Homework 5

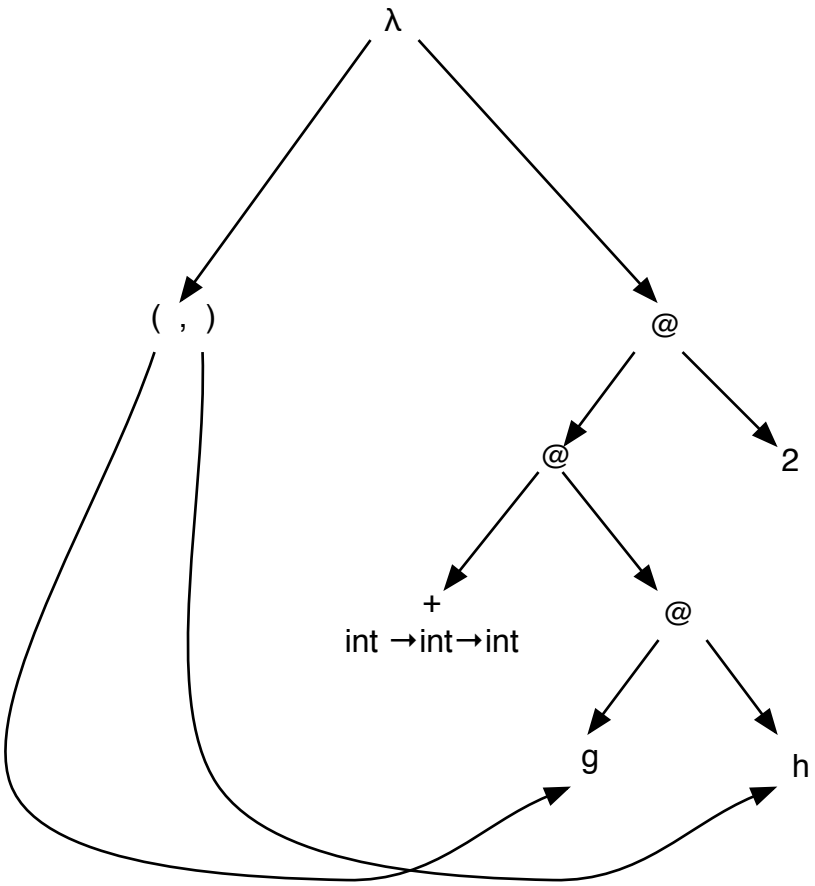
Due Thursday, 10/6/11

Please turn in your homework solutions online at `http://www.dci.pomona.edu/tools-bin/cs131upload.php` before midnight on the due date.

1. (10 points) **Parse Graph**

   Use the parse graph below to calculate the Haskell type for the function

   ```
   fun f(g,h) = g(h) + 2;
   ```



   Be sure to show your work!

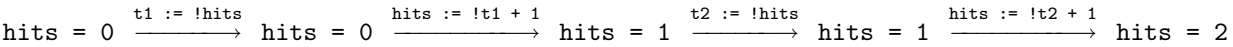
2. (15 points) **Type Inference to Detect Race Conditions**

The general techniques from our type inference algorithm can be used to examine other program properties as well. In this question, we look at a non-standard type inference algorithm to determine whether a concurrent program contains race conditions. Race conditions occur when two threads access the same variable at the same time. Such situations lead to non-deterministic behavior, and these bugs are very difficult to track down since they may not appear every time the program is executed. For example, consider the following program, which has two threads running in parallel:

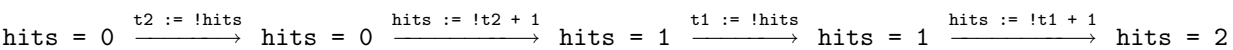
```
Thread 1:                    Thread 2:
  t1 := !hits;                 t2 := !hits;
  hits := !t1 + 1;             hits := !t2 + 1;
```

In the above code, ":=" is used for assignment, "!" is used to extract the value of a variable. Thus `x := !x + 1` *increases the value of variable x by 1.*

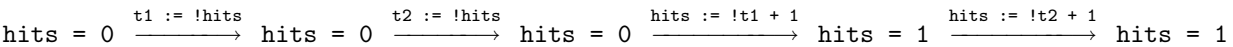
Since the threads are running in parallel, the individual statements of Thread 1 and Thread 2 can be interleaved in many different ways, depending on exactly how quickly each thread is allowed to execute. For example, the two statements from Thread 1 could be executed before the two statements from Thread 2, giving us the following execution trace:



