Lab 9: Operational Semantics

Perform the following substitutions. Capture-avoid where necessary! Choose any new variable names to preserve α-equivalence.

1. $[(\lambda x. x)/y](\lambda x. \lambda x. y)$

2. $[(\lambda x. y)/z](\lambda x. \lambda y. z)$

3. $[(\lambda x. x)/y](\lambda y. \lambda x. x)$

4. $[(\lambda x. x)/f]\big(\lambda y. \lambda x. f\,(yx)\big)$

5. $[(\lambda z. z)/y][(\lambda x. y)/x](\lambda y. \lambda x. x)$

6. $[(\lambda y. x\ y)/z](\lambda y. \lambda x. z\ x\ (\lambda z. y\ z)\ z)$

Consider this (ambiguous) grammar for Python boolean expressions

```
<expr> ::= <expr> and <expr>
         | <expr> or  <expr>
         | not <expr>
         | True   |  False
```

With this associativity and precedence:

`or`	left	increasing precedence ↓
`and`	left	
`not`	N/A	

Write down both **small-** and **big-step semantics** for these expressions. Operands evaluate **left-to-right**. Make sure to enable short-circuiting of `and` and `or`.
With extra time: implement an evaluator for these expressions in OCaml.

Lab 9: Operational Semantics

Perform the following substitutions. Capture-avoid where necessary! Choose any new variable names to preserve α-equivalence.

1. $[(\lambda x.x)/y](\lambda x.\lambda x.\boxed{y})$

   $\lambda x.\lambda x.(\lambda x.x)$

2. $[(\lambda x.y)/z](\lambda x.\lambda y.\boxed{z})$

   $\lambda x.\lambda \boxed{y}.(\lambda x.\boxed{y}) \longrightarrow \lambda x.\lambda z.(\lambda x.y)$

3. $[(\lambda x.x)/y](\lambda y.\lambda x.x)$ *no free y*

   $\lambda y.\lambda x.x$

4. $[(\lambda x.x)/f](\lambda y.\lambda x.\boxed{f}(yx))$

   $\lambda y.\lambda x.(\lambda x.x)(yx)$

5. $[(\lambda z.z)/y][(\lambda x.y)/x](\lambda y.\lambda x.x)$ *no free x*

   $[(\lambda z.z)/y](\lambda y.\lambda x.x)$ *no free y* $\longrightarrow \lambda y.\lambda x.x$

6. $[(\lambda y.x\ y)/z](\lambda y.\lambda x.\boxed{z}\ x\ (\lambda z.y\ z)\ \boxed{z})$

   $\lambda y.\lambda \boxed{x}.(\lambda y.\boxed{x}\ y)\ \boxed{x}\ (\lambda z.y\ z)\ (\lambda y.\boxed{x}\ y) \longrightarrow \lambda y.\lambda u.(\lambda y.x\ y)\ u\ (\lambda z.y\ z)\ (\lambda y.x\ y)$

Consider this (ambiguous) grammar for Python boolean expressions

```
<expr> ::= <expr> and <expr>
         | <expr> or  <expr>
         | not <expr>
         | True   |  False
```

With this associativity and precedence:

`or`	left	increasing precedence ↓
`and`	left	
`not`	N/A	

Write down both **small- and big-step semantics** for these expressions. Operands evaluate **left-to-right**. Make sure to enable short-circuiting of `and` and `or`.
With extra time: implement an evaluator for these expressions in OCaml.

Big-step

$$\frac{}{\texttt{True} \Downarrow \top} \qquad \frac{e \Downarrow \bot}{\texttt{not}\ e \Downarrow \top} \qquad \frac{e_1 \Downarrow \top \quad e_2 \Downarrow v}{e_1\ \texttt{and}\ e_2 \Downarrow v} \qquad \frac{e_1 \Downarrow \bot \quad e_2 \Downarrow v}{e_1\ \texttt{or}\ e_2 \Downarrow v}$$

$$\frac{}{\texttt{False} \Downarrow \bot} \qquad \frac{e \Downarrow \top}{\texttt{not}\ e \Downarrow \bot} \qquad \frac{e_1 \Downarrow \bot}{e_1\ \texttt{and}\ e_2 \Downarrow \bot} \qquad \frac{e_1 \Downarrow \top}{e_1\ \texttt{or}\ e_2 \Downarrow \top}$$

Small-step

$$\frac{e \to e'}{\texttt{not}\ e \to \texttt{not}\ e'} \qquad \frac{e_1 \to e_1'}{e_1\ \texttt{and}\ e_2 \to e_1'\ \texttt{and}\ e_2} \qquad \frac{e_1 \to e_1'}{e_1\ \texttt{or}\ e_2 \to e_1'\ \texttt{or}\ e_2}$$

$$\frac{}{\texttt{not True} \to \texttt{False}} \qquad \frac{}{\texttt{False and}\ e \to \texttt{False}} \qquad \frac{}{\texttt{True or}\ e \to \texttt{True}}$$

$$\frac{}{\texttt{not False} \to \texttt{True}} \qquad \frac{}{\texttt{True and}\ e \to e} \qquad \frac{}{\texttt{False or}\ e \to e}$$

Consider this (ambiguous) grammar for Python boolean expressions

```
<expr> ::= <expr> and <expr>
         | <expr> or  <expr>
         | not <expr>
         | True  |  False
```

With this associativity and precedence:

`or`	left
`and`	left
`not`	N/A

increasing precedence

Write down both **small- and big-step semantics** for these expressions. Operands evaluate **left-to-right**. Make sure to enable short-circuiting of `and` and `or`.

Implement an evaluator in OCaml.

```ocaml
type expr =
  | And of expr * expr
  | Or  of expr * expr
  | Not of expr
  | True | False
```

```ocaml
let rec eval (e : expr) : bool =
  match e with
  | True    -> true
  | False   -> false
  | Not e1  -> not (eval e1)
  | And (e1,e2) -> eval e1 && eval e2
  | Or  (e1,e2) -> eval e1 || eval e2
```

These built-in operators short circuit!

Consider this (ambiguous) grammar for Python boolean expressions

```
<expr> ::= <expr> and <expr>
         | <expr> or  <expr>
         | not <expr>
         | True  |  False
```

With this associativity and precedence:

or	left
and	left
not	N/A

increasing precedence

Implement a lexer and parser for s-expressions.
(Refresh your repo to get the skeleton)

more lax than before

```
()     5     (() 2 hello)

(f (g) h (i jkl m))
```

```
<sexpr> ::= <atom> | (<sexpr> ⋯ <sexpr>)
 <atom> ::= ⋯
```

lexer.mll

```
{ open Parser }

let whitespace = [' ' '\t' '\n' '\r']+
let atom = [^ ' ' '\t' '\n' '\r' '(' ')']+

rule read = parse
  | '(' { LPAREN }
  | ')' { RPAREN }
  | whitespace { read lexbuf }
  | atom { ATOM (Lexing.lexeme lexbuf) }
  | eof { EOF }
```

parser.mly

```
%{ open Utils %}

%token<string> ATOM
%token LPAREN RPAREN
%token EOF

%start <string Utils.sexpr> sexpr

%%

sexpr: e=expr; EOF { e }

expr:
  | a=ATOM { Atom a }
  | LPAREN; ss=expr*; RPAREN { List ss }
```