State Monad:

- The **state monad** is a built-in monad in Haskell that allows for chaining of a state variable through a series of function calls, to simulate stateful code. It is defined as:
  newtype State s a = State { runState :: (s -> (a,s)) }
- The Haskell type State describes functions that consume a state and produce both a result and an updated state, which are given back in a tuple.
- A few basic operations are provided below. Furthemore, "State s" is an instance of Functor, Applicative, and Monad, so you have connectives to chain up basic operations too.

1. **-- "get" reads and returns the current value of the state variable.**
   get :: State s s
   get = State (\s -> (s,s))

2. **-- "put s1" sets the state variable to s1. It returns the 0-tuple because there**
   -- is no information to return.
   put :: s -> State s ()
   put newState = State (\s -> (newState, ()))

3. **-- functionize prog s0 runs prog starting with initial state value s0 and gives**
   -- you the final answer.
   functionize :: State s a -> s -> a

- E.g. We want to build a binary tree out of given elements, inorder, balanced.
  I.e. buildTree [a, b, c, d, e, f, g] means at d, which is the root, recursively create a,b,c in left subtree, and e,f,g in right subtree. We need to count length for halving. A non-obvious but linear-time strategy is:
  - Count length just once.
  - State variable holds unused elements (initially all).
  - Recursive helper takes parameter n, uses the first n elements from state var to build tree. Algorithm:
    - Split n into n = m1 + 1 + m2, m1 is left subtree size, m2 is right subtree size, and 1 element for the middle node.
    - Recursive call: build tree of m1 elements, this will be the left subtree.
    - Take out one element from state var, this will be for the middle node.
    - Recursive call: build tree of m2 elements, this will be the right subtree.
    - Compose the middle node, it's my answer

  Here's the code:

```
-- Recall: data BinTree a = BTNil | BTNode a (BinTree a) (BinTree a)

-- buildTreeHelper n: Use n elements from [a] state var to build tree.
-- Precondition: n <= length of state var.
buildTreeHelper :: Int -> State [a] (BinTree a)
buildTreeHelper 0 = pure BTNil
buildTreeHelper n =
  buildTreeHelper m1        -- Make left subtree, m1 elements, call it lt.
  >>= \lt -> get
  >>= \(x:xt) -> put xt     -- Which elements remaining? Take one for myself.
  >> buildTreeHelper m2     -- Make right subtree, m2 elements, call it rt.
  >>= \rt -> pure (BTNode x lt rt) -- Put it together, this is my answer.
```

```
  where
    n' = n - 1
    m1 = div n' 2
    m2 = n' - m1

buildTree :: [a] -> BinTree a
buildTree xs = functionize (buildTreeHelper (length xs)) xs
-- Whole list for initial state. Use all to build tree.
```

In the example above,
```
>>= \lt -> get
>>= \(x:xt) -> put xt
```
you need to use bind, >>=, to get the return value of get. In this case, (x:xt) is the return value of get.

