Walter Cazzola

Home Page
ADAPT Lab.
Curriculum Vitae
Research Topic

Didactics

LP 2014-15
PA 2014-15
TSP 2014-15

PhD: ADT Curricula

LP 2013-14
PA 2013-14
TSP 2013-15

Thesis Proposals

How to Change Degree Program

Publications

Funded Projects

Research Projects

Related Events

Exam of Programming Languages

29 January 2015

Disclaimer. Note that to have a running solution for an exercise is not enough: you need a well-cooked solution that proves your ability to use what explained during the classes. All the exercises have the same value: 11; Exercise are valued separately and it is necessary to get at least 6 points each exercise to pass the exams, i.e., 18 achieved with only 2 exercises means to fail the exam.

Exercise OCaML/ML: Playing with Numbers.

Number theory taught us that the natural numbers can be expressed through a symbol and the successive operator as in 0, succ(0), succ(succ(0)), ...

After that consideration it is possible to build an abstract data type implementing the natural numbers algebra with its basic operations (+, -, * and /) without using **any** pre-implemented arithmetic.

The exercise consists of implementing such an abstract data type as a module compliant to the following interface

```
module type NaturalI =
  sig
    type natural
    exception NegativeNumber
    exception DivisionByZero

    val ( + ) : natural -> natural -> natural
    val ( - ) : natural -> natural -> natural
    val ( * ) : natural -> natural -> natural
    val ( / ) : natural -> natural -> natural

    val eval : natural -> int
    val convert : int -> natural
  end;;
```

Please note that

- +, –, * and / have the usual meaning except they work on natural numbers, so they can't return neither a negative number (an exception should be raised) nor a real number (the division returns the result trunked to the integral part)
- the implementation for +, –, * and / can't use any traditional arithmetic operator
- `convert` and `eval` connect your natural numbers with the integers provided by OCaML, i.e., the former converts an integer into your internal representation and the later does the vice versa.
- probably you will have to implement more operators (e.g., something to compare your numbers could be useful to implement the other operators)

To be an acceptable solution your module should compile as follows:

```
[10:37]cazzola@surtur:~/lp/ocaml>ocamlc -c NaturalI.mli
[10:37]cazzola@surtur:~/lp/ocaml>ocamlc -c «file».ml (* This is your file *)
[10:37]cazzola@surtur:~/lp/ocaml>ocamlc -o main «file».cmo main.ml
```

where the `main.ml` file is:

```
open Natural.N ;;

let main () =
   Printf.printf "  7 +  18 :- %3d\n" (eval ((convert   7) + (convert  18)) ) ;
   Printf.printf "125 + 252 :- %3d\n" (eval ((convert 125) + (convert 252)) ) ;
   Printf.printf " 25 -  18 :- %3d\n" (eval ((convert  25) - (convert  18)) ) ;
   try
     Printf.printf " 18 -  25 :- %3d\n" (eval ((convert  18) - (convert  25)) )
   with NegativeNumber -> Printf.printf "Exception: NegativeNumber\n" ;
   Printf.printf "  5 *   3 :- %3d\n" (eval ((convert   5) * (convert   3)) ) ;
   Printf.printf " 25 *   7 :- %3d\n" (eval ((convert  25) * (convert   7)) ) ;
   Printf.printf "125 /  25 :- %3d\n" (eval ((convert 125) / (convert  25)) ) ;
   Printf.printf "125 /  23 :- %3d\n" (eval ((convert 125) / (convert  23)) ) ;
   try
     Printf.printf "  1 /   0 :- %3d\n" (eval ((convert   1) / (convert   0)) )
   with DivisionByZero -> Printf.printf "Exception: DivisionByZero\n" ;
   Printf.printf "  1 /   2 :- %3d\n" (eval ((convert   1) / (convert   2)) ) ;
   Printf.printf "  0 / 100 :- %3d\n" (eval ((convert   0) / (convert 100)) ) ;;
```

```
let() = main() ;;
```

The following is the expected behavior:

```
[10:42]cazzola@surtur:~/lp/ocaml>main
  7 +  18 :-  25
125 + 252 :- 377
 25 -  18 :-   7
Exception: NegativeNumber
  5 *   3 :-  15
 25 *   7 :- 175
125 /  25 :-   5
125 /  23 :-   5
Exception: DivisionByZero
  1 /   2 :-   0
  0 / 100 :-   0
```

Exercise Erlang: Distributed Combinatorics.

With the expression **permutation of m possible values took n by n with repetitions** we refer to a tuple of n elements extracted from the m available (considering that the extracted value is not removed from the extractable set of values). Particularly interesting is the problem of generating all the possible permutations where the generated tuple set contains **all** and **only** the possible permutations. With a sequential language it is a pretty easy exercises even if it can get slow on the increasing of n and m.

