della lista

  count: INTEGER
      -- Il numero di elementi della lista

feature -- spostamenti cursore
  start
      -- Sposta `active_element' al primo elemento
    do
      active_element:= first_element
    ensure
      active_element = first_element
    end

  last
      -- Sposta `current_element' all'ultimo elemento
    do
      active_element:= last_element
    ensure
      active_element = last_element
    end

  forth
      -- Sposta `active_element' al successivo elemento, se esiste
    do
      if attached active_element as ae then
        active_element:= ae.next
      end
    ensure
      attached old active_element as ae implies active_element = ae.next
    end

invariant
  count >= 0
  attached last_element as le implies le.next = Void
  count = 0 implies (first_element = last_element) and (first_element = Void)
    and (first_element = active_element)
  count = 1 implies (first_element = last_element) and (first_element /= Void)
    and attached active_element as ae implies (ae = first_element)
  count = 2 implies (first_element /= last_element) and (first_element /= Void)
    and (last_element /= Void)
    and attached active_element as ae implies (ae = first_element or ae = last_element)
    and attached first_element as fe implies (fe.next = last_element)
  count > 2 implies (first_element /= last_element) and (first_element /= Void)
    and (last_element /= Void)
    and attached first_element as fe implies (fe.next /= last_element)

end
```

Aggiungere alla classe *INT_LINKED_LIST* una feature *insert_after* (*a_value, target*) che inserisce un elemento contenente *a_value* dopo la prima occorrenza dell'elemento contenente *a_value*, se presente nella lista, e lo inserisce alla fine della lista altrimenti. Le linee vuote sono solo indicative e non corrispondono alle linee di codice effettivamente necessarie.

```
feature -- operazioni fondamentali
  insert_after (a_value, target: INTEGER)
-- Aggiunge `a_value' dopo la prima occorrenza di `target', se esiste
-- altrimenti aggiunge `a_value' alla fine della lista.

    require
      ____________________________________________
      ____________________________________________
    local
      ____________________________________________
      ____________________________________________
    do
      ____________________________________________
      ____________________________________________
      ____________________________________________
      ____________________________________________
      ____________________________________________
      ____________________________________________
      ____________________________________________
      ____________________________________________
      ____________________________________________
      ____________________________________________
      ____________________________________________
      ____________________________________________
      ____________________________________________
      ____________________________________________
      ____________________________________________
    ensure
      ____________________________________________
      ____________________________________________
end
```

SOLUZIONE:

asdasda

```
feature -- operazioni fondamentali
  insert_after (a_value, target: INTEGER)
-- Aggiunge `a_value' dopo la prima occorrenza di `target', se esiste
-- altrimenti aggiunge `a_value' alla fine della lista.
  local
    current_element, pre_current, new_element: like first_element
  do
    create new_element.set_value(a_value)
    from
      pre_current = Void
      current_element = first_element
    invariant
      pre_current /= Void implies pre_current.value /= target
    until
      current_element = Void or else current_element.value = target
    loop
      pre_current := current_element
      current_element := current_element.next
    end
    if current_element /= Void then
      new_element.link_after(current_element)
      if last_element = current_element then
        last_element := new_element
      end
    else
      if count = 0 then
        first_element := new_element
        last_element := new_element
      else
        if attached last_element as le then
          new_element.link_after(le)
        end
        last_element := new_element
      end
    end
    count := count + 1
  ensure
    count := old count + 1
    not old has(target) implies
      (attached last_element as le implies le.value = a_value)
    old has(target) implies
      (attached get_element(target) as ge implies
        (attached ge.next as gen implies gen.value = a_value))
  end
```

N.B.: I due invarianti che coinvolgono l'uso delle feature *has()* e *get_element()* non sono stati considerati ai fini della correzione degli elaborati, dal momento che tali feature – pur essendo state discusse a lezione – non erano presenti nel testo dell'esercizio.

Esercizio 3) [10 punti]

Elencare i tipi di features che Eiffel mette a disposizione per la scrittura dei programmi e discutere similarità e differenze sia dal punto di vista interno (cioè quello di chi sviluppa le features) che dal punto di vista esterno (cioè quello di chi le usa), presentando anche una visione grafica della loro organizzazione.

SOLUZIONE:

Eiffel mette a disposizione 3 differenti tipi di features:

1. procedure
2. funzioni
3. attributi

Procedure e funzioni possono ricevere parametri di ingresso, ognuno dei quali deve avere un tipo di dato conforme a quello presente nella dichiarazione, mentre gli attributi non possono ricevere parametri di ingresso.

Dal punto di vista **esterno**, cioè di chi utilizza le feature, funzioni e attributi restituiscono come risultato della loro invocazione un valore, che deve avere un tipo di dato conforme a quello presente nella dichiarazione, mentre la procedura non restituisce alcun valore. Inoltre, il valore restituito dalla funzione è ottenuto da una computazione, mentre quello restituito dall'attributo è ottenuto da una lettura della memoria.

Dal punto di vista **interno**, cioè di chi sviluppa le feature, sia procedure che funzioni – dette collettivamente ruotine – svolgono il loro compito mediante una computazione, mentre l'attributo utilizza solo la memoria.

La figura sottostante rappresenta graficamente la situazione.