Lab 8: Formal Grammars

```
<a> ::= <b> <c>
<b> ::= <b> y | x
<c> ::= <d>   | <d> y
<d> ::= z <b> | z
```

1. Give a leftmost derivation for `xyz`.
2. Draw a parse tree for `xyz`.
3. Find a sentence of < 5 symbols with more than one parse tree. Draw the two trees.

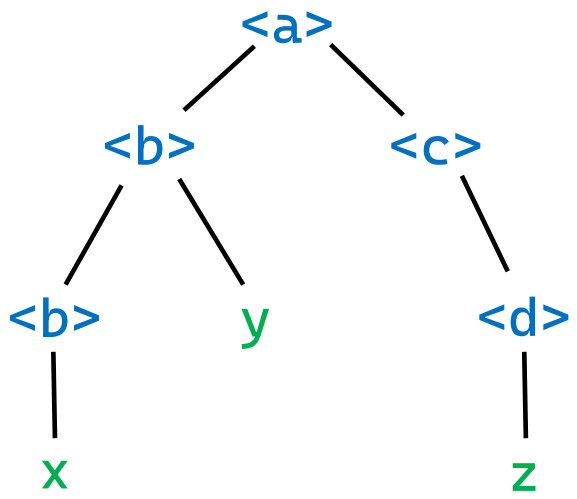
Design an unambiguous grammar for Python boolean expressions, namely values `True` and `False` and operators `and`, `or`, and `not`.

Challenge: implement an *ocamllex* lexer and *menhir* parser for the above grammar.

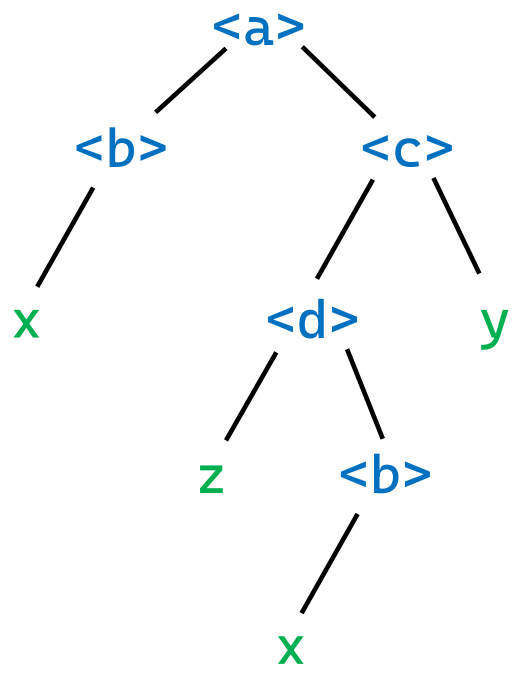
```
<a> ::= <b> <c>
<b> ::= <b> y | x
<c> ::= <d>   | <d> y
<d> ::= z <b> | z
```

1. Give a leftmost derivation for xyz.
2. Draw a parse tree for xyz.

```
<a>
<b> <c>
<b> y <c>
x y <c>
x y <d>
x y z
```

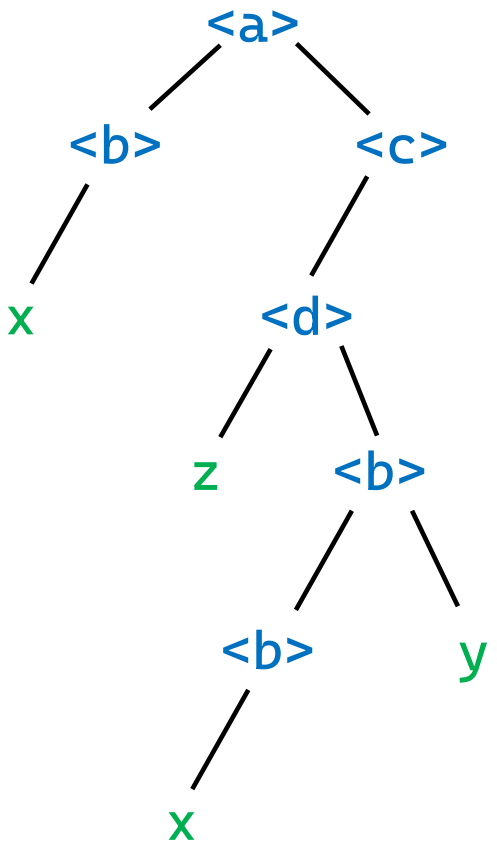


```
<a> ::= <b> <c>
<b> ::= <b> y | x
<c> ::= <d>   | <d> y
<d> ::= z <b> | z
```



3. Find the shortest sentence with more than one parse tree. Draw the two trees.

xzxy



Design an unambiguous grammar for Python boolean expressions, namely values `True` and `False` and operators `and`, `or`, and `not`.

```
  <bexp> ::= <eor>
   <eor> ::= <eor>  or  <eand> | <eand>
  <eand> ::= <eand> and <enot> | <enot>
  <enot> ::= not <enot>        | <econst>
<econst> ::= True | False
```

Compare with Python implementation:

```
or_test  ::= and_test | or_test "or" and_test
and_test ::= not_test | and_test "and" not_test
not_test ::= comparison | "not" not_test
```

lexer.mll

```
{ open Parser }

let whitespace = [' ' '\t' '\n' '\r']+

rule read =
  parse
  | "True"  { CONST true }
  | "False" { CONST false }
  | "and" { AND } | "or"  { OR }
  | "not" { NOT }
  | whitespace { read lexbuf }
  | eof { EOF }
```

ast.ml

```
type expr =
  | Const of bool
  | And of expr * expr
  | Or of expr * expr
  | Not of expr

type prog = expr
```

parser.mly

```
%{ open Ast %}

%token<bool> CONST
%token NOT AND OR
%token EOF

%start <Ast.prog> prog

%%

prog: e = bexp; EOF { e }
bexp: e = eor { e }

eor: e = eand; { e }
  | e1 = eor; OR; e2 = eand { Or (e1,e2) }

eand: e = enot; { e }
  | e1 = eand; AND; e2 = enot { And (e1,e2) }

enot: e = const; { e }
  | NOT; e = enot { Not e }

const: b = CONST { Const b }
```

lexer.mll

```
{ open Parser }

let whitespace = [' ' '\t' '\n' '\r']+

rule read =
  parse
  | "True"  { CONST true }
  | "False" { CONST false }
  | "and" { AND }  | "or"  { OR }
  | "not" { NOT }
  | "(" { LPAREN } | ")" { RPAREN }
  | whitespace { read lexbuf }
  | eof { EOF }
```

ast.ml

```
type expr =
  | Const of bool
  | And of expr * expr
  | Or of expr * expr
  | Not of expr

type prog = expr
```

parser.mly

```
%{ open Ast %}

%token<bool> CONST
%token NOT AND OR
%token EOF
%token LPAREN RPAREN

%left OR
%left AND
%left NOT

%start <Ast.prog> prog

%%

prog: e = bexp; EOF { e }

bexp:
  | e1 = bexp; OR;  e2 = bexp { Or  (e1, e2) }
  | e1 = bexp; AND; e2 = bexp { And (e1, e2) }
  | NOT; e = bexp; { Not e }
  | LPAREN; e = bexp; RPAREN { e }
  | b = CONST { Const b }
```