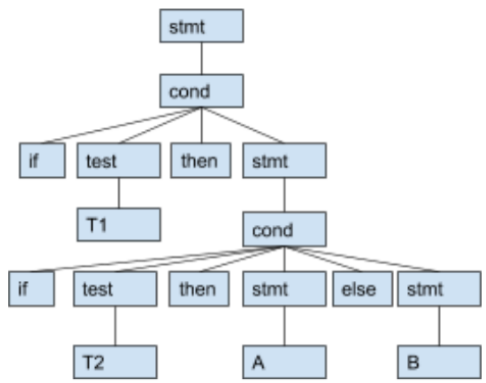
```haskell
-- Translating EBNF to ParserLib
data Expr
    = Num Integer
    | Bln Bool
    | Var String
    | Prim2 Op2 Expr Expr
    -- Let [(name, rhs), ...] body
    | Let [(String, Expr)] Expr
    | Lambda String Expr -- Lambda var body
    | App Expr Expr -- App func arg
    -- Cond test then-branch else-branch
    | Cond Expr Expr Expr
    deriving (Eq, Show)

data Op2 = Eq | Lt | Plus | Minus | Mul |
    Div | Mod
    deriving (Eq, Show)

-- Z ::= X | Y
Z = X <|> Y
-- block ::= "[" X "]"
block = between (char '[' *> whitespaces)
                (char ']' *> whitespaces) X

-- cmp ::= "==" | "<"
-- stringToOp is MY helper that maps "=="
    -> Eq, "<" to Lt, etc.
cmp = do
    op <- operator "==" <|> operator "<"
    return . Prim2 $ stringToOp op

-- infix ::= arith [ cmp arith ]
-- Since [...] means "zero or one", rather
    than "zero or more", we actually rewrite
     this as `arith | arith cmp arith`
infix_ = (do
            l <- arith
            c <- cmp
            r <- arith
            return $ c l r) <|> arith

-- atom ::= "(" block ")" | literal | var
atom = literal <|> (fmap Var var) <|>
    between
        (char '(' *> whitespaces)
        (char ')' *> whitespaces)
        block
```

```haskell
-- cond ::= "if" block "then" block "else"
    block
cond = do
    keyword "if"
    a <- block
    keyword "then"
    b <- block
    keyword "else"
    c <- block
    return $ Cond a b c

-- equation ::= var "=" block ";"
equation = do
    v <- var
    char '=' *> whitespaces
    e <- block
    char ';' *> whitespaces
    return (v, e)

-- literal ::= integer | boolean
literal = int <|> bool

-- boolean ::= "True" | "False"
bool = do
    b <- keyword "True" <|> keyword "False"
    return . Bln $ if b == "True" then True
         else False

int = fmap Num integer
-- identifier: any alphabetical string
    EXCLUDING the ones apassed in the list
var = identifier ["if", "then", "else", "
    let", "in", "True", "False"]

-- Lab 11, parses:
--      S ::= identifier | "(" S { S } ")"
data SExpr = Ident String | List [SExpr]
    deriving (Eq, Show)

sexpr :: Parser SExpr
sexpr = fmap Ident (identifier []) <|>
            between
                (char '(' *> whitespaces)
                (char ')' *> whitespaces)
                (do
                    s <- some sexpr
                    return (List s))
```

```haskell
mainParser :: Parser SExpr
mainParser = whitespaces *> sexpr <* eof

-- Miscellaneous interpreter things
data Value = VN Integer
           | VB Bool
           | VClosure (Map String Value)
              String Expr
    deriving (Eq, Show)

data Error = VarNotFound | TypeError |
    DivByZero deriving (Eq, Show)

-- Core interpreter logic.
interp :: Map String Value -> Expr ->
    Either Error Value

-- Base cases, unwrap value.
interp _ (Num i) = pure $ VN i
interp _ (Bln b) = pure $ VB b

-- Evaluate an if condition, then evaluate
    the branch taken.
interp env (Cond a b c) = do
    cond <- interp env a
    assertBool cond
    interp env $ case cond of
        VB True   -> b
        VB False  -> c

-- Recursively evaluate all bindings in a
    let block.
interp env (Let [] blk) = interp env blk
interp env (Let ((name, expr):xs) blk) = do
    v <- interp env expr
    interp (Map.insert name v env) (Let xs
       blk)

-- Apply an argument to lambda.
