Assignment 3
COMP 302 Programming Languages and Paradigms

Brigitte Pientka
MCGILL UNIVERSITY: School of Computer Science

Due Date: 31 Oct, 2013

Your homework is due at the beginning of class on 31 Oct, 2013. All code files must be submitted electronically, and your program must compile. There are three directories, `backtrack`, `memo`, and `metric`. In each of the directories, you will find a file `sources.ml` which reads in multiple files. You can simply type `#use ''sources.ml'';;` in the interactive OCaml interpreter and it will load all the files contained in the file.

Q1. 20 points In September 1999, NASA lost a $125 million Mars orbiter because one engineering team used metric units while another used English units for a key spacecraft operation - a very costly mistake which could easily have been avoided using a language with a strong type and module system.

In this question, we are using OCaml's module system to do better and separate different units clearly from each other while still sharing code. In the directory `metric`, we give a simple module type `METRIC` for representing a metric in the file `metric.ml`.

```
module type METRIC =
sig
  type t
  val unit : t
  val plus : t -> t -> t
  val prod : t -> t -> t
  val toString : t -> string
  val toFloat : t -> float
  val fromFloat : float -> t
end;;
```

Q1.1 5 points Your task is to provide a module `Float` which provides an implementation for the type `METRIC`. We then want to use the module `Float` to create different representations for `Meter`, `KM`, `Feet`, `Miles`, `Celsius`, `Fahrenheit`, and `Hour`.

Q1.2 5 points Next, define a functor `Speed` which implements the module type `SPEED`. We want to be able to compute the speed km per hour as well as miles per hour. Hence, the functor Speed must be parameterized.

Q1.3 2 points Show how to use the functor `Speed` to obtain an implementation for computing miles per hour in the module `MilesPerHour` and and implementation computing kilometers per hour in the module `KMPerHour`.

Q1.4 8 points It is also often useful to convert between different metrics.

Define a module type `CONVERSION` which specifies the following conversion functions:

`feet2meter`	meter = feet * 0.3048
`fahrenheit2celsius`	celsius = (fahrenheit - 32) / 1.8
`miles2KM`	km = miles * 1.60934
`MilesPerHour2KMPerHour`	

Then implement a module which provides these conversion functions.

Q2 35 points Parsing using exceptions.

In this question, we are implementing a parser for a simple language of arithmetic expressions with +,*, and ()'s. The `n` represents an integer and is a terminal symbol. Top-level start symbol for this grammar is `E`.

Final Expression	E	::=	S ;
S-Expression (expression with plus)	S	::=	P + S | P
P-Expression (expression with times)	P	::=	A * P | A
A-expression (atomic expression)	A	::=	n | (S)

The goal is to implement the parser using exceptions.

The fundamental goal of a parser is to take a string and produce an abstract syntax tree. We have implemented a simple lexer which translates a string into a list of tokens (see `lexer.ml`). This kind of pre-processing will make it easier for the actual parser to generate an abstract syntax tree from the list of tokens. For example, given the string "(2 + 3) * 4;" the function `Lexer.lex` produces a list of tokens as follows:

```
# Lexer.lex "(2␣+␣3)␣*␣4;";;
- : Lexer.token list =
[Lexer.LPAREN; Lexer.INT 2; Lexer.PLUS; Lexer.INT 3; Lexer.RPAREN;
 Lexer.TIMES; Lexer.INT 4; Lexer.SEMICOLON]
#
```

The parser when given a string `''(2 + 3)* 4;''` then first calls the lexer to translate it into a list of tokens and subsequently translates the token list into an abstract syntax tree.

```
# Parser.parse "(2␣+␣3)␣*␣4;";;
- : Parser.exp =
Parser.Prod (Parser.Sum (Parser.Int 2, Parser.Int 3), Parser.Int 4)
#
```

We also implemented a simple evaluator for evaluating artithmetic expressions (see `eval.ml`). The pipeline for evaluating a string denoting an expression is as follws: passes the string denoting the expression to the lexer to obtain a list of tkens; then pass thelist of tokens to the parser to obtain an abstract syntax tree denoting the expression; finally, pass the abstract syntax tree to the evaluator to compute the final result.

```
# Eval.eval "(2␣+␣3)␣*␣4;";;
- : int = 20
#
```

In the directory `backtrack` we have implemented the lexer (see file `lexer.ml`) and parts of the parser for the given grammar in the file `parser.ml` with the corresponding signature in `parser.mli`. Your task is to complete the parser in `parser.ml`. We use *three different exceptions to describe success of parsing a (sub)-expression.*

- **exception** `SumExpr` **of** `exp * L.token list` :
  `SumExpr (s, toklist')` should be raised to indicate that we successfully parsed a list of tokens called `toklist` into an S-Expression `s` and a remaining list of tokens called `toklist'`.
  `toklist'` is what remains from `toklist` after we peeled of all the tokens necessary to build the S-Expression `s`.
- **exception** `ProdExpr` **of** `exp * L.token list`:
  `ProdExpr (p, toklist')` should be raised to indicate that we successfully parsed a list of tokens called `toklist` into an P-Expression `p` and a remaining list of tokens called `toklist'`.
  `toklist'` is what remains from `toklist` after we peeled of all the tokens necessary to build the P-Expression `p`.
- **exception** `AtomicExpr` **of** `exp * L.token list`:
  `AtomExpr (a, toklist')` should be raised to indicate that we successfully parsed a list of tokens called `toklist` into an A-Expression `a` and a remaining list of tokens called `toklist'`.
  `toklist'` is what remains from `toklist` after we peeled of all the tokens necessary to build the A-Expression `a`.