After all four statements execute, the `hits` counter is updated from zero to 2, as expected. Another possible interleaving is the following:



This again adds 2 to `hits` in the end. However, look at the following trace:



This time, something bad happened. Although both threads updated `hits`, the final value is only 1. This is a race condition: the exact interleaving of statements from the two threads affected the final result. Clearly, race conditions should be prevented since it makes ensuring the correctness of programs very difficult. One way to avoid many race conditions is to protect shared variables with mutual exclusion locks. A lock is an entity that can be held by only one thread at a time. If a thread tries to acquire a lock while another thread is holding it, the thread will block and wait until the other thread has released the lock. The blocked thread may acquire it and continue at that point. The program above can be written to use lock `1` as follows:

```
Thread 1:                    Thread 2:
  synchronized(1) {            synchronized(1) {
    t1 := !hits;                 t2 := !hits;
    hits := !t1 + 1;             hits := !t2 + 1;
  }                            }
```

The statement "`sychronized(l) { s }`" aquires lock `l`, executes `s`, and then releases lock `l`. There are only two possible interleavings for the program now:

`hits = 0` $\xrightarrow{\texttt{t1 := !hits}}$ `hits = 0` $\xrightarrow{\texttt{hits := !t1 + 1}}$ `hits = 1` $\xrightarrow{\texttt{t2 := !hits}}$ `hits = 1` $\xrightarrow{\texttt{hits := !t2 + 1}}$ `hits = 2`

and

`hits = 0` $\xrightarrow{\texttt{t2 := !hits}}$ `hits = 0` $\xrightarrow{\texttt{hits := !t2 + 1}}$ `hits = 1` $\xrightarrow{\texttt{t1 := !hits}}$ `hits = 1` $\xrightarrow{\texttt{hits := !t1 + 1}}$ `hits = 2`

All others are ruled out because only one thread can hold lock `l` at a time. Note that while we use assignable variables inside the synchronized blocks, the names we use for locks are constant. For example, the name `l` in the example program above always refers to the same lock.

Our analysis will check to make sure that locks are used to guard shared variables correctly. In particular, our analysis checks the following property for a program P:

> For any variable `y` used in P, there exists some lock `l` that is held by the current thread every time `y` is accessed.

In other words, our analysis will verify that every access to a variable `y` will occur inside the synchronized statement for some lock `l`. Checking this property usually uncovers many race conditions.

Let's start with a simple program containing only one thread:

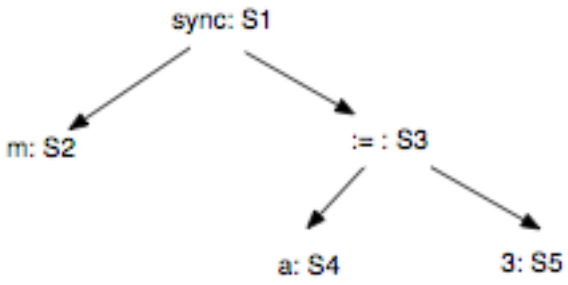
```
Thread 1:
  synchronized (m) {
    a := 3;
  }
```

For this program, our analysis should infer that lock `m` protects variable `a`.

As with standard type inference, we proceed by labeling nodes in the parse tree, generating constraints, and solving them.

Step 1: Label each node in the parse tree for the program with a variable. This variable represents the set of locks held by the thread every time execution reaches the statement represented by that node of the tree. Note that these variables keep track of sets of locks names, and NOT types, in this analysis.

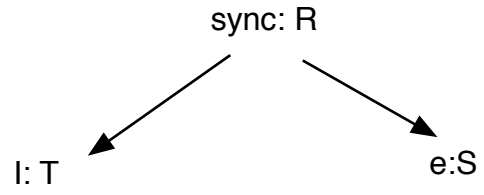
Here is the labeled parse tree for the example:



Step 2: Generate the constraints using the following four rules:

(a) If S is the variable on the root of the tree, then $S = \emptyset$.

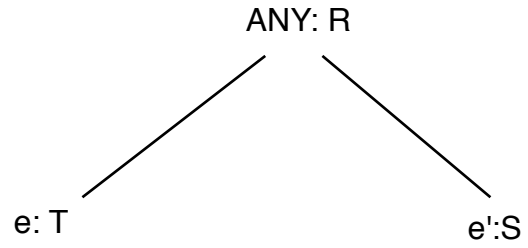
(b) For any subtree matching the form



we add two constraints:

$$\begin{aligned} T &= R \\ S &= R \cup \{\texttt{1}\} \end{aligned}$$

(c) For any subtree matching the form



where ANY matches any node other than a `sync` node, we add two constraints:

$$\begin{aligned} T &= R \\ S &= R \end{aligned}$$

(d) To determine $\texttt{lock}_y$, the lock guarding variable `y`, add the constraint

$$\texttt{lock}_y \in S$$

for each node $\texttt{y} : S$ or $\texttt{!y} : S$ in the tree. In other words, require that $\texttt{lock}_y$ be in the set of locks held at each location `y` is accessed.

