Solution 5: Assignments and control structures

ETH Zurich

1 Assignments

The solution lists the correct statements for each of the subtasks.

1. (c)
2. (a)
3. (b)
4. (d)
5. (c)
6. (d) (e)
7. (a)
8. (c) (e)
9. (b) (e)

2 Reading loops

Version A:

- The result of the comparison using \( = \) will always be False (\textit{STRING} is a reference type).
- The result of the second if statement will always be False, because in the first if statement no assignment was performed (station is \textit{Void}).
- Regardless of the previous observation, the second if-statement is inside the loop, so it would try to move “Central” in every loop iteration after it had been found. This behavior is not incorrect, but it is inefficient, because the same operation is performed again and again while it could be performed only once after the loop.
- The corrected code of version A is shown in Listing ??.

Version B:

- Infinite loop: there is no call to a command that advances the cursor position in the list.
- Possible precondition violation: \texttt{i.item.name} \( \sim \) “Central” will most likely be tested before \texttt{i.after}, therefore trying to access an item when the cursor has already advanced past the end of the list. To get a guaranteed correct order of evaluation, switch the two conditions and use \texttt{or else} instead of \texttt{or}.
- The corrected code of version B is shown in Listing ??.
Listing 1: Version A

```plaintext
explore
  -- Move "Central".
local
  station: STATION
do
  across
    Zurich.stations as i
loop
  if i.item.name ~ "Central" then
    station := i.item
  end
end
if station /= Void then
  station.set_position ([0.0, 0.0])
end
end
```

Listing 2: Version B

```plaintext
explore
  -- Move "Central".
local
  i: like Zurich.stations.new_cursor
do
  from
    i := Zurich.stations.new_cursor
until
  i.after or else i.item.name ~ "Central"
loop
  i.forth
end
if not i.after then
  i.item.set_position ([0.0, 0.0])
end
end
```

3 Next station: loops

```plaintext
note
description: "Route information displays."

class
  DISPLAY
inherit
  ZURICH_OBJECTS

feature
  -- Explore Zurich

  add_public_transport
    -- Add a public transportation unit per line.
    do
      across
        Zurich.lines as i
    loop
      i.item.add_transport
    end
end

update_transport_display (t: PUBLIC_TRANSPORT)
  -- Update route information display inside transportation unit ‘t’.
require
t_exists: t /= Void
local
  i: INTEGER
  s: STATION
```
**Assignments**

**Fall 2014**

```
do
    console.clear
    console.append_line (t.line.number.out + " Willkommen/Welcome")
from
    i := 1
    s := t.arriving
until
    i > 3 or s = Void
loop
    console.append_line (stop_info (t, s))
    s := t.line.next_station (s, t.destination)
    i := i + 1
end
if s /= Void then
    if s /= t.destination then
        console.append_line ("...")
    end
    console.append_line (stop_info (t, t.destination))
end
end

stop_info (t: PUBLIC_TRANSPORT; s: STATION): STRING
    -- Information about stop 's' of transportation unit 't'.
require
    t_exists: t /= Void
    s_on_line: t.line.has_station (s)
local
    time_min: INTEGER
    l: LINE
do
    time_min := t.time_to_station (s) // 60
    if time_min = 0 then
        Result := "<1"
    else
        Result := time_min.out
    end
    Result := Result + " Min.%T" + s.name
end

-- Optional task:
across
    s.lines as i
loop
    l := i.item
    if l /= t.line and
        ((l.next_station (s, l.first) /= Void and not
            t.line.has_station (l.next_station (s, l.first))) or
        (l.next_station (s, l.last) /= Void and not
            t.line.has_station (l.next_station (s, l.last))))
        Result := Result + " " + i.item.number.out
    end
end
end
```
4 Board game: Part 1

There are several possible solutions; we discuss the two most reasonable in our opinion.

The simpler solution only includes three classes:

- **GAME**: encapsulates the logic of the game (start state, the structure of a round, ending conditions).
- **DIE**: provides random numbers in the required range.
- **PLAYER**: stores the state of each player in the game and performs a turn.

We discarded **ROUND** and **TURN**: we consider them parts of the **GAME** and **PLAYER** behavior respectively, rather than separate abstractions. Additionally **PLAYER** and **TOKEN** represent the same abstraction for now.

