# Project Part 2: Parser

New Attempt

**Due** Feb 24 by 11:59pm **Points** 100 **Submitting** a file upload **File Types** java

In this assignment, you will implement the parser for our language. This is the second step in the parsing process which takes the tokens produced by the lexer in the previous part and turns them into an Abstract Syntax Tree (AST), which is a tree-based representation of our language.

Also, note the Crafting Interpreters link below is *highly* recommend to help you understand both the parsing process as well as the implementation approach we're using, especially peek and match. If anything qualifies as a textbook in this class, it's that.

## Submission

You will submit `Parser.java`, which implements your parser. The `Ast` class should be the same, and you do not need to submit any additional tests you've added. You should re-test your parser with the provided test class prior to submitting.

- There will be two test submissions, one on **Friday, February 19** and another on **Monday, February 22**
- The final submission is **Wednesday, February 24**.

## Parser Overview

Recall that the job of the parser is to convert the tokens emitted by the lexer into an Abstract Syntax Tree (AST) which represents the structural meaning of the code. For example, the expressions `1 + 2 * 3` and `1 * 2 + 3` are similar, but their AST's are quite different due to operator precedence:

```
1 + 2 * 3         1 * 2 + 3

    +                 +
   / \               / \
  1   *             *   3
     / \           / \
    2   3         1   2
```

Our parser will be implemented using a method called recursive descent, which means each reference to another rule in our grammar corresponds with a call to the appropriate parse function.

### Error Handling

If the parser is unable to parse something successfully (for example, an unexpected token), then it will throw a `ParseException`. The index of the exception should be the index of the next token (via `token.getIndex()`) if present, otherwise the index after the previous token (which you will need to compute using the length of the literal). You do not need to worry about the edge case of there not being a previous token (as our `source` rule allows empty input which should not cause an exception). Two examples:

- `IF cond THEN ...` -> `new ParseException("Expected DO.", index = 8)`, as `8` is the index of the incorrect `THEN` token.
- `IF cond` -> `new ParseException("Expected DO.", index = 7`, as `7` is the index of the character after the previous token.

### Crafting Interpreters

The Parsing section of Crafting Interpreters provides a good overview for the parsing process and was a starting point for the parser architecture we have provided. Their parser is slightly more complex as they submit more functionality, so make sure you only implement what is defined in our grammar. I highly recommend reading it to help with your understanding of the parsing process and this assignment.

### Grammar

A grammar for our lexer is defined below, which is written in a specific form optimal for recursive descent parsing. You can view a graphical form of our grammar on the following website:

- https://www.bottlecaps.de/rr/ui

```
source ::= field* method*
field ::= 'LET' identifier ('=' expression)? ';'
method ::= 'DEF' identifier '(' (identifier (',' identifier)*)? ')' 'DO' statement* 'END'

statement ::=
    'LET' identifier ('=' expression)? ';' |
    'IF' expression 'DO' statement* ('ELSE' statement*)? 'END' |
    'FOR' identifier 'IN' expression 'DO' statement* 'END' |
    'WHILE' expression 'DO' statement* 'END' |
    'RETURN' expression ';' |
    expression ('=' expression)? ';'

expression ::= logical_expression

logical_expression ::= comparison_expression (('AND' | 'OR') comparison_expression)*
comparison_expression ::= additive_expression (('<' | '<=' | '>' | '>=' | '==' | '!=') additive_expression)*
additive_expression ::= multiplicative_expression (('+' | '-') multiplicative_expression)*
multiplicative_expression ::= secondary_expression (('*' | '/') secondary_expression)*

secondary_expression ::= primary_expression ('.' identifier ('(' (expression (',' expression)*)? ')')?)*

primary_expression ::=
    'NIL' | 'TRUE' | 'FALSE' |
    integer | decimal | character | string |
    '(' expression ')' |
    identifier ('(' (expression (',' expression)*)? ')')?

identifier ::= [A-Za-z_] [A-Za-z0-9_]*
number ::= [+-]? [0-9]+ ('.' [0-9]+)?
character ::= ['] ([^'\\] | escape) [']
string ::= '"' ([^"\n\r\\] | escape)* '"'
escape ::= '\' [bnrt'"\]
operator ::= [<>!=] '='? | 'any character'

whitespace ::= [ \b\n\r\t]
```

### AST Representation

The `Ast` class contains subclasses representing the more specific elements of the AST and the code that is being parsed. There is not a one-to-one relationship between rules in our grammar and the AST. If you have not completed Homework 2, it may be helpful to do that first.

The following classes are contained within `Ast`:

- `Source`: An entire source file (fields + methods).
- `Field`: Field declarations.
- `Method`: Method definitions.
- `Stmt`: Structural parts of the code that perform side effects like assigning variables or modifying control flow.
  - `Expression`: A statement that is simply an expression (such as a function call). Don't confuse this with `Ast.Expr`!
  - `Declaration`: The declaration (and optional initialization) of variables.
  - `Assignment`: The assignment of values. The receiver is an expression and not the name of a variable because assignment also applies to fields like `x.y = z`.
  - `If`: An if statement with an optional else branch. The list of else statements is empty if not defined.
  - `While`: A while statement.
  - `Return`: A return statement. Note that a value must be provided, unlike languages like Java which have `void` return types.
- `Expr`
  - `Literal`: Contains literal values, such as booleans, integers, or strings. You will need to convert values from the lexer into the appropriate type.
    - Nil is represented using `null`. Note this means `getLiteral()` can return `null`, so be careful!
    - Boolean values are represented with the `Boolean` class.
    - Integer values are represented with the `BigInteger` class, which supports arbitrary precision.
    - Decimal values are represented with the `BigDecimal` class, which supports arbitrary precision.
    - Character values are represented with the `Character` class. You will need to remove the surrounding single quotes (`'`) from the literal returned by the lexer and replace any escape characters (hint, see `String#replace`).
    - String values are represented with the `String` class. You will need to remove the surrounding double quotes (`"`) from the literal returned by the lexer and replace any escape characters (hint, see `String#replace`).
  - `Group`: A grouped expression (generally used for changing priority)
  - `Binary`: A binary expression, including additive, multiplicative, comparison, and logical expressions from the grammar. For logical expressions, the operator will be the literal `AND`/`OR` strings used in the grammar.
  - `Access`: An access expression, representing accessing a field or variable. The receiver is an Optional value, which is present to represent field access on objects and empty for variables (effectively, the receiver is the current scope which has access to all defined variables).
  - `Function`: A function call expression. The receiver is an Optional value, which is present to represent method calls on objects and empty for function calls (effectively, the receiver is the current scope which has access to all defined functions).

## Provided Code

The following files are provided to help you help implement the parser.

- Source Files (`src/main/java/plc/project`)
  - Ast.java
  - Parser.java
- Test Files (`src/test/java/plc/project`)
  - ParserTests.java