Here are the constraints generated for the example program:

$$\begin{array}{rcll} S1 & = & \emptyset & \text{(rule 2a)} \\ S2 & = & S1 & \text{(rule 2b)} \\ S3 & = & S1 \cup \{\texttt{m}\} & \text{(rule 2b)} \\ S4 & = & S3 & \text{(rule 2c)} \\ S5 & = & S3 & \text{(rule 2c)} \\ \texttt{lock}_a & \in & S4 & \text{(rule 2d)} \end{array}$$

Step 3: Solve the constraints to determine the set of locks held at each program point and which locks guard the variables:

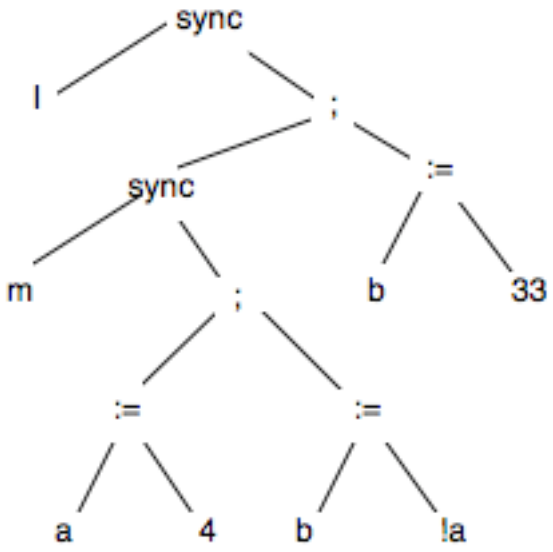
$$\begin{aligned} S2 = S1 &= \emptyset \\ S3 = S4 = S5 &= \{\texttt{m}\} \\ \texttt{lock}_a &\in \{\texttt{m}\} \end{aligned}$$

Clearly, $\texttt{lock}_a$ is `m` in this case, exactly as we expected.

You will now explore some aspects of this analysis:

(a) Here is another program and corresponding parse tree:

```
Thread 1:
  synchronized (l) {
    synchronized (m) {
      a := 4;
      b := !a;
    }
    b := 33;
  }
```



Compute $\texttt{lock}_a$ and $\texttt{lock}_b$ using the algorithm above. Explain why the result of your algorithm makes sense.

(b) Let's go back to the original example, but change Thread 2 to use a different lock:

```
Thread 1:                          Thread 2:
  synchronized(l) {                  synchronized(m) {
    t1 := !hits;                       t2 := !hits;
    hits := !t1 + 1;                   hits := !t2 + 1;
  }                                  }
```

Compute $\texttt{lock}_{t1}$, $\texttt{lock}_{t2}$, and $\texttt{lock}_{hits}$ using the algorithm above. Since there are two threads in the program, you should create two parse trees, one for each thread. Explain the result of your algorithm.

(c) Suppose that we allow assignments to lock variables. For example, in the following program, `l` and `m` are references to locks, and we can change the locks to which those names refer with an assignment statement:

```
Thread 1:                          Thread 2:
  synchronized(!l) {                 synchronized(!m) {
    a := !a + 1;                       x := !b + 3
  }                                    b := 11 + x;
  m := !l;                           }
  synchronized(!m) {
    a := !b + 1;
    b := !a;
  }
```

Describe any problems that arise due to assignments to lock variables, and what the implications for the analysis are. You do not have to show the constraints from this example or change the analysis to handle mutable lock variables. A coherent discussion of the issues is sufficient. Thinking about what the algorithm would compute for $\texttt{lock}_a$, $\texttt{lock}_b$, and $\texttt{lock}_x$ may be useful, however.

3. (10 points) **Type Inference and Bugs**

What is the type of the following Haskell function:

```
append([],l) = l
append (x:l,m) = append(l,m)
```

Write one or two sentences to explain succinctly and informally why append has the type you give. This function is intended to append one list onto another. However, it has a bug. How might knowing the type of this function help the programmer to find the bug?

