Q2. NFA and DFA

Q2.1. Consider the NFA given below. Is this NFA also a DFA?

Yes/No

Q2.2. Which strings will be accepted by the following NFA?

- aaabbb
- aa
- aaaaab
- bbaaa
Q3. NFA to DFA

Consider the following NFA:

When converted to a DFA using the subset construction algorithm from Project 3, we get the following DFA:

Where X, Y and Z are states you'll have to fill in.

Q3.1. In this DFA, which states from the original NFA make up the state X? 0, 1, 2, 3
Q3.2. In this DFA, which states from the original NFA make up the state Y? 0, 1, 2, 3
Q3.3. In this DFA, which states from the original NFA make up the state Z? 0, 1, 2, 3
Q3.4. Which state(s) in the new DFA are final? X, Y, Z
Q3.5. Provide a regex for the NFA / DFA:
Q4. CFG

To represent $\epsilon$ in the CFG, you can either copy and paste the symbol $\epsilon$, type the word `epsilon` or just type the letter e.

Q4.1. Define a CFG that describes the language

$$a^x b^y c^z \text{ where } z \leq x + 2y.$$

Q4.2. Given the following ambiguous CFG, modify it so that it produces the same strings but is not ambiguous.

$$S \rightarrow SaS \mid T$$
$$T \rightarrow bT \mid V$$
$$V \rightarrow c \mid \epsilon$$

Q4.3. Is the below CFG right recursive?

$$S \rightarrow N + S \mid N \ast S \mid N$$
$$N \rightarrow 1 \mid (S)$$

Yes/No

Q5. Can it be parsed?

Indicate if each of the following grammars can be parsed by a recursive descent parser. If not, choose the reason for why it cannot.

Q5.1. Can the below grammar be parsed by a recursive-descent parser?

$$S \rightarrow S \ast S \mid T$$
$$T \rightarrow 1 \mid 2 \mid 3 \mid (S)$$

- Yes
- No, because the grammar is ambiguous i.e., it has more than one leftmost derivation
- No, because the grammar is left recursive
- No, because the grammar is ambiguous and left recursive

Q5.2. Can the below grammar be parsed by a recursive-descent parser?

$$S \rightarrow cS \mid A$$
$$A \rightarrow aA \mid \epsilon$$

- Yes
- No, because the grammar is ambiguous i.e., it has more than one leftmost derivation
- No, because the grammar is left recursive
- No, because the grammar is ambiguous and left recursive
Q5.3. Can the below grammar be parsed by a recursive-descent parser?

\[
S \rightarrow Sa \mid U \\
U \rightarrow Uu \mid \epsilon
\]

- Yes
- No, because the grammar is ambiguous i.e., it has more than one leftmost derivation
- No, because the grammar is left recursive
- No, because the grammar is ambiguous and left recursive

Q6. Writing a Parser

Note: For your reference, we have included the non-imperative definitions for the helper functions you will need to implement the parser.

```
let lookahead toks = match toks with
    | [] -> failwith "no more tokens!"
    | h::_ -> h

let match_token tok toks = match toks with
    | h::t when h = tok -> t
    | _ -> failwith "match error!"
```

Consider the following grammar.

\[
\begin{align*}
\text{Exp} & \rightarrow \text{IfZero} \mid \text{N} \\
\text{IfZero} & \rightarrow \text{ifzero N then Exp else Exp} \\
\text{N} & \rightarrow 0 \mid 1
\end{align*}
\]

We are assuming that a working lexer (or tokenizer) exists and can convert string input into a list of tokens (similar to Project 4a). The goal is to implement a non-imperative recursive-descent parser to parse the grammar described above. To do so, we will define our tokens and the corresponding AST as follows:

```
type token =
    | Tok_ifzero
    | Tok_then
    | Tok_else
    | Tok_0
    | Tok_1

type expr =
    | Num of int
    | IfZero of expr * expr * expr
```

Examples:

```
"0"     | > tokenizer | > parse_Exp
(*) Num(0) *)

"ifzero 0 then 1 else 0" | > tokenizer | > parse_Exp
(*) IfZero(Num(0), Num(1), Num(0)) *)
```
"ifzero 0 then ifzero 1 then 0 else 1 else 0" |> tokenizer |> parse_Exp
(* IfZero(Num(0), IfZero(Num(1), Num(0), Num(1)), Num(0)) *)

**Notes:**
- `parse_Exp` must return type `token list * expr`.
- You don't have to check if the list is empty at the end of parsing.
- You can use `failwith` to handle exceptions.

```ocaml
let rec parse_Exp tok =

and parse_IfZero tok =

and parse_N tok =
```

**Q7. Operational Semantics**

**Q7.1.** What is the difference between lexical/static and dynamic scoping in OpSem?
- Static scoping is for closures and dynamic scoping is for hypotheses.
- Static scoping evaluates a closure with respect to the existing environment, dynamic scoping evaluates a closure on its own.
- Static scoping evaluates the environment from left to right, dynamic scoping evaluates the environment from right to left.

