Introduction to Programming Exam

ETH Zurich

Date: 22 August 2011

Family name, first name: .....

Student number: .....

I confirm with my signature, that I was able to take this exam under regular circumstances and that I have read and understood the directions below.

Signature: .....

Directions:

- Exam duration: 120 minutes.
- Except for a dictionary you are not allowed to use any supplementary material.
- Use a pen (**not** a pencil)!
- Please write your student number onto **each** sheet.
- All solutions can be written directly onto the exam sheets. If you need more space for your solution ask the supervisors for a sheet of official paper. You are **not** allowed to use other paper.
- Only one solution can be handed in per question. Invalid solutions need to be crossed out clearly.
- Please write legibly! We will only correct solutions that we can read.
- Manage your time carefully (take into account the number of points for each question).
- Please **immediately** tell the exam supervisors if you feel disturbed during the exam.

Good luck!

Question	Number of possible points	Points
1	10	
2	12	
3	12	
4	15	
5	15	
Total	64	

1 Multiple choice (10 points)

Put checkmarks in the checkboxes corresponding to the correct statements. A correctly checked or unchecked box is worth 0.5 points. An incorrectly checked or unchecked box is worth 0 points.

Example.

a.	The sun is a mass of incandescent gas.	☒	0.5 points
b.	$2 \times 2 = 4$	☐	0 points
c.	Britney Spears is an honorary doctor of ETH.	☐	0.5 points
d.	"Rösti" is a kind of sausage.	☒	0 points

1. Objects and classes

a.	All types are either reference or expanded.	☒
b.	If an object is of an expanded type, its fields cannot be modified at runtime.	☐
c.	Suppliers of class C can use all the features of class C.	☐
d.	A class can be both a supplier and a client.	☒
e.	If C is a deferred class, then no entity can exist in a program with static type C.	☐

2. Control structures and recursion

a.	A loop invariant is allowed to be violated between the execution of any two instructions in the loop body.	☒
b.	The purpose of the loop variant is to guarantee termination of the loop.	☒
c.	Calling routine r can result in infinite recursion only if the body of r contains a call to r.	☐
d.	For every algorithm containing a loop there is an equivalent recursive algorithm that does not contain loops.	☒
e.	For an **inspect** instruction with n options the equivalent code not involving an **inspect** instruction contains n^2 conditionals.	☐

3. Design by Contract

a.	An empty postcondition is equivalent to the postcondition **True**.	☒
b.	An empty precondition is equivalent to the precondition **False**.	☐
c.	When reasoning about a creation procedure *make*, you are allowed to assume that the class invariant of the object being created holds at the beginning of *make*.	☐
d.	The invariant of a descendant class implies the invariant of its ancestor.	☒
e.	A routine with an empty contract and an empty body is correct.	☒

4. Inheritance and polymorphism

a.	A deferred class cannot inherit from an effective class.	☐
b.	A class C cannot inherit from two different classes *A1* and *A2*, if both *A1* and *A2* have a common ancestor class.	☐
c.	An instruction *o.f* at runtime can result in executing different routines.	☒
d.	An entity of static type C can only be attached to an object of a type that is an ancester of C.	☐
e.	In class C a feature f inherited from class A can only be redefined if f is deferred in A.	☐

2 Inheritance and Polymorphism (12 Points)

Below you can see the class diagram and the source code of the revolutionary new iPlayer, an audio and video player designed by ETH in Zürich, Switzerland.