- The basic idea of state monad is to have a state transition function instead, like s→s, and have some starter function, functionize, that feeds it the initial value. However, we also want it to give an answer and not a state. So s→(s,a) is a function from the old-state to a pair of the new-state and answer.
- E.g.

```
data State s a = MkState (s -> (s, a))

-- Unwrap MkState.
deState :: State s a -> s -> (s, a)
deState (MkState stf) = stf

functionize :: State s a -> s -> a
functionize prog s0 = snd (deState prog s0)

get :: State s s
get = MkState (\s0 -> (s0, s0))
-- old state = s0, new state = old state = s0, answer s0 too.

put :: s -> State s ()
put s = MkState (\s0 -> (s , ()))
-- ignore old state, new state = s, answer the 0-tuple ().

instance Functor (State s) where
   -- fmap :: (a -> b) -> State s a -> State s b
   fmap f (MkState stf) = MkState
      (\s0 ->
        -- Goal: Like stf but use f to convert a to b
        -- old state = s0, give to stf for new state s1 and answer a
        case stf s0 of (s1, a) ->
        -- overall new state is also s1, but change answer to f a
                    (s1, f a))
```

```
testStateFunctor = deState (fmap length program) 10
  where
    program :: State Integer String
    program = MkState (\s0 -> (s0+2, "hello"))
-- should give (12, 5)

instance Applicative (State s) where
    -- pure :: a -> State s a
    -- Goal: Give the answer a and try not to have an effect.
    -- "effect" for State means state change.
    pure a = MkState (\s0 -> (s0, a))
    -- so new state = old state

    -- liftA2 :: (a -> b -> c) -> State s a -> State s b -> State s c
    --
    -- State transition goal:
    --                overall old state
    -- --1st-program--> intermediate state
    -- --2nd-program--> overall new state
    --
    -- (Why not the other order? Actually would be legitimate, but we usually
    -- desire liftA2's order to be consistent with >>='s order.)
    liftA2 op (MkState stf1) (MkState stf2) = MkState
        (\s0 ->
          -- overall old state = s0, give to stf1
          case stf1 s0 of { (s1, a) ->
          -- intermediate state = s1, give to stf2
          case stf2 s1 of { (s2, b) ->
          -- overall new state = s2
          -- overall answer = op a b
          (s2, op a b) }} )

testStateApplicative = deState (liftA2 (:) prog1 prog2) 10
  where
    prog1 :: State Integer Char
    prog1 = MkState (\s0 -> (s0+2, 'h'))
    prog2 :: State Integer String
    prog2 = MkState (\s0 -> (s0*2, "ello"))
-- should give (24, "hello").  24 = (10+2)*2.
```

```
instance Monad (State s) where
  return = pure

  -- (>>=) :: State s a -> (a -> State s b) -> State s b
  -- Goal:
  -- 1. overall old state --1st-program--> (intermediate state, a)
  -- 2. give a and intermediate state to the 2nd program.
  MkState stf1 >>= k = MkState
    (\s0 ->
      -- overall old state = s0, give to stf1
      case stf1 s0 of { (s1, a) ->
      -- k is waiting for the answer a
      --     and also the intermediate state s1
      -- technicality: "(k a) s1" is conceptually right but nominally a
      -- type error because (k a) :: State s b, not s -> (s, b)
      -- Ah but deState can unwrap! (Or use pattern matching.)
      deState (k a) s1 } )
```

Dependency injection, Template method, Mock testing:

- Here is the first version of my file format checker for my toy file format. The first three characters should be A, L, and newline.

```
toyCheckV1 :: IO Bool
toyCheckV1 =
  getChar
  >>= \c1 -> getChar
  >>= \c2 -> getChar
  >>= \c3 -> return ([c1, c2, c3] == "AL\n")
```

- We can also use dependency injection to test it. One way of doing this is to define our own type class for the relevant, permitted operations.

```
class Monad f => MonadToyCheck f where
  toyGetChar :: f Char
-- Simplifying assumptions: Enough characters, no failure.  A practical version
-- should add methods for raising and catching EOF exceptions.
```

  The checker logic should be polymorphic in that type class.

```
toyCheckV2 :: MonadToyCheck f => f Bool
toyCheckV2 =
  toyGetChar
  >>= \c1 -> toyGetChar
  >>= \c2 -> toyGetChar
  >>= \c3 -> return ([c1, c2, c3] == "AL\n")
```

  Only things toyCheckV2 can do: toyGetChar, monad methods, purely functional programming. Because the user chooses f. And toyCheckV2 doesn't even know what it is. All it knows is it can call toyGetChar.

- Now we can instantiate in two different ways, one way for production code, another way for mock testing.

- For production code:

```
instance MonadToyCheck IO where
   toyGetChar = getChar

realProgram :: IO Bool
realProgram = toyCheckV2
```

- For purely functional mock testing:

```
data Feeder a = MkFeeder (String -> (String, a))
-- Again, simplifying assumptions etc.  But basically like the state monad, with
-- the state being what's not yet consumed in the string.

-- Unwrap MkFeeder.
unFeeder :: Feeder a -> String -> (String, a)
unFeeder (MkFeeder sf) = sf

instance Monad Feeder where
   return a = MkFeeder (\s -> (s, a))
   prog1 >>= k = MkFeeder (\s0 -> case unFeeder prog1 s0 of
                              (s1, a) -> unFeeder (k a) s1)

instance MonadToyCheck Feeder where
   -- toyGetChar :: Feeder Char
   toyGetChar = MkFeeder (\(c:cs) -> (cs, c))

instance Functor Feeder where
   fmap f p = p >>= \a -> return (f a)

instance Applicative Feeder where
   pure a = MkFeeder (\s -> (s, a))
   pf <*> pa = pf >>= \f -> pa >>= \a -> return (f a)

testToyChecker2 :: String -> Bool
testToyChecker2 str = snd (unFeeder toyCheckV2 str)

toyTest1 = testToyChecker2 "ALhello"   -- should be False
toyTest2 = testToyChecker2 "AL\nhello" -- should be True
```

