

Mock Exam 2

ETH Zurich

December 2, 2015

Name: _____

Group: _____

Question 1	/ 11
Question 2	/ 16
Question 3	/ 14
Total	/ 41

1 Contracts (11 points)

We are interested in a software system simulating a cellular automaton. The universe is represented by a finite square grid composed of square cells (there is at least 1). Each cell can be in two states: alive or dead. Every cell, depending on its position in the grid, can have from a minimum of 3 neighbors (a cell in a corner) to a maximum of 8 neighbors (a cell in the middle).

The evolution of the automaton from one generation to the next is fully determined by the following set of rules:

- Any living cell with less than 2 living neighbors dies in the next generation.
- Any living cell with 2 or 3 living neighbors lives in the next generation.
- Any living cell with more than 3 living neighbors dies in the next generation.
- Any dead cell with exactly 3 living neighbors becomes alive in the next generation.
- Any dead cell with a number of living neighbors different from 3 stays dead in the next generation.

The evolution from one generation into the next happens by applying the above rules simultaneously to every cell in the grid (see Figures 1 and 2).

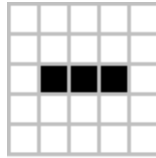


Figure 1: Sample first generation. A black square is a living cell.

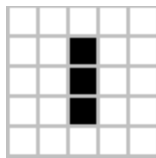


Figure 2: Second generation, computed from the first according to the given set of rules.

Your task is to add appropriate contracts (preconditions, postconditions and class invariants) to the excerpt of class `CELL_GRID` below, so that the informal specification above and the feature comments are reflected in each class interface.

Please note that the number of dotted lines does not indicate the number of missing contracts. It might also be useful to have a look at the excerpt of class `ARRAY_2` shown below.

```
class
  CELL_GRID
create
  make

feature {NONE} -- Initialization
```

```
make (a_dimension: INTEGER)
    -- Initialize grid's dimension to 'a_dimension' and its cells to dead.
    require
        .....
        .....
        .....
        .....
        .....
        .....

    do
        -- Implementation omitted.
    ensure
        .....
        .....
        .....
        .....
        .....
        .....

end

feature -- Access

    dim: INTEGER
        -- Grid dimension.

    cell_at (i, j: INTEGER): BOOLEAN
        -- Value of cell at (i, j).
    require
        .....
        .....
        .....
        .....
        .....
```

```
.....  
  
do  
    -- Implementation omitted.  
ensure  
  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
  
end  
  
feature -- Status Setting  
  
    set_cell_status (b: BOOLEAN; i, j: INTEGER)  
        -- Set status of cell at (i,j).  
    require  
  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
  
do  
    -- Implementation omitted.  
ensure  
  
.....  
.....  
.....  
.....  
.....
```

```
.....

end

feature -- Basic operations

  compute_next_generation
    -- Compute next generation of cells.
  do
    -- Implementation omitted.
  end

feature {NONE} -- Implementation

  current_grid: ARRAY2 [BOOLEAN]
    -- Grid representation as a matrix of boolean cells ("True" means alive for a
    cell).

  new_state_of_cell (i, j, living_neighbors: INTEGER): BOOLEAN
    -- Apply Conway's Game of Life rules to compute new state for cell at (i,j)
    given a number of 'living_neighbors'.

  require
    .....
    .....
    .....
    .....
    .....
    .....
    .....

  do
    -- Implementation omitted.
  ensure
    .....
    .....
    .....
    .....
    .....
    .....
```

```
    end

invariant
    .....
    .....
    .....
    .....
    .....
    .....

end

class
    ARRAY2 [G]

    make_filled ( a_default_value : G; nb_rows, nb_columns: INTEGER)
        -- Create a two-dimensional array which has 'nb_rows'
        -- rows and 'nb_columns' columns, with lower bounds
        -- starting at 1 and filled with value 'a_default_value'.

    item alias "[]" (row, column: INTEGER): G
        -- Entry at coordinates ('row', 'column')

    height: INTEGER
        -- Number of rows

    width: INTEGER
        -- Number of columns

    all_default : BOOLEAN
        -- Are all items set to default values?

end
```

2 Data Structures (16 points)

In this task you are going to implement several operations for a generic class *SET* [*G*].