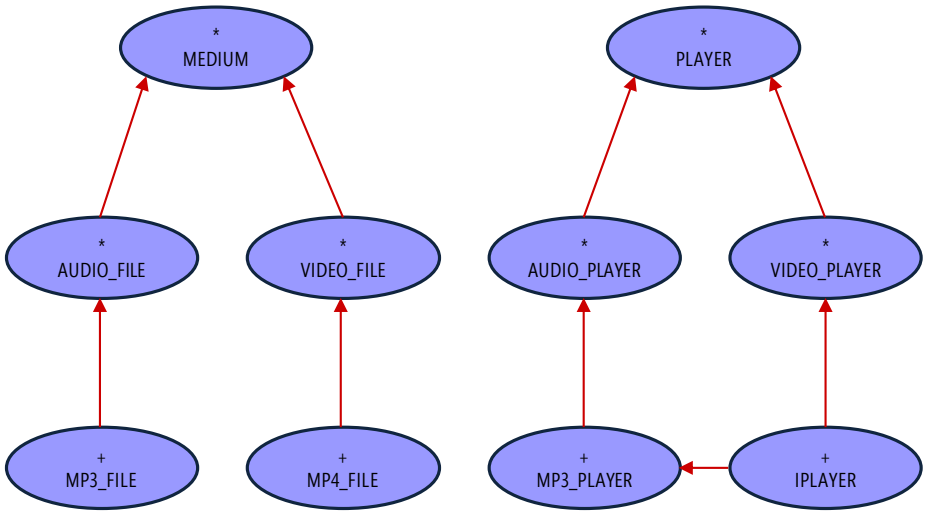

Figure 1: Class digram of iPlayer

```
deferred class MEDIUM

feature
  type: STRING
      -- Type of the medium.
    deferred
    end
end
```

```
deferred class AUDIO_FILE

inherit MEDIUM

feature
  type: STRING
    do
      Result := "audio file"
    end
end
```

```
class MP3_FILE

inherit AUDIO_FILE
    redefine type end

feature
  type: STRING
```

```
    do
      Result := "mp3 file"
    end
end
```

```
deferred class VIDEO_FILE

inherit MEDIUM

feature
  type: STRING
    do
      Result := "video file"
    end
end
```

```
class MP4_FILE

inherit VIDEO_FILE
    redefine type end

create
  make

feature
  type: STRING

  make
    do
      type := "mp4 file"
    end
end
```

```
deferred class PLAYER

feature
  play (m: MEDIUM)
      -- Play a medium 'm'.
    do
      io.put_string ("Player is playing a " + m.type + ".%N")
    end
end
```

```
deferred class AUDIO_PLAYER

inherit PLAYER
    redefine play end

feature
  play (a: AUDIO_FILE)
    do
      io.put_string ("Audio player is playing a " + a.type + ".%N")
```

```
    end
end
```

```
class MP3_PLAYER

inherit AUDIO_PLAYER
    redefine play end

feature
  play (m: MP3_FILE)
    do
      io.put_string ("MP3 player is playing a " + m.type + ".%N")
    end
end
```

```
deferred class VIDEO_PLAYER

inherit PLAYER
    undefine play end

feature
  play (a: VIDEO_FILE)
    deferred
    end
end
```

```
class IPLAYER

inherit
  MP3_PLAYER
    rename play as play_mp3
    select play_mp3 end
  VIDEO_PLAYER
    rename play as play_mp4
    redefine play_mp4 end

feature
  play_mp4 (m: MP4_FILE)
    do
      io.put_string ("iPlayer is playing a " + m.type + ".%N")
    end
end
```

Given the following variable declarations:

```
player: PLAYER
mp3_player: MP3_PLAYER
mp3_file: MP3_FILE
mp4_file: MP4_FILE
iplayer: IPLAYER
```

Indicate, for each of the code fragments below, if it is valid (i.e. will compile) by checking the corresponding box. If the code fragment is not valid (i.e. will not compile), explain why this is the case. If the code fragment compiles, specify the text that is printed to the screen when the code fragment is executed. No other explanations are necessary.

Example:

```
create mp3_file.make
create mp3_player
player.play (mp3_file)
```

Does the code compile? ☐ Yes ☒ No
Output/error description: The code does not compile, because the feature *make* is not a creation procedure of class *MP3_FILE* (in fact, it is not even a valid feature of the class).

Grading Scheme

1 Pt: For stating correctly whether it compiles/doesn't compile.
1 Pt: For providing the correct output (if it compiles) or the reason why it doesn't compile.

Task 1

```
create mp3_file
create player
player.play (mp3_file)
```

Does the code compile? ☐ Yes ☒ No
Type error: creation instruction applies to target of a deferred type.

Task 2

```
create mp3_player
create mp3_file
player := mp3_player
player.play (mp3_file)
```

Does the code compile? ☒ Yes ☐ No
"MP3 player is playing a mp3 file."