In the simple solution we don’t introduce classes for **SQUARE** and **BOARD**. The only information associated with squares in the current version of the game is their index, thus a square can be easily represented with an integer. Also the board in the current version doesn’t have any specific structure (square arrangement); the only property of the board is the number of squares, which probably does not deserve a separate class and instead can be stored in **GAME**.

A more flexible solution additionally includes classes **SQUARE** and **BOARD**. Though **SQUARE** doesn’t contain enough behavior for now, we anticipate that in the future versions of the game there might be squares with special properties and behavior (this anticipation is based on our knowledge of the problem domain, namely that interesting boardgames have squares of different types with different properties).

Introducing class **BOARD** makes the solution more flexible with respect to the arrangement of squares on the board. In the simple version the knowledge about “on which square does a token land if it moves \( n \) steps starting from square \( x \)” is located in class **PLAYER**. Once it becomes more complicated than just \( x + n \), it is better to encapsulate such knowledge in class **BOARD**.

5 MOOC: Assignment, control structures

The order in which the questions and the answers appear here in the solution may vary because they are randomly shuffled at each attempt.

References, Assignment, and Object Structure

- Choose the appropriate initialization values for the variables below: nat_val: NATURAL (0); int_val: INTEGER (0); real_val: REAL (0.0); bool_val: BOOLEAN (False); char_val: CHARACTER (null char); string_val: STRING (Void)

- Suppose to have the following class **PERSON**:

```java
class PERSON
create
    set_friend,
    default_create
```
feature -- Initialization

    set_friend (f: PERSON)
    -- Initialize current object.
    do
        friend := f
    end

feature -- Access

    friend: PERSON
end

In some other class, some objects of type PERSON are created and initialized:

create kima
create jimmy.set_friend (kima)
create buck.set_friend (jimmy)
create rhonda.set_friend (buck)
create kima.set_friend (rhonda)

We claim that there is a cycle in the four objects above. True or False? False

- Determine to whom the following calls apply: set_color ("red"): to Current; my_pic.set_color ("blue"): to the object attached to my_pic; till.friend.friend: to the object attached to till.friend.

- Determine if the following calls are qualified or unqualified: set_color ("red"): unqualified; my_pic.set_color ("blue"): qualified; arno.friend.friend: both qualified; draw: unqualified.

- Assuming you have the following definitions:

  s1: STRING = "Game"
  s2: STRING = "of Thrones"

What can you say about the following Eiffel routine?

join_strings (s1, s2: STRING)
    -- Append s2 to s1.
    do
        s1.append (s2)
    end

It works as expected: s1 has value "Game of Thrones"; This routine produces a side effect on s1.

- What can you say about the following Eiffel routine?

increment (num: INTEGER)
    -- Add 1 to num.
    do
        num := num + 1
    end

It does not work as expected: num is not incremented; It does not compile. In Eiffel you cannot assign directly to a routine argument.
• Suppose to have the following class ITEM:

class ITEM
create {ORDER_LINE}
  set_description

feature {NONE} -- Initialization
  set_description (d: STRING)
    -- Set description for current object.
    do
      description := d
    end

feature -- Basic operations
  set_price (p: INTEGER)
    -- Set price for current object.
    do
      price := p
    end

feature -- Access
  description: STRING
    -- Item description.
  price: INTEGER
    -- Item price.

end

Which of the following is true? Objects of class ITEM can be created from within objects of class ORDER_LINE; Feature set_description can be used as a creation procedure, but cannot be invoked normally (that is, not as a creation procedure) on an object of type ITEM from another class.

• Suppose to have the following class ITEM:

class ITEM

feature -- Basic operations
  set_price (p: INTEGER)
    -- Set price for current object.
    do
      price := p
    end

feature -- Access
  price: INTEGER
In some other class TEST, the following routine is declared:

\[
\textit{swap\_prices} (\textit{item\_1, item\_2}: \textit{ITEM})
\]
\[
\quad \text{-- Swap prices of items.}
\]
\[
\text{local}
\]
\[
\quad \textit{temp}: \textit{INTEGER}
\]
\[
\text{do}
\]
\[
\quad \textit{temp} := \textit{item\_1.price}
\]
\[
\quad \textit{item\_1.set\_price} (\textit{item\_2.price})
\]
\[
\quad \textit{item\_2.set\_price} (\textit{temp})
\]
\[
\text{end}
\]

Assume that in the same class TEST two references of type ITEM are declared:

\[
\textit{item\_one, item\_two}: \textit{ITEM}
\]

