Homework 7

Due Thursday, Nov 1 by 10pm

Handout 9
CSCI 334: Fall 2018

Turn-In Instructions

For text response questions, provide a LaTeX source file and pre-built PDF. For example, for question 3, provide both "`q3.tex`" and "`q3.pdf`" files. For full credit, your LaTeX file should build properly. Be sure to `git add` all necessary files (e.g., images) if your LaTeX depends on it.

For each coding question in this assignment, create a project directory. For example, the source directory for question 1 should be in a folder called "`q1`". You should be able to `cd` into this directory and then run the program by typing the command "`dotnet run`". Each program should be split into two pieces: a "`Program.fs`" file that contains the `main` method and associated program-startup routines (like argument parsers), and another "`Library.fs`" file that contains the function(s) of interest in the question. All library code should be in a module named "`CS334`". Be sure to provide usage output (defined in `main`) for all programs that require arguments. For full credit, your program should both build and run correctly.

Turn in your work using the Github repository assigned to you. The name of the Github repository will have the form `cs334hw7_<your user name>`. For example, my repository would be `cs334hw7_dbarowy`. You should have received an invite to commit in the repository in your email. If you did not receive this email, please contact me right away!

A good way to check that you submitted your assignment correctly is to `git clone` your repository to a new folder and then try building/running everything.

Pair Programming and Honor Code: You may optionally collaborate with one other person on the submission for this assignment. If you work with a partner, choose one of the two partner repositories for your submission. In the other repository, be sure to leave a `collaborators.txt` file that states who you worked with and the name of the repository where the code may be found. If you would like me to pair you with a partner, please let me know.

This assignment is due on Thursday, Nov 1 by 10pm.

Reading

1. **(Required)** "Parser combinators"
2. **(Required)** `https://en.wikipedia.org/wiki/Scope_(computer_science)#Lexical_scoping`
3. **(Required)** `https://en.wikipedia.org/wiki/Scope_(computer_science)#Dynamic_scoping`
4. **(Optional)** "Global Variables (First-Order Case)" in Concepts in Programming Languages by Mitchell, pp. 176–180.

Problems

Q1. (*10 points*) ..... F# Map for Trees

(a) The binary tree datatype

```
type Tree<'a> =
| Leaf of 'a
| Node of Tree<'a> * Tree<'a>
```

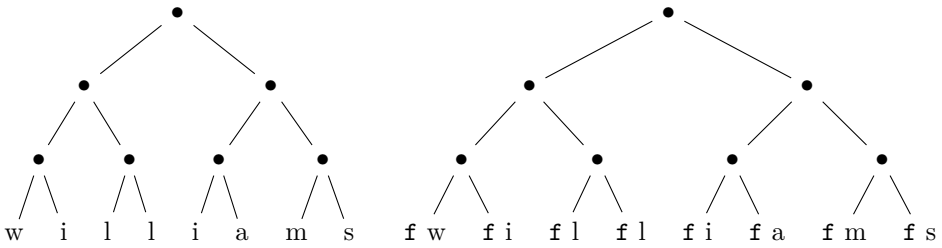
describes a binary tree for any type, but does not include the empty tree (i.e., each tree of this type must have at least a root node).

Write the curried function

```
let maptree f t = ???
```

where `f` is a function and `t` is a tree. `maptree` should return a new tree that has the same structure as `t` but where the values stored in `t` have the function `f` applied to them.

Graphically, if `f` is a function that can be applied to values stored in the leaves of tree `t`, and `t` is the tree on the left, then `maptree f t` should produce the tree on the right.



For example, if `f` is the function `let f x = x + 1` then

`maptree f (Node(Node(Leaf 2, Leaf 3), Leaf 4));;`

should evaluate to `Node (Node (Leaf 3,Leaf 4),Leaf 5)`.

(b) In an XML comment block above your `maptree` definition, explain your definition in one or two sentences (in `<summary>`). You can find documentation on XML comment blocks at the following URL: `https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/xml-documentation`. Be sure to provide `<param ...>` and `<returns>` tags.

(c) What type does F# give to your function? Why isn't it the type `('a → 'a) → Tree<'a> → Tree<'a>`?

The project directory for this question should be called "`q1`". You should be able to run your program on the command line by typing, for example, "`dotnet run`" and output like the kind shown above should be printed to the screen. Be sure to provide several examples that demonstrate that your function works correctly.

