Homework 5

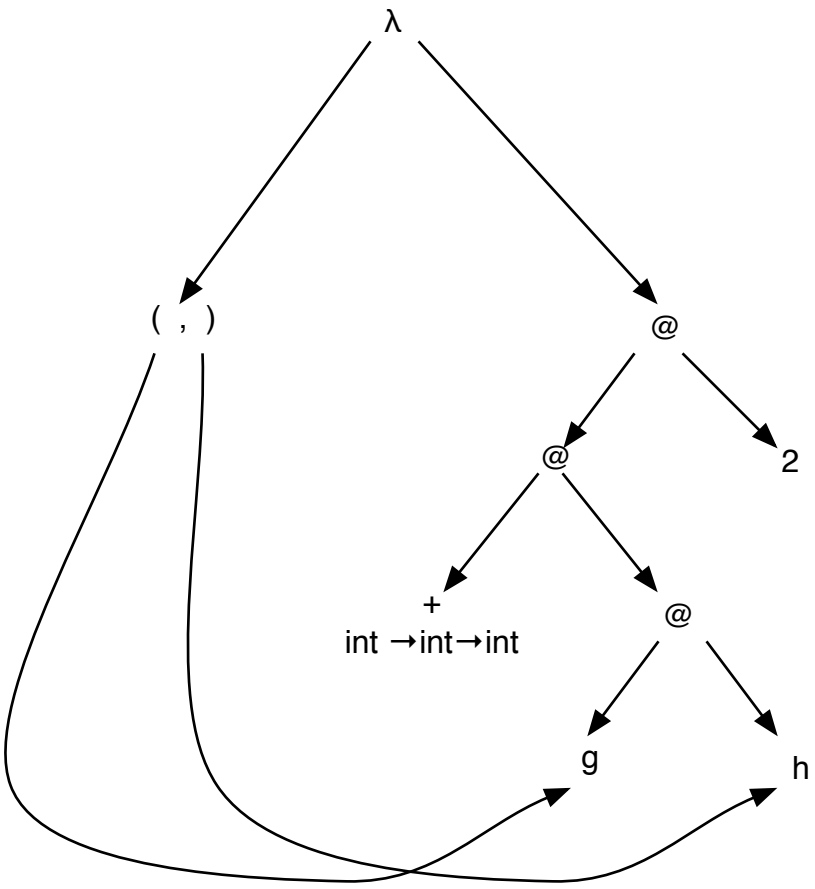
Due Thursday, 10/9/14

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. (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?

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

   Please do problem 6.8 from Mitchell's revised Chapter 6, page 149.

   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))
```

   Notice the extra square brackets in the first clause!

   In your answer to this problem, explain how to fix the definition.

4. (15 points) **Dynamic Typing in Haskell**

   Please do problem 6.11 from Mitchell's revised Chapter 6 page 151.

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

5. (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 braces may 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 5c. 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]]
```