- Here's the code in its entirety:

```
class Monad f => MonadToyCheck f where
   toyGetChar :: f Char
-- Simplifying assumptions: Enough characters, no failure.  A practical version
-- should add methods for raising and catching EOF exceptions.

toyCheckV2 :: MonadToyCheck f => f Bool
toyCheckV2 =
   toyGetChar
   >>= \c1 -> toyGetChar
   >>= \c2 -> toyGetChar
   >>= \c3 -> return ([c1, c2, c3] == "AL\n")
```

```
data Feeder a = MkFeeder (String -> (String, a))
-- Again, simplifying assumptions etc.  But basically like the state monad, with
-- the state being what's not yet consumed in the string.

-- Unwrap MkFeeder.
unFeeder :: Feeder a -> String -> (String, a)
unFeeder (MkFeeder sf) = sf

instance Monad Feeder where
   return a = MkFeeder (\s -> (s, a))
   prog1 >>= k = MkFeeder (\s0 -> case unFeeder prog1 s0 of
                           (s1, a) -> unFeeder (k a) s1)

instance MonadToyCheck Feeder where
   -- toyGetChar :: Feeder Char
   toyGetChar = MkFeeder (\(c:cs) -> (cs, c))

instance Functor Feeder where
   fmap f p = p >>= \a -> return (f a)

instance Applicative Feeder where
   pure a = MkFeeder (\s -> (s, a))
   pf <*> pa = pf >>= \f -> pa >>= \a -> return (f a)

testToyChecker2 :: String -> Bool
testToyChecker2 str = snd (unFeeder toyCheckV2 str)

toyTest1 = testToyChecker2 "ALhello"   -- should be False
toyTest2 = testToyChecker2 "AL\nhello" -- should be True
```

- E.g.

```
*Main> unFeeder toyCheckV2 "ALBERT"
("ERT",False)
*Main> unFeeder toyCheckV2 "AL\nBERT"
("BERT",True)
```

Context Free Grammar:

- A context-free grammar is a set of recursive rules used to generate patterns of strings.
- Parser programs in compilers can be generated automatically from context-free grammars.
- Context-free grammars have the following components:
  - A **set of terminal symbols** which are the characters that appear in the language/strings generated by the grammar. Terminal symbols never appear on the left-hand side of the production rule and are always on the right-hand side.
  - A **set of nonterminal symbols (or variables)** which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols. These are the symbols that will always appear on the left-hand side of the

production rules, though they can be included on the right-hand side. The strings that a CFG produces will contain only symbols from the set of nonterminal symbols.
    - A **set of production rules** which are the rules for replacing nonterminal symbols. Production rules have the following form: **variable → string of variables and terminals**.
    - A **start symbol** which is a special nonterminal symbol that appears in the initial string generated by the grammar.
- To create a string from a context-free grammar, follow these steps:
    - Begin the string with a start symbol.
    - Apply one of the production rules to the start symbol on the left-hand side by replacing the start symbol with the right-hand side of the production.
    - Repeat the process of selecting nonterminal symbols in the string, and replacing them with the right-hand side of some corresponding production, until all nonterminals have been replaced by terminal symbols. Note, it could be that not all production rules are used.
- E.g. A context-free grammar looks like this bunch of rules:
  Rule 1. E → E + E
  Rule 2. E → M
  Rule 3. M → M × M
  Rule 4. M → A
  Rule 5. A → 0
  Rule 6. A → 1
  Rule 7. A → (E)
  E, M, A are **non-terminal symbols** or **variables**. When you see them, you apply rules to expand. One of them is designated as the **start symbol**. You always start from it. Here, E is the start symbol.
  +, ×, 0, 1, (,) are **terminal symbols**. They are the characters you want in your language.
- **Derivation/generation** is a finite sequence of applying the rules until all non-terminal symbols are gone. We often aim for a specific final string.
- E.g.

E → M	(By Rule 2)
→ M × M	(By Rule 3)
→ A × M	(By Rule 4)
→ 1 × M	(By Rule 6)
→ 1 × A	(By Rule 4)
→ 1 × (E)	(By Rule 7)
→ 1 × (E + E)	(By Rule 1)
→ 1 × (M + E)	(By Rule 2)
→ 1 × (A + E)	(By Rule 4)
→ 1 × (0 + E)	(By Rule 5)
→ 1 × (0 + M)	(By Rule 2)
→ 1 × (0 + M × M)	(By Rule 3)
→ 1 × (0 + A × M)	(By Rule 4)
→ 1 × (0 + 1 × A)	(By Rule 6)
→ 1 × (0 + 1 × 1)	(By Rule 6)

- Context-free grammars can support matching parentheses and unlimited nesting.

