Homework 1
COMP 302 Programming Languages and Paradigms

Brigitte Pientka
MCGILL UNIVERSITY: School of Computer Science

Due Date: 24 September 2014

Your homework is due at the beginning of class on Sept 24, 2015. All code files must be submitted electronically using handin, and your program must compile. Solutions should correct (i.e. produce the correct result), be elegant and compact, and take advantage of OCaml's pattern matching. Please consult the style guides posted on the course website to get more information regarding what constitutes good style.

The answer to Q0 should be submitted as a txt-file with the name `Q0.txt`; the answer to Q1 should be submitted as a txt-file with the name `Q1.txt`. For the remaining questions, please fill in the template code given on mycourses and submit the files without changing their names.

Q0. Copyright and Collaboration policy [3 points]

Read the copyright and collaboration policy of this course. For each of the three scenarios below, write 2-3 sentences explaining who is in violation with the copyright and collaboration policy of the course and why.

Scenario 1: Bob and Tom are working on question Q3 of the homework and have already spent 1h trying to get the code to compile and the deadline is drawing near. They decide to ask CS wiz Hanna who already completed the assignment for help. They explain to her that they get a type error when trying to add two numbers using the operation +. She asks: "Are you sure you are adding two numbers of the same type? You can use "+" to add two integers or "`+.`" to add two floating point numbers". Sure enough, Bob and Tom used the wrong symbol to add two integers, change their code and submit their fixed code.

Scenario 2: Bob and Anne work together on all their homework and early on decide to use github, a web-based Git repository hosting service, to better collaborate. Tom finds their repository after searching for material for COMP302 on the web and re-uses their solution for HW2 and submits it with minor modifications.

Scenario 3: John and Matt agreed to be in one team and have started to work already on question 1 of the homework. Matt talks over skype with Abbey who is also in his class. He mentions that he and John are really stuck on Q4 of the COMP302 homework. Abbey wants to help and sends him her solution in OCaml. A week later, John and Matt submit the solution to Q4; Abbey also submits her homework.

Q1. Parsing, Type-checking and Variable Binding [7 points]

Consider the programs in the file `hw1-fixme.ml`. Try to compile the program either by typing into the caml-toplevel `#use ‘‘hw1-fixme.ml’’;;` (note `#use` is a command in OCaml) or in a shell `ocaml hw1-fixme.ml`.

Your task is: Explain step-by-step what errors arise when trying to compile the file `hw1-fixme.ml`; write down the error message you encounter by copy and pasting your errors in your file Q1.txt and state below each error how you fix it to proceed to the next error until your program is error free. Classify your error messages into “Syntax Error” (caught before typing and evaluating a program), “Type Error” (caught before executing the program), and “Run-time Error” (caught during evaluation of the program).

Q2. Zipping and Unzipping[30 points]

In this question we investigate zipping two lists of the same length together and unzipping a list into its two components.

Q2.1 **(10 points)** Implement a function `zip: ’a list -> ’b list -> (’a * ’b)list` which zips two lists into a single list. You can assume that the two input lists have the same length.

```
# zip [1;2;3;4] ["a"; "b"; "c"; "d"];;
- : (int * string) list = [(1, "a"); (2, "b"); (3, "c"); (4, "d")]
```

Q2.2 **(10 points)** Implement a function `unzip: (’a * ’b)list -> ’a list *’b list` which unzips a list into its two components.

```
# unzip [(1, "a"); (2, "b"); (3, "c"); (4, "d")];;
- : int list * string list = ([1; 2; 3; 4], ["a"; "b"; "c"; "d"])
```

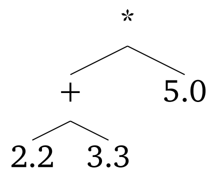
Q2.3 **(10 points)** Prove using induction that your functions satisfy the following property: `unzip (zip l1 l2)= (l1, l2)`

Q3. Pocket Calculator [60 points]

