Assignment 3

Advanced Functional Programming, 2019
(Avancerad funktionell programmering, 2019)

due 23 December 2019, 23:59

1 Stack permutations (`perm.rkt`, 3 points)

Implement the stack permutation algorithm from the previous assignments in Racket. The same requirements and grading as in assignment 2 apply, with the exception that you do not need to supply any tests or write a report. (Of course, it is still a good idea to test your program; perhaps with the same/similar tests that you used in previous assignments.)

Task

Write a Racket module (`perm.rkt`) that returns `#t` if the input list is a stack permutation of the output list, or `#f` otherwise. Your code needs to be efficient, it should be able to handle lists of up to 1 million elements in under 1 second.

Input / Output

The input has a similar format to the one in the Haskell assignment, but this time we will be using Racket lists. Similarly, the output should be Racket boolean values, `#t` or `#f`.

Samples

Here are some sample calls for the program:

```
> (require "perm.rkt")
> (perm '(1 2 3) '(3 1 2))
#t
> (perm '(1 2 3) '(2 3 1))
#f
```

2 Graph k-coloring (`kcolor.rkt`, 4 points)

Implement the k-coloring algorithm from the previous assignments, this time in Racket. You do not need to supply any tests.

Task

Much like in the previous assignments, you are asked to create a graph implementation and use it to solve the k-coloring problem. This time you do not need to hand in any tests for your graph or your algorithm. However, it is still recommended to have a well-defined interface for your graph implementation. Remember that even though the algorithm implemented is in NP and there is no strict time limit requirement, you should still try and write code that is reasonably efficient.

Input / Output

The input has a format similar to the one in the Haskell assignment, but this time we will be using Racket lists. Similarly, the output should contain a list of lists, or `#f` if no k-coloring can be found.

Samples

Here are some sample calls for the program:

```
> (require "kcolor.rkt")
> (kcolor '((1 (2 3)) (2 (1 3)) (3 (1 2))) 3)
'((1 a) (2 b) (3 c))
> (kcolor '((1 (2 3)) (2 (1 3)) (3 (1 2))) 2)
#f
```

3 Contracts (`contracts.rkt`, 3 points)

The following code snippet, where `...` marks the body of each definition, contains some lines from a Racket module that implements some operations on directed graphs with weights on their edges:

```
(module graph racket
 (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.
)
```

The graph could be implemented in various ways, e.g., by a functional implementation where `add_vertex` and `add_edge` return a *new* graph (leaving the old unchanged), or by imperative one that mutates the graph in-place (and returns this graph). However, *your contract definition must not assume any particular implementation!*

Task

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

For `add_vertex`:

- The name of a vertex is an integer number.
- No two vertices have the same name.

For `add_edge`:

- New edges are added only between existing vertices.
- Edges have an integer weight.
- There is at most one edge between each ordered pair of vertices.
- The edge exists in the resulting graph.

For `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.

[1] It is *not* required to define the `graph` module for this task, but you may want to have one for testing your contracts.

4 Relational Algebra (`relations.rkt`, 5 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 1.

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 1: Two relational tables (pers and dept).

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):

`(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.

`(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.

`(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.

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

`(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 1.

```
#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.

5 Reflection (`reflection.pdf`, 5 points)

In this course, we have examined three different functional programming languages: Erlang, Haskell, and Racket. This final exercise asks you to take a moment and reflect on these languages, what you have learned from them and write your thoughts in a comprehensive way. There are no "right" or "wrong" answers in this exercise, but your grade will be determined by how thorough (obs: not long!) and clear your answers are. (Remember that providing concrete examples help in making arguments clear.)

- What is your overall opinion of each language? (Perhaps taking into account also other languages, functional or not, that you may know.) Which of their characteristics do you find interesting/uninteresting and why?
- How easy/difficult was to familiarize yourself with each of these languages (in case you did not know them beforehand)? Which features that were similar to previous languages you knew made them easy/difficult to grasp "the philosophy" of the language?
- For each language, find two defining, positive features (pros) that separate it from the others and briefly describe them. Describe also why you chose these features and what they can be used for. Give at least one use case example for each feature.
- For each language, find one negative feature (con), explain why it is a negative, and explain how the other languages do it better, including some examples.
- Which of the three languages is your favorite and why? This question is purely subjective but make sure to give concrete reasons and examples.
- What would you have designed differently for (one/some of) these languages and what consequences would that have for them? What can each of these languages learn from the others?

Submission instructions

- Each student must send their **own individual submission**.
- For this assignment you must submit a single `afp_assignment3.zip` file at the relevant section in Studentportalen.
- `afp_assignment3.zip` should contain six files (without any directory structure):
  - the four programs requested (`perm.rkt`, `kcolor.rkt`, `contracts.rkt`, and `relations.rkt`) that should conform to the specified interfaces regarding exported functions, handling of input and format of output.
  - the PDF containing your answers to the reflection questions `reflection.pdf`.
  - 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!