Backus-Naur Form (BNF):

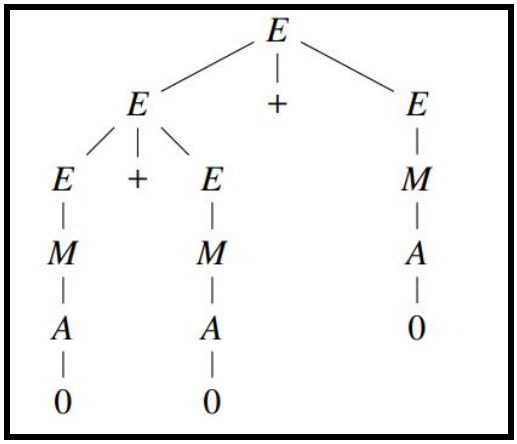
- **Backus-Naur Form** is a computerized, practical notation for CFGs.
- Surround non-terminal symbols by **<>**.
- Allow multi-letter names.
  Note: In some versions, we don't need <> around non-terminal symbols.
- Merge rules with the same LHS.
- In some versions, we surround terminal strings by single or double quotes.
- Use **::=** for →.
- Our example grammar in BNF:
  <expr> ::= <expr> "+" <expr> | <mul>
  <mul> ::= <mul> "*" <mul> | <atom>
  <atom> ::= "0" | "1" | "(" <expr> ")"

Extended Backus-Naur Form (EBNF):

- Use **{...}** for 0 or more occurrences.
- Use **[...]** for 0 or 1 occurrences.
- In some versions, no <> is needed around non-terminal symbols.

Parse Tree/Derivation Tree:

- A **parse tree/derivation tree** presents a derivation with more structure (tree), less repetition.
- E.g. This example generates 0 + 0 + 0.

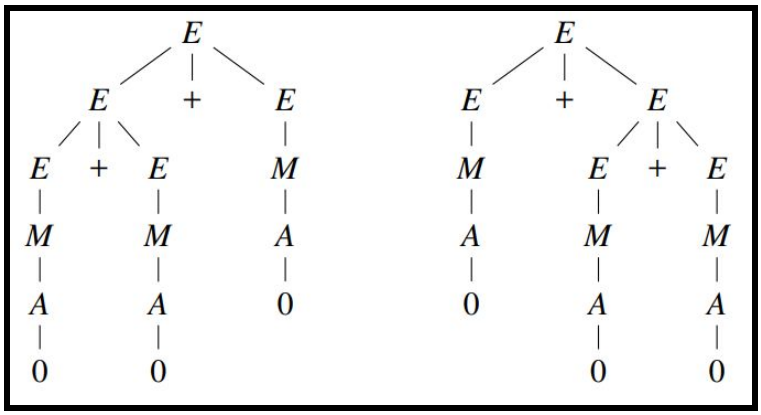


This is how we would write the example using derivation:

E → E + E (By Rule 1)
→ E + E + E (By Rule 1)
→ M + E + E (By Rule 2)
→ M + M + E (By Rule 2)
→ M + M + M (By Rule 2)
→ A + M + M (By Rule 4)
→ A + A + M (By Rule 4)
→ A + A + A (By Rule 4)
→ 0 + A + A (By Rule 5)
→ 0 + 0 + A (By Rule 5)
→ 0 + 0 + 0 (By Rule 5)

- In parse trees:
  - Internal nodes are non-terminal symbols.
  - Both operators and operands are terminal symbols at leaves.
  - The whole string is recorded, just scattered.

  - The purpose is to help visualize derivation and grammar as well as making writing the derivations easy and simple.
- When 2 or more different trees generate the same output, we say that the grammar is **ambiguous**.
  E.g. Two different trees generate the same 0 + 0 + 0:



  We try to design unambiguous grammars.
  CFG ambiguity is undecidable.
  Equivalence of two CFGs is also undecidable.
- **Note:** Generally, the reason we have ambiguity in languages is because there are more than 1 calls to the same item in 1 line.
  E.g. In the above language, we have
  <expr> ::= <expr> + <expr>
  <mul> ::= <mul> * <mul>
  Hence, if we have 0+0+0 or 0*0*0, we can use either <expr> or <mul> for expansion.
- Here is an unambiguous grammar that generates the same language as our ambiguous grammar example from above.
  <expr> ::= <expr> “+” <mul> | <mul>
  <mul> ::= <mul> “*” <atom> | <atom>
  <atom> ::= “0” | “1” | “(“ <expr> “)”