interp env (App f e) = do
    lambda <- interp env f
    case lambda of
        VClosure closure v body -> do
            arg <- interp env e
            interp (Map.insert v arg
               closure) body
        _ -> Left TypeError
```

```
-- Lab 12, mutual recursion
{-     let f x = expr1
           g x = expr2
           h x = expr3
       in body

   is represented by

       LetRecFuns [ ("f", "x", expr1),
                    ("g", "x", expr2),
                    ("h", "x", expr3) ]
                  body
-}

interp env (LetRecFuns defs body) = interp
    (buildEnv env defs) body
  where
    buildEnv env [] = env
    buildEnv env ((fName, var, expr):defs)
       = fEnv
      where
        fEnv = buildEnv (Map.insert fName f
             env) defs
-- The trick is here, for each function
    define two closures that takes the other
     one in; this abuses the fact that
    Haskell *does* have forward references.
    We need to do this recursively, though.
        f = VClosure (buildEnv (Map.insert
            fName f' env) defs) var expr
        f' = VClosure (buildEnv (Map.insert
             fName f env) defs) var expr

-- EBNF refresher
-- either X or Y
X | Y
-- val, followed by ZERO or MORE ‘op val‘
    pairs. Use chainl1 in ParserLib for left
    -associative, and chainr1 for right-
    associative.
val { op val }
-- same as above, except ONE or NONE
val [ op val ]
-- terminal string
"if", e.g. "if" block "else" block "then"
    block
```

```
-- Ambiguous grammars. Consider:
<stmt> ::= <cond> | "A" | "B" | "C"
<cond> ::= "if" <test> "then" <stmt>
                 | "if" <test> "then" <stmt>
                      "else" <stmt>
<test> ::= "T1" | "T2"

For the input "if T1 then if T2 then A else
     B", two interpretations possible:
1. if T1 then {if T2 then {A} else {B}}
2. if T1 then {if T2 then {A}} else {B}

In general case, CFG ambiguity is
    undecidable!

Parse tree (NOT ABSTRACT SYNTAX TREE) for
    #1 shown below.
```




```
For AST, get rid of all the useless
    terminals "if" "then" from the parse
    tree. AST is the form that ParserLib
    generates.

-- Monad stuff, Lab 9
postInc :: State Int Int
postInc = StateOf (\i -> (i+1, i))

numberHelper :: BT v -> State Int (BT Int)
numberHelper Null = return Null
numberHelper (Node left _ right) = do
  left <- numberHelper left
  curr <- postInc
  right <- numberHelper right
  return (Node left curr right)
```

```
-- Monad/Applicative/Functor shit from A2
instance Functor FreeGameMaster where
-- fmap :: (a -> b) -> FreeGameMaster a ->
    FreeGameMaster b
-- or fmap = liftM
    fmap f (Pure a) = Pure (f a)
    fmap f (GMAction lo hi fx) = GMAction
        lo hi (\msg -> fmap f (fx msg))

instance Applicative FreeGameMaster where
-- pure :: a -> FreeGameMaster a
    pure = Pure
-- (<*>) :: FreeGameMaster (a -> b) ->
    FreeGameMaster a -> FreeGameMaster b
--  or (<*>) = ap
    (Pure f) <*> rhs             = fmap f rhs
    (GMAction lo hi f) <*> rhs = GMAction
        lo hi (\msg -> (f msg) <*> rhs)

instance Monad FreeGameMaster where
-- return :: a -> FreeGameMaster a
    return = Pure
-- (>>=) :: FreeGameMaster a -> (a ->
    FreeGameMaster b) -> (FreeGameMaster b)
    (Pure a) >>= f             = f a
    (GMAction lo hi fx) >>= f = GMAction lo
         hi (\m -> (fx m) >>= f)

-- Mutable memory monad
newtype MutM a = MkMutM (IntMap Value ->
    Either String (IntMap Value, a))

runMutM (MkMutM f) = f

instance Monad MutM where
    return a = MkMutM (\s -> Right (s, a))
    MkMutM f >>= k = MkMutM (\s -> case f s
         of Left msg -> Left msg
        Right (s', a) -> runMutM (k a) s')

getMem = MkMutM (\s -> Right (s, s))
modMem f = MkMutM (\s -> Right (f s, ()))
new val = do
    n <- fmap IntMap.size getMem
    store n val
    pure n
store a val = modMem (IntMap.insert a val)
```