Then the following happens:

\[
\text{create \textit{item\_one}}
\]
\[
\text{\textit{item\_one.set\_price} (7)}
\]
\[
\text{create \textit{item\_two}}
\]
\[
\text{\textit{item\_two.set\_price} (4)}
\]
\[
\text{\textit{swap\_prices} (\textit{item\_two, item\_one})}
\]
\[
\text{\textit{print} (\textit{item\_one.price}.out)}
\]
\[
\text{\textit{print} (\textit{item\_two.price}.out)}
\]

What will be printed on the console? 47

Control Structures

- Complete the code of the following function \textit{maximum} by choosing the correct instructions:

\[
\textit{maximum} (a, b: \textit{INTEGER}): \textit{INTEGER}
\]
\[
\quad \text{-- The maximum between a and b.}
\]
\[
\text{do}
\]
\[
\quad \text{if } a > b \text{ then}
\]
\[
\quad \quad \text{Result} := a
\]
\[
\quad \text{else}
\]
\[
\quad \quad \text{Result} := b
\]
\[
\text{end}
\]

Complete the code of the following function \textit{print\_relation} by choosing the correct instructions:

\[
\textit{print\_relation} (a, b: \textit{INTEGER})
\]
\[
\quad \text{-- Prints if } a > b, a < b \text{ or } a = b.
\]
\[
\text{do}
\]
\[
\quad \text{if } a > b \text{ then}
\]
\[
\quad \quad \text{\textit{print} (a.out + ”>” + b.out)}
\]
\[
\quad \text{else}
\]
\[
\quad \quad \text{if } a < b \text{ then}
\]
\[
\quad \quad \quad \text{\textit{print} (a.out + ”<” + b.out)}
\]
else
    print ("The 2 numbers are equal.")
end
end

• Complete the code of the following function remainder by choosing the correct instructions. Assume d1 and d2 are positive.

\[
\text{remainder} (d1, d2: \text{INTEGER}): \text{INTEGER}
\]

\[
\text{-- Compute the remainder of integer division between d1 and d2.}
\]
do
    from
    Result := d1
    until
        Result <= d2
        loop
            Result := Result \text{-} d2
        end
        \text{-- nothing here}
    end

• Complete the code of the following function absolute\_value by choosing the correct instructions:

\[
\text{absolute\_value} (a: \text{INTEGER}): \text{INTEGER}
\]

\[
\text{-- Absolute value of a.}
\]
do
    if \ a >= 0 \ then
        Result := a
    else
        Result := \text{-}a
    end
end

• Assuming that \text{c} is a CHARACTER, what will the following instruction print, if executed with \text{c} = \text{0}?\n
\[
\text{inspect c}
\]
\[
\text{when '1'..'9' then}
\text{print ("number")}
\text{when 'a'..'z' then}
\text{print ("lower case letter")}
\text{when 'A'..'Z' then}
\text{print ("upper case letter")}
\text{when '!', '@', '%' then}
\text{print ("special character")}
\text{else}
\text{print ("unexpected character")}
end
\]

It will print “unexpected character”.\n
• Complete the code of the following function euclid by choosing the correct expressions for the loop invariant and the loop variant:

```
module euclid (a, b: INTEGER): INTEGER
    -- Greatest common divisor of a and b.
require
    a_positive: a > 0
    b_positive: b > 0
local
    m, n: INTEGER
do
    from
        m := a
        n := b
    invariant
        euclid (a, b) = euclid (m, n)
    variant
        m + n
    until
        m = n
    loop
        if m > n then
            m := m - n
        else
            n := n - m
        end
    end
Result := m
end
```

Listing 3: Class WORD_GAMES

**note**

**description:** "Objects of this class store and manage a list of books."

class

```
LIBRARY
```

inherit

```
ANY
    redefine
default_create
end
```

feature {NONE} -- Initialization

```
default_create
    -- Create an empty library.
    do
        create all_books.make
    end
```
feature -- Access

is_copy_available (title, author: STRING): BOOLEAN
    -- Is a copy with given title and author available?
    do
        from all_books.start
        until all_books.after or Result
        loop
            if all_books.item.title ~ title and all_books.item.author ~ author and all_books.item.
            number_of_available_copies > 0 then
                Result := True
            end
        all_books.forth
    end
end

