# Parse Tree

## How does it work?

The parse tree begins with a configuration in the definition of the language based on our grammar. This configuration will allow us to facilitate the parser of our grammar by defining our reserved words, reserved operators, what is a comment, etc.

The parser functions that were defined have two approaches to fulfill their purpose depending on what is needed.

### First approach - Applicatives

The first approach is the use of applicatives. This approach was used in most of the parse tree functions, where it seeks to match the input using combinators between functors and applicative functions with "priority" to choose to return one of the two parameters of its functions to fulfill the parser of that function.

This approach is very useful when the combination we need to perform is independent, in other words, it does not need to generate an intermediate result to be used in another combination to form a final combination.

Examples with this approach:

```
listOfBool :: SymbolTable -> Parser (List, SymbolTable)
listOfBool symTa =
  (\b -> (lb b, newSymTable b symTa))
    <$ whiteSpace
    <* string "List:Bool"
    <* whiteSpace
    <* string "["
    <* whiteSpace
    <*> sepBy boolValue (char ',')
    <* whiteSpace
    <* string "]"
    <* whiteSpace
  where
    lb = ListBool
    newSymTable b = insertList (show b) (lb b)
```

```
strComparison :: Parser Comparison
strComparison =
  ComparisonString
    <$> literal
    <* whiteSpace
    <*> boolComparator
    <* whiteSpace
    <*> literal
    <* whiteSpace
```

### Second approach - Monads

This approach is very useful when the combination being searched needs intermediate results by other intermediate combinations to meet the final combination.

This type of approach can be easily applied using monadic operations such as the bind operator (¿¿=) or the sequential operator (¿¿).

Example:

```
code :: SymbolTable -> Parser (Code, SymbolTable)
code symTable =
  whiteSpace *> funcs symTable >>= \(f, symTable1) ->
    whiteSpace *> doNotation symTable1 >>= \(d, symTable2) ->
      whiteSpace *> return (Code f d, symTable2)
```

But this notation becomes very difficult to read when the combination being searched is larger, as is the case with some of our parsers. In this case, Haskell provides us with the do notation to perform the same operation but in a more readable way:

Example:

```
taskComparison :: SymbolTable -> Parser (Comparison, SymbolTable)
taskComparison symTable = do
  (s1, symTable1) <- task' symTable
  whiteSpace
  cmp <- boolComparator
  whiteSpace
  (s2, symTable2) <- task' symTable1
  whiteSpace
  return (ComparisonTask s1 cmp s2, symTable2)
```

## Are there specific grammar rules?

1. The comments are ignored by the parser thanks to the configuration of the language definition that was previously explained. The way to declare them is using the symbols "//" where that entire line is considered a comment.

2. The spaces between the terminals and non-terminals of our grammar are not explicitly defined, but our parser needs to have at least one space between a reserved word with what follows and the same with the operators. If this condition is met, the user can put more than one space or line break if that is desired for the code format.

3. According to the AST, to declare the elements that a list will contain, they need to be separated by a comma, but it is not possible to have a comma after the last element inserted.

4. There are two types of string. This is because some strings are intended to represent an identifier such as the status or tag of a task and the role of a member, while the others can have text with letters, numbers and symbols defined in our AST.

5. According to the AST, functions may or may not be declared, but the structure of the "do" must be defined yes or yes. This structure is similar

to Haskell's do notation, where you can define variables that store statements and be able to display these results in the console using print, which also receives a statement.