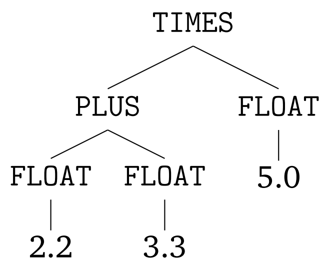
We consider here a simple pocket calculator for arithmetic expressions constructed from floating point numbers (`FLOAT`), addition (`PLUS`), subtraction (`MULT`),multiplication (`TIMES`), in addition to sinus (`SIN`, cosinus (`COS`), and exponential (`EXP`). We represent the arithmetic expressions using a recursive data type in OCaml as follows:

```
type exp =
  | PLUS of exp * exp
  | MINUS of exp * exp
  | TIMES of exp * exp
  | DIV of exp * exp
  | SIN of exp
  | COS of exp
  | EXP of exp * exp
  | FLOAT of float
```

We can then for example represent the arithmetic expression $(2.2 + 3.3) * 5.0$ directly as `TIMES (PLUS (FLOAT 2.2, FLOAT 3.3), FLOAT 5.0)`. This representation directly encodes the syntactic structure, i.e. the abstract syntax tree, of the arithmetic expression. This becomes particularly obvious when we write the expression as a tree where nodes correspond to a particular arithmetic operation.



is represented as



Q3.1 **(10 points)** Implement the function `eval: exp -> float` which takes an arithmetic expression of type `exp` as input and returns its corresponding floating point value.

```
# eval (TIMES (PLUS (FLOAT 2.2, FLOAT 3.3), FLOAT 5.0));;
- : float = 27.5
```

Next we consider translating an expression into a series of instructions for an abstract machine of a simple pocket calculator. The calculator uses instructions for addition (`Plus`), subtraction (`Minus`), multiplication (`Times`), and division (`Div`). In addition, it has instructions for computing the sinus (`Sin`), cosinus (`Cos`), and exponential (`Exp`).

The *instruction set* of the calculator is modelled using the following datatype in OCaml.

```
type instruction =
  Plus | Minus | Times | Div | Sin | Cos | Exp | Float of float
```

The arithmetic expression $(2.2 + 3.3) * 5.0$ can be also viewed as a list of instructions:

```
[Float 2.2; Float 3.3; Plus; Float 5.0; Times]
```

Q3.2 **(20 points)** Implement a function `to_instr:exp -> instruction list` that takes as input an arithmetic expression and returns a list of instructions.

Hint: An expression of type `exp` describes the abstract syntax tree of an arithmetic expression. Translating such an expression to a list of instructions amounts to traversing the tree and flattening it.

```
# to_instr (TIMES (PLUS (FLOAT 2.2, FLOAT 3.3), FLOAT 5.0));;
- : instruction list = [Float 2.2; Float 3.3; Plus; Float 5.; Times]
```

We now want to implement a calculator by executing a list of instructions. The calculator is viewed as a *stack machine* where a *stack* is a list of floating point numbers and it is processed according to the instruction set.

type `stack = float list`

Intuitively, we store intermediate values on the stack. The instructions tell us what to do with these values.

- In the beginning the stack is empty.
- When we encounter a floating point number we simply push it onto the stack.
- When we encounter the instruction `Plus` we simply add the top two elements of the stack `[a;b;c ...]` returning a new stack `[a + b; c ...]` where we replace the top two elements with a single element describing their sum. Similarly we proceed when we encounter other instructions such as `MINUS`, `TIMES`, `DIV` which all work on the top two elements of the stack.
- When we encounter one of the instructions `SIN`, `COS`, `EXP` we apply the corresponding function to the top element of the stack, i.e. for example given the instruction `SIN` and the stack `[a; b; c ...]` we return `[sin a; b; c ...]`.

Q3.3 **(15 points)** Implement a function `instr:instruction -> stack -> stack` which takes an instruction and a stack and returns a new stack according to the description above.

```
# instr Times [5.0; 5.5];;
- : float list = [27.5]
# instr Plus [2.2; 3.3; 5.0];;
- : float list = [5.5; 5.]
# instr (Float 4.2) [2.2; 3.3; 5.5];;
- : float list = [4.2; 2.2; 3.3; 5.5]
```

Q3.4 **(15 points)** Implement a function `prog:instruction list -> float` which takes a list of instructions and computes the resulting floating point value.

Hint: Implement a helper function that takes a list of instructions and a stack and returns the final floating point value.

```
# prog [Float 2.2; Float 3.3; Plus; Float 5.; Times];;
- : float = 27.5
```