Q2. (*10 points*) ..... F# Reduce for Trees

The binary tree datatype

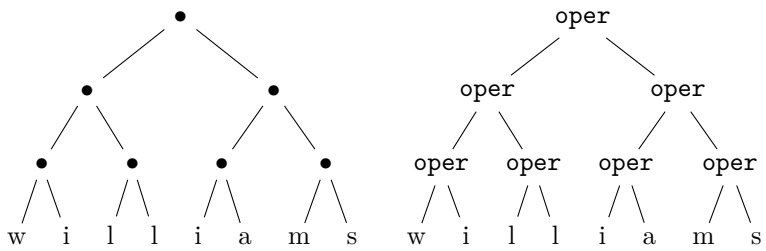
```
type Tree<'a> =
| Leaf of 'a
| Node of Tree<'a> * Tree<'a>
```

describes a binary tree for any type, but does not include the empty tree (i.e., each tree of this type must have at least a root node).

(a) Write a function

`treduce : ('a → 'a → 'a ) → Tree<'a> → 'a`

that combines all the values of the leaves using the binary operation passed as the first parameter. In more detail, if `oper : 'a → 'a → 'a` and `t` is the nonempty tree on the left in this picture,



then `treduce oper t` should be the result obtained by evaluating the tree on the right. For example, if `f` is the function

```
let f x y = x + y
```

then `treduce f (Node(Node(Leaf 1, Leaf 2), Leaf 3))` $= (1 + 2) + 3$ and the output is `6`.

(b) In an XML comment block above your `maptree` definition, explain your definition of `treduce` in one or two sentences (in `<summary>`). You can find documentation on XML comment blocks at the following URL: `https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/xml-documentati`
Be sure to provide `<param ...>` and `<returns>` tags.

The project directory for this question should be called "`q2`". You should be able to run your program on the command line by typing, for example, "`dotnet run`" and output like the kind shown above should be printed to the screen. Be sure to provide several examples that demonstrate that your function works correctly.

Q3. (*10 points*) ..... Currying

Show that the ML types $'a \to ('b \to 'c)$ and $('a * 'b) \to 'c$ are essentially equivalent.

(a) Define higher-order ML functions

$$\texttt{Curry} : (('a * 'b) \to 'c) \to ('a \to ('b \to 'c))$$

and

$$\texttt{UnCurry} : ('a \to ('b \to 'c)) \to (('a * 'b) \to 'c)$$

(b) For all functions $\texttt{f} : ('a * 'b) \to 'c$ and $\texttt{g} : 'a \to ('b \to 'c)$, the following two equalities should hold (if you wrote the right functions):

$$\texttt{UnCurry}(\texttt{Curry}(\texttt{f})) = \texttt{f}$$

$$\texttt{Curry}(\texttt{UnCurry}(\texttt{g})) = \texttt{g}.$$

Explain why each is true, for the functions you have written. Your answer can be 3 or 4 sentences long. Try to give the main idea in a clear, succinct way. We are more interested in the insight than in the number of words.

Note that you must *explain* why the equations hold to receive full credit.

Hint: One way to do this is to apply both sides of each equation to the same argument(s) and describe how each side evaluates to the same term. For example, show that

$$\texttt{UnCurry}(\texttt{Curry}(\texttt{f}))(\texttt{s}, \texttt{t}) = \texttt{f}(\texttt{s}, \texttt{t})$$

and

$$\texttt{Curry}(\texttt{UnCurry}(\texttt{g}))\ \texttt{s}\ \texttt{t} = \texttt{g}\ \texttt{s}\ \texttt{t}$$

for any `s` and `t`.

Supply your answer as a LaTeX file and call it "`q3.tex`". Also provide a PDF for your LaTeX.

Q4. (*10 points*) ..... Static and Dynamic Scope

If a program variable `x` is *used* in the body of a function `f`, but `x` is *not declared* inside `f` itself, then the value of `x` depends a declaration outside `f`. In this situation, the location of the storage for `x` is outside the stack frame for `f`. Because `x` must be declared *somewhere*, access to the value of `x` involves locating the appropriate stack frame elsewhere on the stack.

There are two different procedures for locating `x`. Programming languages generally choose one rule or the other.

- Static scope: The variable `x` refers to storage that is declared in the *closest enclosing scope* of the program *text*. Because this rule depends only on program text (source code), it is also called lexical scope.
- Dynamic scope: The variable `x` refers to storage associated with the *most recent* stack frame still on the stack.

In other words, each scope rule is an *algorithm* that searches the call stack for variable storage.

Consider the following program fragment, written in F#:

```
1    let result() =
2      let x = 2
3      let f = fun y -> x + y
4      let x = 7
5      x +
6        f x
```

(a) Under static scoping, what is the value of `x + f x` in this code? During the execution of this code, the value of `x` is used three different times (on lines 3, 5, and 6). For each line where `x` is used, state the value of `x` and explain why these are the appropriate values under static scoping.

