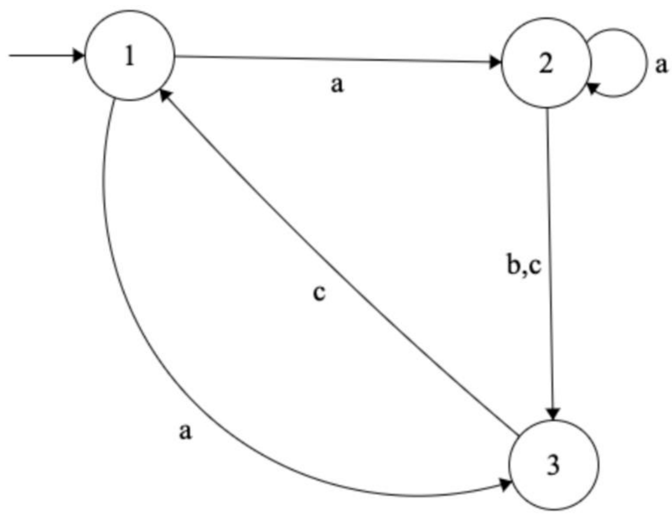
CMSC 330 Exam 2 Spring 2022 Solutions

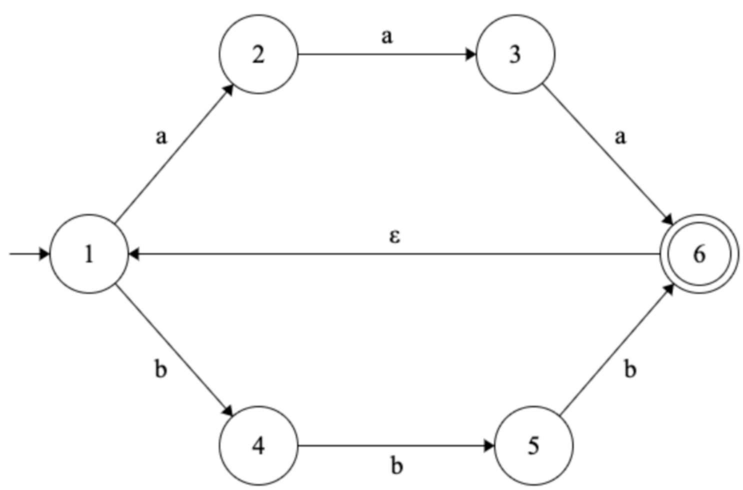
Q2. NFA and DFA

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



Yes/No

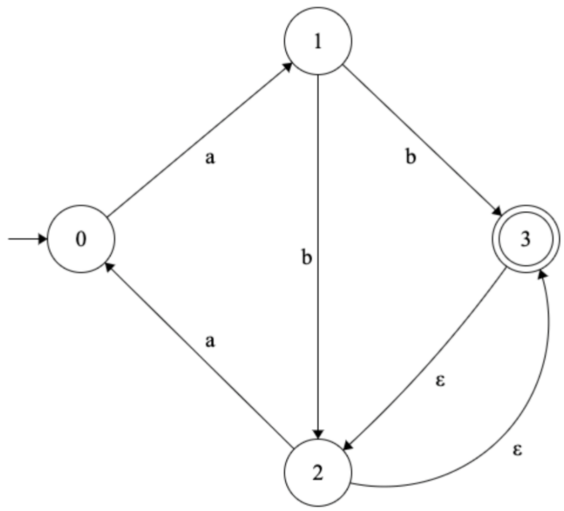
Q2.2. Which strings will be accepted by the following NFA?