Implement the functions `parseSumExp`, `parseProdExp`, and `parseAtom` according to the grammar rules given above in the file `parser.ml`.

Q 35 points Memoization

In the absence of effects, a function will always evaluate to the same value when applied to the same arguments. Therefore, applying a particular function to the same arguments more than once will often result in needless work. Memoization is a simple optimization that helps to avoid this inefficiency. The idea is that you equip a function with some data structure that maps the arguments that the function has been called on to the results produced. Then, whenever the function is applied

to any arguments, you first check to see if it has been applied to those arguments previously: if it has, the cached result is used instead of computing a new one; if it hasnt, the computation is actually performed and the result is cached before being returned.

If you think of a graph of a function as a set of (input, output) pairs, rather than a doodle on a piece of paper representing such a set, this mapping is really storing the subset of the graph of its associated function that has been revealed so far. The optimization should let us compute each (input, output) pair in the graph exactly once and refer to the already discovered graph for inputs we need more than once.

In this question, we will use a dictionary (see `dict.ml`) to store (input,output) pairs.

Case Study: Fibonacci We will first work through implementing this idea using the familiar Fibonacci function as a case study. Recall the nave implementation of the Fibonacci sequence, provided in `fib.ml`.

```
module type FIBO =
sig
  (* on input n, computes the nth Fibonacci number *)
  val fib : int -> Big_int.big_int
end

module Fibo : FIBO =
struct
  let rec fib n = match n with
    | 0 -> Big_int.big_int_of_int 0
    | 1 -> Big_int.big_int_of_int 1
    | _ -> Big_int.add_big_int (fib(n-2)) (fib(n-1))
end
```

Note we are taking advantage of the OCaml library for big integers. You will need to first load the library containing big integers by typing into the OCaml interactive interpreter

```
# #load "nums.cma";;
```

15 points Finish the `MemoedFibo` functor in `memfib.ml` by writing a memoized version of Fibonacci.

You should represent the (input,output) mapping using a reference containing a persistent dictionary of type `D.dict`, where `D` is the argument to `MemoedFibo`. The mapping should be shared between all calls to `MemoedFibo.fib`, so that results are reused between multiple top-level calls. But note, that we do not expose this dictionary to the outside.

If you dont know where to start, one good strategy is to use a pair of mutually recursive functions: make one function in the pair the `fib` function required by the signature; make the other function responsible for checking and updating the mapping. The benefit to this strategy is that it lets you separate memoizing from the function being memoized.

3 points Instead of hand-rolling a new version of every function that wed like to memoize, it would be nice to have a higher order function that produces a memoized version of any function. A totally reasonablebut wrongfirst attempt at writing such an automatic memoizer as shown below and can be found in the file `memo.ml`.

```
module PoorMemoizer (D:DICT) : (POORMEMOIZER with type key = D.Key.t) =
struct

  type key = D.Key.t

  let memo f =
    let hist : 'a D.dict ref = ref D.empty in
    let rec f_memoed x = match D.lookup (!hist) x with
      | Some b -> b
      | None ->
          let res = f x in
            (hist := D.insert (!hist) (x,res);
             res)
    in
      f_memoed

end;;

module PoorAutoMemoFibo : FIBO =
struct
  module PM = PoorMemoizer (ID)
  let fib = PM.memo Fibo.fib
end

module PMF = PoorAutoMemoFibo
```

What is wrong with this code? For example, apply the functor and use it to memoize an implementation of Fibonacci. You should observe that it is much slower than the hand-rolled version you wrote. Why?

Note: It is useful to actually look at the result which is described as a big integer, you might want to convert it to a string using for example the library function `Big_int.string_of_big_int`.

15 points Finish the `Memoizer` functor in `memfib.ml` by writing an automatic memoizer that doesnt have the problems of the `PoorMemoizer`. Notice that `Memoizer` ascribes to a different signature than `PoorMemoizer`. Functions that can be memoized by `Memoizer` take a new argument: rather than having type `key -> a` they have type `(key -> a)-> (key -> a)`. When implementing `Memoizer`, assume that any function you memoize uses this new first argument instead of directly calling itself recursively.

To illustrate how we would want to use the function `memo`, take a look at the module `AutoMemoedFibo` in the file `memfib.ml`.

2 points What happens if you use Memoizer to memoize a function that has effects? In particular, what happens if you memoize a function that prints things?

Q4 10 points Draw an environment diagram for the following expression and explain what the final result of its evaluation will be.

```
let y = ref 1 in
let x = 2 in
let f = (let x = !y + 3in (y := 4 ; fn u => u + x + !y)) in
let x = 3 in
  (y := !y + 2;
  f(!y + 3) )
```