A set is a collection of distinct objects. Every element of a set must be unique; no two members may be identical. All set operations preserve this property. The order in which the elements of a set are listed is irrelevant (unlike for a sequence or tuple). Therefore the two sets {5, 10, 12} and {10, 12, 5} are identical.

There are several fundamental operations for constructing new sets from given sets.

- Union: The union of *A* and *B*, denoted by $A \cup B$, is the set of all elements that are members of either *A* or *B*.
- Intersection: The intersection of *A* and *B*, denoted by $A \cap B$, is the set of all elements that are members of both *A* and *B*.
- Relative complement of *B* in *A* (also called the set-theoretic difference of *A* and *B*), denoted by $A \setminus B$ (or $A - B$), is the set of all elements that are members of *A* but not members of *B*.

The Jaccard index (or coefficient) measures similarity between sample sets, and is defined as the size of the intersection divided by the size of the union of the sample sets (see Figure 3). If both sets are empty the Jaccard coefficient is defined as 1.0.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Figure 3: Jaccard index definition for non-empty sets *A* and *B*.

Your task is to fill in the gaps of class *SET* [*G*] below. Please note:

- Your code should satisfy the contracts and provide new contracts where necessary.
- The set should never contain **Void** elements.
- The number of dotted lines does not indicate the number of missing contract clauses or code instructions.
- The implementation of class *SET* [*G*] is based on an arrayed list. The arrayed list is set up to use object comparison, so features like *has* and *prune* use object equality instead of reference equality when comparing elements from the set. The following features of class *ARRAYED_LIST* may be useful:

```
class ARRAYED_LIST [G]
  feature
    has (v: G): BOOLEAN
      -- Does current include 'v'?

    start
      -- Move cursor to first position if any.

    extend (v: G)
      -- Add 'v' to the end.
```

```
    prune (v: G)
      -- Remove first occurrence of 'v', if any, after cursor position.
      -- Move cursor to right neighbor.

    -- Other features are omitted.
  end
```

```
class
  SET [G]

create
  make_empty

feature {NONE} -- Initialization

  make_empty
    -- Create empty Current.
  do
    create content.make (0)
    content.compare_objects
  ensure
    empty_content: content.is_empty
  end

feature -- Access

  count: INTEGER
    -- Cardinality of the current set.
  do
    Result := content.count
  end

  is_empty: BOOLEAN
    -- Is current set empty?
  do
    .....
    .....
    .....

  end

  has (v: G): BOOLEAN
    -- Does current set contain 'v'?
  require
    .....
    .....
```


do

.....
.....
.....
.....

end

add (*v*: *G*)

-- Add 'v' to the current set.

require

.....
.....

do

.....
.....
.....
.....

ensure

.....
.....
.....

end

remove (*v*: *G*)

-- Remove 'v' from the current set.

require

.....
.....

do

.....

```
.....
.....
.....
ensure
.....
.....
.....
end
duplicate: like Current
  -- Deep copy of Current.
do
  create Result.make_empty
  across content as c
  loop
    Result.add (c.item)
  end
ensure
  same_size: Result.count = count
  same_content: across content as c all Result.has (c.item) end
end
feature -- Set operations.
union (another: like Current): like Current
  -- Union product of the current set and 'another' set.
require
.....
.....
do
.....
.....
.....
.....
.....
.....
```

.....
.....
.....
.....
.....
ensure

.....
.....
end

intersection (*another*: **like Current**): **like Current**
-- Intersection product of the current set and 'another' set.

require

.....
.....
do

.....
.....
ensure

.....
end

difference (*another*: **like Current**): **like Current**

-- Set-theoretic difference of the current set and 'another' set.

require

.....
.....
do

.....
.....
ensure

.....
end

feature -- Set metrics.

jaccard_index (*another*: **like Current**): *REAL_64*

-- Jaccard similarity coefficient between current set and 'another' set.

require

.....
.....
do

```
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
.....  
  
ensure  
  
.....  
.....  
  
end  
  
feature {NONE} -- Implementation  
  
  content: ARRAYED_LIST[G]  
    -- Items of the set.  
  
invariant  
  
  content_exists: content /= Void  
  content_object_comparison: content.object_comparison  
  non_negative_cardinality: count >= 0  
  
end
```

3 Recursion (14 points)

The N-queens problem is the problem of positioning N queens on an $N \times N$ board such that no queen can attack another (i.e., share the same row, column, or diagonal). The N-queens problem can be solved recursively: having a solution for the first 4 rows of the board can be used to build a solution for the 5th row, as is being done in Figure 4.