(b) Under dynamic scoping, what is the value of `x + f x` in this code? For each line where `x` is used, state the value of `x` and explain why these are the appropriate values under dynamic scoping.

(c) Which scope rule does F# use? How do you know?

Supply your answer as a LaTeX file and call it "`q4`.tex". Also provide a PDF for your LaTeX.

Q5. (*30 points*) ..... Parsing with Combinators

(a) Given the following grammar in Backus-Naur Form,

```
<expr> ::= <var>
         |  <abs>
         |  <app>
<var>  ::= α ∈ {a...z}
<abs>  ::= (L<var>.<expr>)
<app>  ::= (<expr><expr>)
```

and the algebraic data type,

```
type Expr =
| Variable of char
| Abstraction of char * Expr
| Application of Expr * Expr
```

provide an implementation for the parser function

```
parse(s: string) : Expr option
```

In other words, given a `string s` representing a valid expression, `parse` should return `Some` abstract syntax tree (the `Expr`) of the expression. When given a `string s` that is not a valid expression, `parse` should return `None`.

You may use any of the combinator functions defined in the assigned reading on parser combinators in your solution. You may find the `pletter`, `pchar`, `pstr`, `pseq`, `pbetween`, `pleft`, `peof`, `<|>`, and `|>>` combinators to be the most useful.

Note that because your expression parser is recursive, we need to do a little bookkeeping to keep F# happy. The first parser that appears your implementation should be written like:

```
let expr, exprImpl = recparser()
```

`recparser` defines two things: a parser called `expr` and an implementation for that parser called `exprImpl`. Later, once you have defined all of the parsers that `expr` depends on, write:

```
exprImpl := (* your expr parser implementation here *)
```

Note: use of `recparser` is admittedly a little bit of a hack. We use it because F# does not use laziness by default, which means that F# gets unhappy when it notices that we define recursive functions of other recursive functions. `recparser` is one way around this problem. You should only need to use `recparser` once in this problem. To be *crystal clear*, your code should probably have at least the following definitions in it:

```
let expr, exprImpl             = recparser()
let variable    : Parser<Expr> = (* variable parser implementation *)
let abstraction : Parser<Expr> = (* abstraction parser implementation *)
let application : Parser<Expr> = (* application parser implementation *)
exprImpl                      := (* expr parser implementation *)
```

(b) Provide a function `prettyprint(e: Expr) : string` that turns an abstract syntax tree into a string. For example, the program fragment

```
let asto = parse "((Lx.x)(Lx.y))"
match asto with
| Some ast -> printfn "%A" (prettyprint ast)
| None     -> printfn "Invalid program."
```

should print the string

```
Application(Abstraction(Variable(x), Variable(x)), Abstraction(Variable(x), Variable(y)))
```

(c) Be sure to document all of your functions using XML comment blocks.

The project directory for this question should be called "`q5`". You should be able to run your program on the command line by typing, for example, "`dotnet run "((Lx.x)(Lx.y))"`" and output like the kind shown above should be printed to the screen.

Q6. (*10 points*) ..... Five 5's (Bonus)

Consider the well-formed arithmetic expressions using the numeral 5. These are expressions formed by taking the integer literal 5, the four arithmetic operators `+`, `-`, `*`, and `/`, parentheses. Examples are `5`, `5 + 5`, and `(5 + 5) * (5 - 5 / 5)`. Such expressions correspond to binary trees in which the internal nodes are operators and every leaf is a 5. Write an F# program that answers each of the following questions:

(a) What is the smallest positive integer that cannot be computed by an expression involving exactly five 5's?

(b) What is the largest prime number that can computed by an expression involving exactly five 5's?

(c) Generate an expression that evaluates to that prime number.

You should start by defining a `type` to represent arithmetic expressions involving 5 and the arithmetic operations, as well as an `eval` function that evaluates them. This question involves only integer arithmetic, so be sure to use `int` data. You should then write code that generates all expressions containing a fixed number of 5's.

Answering the questions will involve an exhaustive search for all numbers that can be computed with a fixed number of 5's, so good programming techniques are important. Avoid unnecessary operations. You will also have to address the possibility of division by zero inside `eval` using an exception handler.

Be sure to document your code using XML comment blocks.

The project directory for this question should be called "`q6`". You should be able to run your program on the command line by typing, for example, "`dotnet run`" and the program's output should clearly indicate the answers to the questions above.