Programming Assignment #3

CS 6353 Section 1, Spring 2016

In this programming assignment, you will be build a type inference engine, using the unification algorithm discussed in class, using Haskell. The requirements for your submission are listed in sections 1 and 2 below. *Follow the requirements carefully and precisely – they will be thoroughly tested.* Section 3 provides a few example outputs.

As usual, I will grade your program by running it against a large test suite. You will be graded by the percentage of test cases you get correct.

The Due Date for this assignment is April 18th at 8:00am.

1. Program Specification

Your program will construct a type environment, and then perform unifications in that type environment, until either the program ends, or a unification fails. For each unification, you will be outputting the most generic type that unifies the two inputs (not the set of substitutions that would unify the type).

That means, as you learn specific information about a generic type, you must keep track of how that information is bound.

For instance, you may be first asked to unify `a and [`b]. After that unification, our type environment must remember that `a should be substituted with [`b]. So, if the next query is to unify `a and `c, the most specific generic type will again be [`b].

1. The grammar for the type language is as follows:

```
TYPE ::= TYPEVAR | PRIMITIVE_TYPE | FUNCTYPE | LISTTYPE;
PRIMITIVE_TYPE ::= 'int' | 'real' | 'str';
TYPEVAR ::= '`' VARNAME;  // Note, the character is a backwards apostrophe!
VARNAME ::= [a-zA-Z][a-zA-Z0-9]*; // Initial letter, then can have numbers
FUNCTYPE ::= '(' ARGLIST ')' -> TYPE | '(' ')' -> TYPE;
ARGLIST ::=   TYPE ',' ARGLIST | TYPE;
LISTTYPE ::= '[' TYPE ']';
```

2. A single input consists of a pair of types on a single line, separated by an ampersand:

```
UNIFICATION_QUERY ::= TYPE '&' TYPE NEWLINE;
NEWLINE ::= '\n';
```

3. White space (tabs and spaces) cannot occur in the middle of a TYPEVAR or PRIMITIVE_TYPE, but can be added anywhere else. This is true for both input and output.

4. When a UNIFICAITON_QUERY does not match the above grammar, your program shall output only "ERR" followed by a newline, and then exit.

5. Valid input to your program follows this grammar:

```
VALID_INPUT ::= QUERY_LIST 'QUIT' '\n';
```

```
QUERY_LIST ::= UNIFICATION_QUERY QUERY_LIST |    ;
```

6. After you read in each unification query, you must output the correct result, which should either be:
a) The most general unifier, if it exists. It should follow the type grammar above.
b) If no most general unifier exists, output: "BOTTOM" (all caps) on a line by itself, and then exit.

7. When the program input consists of a "QUIT" command instead of a UNIFICATION_QUERY, your program should exit without reading further input or producing further output.

8. When your program outputs either "ERR" or "BOTTOM", you must not read any additional input, or produce any additional output.

9. All input is case sensitive. For instance, `a and `A represent different type variables.

10. If unification would create a recursive type, you must output "BOTTOM".

11. Different primitive types cannot unify with each other.

12. When you unify two type variables, it does not matter which name you use as the proper name. For instance, if you're asked to unify `a and `b, either `a or `b are acceptable as an output.

2. Other Requirements

You will receive a 0 on this if any of these requirements are not met!

1. The program must be written entirely in Haskell, with no third party libraries.
2. You must submit the source of your program in a single Haskell file, named NETID.hs, with your NetId in place of NETID. Make sure you get your NetID right, you will get a zero if you use the wrong ID.
3. The program must be compilable with GHC, the Glasgow Haskell Compiler – I will not run any other compiler or interpreter.
4. I will compile your code using the command: ghc –o unify NETID.hs (where NETID is replaced with your NETID).
5. The program must compile and run in the reference environment. Even if it works on your desktop, if it doesn't work in the reference environment, you will get a 0.
6. You must submit your homework before 8:00am on the due date.
7. You must submit the homework through the course website, unless otherwise pre-approved by the professor.
8. You must submit ONLY the Haskell source file.
9. You may not give or receive any help from other people on this assignment.
10. You may use references on the Internet to teach yourself Haskell. For instance, there is a list of Haskell tutorials at https://www.haskell.org/haskellwiki/Tutorials
11. You may NOT use code from any other program, no matter who authored it. This includes 3rd party libraries.

3. Test Cases

Below are four sample test cases for you, which I will use in my testing. Typically, I use anywhere from 20-50 test cases, and will definitely use these three. I strongly recommend you create your own test harness and come up with a large number of test cases to help you get the best possible grade.

For test cases, what one would type on the command line is **BLACK**, input is in **GREEN**, and output is in **BLUE**.

Case 1

```
./unify
(int, int) -> `a & (`a, int) -> int
(int, int)->int
QUIT
```

Case 2

```
./unify
[`a] & int
BOTTOM
```

Case 3

```
./unify
(int, int) & (`a, int)
ERR
```

Case 4

```
./unify
(`a, `b) -> int & ([`b], `c) -> `d
([`c], `c) -> int
`z & int
int
`a & int
BOTTOM
```

Note that, for the first output, it's equally acceptable to output:

```
([`b], `b) -> int
```

For the third unification in this case, it should fail because `a is already bound to a list type (specifically, a list that contains items of type `c).

4. Reference Environment

The reference environment is once again kimota.net. Your same login credentials will continue to work.