Homework 1
COMP 302 Programming Languages and Paradigms

Brigitte Pientka
MCGILL UNIVERSITY: School of Computer Science

Due Date: 18 September 2014

Your homework is due at the beginning of class on Sept 18, 2014. All code files must be submitted electronically, and your program must compile. 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 3 points Copyright and Collaboration policy.

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 is an excellent student and diligently writes down all his class notes and also types up all his homeworks and his solutions. This is an incredible resource, he wishes to make them available for other students to study. He decides to post his material including assignments and midterm with solutions to the wikinotes.

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 7 points Compiling, Type-checking and evaluation/binding warm-up.

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.

Q2 10 points Programming warm-up.

The Newton-Raphson method can be used to find the roots of a function; in particular, it can be used for computing the square root of a given integer. Given a good initial approximation, it converges rapidly and is highly effective for computing square roots, solving the equation

$$a - x^2 = 0$$

To compute the square root of a, choose any positive `x0`, say `1`, as the first approximation. If `x` is the current approximation then the next approximation `next` is

$$(a/x + x)/2$$

Stop as soon as the difference becomes too small.

To compute the square root of a, implement a function `findroot x acc` where `x` approximates the square root of a with accuracy `acc`, i.e. the absolut difference between the current approximation and the next approximation is smaller than `acc`. We use `epsilon_float` as the desired accuracy, which is in Ocaml the difference between 1.0 and the smallest exactly representable floating-point number greater than 1.0.

Your function `findroot` should have type `float -> float -> float`. Note that we made `findroot` a local function to be defined inside the function `square_root`.

Remark: You can compute the absolut value of a floating point number in Ocaml with the library function `abs_float: float -> float`.

Q3 25 points The House of Cards

We revisit here our data types for suit, rank, cards and a hand of cards which we defined in class. Please consult the posted notes and code for further explanation.

```
type suit = Clubs | Spades | Hearts | Diamonds
type rank = Six | Seven | Eight | Nine | Ten |
            Jack | Queen | King | Ace

type card = rank * suit
```

Here we concentrate here on sorting cards in a given hand.

Q3.1 (5 points) Implement a function `dom_rank:rank -> rank -> bool` which when we call `dom_rank r1 r2` returns true, if `r1` is greater than or equal to `r2`. In addition to pattern matching, you can exploit logical operators such as disjunction (written as `a || b` in OCaml), negation (written as `not a` in OCaml), or conjunction (written as `a && b` in OCaml).

Q3.2 (10 points) Implement a function `insert:card -> hand -> hand` which inserts a given card into a hand which is already sorted.

Q3.3 (10 points) Implement a function `sort: hand -> hand` which sorts a hand describing a collection of cards using insertion sort. The basic idea of insertion sort can be summarized as follows: assuming you already have sorted the smaller hand, we are inserting a new card such that the resulting hand remains sorted.

Q4 40 points We represent binary numbers as a list of increasing integers where each element is a power of two.

type `Nat = int list` *(∗ increasing list of weights, each a power of two ∗)*

Example: $5 = [1, 4]$ or $15 = [1, 2, 4, 8]$ or $17 = [1, 16]$

The sparse representation is a more compact way of representing numbers than using a dense representation where $17 = 10001$.

Q4.1 (10 points) Implement a function `inc:Nat -> Nat` which increments a given sparse binary number.

Q4.2 (10 points) Implement a function `dec:Nat -> Nat` which decrements a given sparse binary number (i.e. subtract one).

Q4.3 (10 points) Implement a function `add:Nat -> Nat` which adds two sparse binary numbers.

Q4.4 (10 points) We can easily translate sparse binary numbers to integers. Implement a function `sbinToInt:Nat -> Int` using a local helper function which translates a given sparse binary number to an integer tail-recursively using an accumulator.

Q5 (15 points) In logic, a formula is in negation normal form if the negation operator is only applied to atoms and the only other allowed Boolean operators are conjunction and disjunction. For example, $\neg(q \wedge p) \vee \neg\neg p$ is not in negation normal form, while the equivalent translated formula $(\neg q \vee \neg p) \vee p$ is.

In classical logic and many modal logics, every formula can be brought into this form by replacing implications and equivalences by their definitions, using De Morgan's

laws to push negation inwards, and eliminating double negations. This process can be represented using the following rewrite rules:

$$\begin{array}{lcl} \neg A & \Longrightarrow & A \\ \neg(A \wedge B) & \Longrightarrow & \neg A \vee \neg B \\ \neg(A \vee B) & \Longrightarrow & \neg A \wedge \neg B \\ A \supset B & \Longrightarrow & \neg A \vee B \end{array}$$

We can define a data type for propositions as follows:

```
type prop =
  | Atom of string
  | Neg of prop
  | Conj of prop * prop
  | Disj of prop * prop
  | Impl of prop * prop
```

Here are a few example to show how formulas are represented:

$\neg(q \wedge p) \vee \neg\neg p$	`Disj (Neg(Conj (Atom "q", Atom "p")), Neg(Neg (Atom "p")))`
$(\neg q \vee \neg p) \vee p$	`Disj(Disj (Neg (Atom ‘‘q’’), Neg (Atom ‘‘p’’)), Atom ‘‘p’’)`

Implement a function `nnf:prop -> prop` which accepts a proposition and returns its translation into negation normal form. As a consequence, the result does not contain any implications and all negations are pushed to the atoms