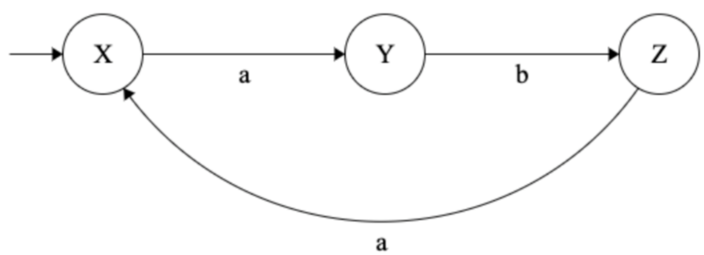
- aaabbb
- aa
- aaaaab
- bbbaaa

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:

ab(aab)*

Q4. CFG

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

Q4.1. Define a CFG that describes the language

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

```
S -> aSU | T
T -> bTUU | ϵ
U -> c | ϵ
```

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

$$\begin{aligned} S &\rightarrow SaS \mid T \\ T &\rightarrow bT \mid V \\ V &\rightarrow c \mid \epsilon \end{aligned}$$

Rewrite: `S -> TaS | T`

Q4.3. Is the below CFG right recursive?

$$\begin{aligned} S &\rightarrow N + S \mid N * S \mid N \\ N &\rightarrow 1 \mid (S) \end{aligned}$$

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?

$$\begin{aligned} S &\rightarrow S * S \mid T \\ T &\rightarrow 1 \mid 2 \mid 3 \mid (S) \end{aligned}$$

- 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

Partial credit for options 2 and 3.

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

$$\begin{aligned} S &\rightarrow cS \mid A \\ A &\rightarrow aA \mid \epsilon \end{aligned}$$

- 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.

$$\mathrm{Exp} \rightarrow \mathrm{IfZero} \mid \mathrm{N}$$
$$\mathrm{IfZero} \rightarrow \texttt{ifzero}\ \mathrm{N}\ \texttt{then}\ \mathrm{Exp}\ \texttt{else}\ \mathrm{Exp}$$
$$\mathrm{N} \rightarrow 0 \mid 1$$

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 recurive-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.

```
let rec parse_Exp toks =
  match lookahead toks with
  | Tok_ifzero -> parse_IfZero toks
  | Tok_0 | Tok_1 -> parse_N

and parse_IfZero toks =
  match lookahead toks with
  | Tok_ifzero -> let toks = match_token Tok_ifzero toks in
      let e, toks = parse_N toks in
      let toks = match_token Tok_then toks in
      let e’, toks = parse_Exp toks in
      let toks = match_token Tok_else toks in
      let e’’, toks = parse_Exp toks in
      (toks, IfZero(e, e’, e’’))
  | _ -> failwith “error”

and parse_N toks =
  match lookahead toks with
  | Tok_0 -> let toks = match_token Tok_0 toks in (toks, Num(0))
  | Tok_1 -> let toks = match_token Tok_1 toks in (toks, Num(1))
  | _ -> failwith “error”
```

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 ?.

$$\frac{\dfrac{}{A;\ e_1 \Rightarrow v_1} \qquad \dfrac{A;\ e_2 \Rightarrow (A', \lambda x.\ e) \qquad A', x : v_1;\ e \Rightarrow v_2}{A;\ e_2\ v_1 \Rightarrow v_2}}{A;\ e_1\ ?\ e_2 \Rightarrow v_2}$$

Describe what this **magic** operator does.

Hint: Recall closures from OCaml.

? applies the value of `e1` to the function `e2` OR ? is the pipeline operator |> from OCaml.

Q7.3. Using the given rules, fill in the blanks the complete the derivation below:

$$\frac{}{A;\ \mathrm{n} \Rightarrow \mathrm{n}} \qquad \frac{A(\mathrm{x}) = \mathrm{v}}{A;\ \mathrm{x} \Rightarrow \mathrm{v}}$$

$$\frac{A;\ \mathrm{e}_1 \Rightarrow \mathrm{v}_1 \qquad \mathrm{v}_2 \text{ is true if } \mathrm{v}_1 \text{ is } 0\text{, otherwise } \mathrm{v}_2 \text{ is false}}{A;\ \text{equals0 } \mathrm{e}_1 \Rightarrow \mathrm{v}_2}$$

