Assignment 2

(due 3 December 2013, 23:59)

Advanced Functional Programming 2013

1 Contracts

`contracts.rkt`, 4 points

The following code snippet contains some lines from a Racket module that implements some operations on directed graphs:

```
(module graph racket
  (require racket/include)
  (include "contracts.rkt")          ; <- This is the file that you should submit.

  (define new                        ; Returns a new, empty directed graph.
  (define (add_vertex digraph a)     ; Adds a vertex with name 'a' to 'digraph'.
  (define (add_edge digraph a b w)   ; Adds an edge from 'a' to 'b' with weight 'w'.
  (define (has_vertex? digraph a)    ; Checks whether 'digraph' has a vertex named 'a'.
  (define (has_edge? digraph a b)    ; Checks whether vertex 'a' has an edge to 'b'.
  (define (out_neighbours digraph a) ; Returns all vertices that 'a' has an edge to.
  (define (weight digraph a b)       ; Returns the weight of the edge from 'a' to 'b'.
)
```

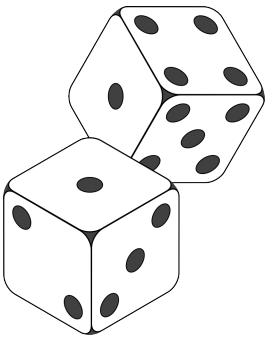
Task

Provide contracts in `contracts.rkt` that check the following properties for the graph module[1]:

- `add_vertex`:
  - The name of a vertex is an integer number.
  - No two vertices have the same name.
- `add_edge`:
  - New edges are added only between existing vertices.
  - Edges have an integer weight.
  - There is at most one edge defined between each ordered pair of vertices.
  - The edge exists in the returned graph.
- `has_edge?`:
  - It is called with existing vertices as arguments.
- `out_neighbours`:
  - It is called with an existing vertex as argument.
  - The result is a list of vertices.
  - There is an edge from the vertex given as argument to each element of the result.
- `weight`:
  - It is called with two existing vertices.
  - There is an edge from the first vertex to the second.

[1] It is **not** required to define the `graph` module, but you may want to use such a module for Task 2.

2 Programmers, too, don't play dice. `dice.erl`, `dice.rkt`, 5 + 5 points



With enough levels of abstraction, every game can be seen as a search for a 'winning' node in a game graph, where edges represent legal moves between game states. If there are no random elements like card shuffling or dice rolling, players can devise strategies that will get them to such a winning node, just by inspecting the graph and the node they are currently at. A similar approach could be applied if the players knew beforehand the order of the cards in a shuffled deck or the result of every dice roll. For the purposes of this puzzle, you are going to play a simple game using such a special dice, for which you already know how it is going to roll.

You are given a game graph whose nodes are labelled 1 to N, with 1 being the starting node and N the only winning node. The graph contains directed edges and may also contain cycles. You are also given a finite sequence of numbers between 1 and 6, which are the results of the dice rolls: the dice will produce every number in this sequence in order, before resuming from the start (the list is 'cyclic' in some sense). To move in the game, you use the result of a dice roll to traverse the corresponding number of edges, starting from your current node and reaching a new node. If you end up in the winning node after a move, you win.

Task

Write two programs, one in Erlang (`dice.erl`) and one in Racket (`dice.rkt`), that return the smallest number of moves that you have to make to reach the winning node. It might also be impossible to reach the winning node no matter how you use the dice rolls, in which case you should return `-1`.

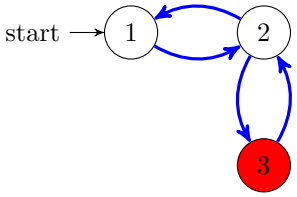
Example

In Figure 1 you can see a small game graph. With the predefined dice rolls `[3,5]` you can win this game in 2 moves:

- 1^{st} move, using the 3: $1 \to 2 \to 1 \to 2$
- 2^{nd} move, using the 5: $2 \to 3 \to 2 \to 3 \to 2 \to 3$

This example is also encoded in the Samples.

If the second move did not end up in the winning node, you should have used the `3` again, then the `5` again etc. until you can safely decide that you can never reach the winning node.



`Dice: [3,5]`

Figure 1: Sample game

Input - Output

The programs take as input an integer N (the number of nodes), a list of node pairs $(NodeA, NodeB)$ (each describing an edge) and a list of numbers between 1 and 6 (representing the results of the dice).

They should return a positive integer, which is the smallest number of moves required to reach the winning node or `-1` if that is not possible.

Hint

If you want a data structure to represent the graph, Erlang has the `digraph` library. Notice that Erlang's `digraph`s are implemented with ETS tables and are therefore **not** purely functional data (for example, modifying a 'copy' of a graph will also modify the original).