This exercise consists in implementing a distributed version of the permutations generator whose master/slave architecture is composed of one master and n slaves. We have a slave (module `generator`) for each element of the tuple and if you consider the permutation set as a table each slave is in charge of generating a column of the table. All slaves are identical. The master (module `combinator`) will spawn the slaves, collects the data and coalesce the collected data in the permutations tuple. The only function the master exports is `start` that receives the n and m information and then starts the whole computation.

Note that

- the values are the numbers from 1 to m,
- the total number of permutations is m^n and
- the output should be lexicographically ordered

The following is an example of the expected behavior:

```
[17:53]cazzola@surtur:~/lp/erlang>erl
3> combinator:start(3,3).
true
1, 1, 1
1, 1, 2
1, 1, 3
1, 2, 1
1, 2, 2
1, 2, 3
1, 3, 1
1, 3, 2
1, 3, 3
2, 1, 1
2, 1, 2
2, 1, 3
2, 2, 1
2, 2, 2
2, 2, 3
2, 3, 1
2, 3, 2
2, 3, 3
3, 1, 1
3, 1, 2
3, 1, 3
3, 2, 1
3, 2, 2
3, 2, 3
3, 3, 1
3, 3, 2
3, 3, 3
4> combinator:start(3,2).
1, 1, 1
1, 1, 2
1, 2, 1
1, 2, 2
2, 1, 1
2, 1, 2
```

```
2, 2, 1
2, 2, 2
5> combinator:start(2,3).
1, 1
1, 2
1, 3
2, 1
2, 2
2, 3
3, 1
3, 2
3, 3
6> combinator:start(2,2)
1, 1
1, 2
2, 1
2, 2
```

All the solutions not conform to the specification will be considered wrong.

Tables' Pretty Printing.

A comma-separated values (CSV) file stores tabular data (numbers and text) in plain-text form. Plain text means that the file is a sequence of characters, with no data that has to be interpreted as binary numbers. A CSV file consists of any number of records, separated by line breaks of some kind; each record consists of fields, separated by a comma. All records have an identical sequence of fields.

CSV is not a single, well-defined format. Rather, in practice the term CSV refers to any file that

1. is plain text using a character set such as ASCII or Unicode,
2. consists of records (one record per line),
3. with the records divided into fields separated by commas, any character can be part of a field including the comma in that case the whole field is quoted
4. where every record has the same sequence of fields,
5. often the first record has the special role of header and describes the various data

Basically the data contained in a CSV file are tables.

The exercise consists of writing a program that uses parser combinators to parse and translate a pool of CSV files. The content of each CSV file must be pretty printed as an ASCII table with the following characteristics:

- each column of the table has a fixed size large as the largest element in the field plus two white spaces
- each column is separated from the next by a "|" symbol
- data inside columns are left-aligned
- a "|" symbol opens and closes every row of the table
- the header is printed as any other row of the file apart that it is separated from the rest of the table by an horizontal row made by "-" symbols.
- an horizontal row made by "-" symbols opens and closes the table
- the horizontal rows have all the same lenght given by the size of each column plus the column separators.

The following is an example of expected behavior (the test files are books.csv and languages.csv)

```
[20:31]cazzola@surtur:~/lp/scala>scala CSVParserCLI books.csv languages.csv
-------------------------------------------------------------------------------
| Author 1     | Author 2  | Title                | Year | Publisher           |
-------------------------------------------------------------------------------
| J. Hickey    |           | Introduction to OCaML | 2007 | Cambridge  Press    |
| J. Armstrong |           | Programming Erlang    | 2011 | Pragmatic Bookshelf |
| D. Wampler   | A. Payne  | Programming Scala     | 2009 | O'Really            |
| M. Lutz      |           | Learning Python       | 2007 | O'Reilly            |
| M. Pilgrim   |           | Dive into Python 3    | 2009 | Apress*             |
| R. Laddad    |           | AspectJ in Action     | 2003 | Manning             |
-------------------------------------------------------------------------------


-------------------------------------------------------------------------
| Language | Inventors                                    | Year | Version |
-------------------------------------------------------------------------
| Python   | Guido Van Rossum                             | 1989 | 3.4.2   |
| Java     | J. Gosling, M. Sheridan & P. Naughton        | 1995 | 8.0     |
| Erlang   | J. Armstrong, R. Virding & M. Williams       | 1986 | 5.10.4  |
| Scala    | Martin Odersky                               | 2001 | 2.11.2  |
| OCaML    | X. Leroy, J. Vouillon, D. Doligez & D. Rémy  | 1996 | 4.02.1  |
| AspectJ  | Gregor Kiczales                              | 2001 | 1.8     |
| Fortran  | John W. Backus                               | 1957 | 2008    |
-------------------------------------------------------------------------
```