CSCI 3155: Principles of Programming Languages
Exercise sheet #11
25 June 2007

Group name:_____

Functional Programming II

This exercise is mostly a lab exercise, using Standard ML (specifically, *Standard ML of New Jersey*, which is one of the five most prominent implementations of the language).

You can interact with Standard ML using command-line actions or by using EMACS interaction. The accompanying editor interaction sheet describes how to interact with SML.

Some of today's exercises ask for source code submissions. Leave your source code in a directory called `worksheet-11/$n.sml`, where `$n` is the exercise number. Leave source code where the exercise questions ask for source code, and only write down text on the exercise sheet where asked for. (You can also use SML comments to leave your answers to text questions).

Hints:

- In today's exercise, you may run across error messages "`Warning: type vars not generalized because of value restriction`". These error messages are due to parts of SML that we will not discuss. You can always work around this problem by changing the offending expression x to read fn v ⇒ x v
- You can use use "file.sml"; to reuse your previous exercise results.
- The SML comment syntax is (* ... *)

Make sure to save all of your source files. The source files are the `.sml` files you pass to the sml runtime. You will not get full credit if you save the compiler output instead of the source files!

SML of New Jersey won't count how often you interact with it, so feel free to experiment!

Exercise 1. As part of the last worksheet, you constructed two helper functions for dealing with lists. Now that you have learned about *pattern matching*, you can write cleaner versions of the same functions.

Feel free to check your previous implementation for hints as to how you can recurse properly!

(a) Construct a function `map` that takes a function `f` as parameter and applies `f` to each element of the list, using pattern matching

(b) Construct a function `reverse` that reverses an *arbitrary* list, using pattern matching.

Note that SML already provides these two functions as part of its initial environment, as `rev` and `map`.

Exercise 2. SML provides an infix *function composition* operator, "`o`".

(a) Determine the type of "`o`".

(b) Explain its type.

Exercise 3. The programming language Haskell treats the type "string" as an alias to the type "list of characters".

(a) Discuss this approach, using our criteria.

(b) SML takes a similar approach. Explore the types of the functions `explode` and `implode`, then determine (by experiment) what it is that they do.

(c) Implement a function that replaces blanks in strings by dashes ("-"). Use the "`o`" operator. Hide any auxiliary definitions you make within a `let` block.

Exercise 4. Two-valued boolean logic (often imprecisely just called "boolean logic") assumes two literal values, TRUE and FALSE. On these values, we normally define three binary operations AND, OR, IMPLIES, and one unary operation NOT. Below is a full semantics of all literals and values:

$$
\begin{aligned}
[\![\mathsf{TRUE}]\!] &= \top \\
[\![\mathsf{FALSE}]\!] &= \bot \\
[\![\mathsf{NOT}(x)]\!] &= \begin{cases} \top & \iff [\![x]\!] = \bot \\ \bot & \iff [\![x]\!] = \top \end{cases} \\
[\![\mathsf{AND}(x,y)]\!] &= \begin{cases} \top & \iff [\![x]\!] = \top \text{ and } [\![y]\!] = \top \\ \bot & \text{otherwise} \end{cases} \\
[\![\mathsf{OR}(x,y)]\!] &= \begin{cases} \top & \iff [\![x]\!] = \top \text{ or } [\![y]\!] = \top \\ \bot & \text{otherwise} \end{cases} \\
[\![\mathsf{IMPLIES}(x,y)]\!] &= \begin{cases} \bot & \iff [\![x]\!] = \bot \text{ or } [\![y]\!] = \top \\ \top & \text{otherwise} \end{cases}
\end{aligned}
$$

where $\top$ and $\bot$ are two values in some set we use to interpret two-valued boolean logic.

(a) From the above, extract a BNF grammar of two-valued boolean logic expressions.

(b) Develop an SML datatype boolexpr that we can use to represent all two-valued logic expressions.

(c) Implement an *interpreter* for boolexpr. This interpreter is precisely an implementation of the denotational semantics. Interpret $\top$ as true and $\bot$ as false. The name of your interpreter should be sem.

You may use not and the boolean short-circuit operators andalso and orelse.

(d) Which type represents your object language? Which type represents your meta-langauge?

(e) Test your interpreter with the following expressions:

```
TRUE
NOT(TRUE)
AND(FALSE, TRUE)
IMPLIES(NOT(TRUE), FALSE)
IMPLIES(AND(TRUE, NOT(FALSE)), FALSE)
```

Exercise 5. Compilers often use types such as boolexpr to represent input programs. In this exercise we will have a look at what optimising compilers then do with such expressions.

Copy your definition of the datatype boolexpr into a new program.

(a) In real programs, we must usually deal with variables occurring in expressions. Add a value constructor

```
VAR : string -> boolexpr
```

to your datatype.

(b) Examine your previous definition of sem. Assume that you don't know anything about the values of VAR constructions. In this context, you can still evaluate *some* expressions, though not all of them. What would you have to change in sem to adapt it to this new type? Explain[1].

(c) Examine again your results for the previous exercise. Observe that the semantics of IMPLIES look very similar to the semantics of OR. Explain how you can use OR and NOT to describe the (semantically) same construct.

(d) In compiler implementation, it is often helpful to simplify expressions as much as possible: if our expressions are less complex, we need less complex logic to deal with them. Write a program map_deimply that traverses through boolexpr and applies the simplification you implemented above.

(e) Test your transformation with the following expressions:

```
IMPLIES(NOT(TRUE), FALSE)
IMPLIES(IMPLIES(TRUE, FALSE), IMPLIES(FALSE, TRUE))
```

[1] Do not implement a new sem. There are two popular techniques– one pure, one impure– to solve this program elegantly; we have discussed neither in class.

Exercise 6. In the previous exercise, you traversed over boolexpr values to find all places to which you could apply a transformation. You can abstract over this traversal.

(a) Implement a function

$$\mathsf{mapb} : (\mathsf{boolexpr} \to \mathsf{boolexpr}) \to \mathsf{boolexpr} \to \mathsf{boolexpr}$$

that performs the *traversal* part of map_deimplify only. Your function takes two parameters, a function f and a boolexpr e. Your function should visit any sub-boolexpr of e and apply f to it.

(b) Reimplement map_deimplify using mapb.

(c) Find one or two logical simplifications you can apply to boolexprs (hint: consider double negation, DeMorgan's laws if you know them, or how AND/OR behave if one of their parameters is TRUE or FALSE). Write a function that applies these simplifications *locally*. Apply your simplification via mapb. Illustrate that it works.