op ::= + | * | …

\[ e ::= n \quad | \quad \text{true} \quad | \quad \text{false} \quad | \quad \text{(op e e)} \quad | \quad \text{(if e e e)} \]

\[ \text{(if false 3 (+ 4 5))} \]
“(if false
3
(+ 4 5))”

([if, false, 3, [+ 4 5]])

IfE (BoolE False)
  (NumE 3)
  (OpE Add (NumE 4) (NumE 5))
\((* \ (\ + \ 1 \ 2) \ (\ + \ 1 \ 2))\)
(with [x (+ 1 2)]
   (* (+ 1 2) (+ 1 2)))
(with [x (+ 1 2)]
  (* x x))
op ::= + | * | ...

e ::= n
| true | false
| (op e e)
| (if e e e)

data Op = Add | Mul | ...
data Expr = NumE Int
| BoolE Bool
| OpE Op Expr Expr
| IfE Expr Expr Expr Expr
op ::= + | * | ...

\[ e ::= n \]
  \[ | \text{true} | \text{false} \]
  \[ | (\text{op} \ e \ e) \]
  \[ | (\text{if} \ e \ e \ e) \]
  \[ | x \]
  \[ | (\text{with} \ [x \ e] \ e) \]

data Op = Add | Mul | ...
data Expr = NumE Int
  \[ | \text{BoolE} \ Bool \]
  \[ | \text{OpE} \ Op \ Expr \ Expr \]
  \[ | \text{IfE} \ Expr \ Expr \ Expr \]
  \[ | \text{VarE} \ Var \]
  \[ | \text{WithE} \ Var \ Expr \ Expr \]
(with [x (+ 1 2)]
  (* x x))
=> 9
(with [x (+ 1 x)]
  (* x x))
=> <error>
(with [x (+ 1 2)]
  (+ x (with [y (* x x)]
        (+ x y))))
=> 15
(with [x (+ 1 2)]
  (+ x (with [x (* x x)]
         (+ x x))))

=> 21
(with [x (+ y 2)]
  (+ x 4))
=> <error>
(with [x (+ y 2)]
    (with [y 4]
        (+ x y)))
=> <error>

interp :: Expr -> Expr
interp expr = case expr of
  NumE _ -> expr

  ... 

  WithE var boundExpr body ->
    interp (subst var (interp boundExpr) body)
  VarE var -> error (var ++ " : unbound")
  FunE _ _ -> expr
  AppE fun arg -> case interp fun fun of
    FunE var body ->
      interp (subst var (interp arg) body)
data Expr = NumE Int | ...  
| OpE Op Expr Expr 
| IfE Expr Expr Expr 
| VarE Var 
| WithE Var Expr Expr 

subst :: Var -> Expr -> Expr -> Expr
subst v boundExpr body =  
  let recur = subst v boundExpr in
  case body of
    NumE _ -> body
    ...
    VarE v' | v == v' -> boundExpr
    | otherwise -> body
    WithE v' boundExpr' body'
    | v == v'   -> WithE v' (recur boundExpr') body'
    | otherwise -> WithE v' (recur boundExpr') (recur body')
    FunE v' body' | v == v' -> body
    | otherwise -> Fun v' (recur body)
    AppE fun arg -> AppE (recur fun) (recur arg)
(+ (* 3 3) (* 4 4))
(with [sqr (fun (x) (* x x))]
  (+ (* 3 3) (* 4 4)))
(with [sqr (fun (x) (* x x))]
  (+ (sqr 3) (sqr 4)))
((fun (x) (* x x))
  (+ 1 2))
=> 9
((fun (f) (f (f 3))))

(fun (x) (+ x 2)))

=> 7
\[
((\text{fun } (x) \\
\quad (\text{fun } (f) (f (f x))))))
\]

2)

\[
\Rightarrow (\text{fun } (f) (f (f 2)))
\]
(+ 3 (fun (x) (+ x x)))
=> <error>
(with [f (fun (x) (+ x y))]  
  (with [y 5]  
    (f y)))  
=> <error>
(with [y 2]
  (with [f (fun (x) (+ x y))]
    (with [y 5]
      (f y))))
=> 7
(((fun (x) (* x x))
 (+ 1 2))

(with [x (+ 1 2)]
 (* x x)))
((fun (x) <body>))
<arg>)

(with [x <arg>]
<body>)
(with [x 2]
  (with [y 3]
    (+ (* (+ 3 4) (+ 1 2))
      (+ (* 2 x) (* y 3))))))
type Env = [(Var, Expr)]

emptyEnv = []

extendEnv var val env =

lookup :: Eq a => a -> [(a, b)] -> Maybe b
```haskell
interp :: Expr -> Env -> Expr
interp expr env = case expr of
  NumE _ -> expr
  VarE v -> case lookup v env of
    Just result -> result
    _ -> error (v ++ " : unbound")
  FunE var body -> expr
  AppE fun arg -> case interp fun env of
    FunE var body ->
      interp body (var, (interp arg env)):env
    bad -> error ("not a fun: " ++ (show bad))
```

```
data Expr = NumE Int | BoolE Bool |
  | OpE Op Expr Expr |
  | IfE Expr Expr Expr |
  | VarE Var |
  | FunE Var Expr |
  | AppE Expr Expr `
(with [y 2]
  (with [f (fun (x) (+ x y))]
    (with [y 5]
      (f y))))

=> 10 (!)
data Expr = NumE Int | BoolE Bool |
            OpE Op Expr Expr |
            IfE Expr Expr Expr |
            VarE Var |
            FunE Var Expr |
            AppE Expr Expr |

interp :: Expr -> Env -> Expr
interp expr env = case expr of
  NumE _ -> expr
  VarE v -> case lookup v env of
    Just result -> result
    _ -> error (v ++ " unbound")
  FunE var body -> expr
  AppE fun arg -> case interp fun env of
    FunE var body ->
      interp body (var, (interp arg env)):env
    bad -> error ("nota fun: " ++ (show bad))
data Expr = NumE Int | VarE Var | FunE Var Expr | AppE Expr Expr

data Val = NumV Int | BoolV Bool | FunV Var Expr Env

interp :: Expr -> Env -> Val
interp expr env = case expr of
  NumE n -> NumV n
  VarE v -> case lookup v env of
    Just result -> result
    _ -> error (v ++ " : unbound")
  FunE var body -> FunV var body env
  AppE fun arg -> case interp fun env of
    FunV v body closEnv ->
      interp body ((v, interp arg env):closEnv)
    bad -> error ("not a fun: " ++ (show bad))
(with [y 2]
  (with [f (fun (x) (+ x y))]
    (with [y 5]
      (f y))))
=> 7
(with [f (fun (x) (* x x))] (f (+ 1 2)))
(with [f (fun (_) (* 3 4))]
  (f (+ 1 2)))
(with [f (fun (_) (* 3 4))]
    (f (/ 1 0)))
int i = 0;
someFunction(++i);
printf("i = %d\n", i);
λ> :t readFile
readFile :: FilePath -> IO String
Concepts

- Substitution (β-reduction)
- Lexical scope
- α-renaming
- Call-by-{value, name, need} (eager vs. lazy)
- λ-abstraction (anonymous functions)
- Desugaring
- Environments
- Closures