**Q7.2.** Consider the following semantics that uses a mystery `magic` operator `?`.

```
\[
\frac{A; \ e_1 \Rightarrow v_1}{A; \ e_2 \Rightarrow (A', \lambda x. \ e) \ A', \ x : v_1; \ e \Rightarrow v_2}
\]
```

Describe what this `magic` operator does.

**Hint:** Recall closures from OCaml.
Q7.3. Using the given rules, fill in the blanks the complete the derivation below:

\[
\frac{\frac{A(x) = v}{A; \ x \Rightarrow v}}{A; \ n \Rightarrow n}
\]

\[
\frac{A; \ e_1 \Rightarrow v_1}{v_2 \text{ is true if } v_1 \text{ is 0, otherwise } v_2 \text{ is false}}
\]

\[
\frac{A; \ e_2 \Rightarrow v_1}{A; \ \text{equals0} \ e_1 \Rightarrow v_2}
\]

\[
\frac{A; \ e_1 \Rightarrow \text{true}}{A; \ e_2 \Rightarrow v_1} \quad \frac{A; \ e_1 \Rightarrow \text{false}}{A; \ e_3 \Rightarrow v_1}
\]

\[
\frac{A; \ \text{if} \ e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow v_1}{A; \ e_1 + e_2 \Rightarrow v_3}
\]

\[
\frac{(\#3)}{A, (\#1); \ x \Rightarrow 5 \text{ true if 5 is 0, otherwise false}} \quad \frac{(\#3)}{A, (\#1); \ x \Rightarrow 5 \text{ true if 5 is 0, otherwise false}}
\]

\[
\frac{A, (\#1); \ equals0 \ x \Rightarrow (\#2)}{A, (\#1); \ equals0 \ x \Rightarrow (\#2)} \quad \frac{A, (\#1); \ 5 \Rightarrow 5}{A, (\#1); \ 5 \Rightarrow 5}
\]

\[
\frac{A, (\#1); \ \text{if equals0} \ x \text{ then } 21 \text{ else } x + 5 \Rightarrow (\#6)}{A, (\#1); \ \text{if equals0} \ x \text{ then } 21 \text{ else } x + 5 \Rightarrow (\#6)}
\]

Notes:
- If (\#5) is not visible, please scroll to the right to ensure the entire LaTeX is visible.
- The blanks refer to the part of derivation (judgement/hypothesis) that should exist in the position of the blank.

Blank #1:
Blank #2:
Blank #3:
Blank #4:
Blank #5:
Blank #6:

Q8. Lambda Calculus

To represent λ, you may either copy and paste the symbol λ or just type the characters L or \ in your solutions.

Q8.1. Which of the following are free variables in the lambda calculus expression?

\[
\lambda a. \ b \ \lambda y. \ y \ x \ \lambda p. \ p \ y
\]

- a
- b
- y
- x
- p
Q8.2. Consider the following lambda calculus expression,

\((\lambda x. \, \lambda y. \, x \, y \, \lambda x. \, x \, y) \, (\lambda z. \, z) \, (\lambda z. \, w)\)

Make parentheses explicit in the above expression.

Give a valid \(\alpha\)-conversion for the expression.

Q8.3. Reduce the following lambda calculus expression to the \(\beta\)-normal form using both CBN and CBV.

\((\lambda x. \, (\lambda y. \, y \, a) \, x) \, ((\lambda x. \, x) \, (\lambda y. \, y \, b))\)

Show each step, including any \(\beta\)-reduction or \(\alpha\)-conversion. If there is infinite recursion, write "Infinite Recursion".

**Call-by-name:**

**Call-by-value:**

Q8.4. Consider the following encodings,

\(\text{true} = (\lambda x. \, \lambda y. \, x)\)
\(\text{false} = (\lambda x. \, \lambda y. \, y)\)
\(\text{not} = (\lambda x. \, x \, \text{false} \, \text{true})\)
\(\text{or} = (\lambda x. \, \lambda y. \, x \, \text{true} \, y)\)

Prove that \(\text{not} \, (\text{or} \, \text{false} \, \text{true}) = \text{false}\)

**Hint:** Replace the bindings for their lambda-calculus expressions and show that the left side reduces to \(\text{false}\), which is \((\lambda x. \, \lambda y. \, y)\).