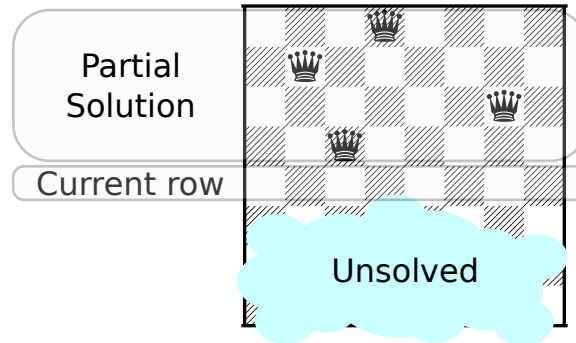


Figure 4: An example of a partial solution

A safe location is one which cannot be attacked by any of the currently placed queens.

A routine to solve the N-queens problem, *complete* (*partial*: *SOLUTION*), does as follows: if the partial solution is not yet complete, then for each safe location in the current row, add the safe location to the solution and use this new solution to solve the problem for the next row. The *current row* is *partial.row_count* + 1; for example in Figure 4 the partial solution has *row_count* equal to 4, thus the current row is 5. If the solution is already complete then it is added to the list of solutions.

You must complete the implementation of *PUZZLE* (which has an attribute *solutions* to store all solutions) below by filling in the body of *complete* and *attack_each_other*. Note that a solution can be added to the list of solutions using the *extend* feature from *LIST*.

```

note
    description: "N-queens puzzle."

class
    PUZZLE

feature -- Access

    size: INTEGER
        -- Size of the board.

    solutions: LIST [SOLUTION]
        -- All solutions found by the last call to 'solve'.

feature -- Basic operations

    solve (n: INTEGER)
        -- Solve the puzzle for 'n' queens.
    require
        solvable: n > 3 -- All puzzles with size > 3 are solvable
    do
    
```

```

size := n
create {LINKED_LIST [SOLUTION]} solutions.make
complete (create {SOLUTION}.make_empty)
ensure
    solutions_exists: not solutions.is_empty
    complete_solutions: across solutions as s all s.item.row_count = n end
end
    
```

feature {*NONE*} -- Implementation

```

complete (partial: SOLUTION)
    -- Find all complete solutions that extend the partial solution 'partial'
    -- and add them to 'solutions'.
    
```

```

require
    partial_exists: partial /= Void
local
    
```

do

```

.....
.....
.....
do
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
    
```

```
.....  
.....  
.....  
.....  
  
end  
  
under_attack (partial: SOLUTION; c: INTEGER): BOOLEAN  
  -- Is column 'c' of the current row under attack  
  -- by any queen already placed in partial solution 'partial'?  
require  
  partial_exists: partial /= Void  
local  
  current_row, row: INTEGER  
do  
  current_row := partial.row_count + 1  
  from  
    row := 1  
  until  
    Result or row > partial.row_count  
  loop  
    Result := attack_each_other (row, partial.column_at (row), current_row, c)  
    row := row + 1  
  end  
end  
  
attack_each_other (row1, col1, row2, col2: INTEGER): BOOLEAN  
  -- Do queens in positions ('row1', 'col1') and ('row2', 'col2') attack each other?  
do  
.....  
.....  
.....  
.....  
.....  
.....  
  
end  
  
end
```



```
note
  description: "Possibly partial solution to the n-queens problem."

class
  SOLUTION

create
  make_empty

feature {NONE} -- Initialization

  make_empty
    -- Create an empty partial solution
    -- (no rows filled)
  do
    ...
  end

feature -- Access

  row_count: INTEGER
    -- Number of rows filled.
  do
    ...
  end

  column_at (r: INTEGER): INTEGER
    -- Column in which a queen is placed in row 'r'.
  require
    valid_row: 1 <= r and r <= row_count
  do
    ...
  end

feature -- Extension

  extended_with (c: INTEGER): SOLUTION
    -- Returns a new solution that is the Current solution with
    -- one more row, in which the queen is in column 'c'.
    -- Current is left unchanged.
  require
    column_positive: c > 0
  do
    ...
  end

end
```