UVa ID (again): ____________________

<table>
<thead>
<tr>
<th>Problem</th>
<th>Max Points</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1—Definitions</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>2—Reason Functional Programming</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>3—Higher-Order Functions</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>4—Regular Expressions and Automata</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>5—Ambiguity</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>6—Interpreter Stages</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Extra Credit</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>100</strong></td>
<td></td>
</tr>
</tbody>
</table>

Honor Pledge:

How well do you think you did? ____________________________________________
1. Definitions (15 points)

For each of the following terms, briefly describe or define its meaning. Each definition is worth three points.

(a) First-class functions:
First-class functions are treated like values—they may be passed as values, returned from other functions, and bound to variables.

(b) Linear recursion:
In linear recursion, expression length (and memory) use grows linearly with the input size.

(c) Closure:
A closure is the combination of a function and its surrounding state. The function has access to the bindings of surrounding state even if the function is invoked outside of the scope where those values are bound.

(d) Maximal munch rule:
The maximal munch rule (in the context of lexing) states that we should pick the longest possible substring from the input that matches a lexing rule (regular expression).

(e) Language of a Context-Free Grammar (i.e., $L(G)$):
The language of a context-free grammar is the set of token sequences for which there are valid derivations given the grammar’s productions. More formally:

$$L(G) = \{a_1 \cdots a_n \mid S \xrightarrow{*} a_1 \cdots a_n, S \text{ is the starting symbol of } G, a_1, \ldots, a_n \text{ are terminal in } G\}$$
2. Reason Functional Programming (16 points)

Consider the function dfa_accepts for determining if a string is in the language of a DFA. Recall that a DFA has no epsilon transitions. Recall that a DFA never has two edges leaving the same state with the same label going to different destination states. For example, consider the DFA accepting the regular language denoted by the regular expression $c \mid a(aa)^*b$

let edges = [ ("q0", "a", "q1"), /* in state q0, on a, goto q1 */ ("q0", "c", "q2"), /* in state q0, on c, goto q1 */ ("q1", "b", "q2"), /* in state q1, on b, goto q2 */ ("q1", "a", "q0") ]; /* in state q1, on a, goto q0 */
let final = [ "q2" ];
let start = "q0";
List.iter (fun input => {
    Printf.printf "%s : %b\n" input (dfa_accepts start edges final input)
}) [ "a", "ab", "c", "aab", "aaab" ];

let mem_test = List.mem 3 [1,2,3,4,5,6];
let mem_test2 = List.mem 3 [1,2,4,5,6];
let sub = String.sub "abcdefg" 4 2;

Printf.printf "mem: %b, mem2: %b, sub: %s
" mem_test mem_test2 sub;

This yields the following output:

a : false
ab : true
c : true
aab : false
aaab : true
mem: true, mem2: false, sub: ef

Complete the following function by filling in each blank with a single identifier, keyword, operator or constant. You must write a correct Reason program.

let rec dfa_accepts = fun state edges final input => {
    switch (input) {
        | "" => List.mem state final
        | s => {
            let destination_list = List.filter (fun (src, symb, dest) => {
                src == state &&
                symb == (String.sub input (or s) 0 1)
            }) edges;
            switch (destination_list) {
                | [ (__,__,new_state) ] => {
                    let new_input = String.sub input 1 ((String.length input) - 1);
                    dfa_accepts new_state edges final new_input
                }
                | _ => false
            }
        }
    }
};


3. Higher-Order Functions (20 points)

For this question, do your best to adhere to Reason (preferable) or OCaml syntax. Consider the following function fold_left, which combines elements of a list into a single value:

```ocaml
let rec fold_left = fun (f : 'a => 'b => 'a) (accumulator : 'a) (lst : list 'b) : 'a => { switch (lst) { | [] => accumulator 
| [hd, ...tl] => fold f (f accumulator hd) tl 
};
};
```

(a) (10 points) In the space below and using the fold_left function, implement the function map, which takes a function f and a list as input and return a new list l' where f has been applied to each element of l. Elements should appear in the same order in l and l' (i.e., element e from l and element (f e) should appear at the same offset in the list). Use of fold_left is required for full credit. You may not use library functions.

```ocaml
let map = fun (f : 'a => 'b) (l : list 'a) : list 'b => { let reversed_list = fold (fun l i => [i, ...l]) [] l; fold (fun l i => [(f i), ...l]) [] reversed_list
};
```

(b) (10 points) In the space below and using the fold_left function, implement the function filter, which takes a function f and a list as input and returns a new list l'. The new list l' contains the elements from l (in any order) for which f applied to the element returns true. Use of fold_left is required for full credit. You may not use library functions.

```ocaml
let filter = fun (f : 'a => bool) (l : list 'a) : list 'a => { let reversed_list = fold (fun l i => [i, ...l]) [] l; fold (fun l i => [(f i), ...l]) [] reversed_list
};
```
4. Regular Expressions and Automata (21 points)

For this question, regular expressions may be single character (a), epsilon (ε), concatenation (AB), union (A|B), Kleene star (A*), plus (A+), and option (A?).

(a) (7 points) Write a regular expression (over the alphabet Σ = {a, b, c, d}) for the language of strings in which all of the letters are in order, there are an even number of occurrences of a and c, and there are an odd number of occurrences of b and d. Use at most 25 symbols in your answer.

(aa)*b(bb)*(cc)*d(dd)*

(b) (7 points) Draw an NFA (or DFA) that accepts the language from the above problem. Use at most fifteen states in your answer.

![NFA Diagram]

(c) (7 points) Consider the following NFA which operates over the alphabet Σ = {a, b, c}:

![NFA Diagram]

Draw an equivalent DFA. Use no more than six states total.

![DFA Diagram]
5. Ambiguity (16 points)

Consider the following grammar $G_1$:

$$
S \rightarrow E \\
E \rightarrow \text{if } E \text{ then } E \text{ else } E \\
E \rightarrow \text{if } E \text{ then } E \\
E \rightarrow \text{true} \mid \text{false} \\
E \rightarrow E \text{ or } E
$$

(a) (6 points) Show that this grammar is ambiguous using the string:

“if true then if false then false else true”

(b) (10 points) Rewrite the grammar to eliminate left recursion. That is, provide a grammar $G_2$ such that $L(G_1) = L(G_2)$ but $G_2$ admits no derivation $X \rightarrow^* X\alpha$.

$$
S \rightarrow E \\
E \rightarrow \text{if } E \text{ then } E \text{ else } E \text{ N} \\
E \rightarrow \text{if } E \text{ then } E \text{ N} \\
E \rightarrow \text{true} \text{ N} \mid \text{false} \text{ N} \\
N \rightarrow \text{or } E \text{ N} \mid \varepsilon
$$
6. **Interpreter Stages (12 points)**

The following diagram shows the stages of an interpreter. Label each of the nine unlabeled elements. Unlabeled elements may be: a *generating tool* used in interpreter construction, a *representation of the subject program*, a *stage* of the interpreter, or a *formalism* used to guide or generate a stage of the interpreter. Interpreter stages (gray) are worth two points each; all other blanks are worth one point each.
Extra Credit (at most 3 points)

(a) What do you like about this class so far? What would you like me to keep doing?

(b) What do you dislike about this class? Is there anything you would suggest changing?

(c) Give two additional use cases for regular expressions (outside of lexical analysis).