Module 0
Getting Started

Compilers and Execution Environments (ID2202)
7.5 hp

David Broman
KTH Royal Institute of Technology

Friday 29th October, 2021

Abstract

Welcome to Module 0 of the course! This is a pre-module, given before the actual course modules start. Hence, this should be seen as a getting starting module, which gets you up to speed in programming in a typed functional language called OCaml.

The assignments in module 0 are all optional, but highly recommended to complete during the first week of the course.

Contents

1 Typed Functional Programming in OCaml

The goals of the tasks in this module is to practice some fundamental aspects of typed functional programming in OCaml.

1.1 Prime numbers

In this exercise, you will create a few small functions and operate on lists.

Recommended Reading

Before you start, we recommend that you

- listen to all videos on the Canvas page about OCaml and try out all examples yourself.
- go through the slides from lectures 2 and 3
- take a look at `https://ocaml.org/`

Task

Your task is to create a program that includes two functions:

- Function 1: `primes : int -> int list` which takes as argument the number of prime numbers that should be generated, and returns a list of these prime numbers. For instance, a call `primes 5` should return a list `[2; 3; 5; 7; 11]`.
- Function 2: `pretty : int list -> string` takes as input a list of integers, and returns a string with a comma separated list. For instance, if used together with function `primes`, then the expression `pretty (primes 7)` should return a string value `"2,3,5,7,11,13,17"`. Note that no spaces should be added and it is not allowed to have a comma as the last character. No newlines are allowed in the string.

For the automatic grader to work, the solution must be stored in your Git repository in folder `solutions/module0/primes/` and the generated executable must be called `primes` and stored in this folder. When the program is invoked, the program argument should state the number of primes that should be pretty printed to the standard output. For instance, to generate 7 prime numbers, the program is invoked with command line `./primes 7`. Note that none of the functions above are allowed to use any side effects (e.g., you are not allowed to use `for` or `while` loops). Using recursion or functions such as `map`, `fold_left` and `concat` are highly recommended. The program must return the following return codes:

- Return code 0 if the input argument value is between 1 and 100. In such a case, the prime numbers should be printed to standard output.

  exit 0

  use the "pretty"
- Return code 1 if the program was started with a program argument that is a well-defined positive integer number, but outside the allowed range (1 to 100).
- Return code 2 if the number of program arguments are not exactly one, or if the program argument is not an integer.

Grading tag

Use the following tag when submitting this exercise: `#module0-primes`

1.2 Binary Trees

In this exercise, you will use algebraic data types, and operate over data structures. You will also perform some simple manual string parsing.

Recommended Reading

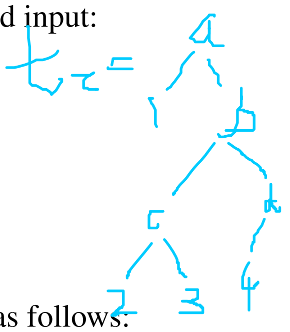
- refresh your knowledge about binary trees by watching the course videos
- you may also take a quick look at the Wikipedia pages for `Binary tree` and `Tree traversal`

Task

In this task, you will read in a binary tree from standard input, execute a command (depending on the program argument) and output the result to standard output.

The binary tree is given in pre-order as part of the standard input, where each node is given on a separate line. Each node is marked as either a `Leaf` or as a `Node`. The nodes always have two children, (either a leaf or another node), and a leaf has no children. Each leaf holds an integer value and each node holds a string value. The implementation should use an algebraic data type when defining the type of the binary tree.

The first step of the program is to read the data from the standard input and to construct the internal binary tree. For instance, suppose that the following is given as standard input:

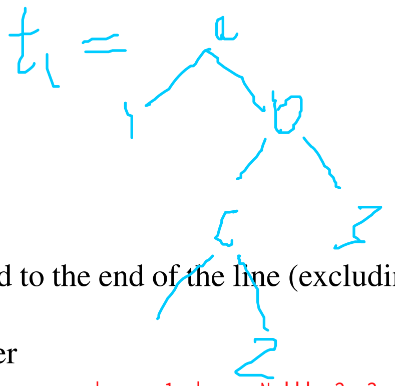


```
Node:this text
Node:why
Leaf:13
Leaf:4
Leaf:8
```

The above input string corresponds to a binary tree, which could be illustrated as follows:

```
      "this text"
       /      \
    "why"      8
     /  \
    13   4
```

pre-order:t, w, 13, 4, 8
in-order:13, w, 4, t, 8
post-order:



Note how the strings are parsed from the colon `:` character and to the end of the line (excluding end-of-line character).

The solution must be stored in your Git repository in folder

```
solutions/module0/binary-tree/
```

pre-order:a, 1, b, c, Nulll, 2, 3
in-order:1, a, Nulll, c, 2, b, 3
post-order:

and the generated executable must be called `binary-tree` and stored in this folder. Only the first program argument is used, and it states what the program should do with the tree. For instance, the program can be invoked as follows:

```
./binary-tree pre-order
```

The following list gives the required commands:

- Command `pre-order`. Pretty prints the parsed tree in pre-order format. If the implementation is correct, it should give the same result as what is given as standard input.
- Command `in-order`. Pretty prints the tree in in-order format. The output of the above tree would be:

  ```
  Leaf:13
  Node:why
  Leaf:4
  Node:this text
  Leaf:8
  ```

- Command `post-order`. Pretty prints the tree in post-order format. The output of the above tree would be:

  ```
  Leaf:13
  Leaf:4
  Node:why
  Leaf:8
  Node:this text
  ```

- Command `list`. The task is to create a function that traverses the tree and returns a list of integers that represents all integer values of the leaves, when visited using any of the above traversal orders (note that the order of visited leafs will be the same). The list should be pretty printed as a comma separate list (please reuse the function from the previous exercise). Note that you should use an accumulator when you traverse the tree and when you create the list. As a consequence, the returned list will be in reverse order, but it should be printed in the same order as the tree was traversed.
- Command `size`. Prints out to standard output the total number of nodes and leafs of the tree. In the example above, the printed value would be `5`.
- Command `depth`. Prints out to standard output the maximal depth of the tree. In the above example, the printed value would be `2`.

必须输入参数，且只允许输入一个参数

Return code 0 should be returned if one of the above commands is given and the standard input can be parsed correctly. Return code 1 should be given if there is an error during parsing, if an empty binary tree is supplied, if the argument is unknown, or if not exactly one program argument is supplied.

Grading tag

Use the following tag when submitting this exercise: `#module0-binary-tree`