Racket unfortunately doesn't have a graph library.

Samples

Here are some sample calls for the two programs:

Erlang

```
1> dice:dice(3, [{1,2}, {2,1}, {2,3}, {3,2}], [3,5]).
2
2> dice:dice(4, [{1,2}, {2,3}, {3,4}], [1]).
3
3> dice:dice(3, [{1,2}, {2,3}], [4,2,6]).
-1
```

Racket

```
> (require dice)
> (dice 3 '((1 2) (2 1) (2 3) (3 2)) '(3 5)).
2
> (dice 4 '((1 2) (2 3) (3 4)) '(1)).
3
> (dice 3 '((1 2) (2 3)) '(4 2 6)).
-1
```

3 Relational Algebra

relations.rkt, 6 points

The relational algebra operates on named relations. A named relation can be seen as a table where each row is a tuple of the relation and each column is identified by a name. In addition all rows are different (i.e., a relation is a set of tuples). You can see two examples of relations in Figure 2.

name	title	department	salary
"John"	"Accountant"	"Finance"	6000
"Rob"	"Salesman"	"Sales"	5000
"Bill"	"Manager"	"Sales"	10000
"Ben"	"Driver"	"Logistics"	4500

department	location	head
"Finance"	"Paris"	"John"
"Sales"	"London"	"Bill"
"It-support"	"Paris"	"Mark"

Figure 2: Sample relations

Task

Implement a language covering a subset of the relational algebra. The language has the following operators (it is up to you to decide whether each of them is a procedure or a macro):

1. `(make-relation (<colname> ...) ((<data> ...) ...))`
   Creates a new relation. The `colname`s are the identifiers of the columns. The data is given as a list of lists, where each inner list is a row. All rows must have the same length. Data can be integers or strings.

2. `(project <rel> (<colname> ...))`
   Returns the projection of the relation `rel` on the given `colname`s. The result is a relation containing only those columns from the original relation. The order of the columns in the new relation is the order of the given `colname`s.

3. `(restrict <rel> <condition>)`
   Restricts the relation `rel` by keeping only the rows that satisfy the condition. The condition can be one of the following:

   - `(<op> <val1> <val2>)`, `op` $\in$ {`>`,`<`,`=`,`<=`,`>=`,`!=`}, and `val1`,`val2` are values or column names. True for a row if the value(s) in the corresponding column(s) of that row make the condition true. The operators `=` and `!=` are used for both integers and strings, while the other four ones are only meaningful for integers.

   - `(or <cond1> <cond2>)`, `(and <cond1> <cond2>)`, `(not <cond>)`
     True for a row when respectively `cond1` or `cond2` is true; `cond1` and `cond2` are true; `cond` is false for that row.

4. `(relation <relname> <reldef>)`
   Binds the identifier `relname` to the relation given by `reldef`.

5. `(equal? <rel1> <rel2>)`
   Returns `#t` or `#f` depending on whether `rel1` and `rel2` are equal. Two relations are equal if and only if they have the same column names and they contain the same tuples. The order of columns matters, but the order of the tuples does not matter; remember that a relation is a set.

You may also want to define a printing operator (e.g. `(show <rel>)`), but you are not required to do so.

Sample

Here is an example of a program in our language using the relations shown in Figure 2.

```
#lang s-exp "relations.rkt"
(relation pers (make-relation
                (name title department salary)
                (("John" "Accountant" "Finance" 3000)
                 ("Rob" "Salesman" "Sales" 5000)
                 ("Bill" "Manager" "Sales" 10000)
                 ("Ben" "Driver" "Logistics"  4500))))
(relation dept (make-relation
                (department location head)
                (("Finance" "Paris" "John")
                 ("Sales" "London" "Bill")
                 ("It-support" "Paris" "Mark"))))
(show (project (restrict pers (salary . >= . 5000)) (name title)))
(equal? (project dept (location))
        (make-relation (location)
                       (("Paris") ("London"))))
```

If we run the program the output can be the following (for a simple definition of `show`):

```
((name title)
 (("Rob" "Salesman")
  ("Bill" "Manager")))
#t
```

Note that the output of your `show` operator can be different. That operator will not be tested and is included only for your convenience. On the contrary, the `equal?` operator will be used for testing purposes.

Submission instructions

- Each student must send their **own individual submission**.
- For this assignment you must submit a single `afp13_assignment2.zip` file at the relevant section in Studentportalen.
- `afp13_assignment2.zip` should contain 5 files (without any directory structure):
  - the 4 programs requested (`relations.rkt`, `contracts.rkt`, `dice.erl`, `dice.rkt`) that should conform to the specified interfaces regarding exported functions, handling of input and format of output.
  - a text file named `README.txt` whose first line should be your name. You can include any other comments about your solutions in this file.

Have fun!