Homework 2: The Abstraction Principle

Due: problems 2, 3, and 5 on March 24; 9, 13, 15, 16, 19, 21, 22 on March 31.

In this homework you will learn about the abstraction principle, including module systems and records.

For all Haskell programs, you must run your code with a Haskell interpreter (such as `hugs`). You must also provide evidence that your program is correct (for example, test cases). Hand in a printout of your code and your testing. Some questions just require an answer in English. Be sure to clearly label what problem each function solves with a comment.

Read Schmidt's book, chapter 2.

1. (suggested practice) Do problems 1.1, and 1.2 on page 67 of Schmidt's book.
2. Do problem 2.1 on page 67 of Schmidt's book. This is to be done in Schmidt's notation. Your answer can be handwritten, although you may find it easier to manipulate this kind of calculation from inside a text editor. (Hint: be careful to only do the *well-formed* programs from exercise 1.1. Also, if you do part b first, part a is easy!)
3. Do problem 2.2, part a on page 68 of Schmidt's book. This is to be done in Schmidt's notation. Your answer can be handwritten.
4. (suggested practice) Do problem 2.2, part b and c on page 68 of Schmidt's book.
5. Do problem 4.1 on page 68 of Schmidt's book, but only for part (b) of exercise 1.1.
6. (suggested practice) Do problem 5.1 on page 68 of Schmidt's book.
7. (extra credit) Do problem 6.2 on page 68 of Schmidt's book. For part (b), the question is about the eager evaluation semantics.
8. (extra credit) Do problem 6.3 on page 68 of Schmidt's book.
9. Do problem 7.2 on page 69 of Schmidt's book. Do this (a) on paper, and then (b) code your semantics and typing in Haskell. You can use the following abstract syntax in Haskell notation as the start of your work. You can add other declarations, commands, and expressions if you wish. And you can use variables instead of locations if you like.

```
> type D = Declaration
> type K = Completion
> type C = Command
> type E = Expression
> type L = Location
> type N = Numeral
> type X = IdentifierExpr

> data Declaration = Label I K | D `Also` D | D `Then` D
> data Completion = C `KSemi` K | Stop | Goto X
> data Command = L `Assign` E | C `Semi` C | While E C | Complete K
> data Expression = Num N | Deref L | E `Plus` E | Not E | E `Equals` E
> data Location = Loc Integer
> type Numeral = Integer
> type IdentifierExpr = Name I
> type Identifier = String
```

You should also be able to build on the type checking code for chapter 2 in `WhileNTyping.lhs`. Be sure to test your code; hand in a printout of your testing along with the code.

First, let's make clear what the semantics of the little language in the problem is supposed to be. Consider the command:

```
stop; v := 3
```

This should result in no change to the variable v, but rather should just halt without changing the store. Similarly, the loop

```
while true do stop od
```

is not an infinite loop, but does nothing.

The tricky part of this problem is part (b). To do this right you have to use the idea of a continuation. A continuation represents "the rest of the program", or in our case, a completion. Semantically, the meaning of the "rest of the program" for a given store is the store that it returns. So continuations are functions from stores to stores. In Haskell we might write

```
> type Continuation = Store -> Store
```

Now, to make commands have the meaning we described above, we have to change their semantics. If we use the standard semantics we've been doing so far, we would not be able to have "stop" halt execution (see the above examples). So the basic idea is:

- pass a continuation to the meaning function, and
- in each meaning send the final store to the passed continuation, instead of returning it directly

We'll have to do this for the meaning functions for commands and completions. But it won't be needed for expressions or decarations.

So the semantics would be something like the following, where I'm not showing any of the typing. I'm also omitting the abstraction parts (the semantics of label declarations and goto), since that's what I want the homework problem to be. Pardon the ASCII notation as well.

```
[[__]] : Command -> Environment -> Continuation -> Store -> Store
[[L := E]] e k s = k(update([[L]] e, [[E]] e s, s))
[[C1 ; C2]] e k s = [[C1]] e (\s -> [[C2]] e k s) s
[[while E do C od]] e k s = w(s)
    where w(s) = if [[E]] e s then [[C]] e (\s -> w(s)) s else k(s)
[[K]] e k s = [[K]] e k s       -- as a Completion

[[__]] : Completion -> Environment -> Continuation -> Continuation
[[C ; K]] e k = (\s -> [[C]] e (\s -> [[K]] e k) s)
[[stop]] e k = (\s -> s)
```

You might want to compute how this semantics works on the two commands given above to help you understand this.

10. (suggested practice) Do problem 7.3 on page 69 of Schmidt's book.

11. (extra credit) Do problem 8.2 on page 69 of Schmidt's book.

12. (extra credit) Do problem 8.3 on pages 69-70 of Schmidt's book.

13. Modify the parser, typing and semantics found in the `WhileN*` files of the course library to include recursive procedures. In what you hand in, highlight or circle the changed parts.

14. (extra credit) Do problem 8.4 on page 70 of Schmidt's book.

15. Do problem 10.2 on page 70 of Schmidt's book.

16. Do problem 11.1 on page 70 of Schmidt's book. This can be handwritten. Try to relate your answer to the semantics and be precise.

17. (extra credit) Do problem 11.2 on pages 70-71 of Schmidt's book.

18. Do problem 12.3 on page 71 of Schmidt's book.

19. Do problem 13.1 on page 71 of Schmidt's book.

20. (extra credit) Do either exercise 13.4 or 13.5 on pages 72-73 of Schmidt's book. If you like, you can do this on paper, or in Haskell, or both.

21. Do problem 14.1 on page 73 of Schmidt's book. Do this on paper. Note that there is a typo in the book, part (c); one can't write E.I in the language of this chapter, only X.I.

22. This can be either done as a group problem or individually. Do problem 15.1 on page 73 of Schmidt's book.