Left Recursive vs Right Recursive:

- **<expr> ::= <expr> “+” <mul>**
  That is a left recursive rule. The recursion is at the beginning (left).
- **<expr> ::= <mul> “+” <expr>**
  That is a right recursive rule. The recursion is at the end (right).
- Sometimes they convey intentions of left association or right association, but not always.
- They affect some parsing algorithms.
- Recursive descent parsing is a simple strategy for writing a parser.
- For each non-terminal symbol, we create a procedure based on RHS:
  - Non-terminal Symbol: Procedure call, possibly mutual recursion. (Thus “recursive descent”, also “top-down”.)
  - Left recursion needs special treatment to avoid infinite loops.
  - Terminal Symbol: Consume input and check.
  - Alternatives: Look ahead to choose, or try and backtrack.
- Some options for handling left recursion:
  - Redesign grammar to not have left recursion.

  - Many left recursive rules just express left-associating operators. Can be done without left recursive code.

- E.g.

<sub> ::= <atom> "-" <sub> | <atom>
<atom> ::= "0" | "1" | "(" <sub> ")"

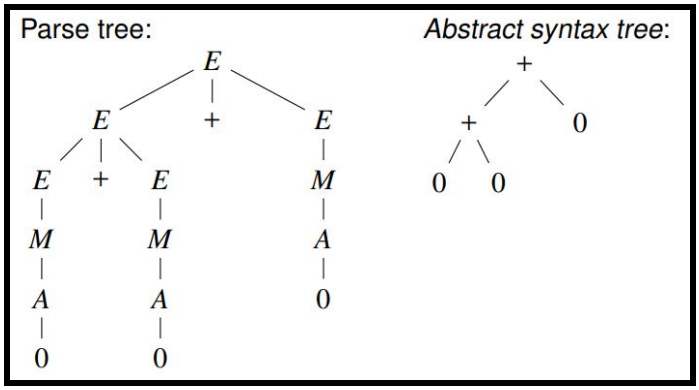
Starting with <sub>, we look at its RHS, which is <atom> "-" <sub> | <atom>.
We see <atom>, which is a non-terminal symbol. Hence, we make a procedure call to <atom>.
Next, we see a terminal symbol. We check if that terminal symbol is “-”. If it is, we continue to <sub>. Otherwise, we go to <atom>.
Since the terminal symbol is “-”, we see <sub>, so we make a procedure call to <sub>.
Pseudo-code of recursive descent parser:

```
sub:
        try (atom;
            read; if not "-" then fail;
            sub ;)
        if that failed: atom;

atom:
        read;
        if "0" or "1": success;
        if "(": sub;
                read; if not ")" then fail;
        else: fail;
```

Abstract Syntax Tree (AST) (vs Parse Tree):

- Abstract Syntax Tree General Points:
  - Internal nodes are operators/constructs.
    An example of a construct is if-then-else.
  - Non-terminal symbols are gone or replaced by constructs.
  - Many terminal symbols are gone too if they play no role other than nice syntax.
    E.g. spaces, parentheses, punctuations
    Those bearing content are replaced by appropriate representations and do not stay as characters.
    E.g. The character '+' is replaced by a data constructor.
    E.g. The character '0' is replaced by the number 0.
  - The purpose is to present only the essential structure and content, ready for interpreting, compiling, analyses.
  - Parsers usually output abstract syntax trees when successful.
- Comparison of Parse Tree and Abstract Syntax Tree:



Lexical Analysis/Tokenization:

- In principle, grammar and parser can work on characters directly, but it is usually messy.
- In practice, we have 2 stages:
  1. We chop character streams into chunks and classify into **lexemes/tokens** and we discard spaces. Furthermore, we typically use objects or data representations instead of the actual strings.
     E.g.
     " (xa * xb)**25 " →
     [Open, Var "xa", Op Mul, Var "xb", Close, Op Exp, NumLiteral 25]
     Here, we use the object or data representation "Open" to denote the open parentheses, "(". Furthermore, notice the space between " and (. In the array, the space isn't shown.
     Note: We can use regular expression to determine what category, variable, number, operator, etc, an item is.
     This step is called **lexical analysis/tokenization**.
  2. Parsing is based on CFG. Terminal symbols are tokens, not characters.