Task 3

```
create mp4_file.make
create iplayer
iplayer.play_mp4 (mp4_file)
```

Does the code compile? ☒ Yes ☐ No
"iPlayer player is playing a mp4 file."

Task 4

```
create mp3_file
player := create {IPLAYER}
player.play (mp3_file)
```

Does the code compile? ☒ Yes ☐ No
"MP3 player is playing a mp3 file."

Task 5

```
create mp3_player
mp3_player.play (create {MP4_FILE}.make)
```

Does the code compile? ☐ Yes ☒ No
Type error: non-compatible actual argument in feature call.

Task 6

```
create {MP3_PLAYER} iplayer
iplayer.play_mp3 (create {MP3_FILE})
```

Does the code compile? ☐ Yes ☒ No
Error: Creation instruction lists explicit creation type which does not conform to type of target.

3 Specifying Software through Contracts (12 points)

In the Western calendar, a year is divided into 12 months, numbered from 1 to 12, and months into days, numbered starting from 1. The 1st, 3rd, 5th, 7th, 8th, 10th, and 12th months have 31 days each, while the 4th, 6th, 9th, and 11th months have 30 days each. The 2nd month usually has 28 days, but it has 29 days in leap years. A year is a *leap year* if and only if either it can be divided by 4 but not by 100, or it can be divided by 400. For example, the years 2000 and 2004 are leap years, but the years 2011 and 2100 are not.

The following deferred class *DATE* defines a simple interface for dates in the Western calendar. Please fill in the missing contracts (preconditions, postconditions, and class invariants) of the class; the contracts must reflect all the details given in the informal specification above. They also must ensure that the following client procedure always executes without contract violations:

```
client (d: DATE)
    require
        d /= Void
    local
        i: INTEGER
    do
        d.set (2011, 8, 22)
        check not d.is_leap (2011) end
        check d.is_leap (2012) end
        from i := 1 until i > 366 loop
            d.proceed
            i := i + 1
        end
        check d.year = 2012 and d.month = 8 and d.day = 22 end
    end
```

Please note that the number of dotted lines is not indicative of the number of missing assertions (contract elements).

```
deferred class
    DATE

feature -- Access

    year: NATURAL
            -- Year of the date.
        deferred
        end

    month: NATURAL
            -- Month of the date.
        deferred
        end

    day: NATURAL
            -- Day of the date.
        deferred
        end

feature -- Status set
```

```
    set (y, m, d: NATURAL)
            -- Set 'year', 'month', and 'day' using 'y', 'm', and 'd', respectively.
        require
            valid_month: 1 <= m and m <= 12 -- +0.5
            valid_day: 1 <= d and d <= days_in_month (y, m) -- +0.5
        deferred
        ensure
            year_set: year = y -- +0.5
            month_set: month = m -- +0.5
            day_set: day = d -- +0.5
        end


feature -- Auxiliary queries

    is_long_month (m: NATURAL): BOOLEAN
            -- Does month 'm' have 31 days?
        deferred
        ensure
            definition: Result = (m = 1 or m = 3 or m = 5 or m = 7 or m = 8 or m = 10
                or m = 12) -- +1
        end

    is_short_month (m: NATURAL): BOOLEAN
            -- Does month 'm' have 30 days?
        deferred
        ensure
            definition: Result = (m = 4 or m = 6 or m = 9 or m = 11) -- +1
        end

    days_in_month (y, m: NATURAL): NATURAL
            -- Number of days in month 'm' of year 'y'.
        require
            valid_month: 1 <= m and then m <= 12 -- +0.5
        deferred
        ensure
            long_months: is_long_month (m) implies Result = 31 -- +0.5
            short_months: is_short_month (m) implies Result = 30 -- +0.5
            feb_in_nonleap: (m = 2 and not is_leap(y)) implies Result = 28 -- +0.5
            feb_in_leap: (m = 2 and is_leap(y)) implies Result = 29 -- +0.5

        end

    is_leap (y: NATURAL): BOOLEAN
            -- Is 'y' a leap year?
        deferred
        ensure
            definition: Result = (y \\ 4 = 0 and y \\ 100 /= 0) or else (y \\ 400 = 0) --
                +1
    end
```

```
feature -- Basic operation

    proceed
            -- Proceed to the next date.
            -- For example, if 'Current' represents the date Dec. 31, 2011 before the call,
            -- then it represents Jan. 1, 2012 afterwards.
        deferred
        ensure
            within_month: old day < days_in_month(old year, old month) implies
                    (old year = year and old month = month and day = old day + 1) --
                        +1
            to_next_month: (old day = days_in_month(old year, old month) and old month <
                12) implies
                    (old year = year and month = old month + 1 and day = 1) -- +1
            to_next_year: (old day = days_in_month(old year, old month) and old month =
                12) implies
                    (year = old year + 1 and month = 1 and day = 1) -- +1
        end

    -- Other features omitted for brevity.

invariant

    valid_month: 1 <= month and then month <= 12 -- +0.5
    valid_day: 1 <= day and then day <= days_in_month (year, month) -- +0.5

end
```