4. (10 points) **Type Inference and Debugging**

Please do problem 6.8 from Mitchell, page 159.

NOTE: There is an important typo in the problem. The (incorrect) definition of `reduce` should be:

```
reduce(f,[x]) = [x]
reduce(f,(x:y)) = f(x,reduce(f,y))
```

The other functions in this problem are written in ML, rather than Haskell, but you should have no trouble translating them into Haskell.

5. (15 points) **Dynamic Typing in ML**

   Please do problem 6.11 from Mitchell, page 160, except write your solution in Haskell, rather than ML. You should use the following to define your type LISP:

```
data LISP = Nil | Symbol String | Number Int | Cons (LISP,LISP) |
            Function (LISP,LISP) deriving (Eq,Show)
```

   For part c, assume that `car` returns `nil` when applied to anything other than a `Cons` cell.

6. (30 points) **Parsing Tuples**

   Given the following BNF:

```
<exp> ::= ( <tuple> ) | a
<tuple> ::= <tuple>, <exp> | <exp>
```

   (a) Draw the parse tree for ((a,a),a,(a)).

   (b) Write a lexer for terms of this form. The tokens are simply "a", "(", ")", and ",". The tokens generated by the lexer should be from the following type:

```
data Tokens = AToken | LParen | RParen | Comma | Error String |
              EOF deriving (Eq,Show)
```

   where `EOF` marks the end of the token list. The main lexer function should be of the form:

```
getTokens :: [Char] -> [Tokens]
```

   As an example, you should get the following results when testing `getTokens`:

```
*Main> getTokens "((a,a),a,(a,a))"
[LParen,LParen,AToken,Comma,AToken,RParen,Comma,AToken,Comma,
LParen,AToken,Comma,AToken,RParen,RParen,EOF]
```

   (c) An alternative grammar for the above language in EBNF is

```
<exp> ::= ( <tuple> ) | a
<tuple> ::= <exp> {, <exp> }*
```

   where the * means the items in parentheses repeat 0 or more times.

   We can rewrite this as BNF as

```
<exp> ::= ( <tuple> ) | a
<tuple> ::= <exp> <expTail>
<expTail> ::= e | , <exp> <expTail>
```

where `e` stands for the empty string.

Recall from class that we can build a table that will direct a parser as follows. The rows of the table will correspond to non-terminals, while the columns will correspond to terminals. The entries are productions from our grammar.

Put production `X` ::= α in entry (`X`,`b`) if either

- `b` $\in$ FIRST(α), or
- $\epsilon \in$ FIRST(α) and `b` $\in$ FOLLOW(`X`).

For any non-terminal `X` and terminal `b`, the production `X` ::= α will occur in the corresponding entry if applying this production can eventually lead to a string starting with `b`.

For this to give us an unambiguous parse, no table entry should contain two productions. (If so, we would have to rewrite the grammar!) Slots with no entries correspond to errors in the parse (e.g., that the string is not in the language generated by the grammar).

We can also write this restriction out as the following two laws for predictive parsing:

i. If `A ::=` α_1 `|` `...|` α_n then for all i $\neq$ j, First(α_i) $\cap$ First(α_j) = $\emptyset$.

ii. If `X` $\rightarrow^*$ ϵ, then First(`X`) $\cap$ Follow(`X`) = $\emptyset$.

Compute First and Follow for each non-terminal of this grammar and show that the grammar follows the first and second rules of predictive parsing.

(d) The following definition can be used to represent abstract syntax trees of the expressions generated by the grammar above:

```
data Exp = A | AST_Tuple [Exp] | AST_Error String deriving (Eq,Show)
```

[The last item is simply there to handle the case of errors.] Thus the tuple (a,a,a) would be represented as `AST_Tuple[A,A,A]`, while the term in part (a) would be represented as `AST_Tuple [AST_Tuple [A,A],A,AST_Tuple[A,A]]`.

Write a predictive recursive descent parser for the grammar in part 6c. It should generate abstract syntax trees of type `Exp`.

Hint: Watch the types of your parsing functions. You should have

```
parseExp :: [Tokens] -> (Exp, [Tokens])
parseTuple :: [Tokens] -> ([Exp], [Tokens])
parseExpTail :: ([Exp], [Tokens]) -> ([Exp], [Tokens])
parse :: [Char] -> Exp
```

where `parse` is the function invoked on the input string. E.g.,

```
*Main> parse "((a,a),a,(a,a))"
AST_Tuple [AST_Tuple [A,A],A,AST_Tuple [A,A]]
```