Walter Cazzola

Home Page
ADAPT Lab.
Curriculum Vitae
Research Topic

Didactics

Publications

Funded Projects

Research Projects

Related Events

Exam of Programming Languages

18 February 2016

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 ML/OCaML: Expressions Solved Step by Step.

One of the exercises you had to learn in your childhood was to solve arithmetic expressions and the easiest way to learn that was to solve them step by step starting from the innermost sub-expressions and proceeding by reducing the complexity of the expression. E.g., ((4+5)*2) → (9*2) → 18.

ML, as well as many other programming languages, can solve any kind of arithmetic expression even the most complicated but it produces immediately the result without showing the intermediate steps.

You have to define a function `print_evaluation` that can parse an expression (contained in a string) and solve it step by step (printing all the intermediate results) as in:

```
open Reduction.ArithExpr;;

let main() =
  let expressions = ["+34"; "+3-15"; "*+34-23"; "+7++34+23";
      "*+*34-34/6-35"; "/+-81*45*/93/52"; "*+/12/14-2/32"] in
  List.iter print_evaluation expressions ;;

main();;
```

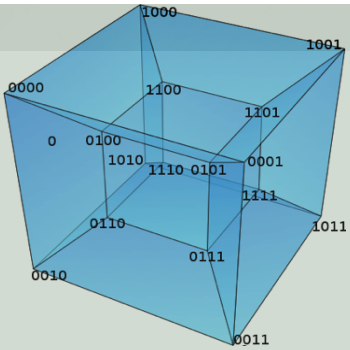
```
[16:18]cazzola@hymir:~/lp/ocaml> main
( 3. + 4. )
7.
( 3. + ( 1. - 5. ) )
( 3. + -4. )
-1.
( ( 3. + 4. ) * ( 2. - 3. ) )
( 7. * -1. )
-7.
( 7. + ( ( 3. + 4. ) + ( 2. + 3. ) ) )
( 7. + ( 7. + 5. ) )
( 7. + 12. )
19.
( ( ( 3. * 4. ) + ( 3. - 4. ) ) * ( 6. / ( 3. - 5. ) ) )
( ( 12. + -1. ) * ( 6. / -2. ) )
( 11. * -3. )
-33.
( ( ( 8. - 1. ) + ( 4. * 5. ) ) / ( ( 9. / 3. ) * ( 5. / 2. ) ) )
( ( 7. + 20. ) / ( 3. * 2.5 ) )
( 27. / 7.5 )
3.6
( ( ( 1. / 2. ) + ( 1. / 4. ) ) * ( 2. - ( 3. / 2. ) ) )
( ( 0.5 + 0.25 ) * ( 2. - 1.5 ) )
( 0.75 * 0.5 )
0.375
```

The only admitted operators are `+`, `-`, `*` and `/` with the traditional meaning. The only available type is float. To simplify the parsing, the expressions are written in polish notation and numbers in the initial expression are only 1-figure integers.

Note Polish notation is a form of notation for arithmetic that places operators to the left of their operands. If the arity of the operators is fixed, the result is a syntax lacking parentheses or other brackets that can still be parsed without ambiguity.

Exercise Erlang: Hamiltonian Path on an Hypercube.

An hypercube is a particular 4-dimensional geometrical figure (look at the picture) where each vertex (of the 16 available vertexes) is always connected to other 4 vertexes.



This configuration can be used to configure the elements in a distributed system balancing the number of connections each node owns. If you number the nodes by using the Grey's code (the Grey's code is a binary numeral system where two successive values differ in only one bit, e.g., 000 → 001 → 011 → 010 → 110 → 111 → 101 → 100) it admits an Hamiltonian path, i.e., it admits a path that connects all nodes and each node is traversed only once.

Given that, you have to build up your hypercube. This is made by 16 processes each labeled with one of the 16 Grey's code and each can communicate only with its four neighbors according to the picture. **Note that the processes are not registered (apart the first one) and they needs the PID of the neighbors to communicate.**

Once the system is up and configured (through the function `create/0` in module `hypercube`) exploits the Hamiltonian path (function `hamiltonian/2`): send a message to the system and let it flow through all nodes without repetitions. The `hamiltonian` function takes two parameters, the first is the message and the second is the path we want to check as Hamiltonian; the path is a list of labels after the Gray's code labels. We get the proof that the given path is Hamiltonian by piggybacking the message with the labels of the processes it passes through. At the end the piggybacked message is returned to the process invoking the function that prints it showing the result. If it is Hamiltonian no label should appear twice. **Note that not all the possible permutations on the Gray's code give a feasible path (this is due to the limited number of connections each nodes has).**

The following is an example of the expected behavior:

```
1> hypercube:create().
The process labeled "0000" just started
The process labeled "0001" just started
The process labeled "0011" just started
The process labeled "0010" just started
The process labeled "0110" just started
The process labeled "0111" just started
The process labeled "0101" just started
The process labeled "0100" just started
The process labeled "1100" just started
The process labeled "1101" just started
The process labeled "1111" just started
The process labeled "1110" just started
The process labeled "1010" just started
The process labeled "1011" just started
The process labeled "1001" just started
The process labeled "1000" just started
true
2> hypercube:hamilton("Hello", hypercube:gray(4)).
{msg,
 {src,"1000",msg,
  {src,"1001",msg,
   {src,"1011",msg,
    {src,"1010",msg,
     {src,"1110",msg,
      {src,"1111",msg,
       {src,"1101",msg,
        {src,"1100",msg,
         {src,"0100",msg,
          {src,"0101",msg,
           {src,"0111",msg,
            {src,"0110",msg,
             {src,"0010",msg,
              {src,"0011",msg,
               {src,"0001",msg,{src,"0000",msg,"Hello"}}}}}}}}}}}}}}}}
```

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

Exercise Scala: Expressions Solved Step by Step.

One of the exercises you had to learn in your childhood was to solve arithmetic expressions and the easiest way to learn that was to solve them step by step starting from the innermost sub-expressions

and proceeding by reducing the complexity of the expression. E.g., ((4+5)*2) → (9*2) → 18.

Scala, as well as many other programming languages, can solve any kind of arithmetic expression even the most complicated but it produces immediately the result without showing the intermediate steps.

You have to define a new arithmetic parse (`ArithmeticParser`) that can parse an expression (contained in a string) and solve it step by step (printing all the intermediate results) as in:

```
[16:34]cazzola@hymir:~/lp/scala>scala StepByStepEvaluator
       "((2 + 7) + ((3 + 9) + 4))"
       "((1 * 7) + (7 * ((3 + 9) + 5)))"
       "((5*(7-2))+(((15/3)+2)-((1422*2)-(3500/4))))"
((2 + 7) + ((3 + 9) + 4))
(9 + (12 + 4))
(9 + 16)
25

((1 * 7) + (7 * ((3 + 9) + 5)))
(7 + (7 * (12 + 5)))
(7 + (7 * 17))
(7 + 119)
126

((5 * (7 - 2)) + (((15 / 3) + 2) - ((1422 * 2) - (3500 / 4))))
((5 * 5) + ((5 + 2) - (2844 - 875)))
(25 + (7 - 1969))
(25 + -1962)
-1937
```

The only admitted operators are `+`, `-`, `*` and `/` with the traditional meaning. The only available type is integer (the division is a division among integers with an integer result). To simplify the implementation, all sub expressions are closed in parenthesis and all the operators are binary.