Grading:

- For each incorrect extra contract clause: -0.5.

4 Data Structures (15 points)

4.1 Background information

A skip list is a data structure that expands on the idea of a linked list. A node in a linked-list has 1 link; each node in a skip list has 4 links, *up*, *down*, *left*, and *right*.

A skip list has the following properties:

- The nodes are arranged into rows; each row is a list of **sorted** elements.
- Every row, except for the bottom row, contains a subset of the elements beneath it, as in Figure 2. This implies that the bottom row contains all the elements in the skip list.
- All nodes are mutually linked, i.e. *node_a.left* = *node_b* iff *node_b.right* = *node_a*, and likewise for *up* and *down*.
- Every row begins with a universal minimal element (represented here by $-\infty$).
- If an element exists in two adjacent rows, then the nodes are linked through the *up*/*down* attributes. This can be seen for the elements 20 and $-\infty$ in Figure 2.

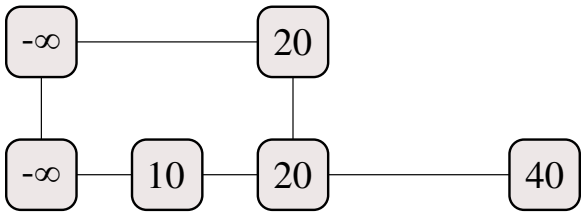

Figure 2: Initial skip list

When a new element is inserted into the skip list, it is first inserted into the bottom row, as in Figure 3.

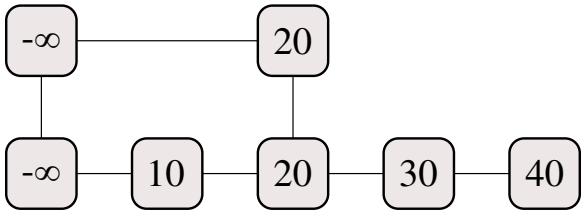

Figure 3: Skip list after insertion of 30 into the bottom row

Whenever a node is added to any row, there is a chance that it will be promoted, adding it to the row above, as in Figure 4. If there is no row above, a new one will be created. This promotion to the row above happens randomly, and a promotion can trigger another promotion (again, randomly).

4.2 Task

For the task the *search* feature is already implemented, and returns the rightmost node in the bottom row of the skip list less-than or equal to the argument *elem*. Feature *is_promoted* randomly returns **True** or **False**, indicating whether to promote a node at any given time. You must implement:

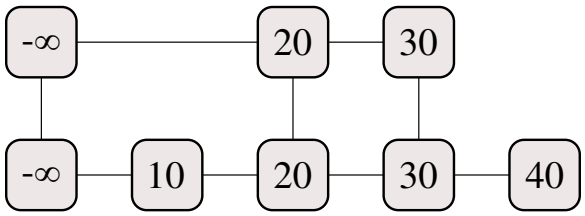

Figure 4: Skip list after promotion of 30-node

- *insert_in_row* (*a_pre*, *a_node*: *SKIP_LINKABLE*) inserts *a_node* directly after *a_pre* with no promotion. An instance of this can be seen in the transformation between Figure 2 and Figure 3.
- *promote* (*a_link*: *SKIP_LINKABLE*) takes *a_link*, which is already inserted in a row, and either promotes it or does nothing. Remember, *promote* can trigger another promotion.
- *insert* (*elem*: *INTEGER*) takes an element and inserts a new node into the correct position in the skip list, including promotion (if any).

