CSCI 131 Spring 2019

Homework 4

Due Thursday, 2/21/2019

Important: For this assignment I would like you to turn in a zipped folder with two files. The first file should be a LaTeX document with the complete solutions to all problems (including code). The second file should have suffix "hs" and contain all of the Haskell programs you are writing for this assignment. All functions should be named exactly as specified in this assignment. We will be testing your code by loading this file and running our code on it. The first line of your Haskell file should be:

```
module Hmwk4 where
```

Putting this line in will allow us to automatically test your program. Leaving it out will result in all of our tests failing, and you receiving a correspondingly low score, making all of us sad! As usual, test your program by loading the "hs" file once you have fully commented your code.

Name your two files `Hmwk4.hs` and `Hmwk4.pdf` (where you should replace *OurNames* by the last names of the people working together on the assignment).

When you are ready to turn in your homework, compress your folder into a zip file. Then go to `https://submit.cs.pomona.edu/2019sp/cs131` and click on the assignment 4 on that page and follow the instructions to submit your work.

Be sure to test your programs one last time before submitting. I've seen students mess up when commenting code in a way that causes everything to break. Code that doesnt compile will get very little credit. We will be using some automatic testing, and code that doesnt compile brings everything to a halt. With a large class we will not have time to go in to manually tweak your code to make it work.

1. (20 points) **Parsing Tuples** This problem is a continuation of the last problem in last week's homework.

   Given the following BNF:

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

   Last week you wrote 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 was of the form:

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

   For this week do the last two parts of the problem:

   (a) 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` $\rightarrow^* \epsilon$, 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.

(b) 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 **??**. 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]]
```

2. (5 points) **Lambda Calculus Reduction**

   Please do problem 4.3 from Mitchell, page 83.

3. (10 points) **Symbolic Reduction**

   Please do problem 4.4 from Mitchell, page 83.

4. (10 points) **Lambda Reduction with Sugar**

   Here is a "sugared" lambda-expression using `let` declarations:

$$\begin{array}{l}\texttt{let}\ \mathit{compose} = \lambda f.\, \lambda g.\, \lambda x.\, f(g\,x)\ \texttt{in} \\ \quad \texttt{let}\ h = \lambda x.\, x + x\ \texttt{in} \\ \qquad ((\mathit{compose}\, h)\, h)\, 3\end{array}$$

   The "de-sugared" lambda-expression, obtained by replacing each `let` $z = U$ `in` V by $(\lambda z.\, V)\, U$ is

$$\begin{array}{l}(\lambda \mathit{compose}. \\ \quad (\lambda h.\ ((\mathit{compose}\ h)\ h)\ 3)\quad (\lambda x.\, x + x)) \\ \quad (\lambda f.\, \lambda g.\, \lambda x.\, f(g\ x))\end{array}$$

   This is written using the same variable names as the `let`-form in order to make it easier to compare the expressions.

   Simplify the desugared lambda expression using reduction. Write one or two sentences explaining why the simplified expression is the answer you expected.

5. (20 points) **Defining Terms in Lambda Calculus**

   In class we defined Church numerals and booleans, and showed how to define more complex functions in the pure lambda calculus. Please show how to define the following functions in the pure lambda calculus. (You may use the functions defined in class in defining these new functions.)

   (a) (5 points)
   Define a function Minus such that Minus m n = m - n if m > n and 0 otherwise. Do not use recursion, but instead define it directly. You may assume that you are given the function Pred defined (but not explained) in class that computes the predecessor of a number (where the predecessor of 0 is 0, predecessor of 1 is 0, 2 is 1, etc.). That is, your answer may include the term Pred without providing a definition of it.

   (b) (5 points)
   Define a function LessThan such that LessThan m n = true iff m < n.

   (c) (5 points)
   Define the recursive fibonacci function fib such that fib 0 = 1, fib 1 = 1, and fib n = fib (n-1) + fib (n-2) for n > 1. *This is tricky as you are not allowed to name the function and use the name in the definition. You must use the Y-combinator and emulate what we did in class defining factorial.*

(d) (5 points)
Use your definition of $\underline{\text{fib}}$ from above to calculate $\underline{\text{fib}}$ 2. Do it step by step, showing all of your work. You may assume that $\underline{\text{fib}}\ \underline{1} = \underline{1}$, $\underline{\text{fib}}\ \underline{0} = \underline{1}$ and that $\underline{\text{Plus}}\ \underline{1}\ \underline{1} = \underline{2}$ without showing all of the reduction steps. All other steps in the reduction should be shown.

6. (10 points) **Fixed Points**

We showed in class that every function f definable in the lambda calculus has a fixed point, i.e., there is a term a such that f(a) = a. In class we defined the function $\underline{\text{Succ}}$ as the successor function. I.e., $\underline{\text{Succ}}\ \underline{\text{n}} = \underline{\text{n+1}}$ for all encodings of integers, $\underline{\text{n}}$. Needless to say, we don't expect the successor function to have a fixed point, yet we proved that every function has a fixed point!

Compute the fixed point of $\underline{\text{Succ}}$. What can you say about it? In particular, why doesn't this contradict our expectations that the successor function on the natural numbers does not have a fixed point. *Yes, this is supposed to be tricky and puzzling, but you need to deal with this type of seemingly contradictory information. A hint is that you should think about what all encodings of numbers look like.*