Zachary Seymour
CS 571
Assignment 3
November 4, 2013

1 Given a Haskell program

```
let fun = x + 1 in fun 3
```

Please encode the program into an expression so that `let` is not used at all.

```
(\x->x+1)3
```

2 Write a Haskell function `sublist lt` that computes all sublists of a list `lt`. The order of the elements in the sublist does not matter.

```
sublist :: [a] -> [[a]]
sublist [] = [[]]
sublist (x:xs) = [x:sub|sub <- sublist xs] ++ sublist xs
```

3 Define a function `replic lt` that replicates each element in `lt` into a list. If the element is in the *k*th position of `lt`, the resulting list contains *k* copies of the same element. You must define this function using the higher-order function of `map`.

I don't see how to do this with just `map`.

```
replic lt = zipWith replicate [1..] lt
```

4 In class, we introduced a function `reverse` to reverse the elements of a list, and function `head` as returning the first element of the list. Define a function `laste` to return the last element of a non-empty list based on `reverse`, `head`, and possible function composition operator (.).

```
laste = head . reverse
```

5 Explain in English what the following function named `mystery` does.

```
mystery n m
      | n == m = [n]
      | n < m = n:(mystery (n+1) m)
      | n > m = n:(mystery (n-1) m)
```

The function is creating a range between n and m, counting up from n to m if n is smaller or counting down from m to n if n is larger.

6 Given the following definition of the propositional formula

```
data Formula
      = Atom Bool                          -- atomic formula
      | And Formula Formula                  -- f /\ f
      | Or Formula Formula                  -- f \/ f
      | Not Formula                          -- not(f)
```

6.1 Write a Haskell function `collect_atoms f` that computes all atomic formulas of a propositional formula `f`

6.2 Write a Haskell function `eval f` to evaluate term f according to standard definitions of propositional logic.

The full code for (1) and (2) follows.

```
data Formula
      = Atom Bool                          -- atomic formula
      | And Formula Formula                  -- f /\ f
      | Or Formula Formula                  -- f \/ f
      | Not Formula                          -- not(f)

instance Show Formula where
   show (Atom a) = "Atom " ++ show a
   show (And a b) = "And (" ++ show a ++ ") (" ++ show b ++ ")"
   show (Or a b) = "Or (" ++ show a ++ ") (" ++ show b ++ ")"
   show (Not a) = "Not (" ++ show a ++ ")"

collect_atoms (Atom x) = [Atom x]
collect_atoms (And a b) = collect_atoms a ++ collect_atoms b
collect_atoms (Or a b) = collect_atoms a ++ collect_atoms b
collect_atoms (Not a) = collect_atoms a

eval (Atom a) = a
eval (And a b) = eval a && eval b
eval (Or a b) = eval a || eval b
eval (Not a) = not (eval a)
```

7 Suppose we would like to have Featherweight Java directly support Java-style interfaces. (You only need to consider the simple case where interfaces are defined as a list of method signatures, and they can be "implemented" by classes and used as object types. No need to consider corner cases such as constants in interfaces, or anonymous class instantiation with interfaces.)

7.1 What changes should be made to the abstract syntax?

We first need to add abstract syntax for an interface definition: `interface I {`$\bar{\texttt{M}}$`}`. Then, we modify the class syntax to include `implements` and the list of methods from `I`:

`class C extends D implements I{`$\bar{\texttt{C}}\ \bar{\texttt{f}}$`; K`$\ \bar{\texttt{M}}\ \bar{\texttt{N}}$`}`

Finally, we need to allow our method syntax to have no body, since this is what is added by the implementing class: `C m(`$\bar{\texttt{C}}\ \bar{\texttt{x}}$`);`.

7.2 What changes should be made to the definition of a top-level Java program? (Recall it was previously defined as a list of classes together with a bootstrapping expression).

The definition needs to be augmented with a list of interfaces implemented by the list of classes.

7.3 What changes should be made to operational semantics?

No changes need to be made to operational semantics, since the lookups for method type and method body still work, since the methods from the interface are added to the method list for the class.

7.4 What changes should be made to expression typing rules?

We add a predicate to rule T-INVK that requires either `C` extend `D` or `C` implements `D` when looking up return type for method `m`.

7.5 What changes should be made to method typing and class typing?

If we add `class C extends D implements I {...}`, then we have another predicate for method typing: if $mtype(\texttt{m}, \texttt{I})$ etc. For class-typing, we add a predicate to rule T-CLASS that requires $\bar{\texttt{N}}$ `OK IN C`, where $\bar{\texttt{N}}$ is the list of methods from our interface as before.