While writing these procedures you are encouraged to use any applicable features already available in the *SKIP_LIST* and *SKIP_LINKABLE* classes (i.e. the features shown below without dotted lines).

4.3 Solution

```
class
  SKIP_LIST [G -> COMPARABLE]

inherit
  ANY
    redefine
      out
    end

create
  make

feature
  make (a_signal: G)
    do
      create rand.set_seed (42)
      minimum := a_signal

      create head.make (minimum)
    end

  out: STRING
    local
      curs: SKIP_LINKABLE [G]
    do
      Result := ""
```

```
    from curs := head
    until curs.down = Void
    loop
      curs := curs.down
    end

    from
    until curs = Void
    loop
      Result := Result + curs.value.out + ","
      curs := curs.right
    end
  end

minimum: G

head: SKIP_LINKABLE [G]

has (elem: G): BOOLEAN
  do
    Result := search (elem).value = elem
  end

search (elem: G): SKIP_LINKABLE [G]
  local
    curs: SKIP_LINKABLE [G]
    done: BOOLEAN
  do
    from curs := head
    until curs = Void or done
    loop
      if elem = curs.value then
        from
        until curs.down = Void
        loop curs := curs.down
        end

        Result := curs
        done := True
      elseif elem > curs.value then
        if curs.right = Void or else
            elem < curs.right.value then
          if curs.down = Void then
            Result := curs
            done := True
          end
        curs := curs.down
        else
          curs := curs.right
        end
      end
    end
```

```
  ensure
    result_exists: Result /= Void
    result_precedes_element: Result.value <= elem
  end

insert (elem: G)
  require
    not has (elem)
  local
    new_link: SKIP_LINKABLE [G]
  do
    create new_link.make (elem)

    insert_in_row (search (elem), new_link)
    promote (new_link)
  ensure
    has (elem)
  end

insert_in_row (a_pre, a_node: SKIP_LINKABLE [G])
  require
    nodes_exist: attached a_pre and attached a_node
    different_nodes: a_pre /= a_node
  do
    a_node.set_right (a_pre.right)
    a_node.set_left (a_pre)

    if a_node.right /= Void then
      a_node.right.set_left (a_node)
    end

    a_pre.set_right (a_node)
  end

promote (a_link: SKIP_LINKABLE [G])
  require
    node_exists: attached a_link
    already_inserted: attached a_link.left
  local
    curs: SKIP_LINKABLE [G]
    new_link: SKIP_LINKABLE [G]
  do
    if is_promoted then
      from curs := a_link
      invariant curs /= Void
      until curs.up /= Void or curs.left = Void
      loop curs := curs.left
      end

      if curs.up = Void then
        curs.set_up (create {SKIP_LINKABLE[G]}.make (minimum))
        curs.up.set_down (curs)
```

```
            end
            curs := curs.up

            create new_link.make (a_link.value)
            insert_in_row (curs, new_link)

            a_link.set_up (new_link)
            new_link.set_down (a_link)

            promote (new_link)
          end
        end

    rand: RANDOM

    is_promoted: BOOLEAN
        do
          Result := (rand.item \\ 2) = 0
          rand.forth
        end
end
```

4.4 Grading Scheme

insert (4 points):

- creating a new link with the element – 1 point
- search for the correct spot to insert – 1 point
- inserting the new link at the correct spot – 1 point
- begin the promotion sequence – 1 point

insert_in_row (4 points):

- point the new node's left to the previous node – 1 point
- point the new node's right to the previous node's right – 1 point
- if the new node's right exists, back point its left to the new node – 1 point
- point the previous node's right to the new node – 1 point

promote (7 points):

- only promote randomly – 1 point
- find left-most node – 2 points
- possibly create a new row (minimal element) – 1 point
- create a new link in the upper row – 1 point
- link the lower and upper levels – 1 point
- loop / recurse – 1 point

5 Agents (15 Points)

5.1 Background information

Classes *LIBRARY* (listing 1) and *BOOK* (listing 2) model a library containing books. Class *APPLICATION* (listing 3) is used to query the library for certain kinds of books. You can assume that the library is filled with books.