feature -- Element change

extend (a_book: BOOK)
    -- Extend library with a_book.
    require
        not book_in_library (a_book)
    do
        all_books.extend (a_book)
    ensure
        one_more: all_books.count = old all_books.count + 1
        book_added: all_books.last = a_book
    end

remove (a_book: BOOK)
    -- Remove a_book from library.
    require
        book_in_library (a_book)
    do
        all_books.start
        all_books.search (a_book)
        all_books.remove
    ensure
        one_less: all_books.count = old all_books.count - 1
        book_not_in_library: not all_books.has (a_book)
    end

feature -- Output

get_all_titles: STRING
    -- Return titles of all books in the library.
    do
        if all_books.is_empty then
            Result := "No book available at the moment"
else  
    Result := ""
from
    all_books.start
until
    all_books.after
loop
    Result.append (all_books.item.title + ", " )
    all_books.forth
end  
−− Remove the last ", "
    Result.remove_tail (2)
end
end

feature {NONE}  
−− Implementation

    all_books: LINKED_LIST [BOOK]  
−− List of books in the library.

feature  
−− Contracts

    book_in_library (some_book: BOOK): BOOLEAN
    do
        Result := all_books.has (some_book)
    end

end

Listing 4: Class WORD_GAMES

note
    description: "The class \{PALINDROME\} implements algorithms that are related to strings."
    author: "hce"
    date: "11.07.2013"

class
    WORD_GAMES

feature  
−− Basic algorithms

    is_palindrome (s: STRING): BOOLEAN
    −− Returns true if 's' is a palindrome.
    require
        input VALID: s /= Void and not s.is_empty
    local
        l_reversed_s: STRING
        i: INTEGER
    do
        −− We start with an empty reversed string.
        l_reversed_s := ""
        Result := false
from
  \(i := s.\text{count}\)
until
  \(i = 0\)
loop
  \[\text{-- We append to the reversed string the characters from}\]
  \[\text{-- the } s, \text{ read from the end to the beginning (in reverse order)}\]
  \(l_{\text{reversed}}.\text{append character}(s.\text{at}(i))\)
  \(i := i - 1\)
end
\[\text{-- If a string is the same as its reversed, then it is palindrome.}\]
if \(l_{\text{reversed}}.\text{is equal}(s)\) then
  \(\text{Result} := \text{true}\)
end
end

Listing 5: Class \texttt{DECIMAL\_TO\_BINARY\_ CONVERTER}

class \texttt{DECIMAL\_TO\_BINARY\_ CONVERTER}

feature -- Conversion

valid_input \((n: \texttt{INTEGER})\): \texttt{BOOLEAN}
  \[\text{-- Is ‘}n\text{’ a valid input for a conversion?}\]
  do
    \(\text{Result} := 0 <= n \text{ and } n <= 10000000\)
  end

to_binary \((n: \texttt{INTEGER})\): \texttt{STRING}
  \[\text{-- Binary representation of a number ‘}n\text{’ expressed in base 10.}\]
  require
    valid_input: valid_input \((n)\)
  local
    \(my\_local: \texttt{INTEGER}\)
  do
    if \(n = 0\) then
      \(\text{Result} := \text{“}0\text{”}\)
    else
      from
        \[\text{-- We will build the result string digit by digit}\]
      \[\text{Result} := \text{“}\]
        \[\text{-- We start from } n \text{ and save it in our temp variable}\]
      \]
\[ my_{local} := n \]

**invariant**

--- Our invariant states that at every iteration the value
--- in our temp variable corresponds to \( n \) divided by \( 2 \)
--- to the power of the number of elements in the result string.
--- The truncation to an integer is necessary because \( \hat{\cdot} \) gives a real.

\[ my_{local} = n \div (2^n \cdot \text{Result}.\text{count} \cdot \text{truncated_to_integer} \]  

**until**

--- We exit the loop when \( my_{local} \) reaches 0

\[ my_{local} = 0 \]

**loop**

--- We build the result string one digit at the time
--- Note that we are using the modulus operator
--- for computing the remainder of integer division

\[ \text{Result}.\text{prepend_integer} (my_{local} \mod 2) \]

--- Now we update \( my_{local} \) using the integer division

\[ my_{local} := my_{local} \div 2 \]

**variant**

--- This is always decreasing and positive

\[ my_{local} + 1 \]

**end**

**ensure**

\[ result_{exists}: result \neq \text{Void} \text{ and not Result.is_empty} \]

**end**

**end**