$$\frac{A;\ \mathrm{e}_1 \Rightarrow \text{true} \qquad A;\ \mathrm{e}_2 \Rightarrow \mathrm{v}_1}{A;\ \text{if } \mathrm{e}_1 \text{ then } \mathrm{e}_2 \text{ else } \mathrm{e}_3 \Rightarrow \mathrm{v}_1} \qquad \frac{A;\ \mathrm{e}_1 \Rightarrow \text{false} \qquad A;\ \mathrm{e}_3 \Rightarrow \mathrm{v}_1}{A;\ \text{if } \mathrm{e}_1 \text{ then } \mathrm{e}_2 \text{ else } \mathrm{e}_3 \Rightarrow \mathrm{v}_1}$$

$$\frac{A;\ \mathrm{e}_1 \Rightarrow \mathrm{v}_1 \qquad A;\ \mathrm{e}_2 \Rightarrow \mathrm{v}_2 \qquad \mathrm{v}_3 \text{ is } \mathrm{v}_1 + \mathrm{v}_2}{A;\ \mathrm{e}_1 + \mathrm{e}_2 \Rightarrow \mathrm{v}_3}$$

$$\dfrac{\dfrac{\dfrac{(\#3)}{A,\ (\#1);\ \mathrm{x} \Rightarrow 5} \qquad \text{true if 5 is 0, otherwise false}}{A,\ (\#1);\ \text{equals0 x} \Rightarrow (\#2)} \qquad \dfrac{\dfrac{(\#3)}{A,\ (\#1);\ \mathrm{x} \Rightarrow 5} \qquad \dfrac{}{A,\ (\#1);\ 5 \Rightarrow 5} \qquad (\#5)}{A,\ (\#1);\ (\#4) \Rightarrow (\#6)}}{A,\ (\#1);\ \text{if equals0 x then 21 else x} + 5 \Rightarrow (\#6)}$$

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: `x:5`
Blank #2: `false`
Blank #3: `A,x:5(x) = 5`
Blank #4: `x + 5`
Blank #5: `10 is 5 + 5`
Blank #6: `10`

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?

`λa. b λy. y x λp. p y`

- a
- b
- y
- x
- p

Q8.2. Consider the following lambda calculus expression,

```
(λx. y λy. x y λx. x y) (λz. z) (λz. w)
```

Make parentheses explicit in the above expression.

```
((λx. (y (λy. ((x y) (λx. (x y)))))) (λz. z)) (λz. w)
```

Give a valid α-conversion for the expression.

```
(λx. y λm. x m λn. n m) (λz. z) (λz. w)
```

Q8.3. Reduce the following lambda calculus expression to the β-normal form using both CBN and CBV.

```
(λx. (λy. y a) x) ((λx. x) (λy. y b))
```

Show each step, including any β-reduction or α-conversion. If there is infinite recursion, write "Infinite Recursion".

Call-by-name:

```
(λx. (λy. y a) x) ((λx. x) (λy. y b))
= (λy. y a) ((λx. x) (λy. y b))
= ((λx. x) (λy. y b)) a
= (λy. y b) a
= a b
```

Call-by-value:

```
(λx. (λy. y a) x) ((λx. x) (λy. y b))
= (λx. (λy. y a) x) (λy. y b)
= (λy. y a) (λy. y b)
= (λy. y b) a
= a b
```

Q8.4. Consider the following encodings,

```
true = (λx. λy. x)
false = (λx. λy. y)
not = (λx. x false true)
or = (λx. λy. x true y)
```

Prove that `not (or false true) = false`

Hint: Replace the bindings for their lambda-calculus expressions and show that the left side reduces to `false`, which is `(λx. λy. y)`.

```
not (or false true)
= not ((λx. λy. x true y) false true)
= not (false true true)
= not ((λx. λy. y) true true)
= not ((λy. y) true)
= not (true)
= (λx. x false true) true
= true false true
= (λx. λy. x) false true
= (λy. false) true
= false
```