For reference, we provide an interface excerpt for the class *ARRAYED_LIST* in listing 4.

5.2 Task

Your task is to fill in the dotted lines to implement the missing code in the classes below. You are not supposed to add declarations of new features or local variables.

5.3 Listings

Listing 1: *LIBRARY* class excerpt

```
class
  LIBRARY

feature -- Access

  books: ARRAYED_LIST [BOOK]

  matched: ARRAYED_LIST [BOOK]

feature -- Basic operations

  match (criterion: FUNCTION [ANY, TUPLE [BOOK], BOOLEAN])
      -- Find a match given 'criterion'.
    do
      matched.wipe_out

      .....

    end
end
```

Listing 2: *BOOK* class excerpt

```
class
  BOOK

feature -- Access

  title: STRING
      -- Title of 'Current'.

  number_of_copies: INTEGER
      -- Number of copies of 'Current'.
end
```

Listing 3: *APPLICATION* class excerpt

```
class
  APPLICATION
create
  make

feature {NONE} -- Initialization

  make
      -- Run application.
    local
      my_library: LIBRARY
    do
      create my_library.make

        -- print the titles of the books having less than 5 copies.

      .....

      .....

        -- print the number of copies of the book "Pushing Ice".

      .....

      .....

    end

feature -- Output

  print_result (res: ARRAYED_LIST [BOOK])
      -- Print query result on console.
    do
      io.new_line
      if res.is_empty then
        print ("No books match the given criterion.")
      else
        from
          res.start
        until
          res.after
        loop
          print (res.item.title + ", " + res.item.number_of_copies.out + "; ")
          res.forth
        end
      end
    end

feature -- Query routines

  copies_less_than (b: BOOK; c: INTEGER): BOOLEAN
```

```
        -- Are there less than 'c' copies of 'b'?
    do
        ...
    end

  title_matches (b: BOOK; s: STRING): BOOLEAN
        -- Does 'b''s title match 's'?
    do
        ...
    end
end
```

Listing 4: *ARRAYED_LIST* interface excerpt

```
class ARRAYED_LIST [G]

feature -- Iteration

  do_all (action: PROCEDURE [ANY, TUPLE [G]])
        -- Apply 'action' to every item, from first to last.
        -- Semantics not guaranteed if 'action' changes the structure;
        -- in such a case, apply iterator to clone of structure instead.

  do_if (action: PROCEDURE [ANY, TUPLE [G]]; test: FUNCTION [ANY, TUPLE [G],
      BOOLEAN])
        -- Apply 'action' to every item that satisfies 'test', from first to last.
        -- Semantics not guaranteed if 'action' or 'test' changes the structure;
        -- in such a case, apply iterator to clone of structure instead.

  there_exists (test: FUNCTION [ANY, TUPLE [G], BOOLEAN]): BOOLEAN
        -- Is 'test' true for at least one item?

  for_all (test: FUNCTION [ANY, TUPLE [G], BOOLEAN]): BOOLEAN
        -- Is 'test' true for all items?

feature -- Element change

  force, extend (v: like item)
        -- Add 'v' to end.
        -- Do not move cursor.

feature -- Removal

  wipe_out
        -- Remove all items.

end -- class ARRAYED_LIST
```

5.4 Grading

The source code for the master solution is part of this exam.

- Class *APPLICATION*
  - 3 points for the correct first invocation of feature *match*.
  - 1 points for the correct first invocation of feature *print_result*.
  - 3 points for the correct second invocation of feature *match*.
  - 1 points for the correct second invocation of feature *print_result*.
  - 2 points for the correct implementation of query *copies_less_than*.
  - 2 points for the correct implementation of query *title_matches*.
- Class *LIBRARY*
  - 3 points for the correct implementation of feature *match*.