Objo Grammar Summary

The notation used in this grammar is as follows:

`nonTerminal`	These symbols represent rules in the grammar
`TERMINAL`	These symbols represent endpoints in the grammar. Essentially, they are `Tokens`.
`()`	Symbols may be grouped with parentheses.
`|`	Acts as a logical OR between symbol groups
`*`	Indicates the preceding symbol group may occur >= 0 times
`+`	Indicates the preceding symbol group may occur >= 1 times
`?`	Indicates the preceding symbol group may occur <= 1 time

The Program

The syntactic grammar is used to parse the linear sequence of tokens into a nested abstract syntax tree. It begins with the first rule that matches an entire Objo program (or a single REPL entry):

```
program         →    declaration*
```

Declarations

A program is a series of declarations. Declarations are the statements that bind new identifiers or any of the other statement types.

```
declaration     →    variableDecl
                |    functionDecl
                |    classDecl
                |    statement

variableDecl    →    VAR IDENTIFIER (COMMA IDENTIFIER)* (EQUAL expression)? terminator

functionDecl    →    FUNCTION method END FUNCTION terminator

classDecl       →    CLASS IDENTIFIER (INHERITS IDENTIFIER)? EOL
                     (getter | setter | methodDecl | foreignMethodDecl | staticMethodDecl)*
                     END CLASS terminator
```

Statements

The remaining statement rules produce side effects but do not introduce new bindings.

```
statement       →    block
                |    breakStmt
                |    continueStmt
                |    doStmt
                |    exitStmt
                |    ifStmt
                |    forStmt
                |    printStmt
                |    quitStmt
                |    returnStmt
                |    selectStmt
                |    whileStmt
                |    expressionStmt

breakStmt       →    BREAK terminator

block           →    (statement)*
```

```
doStmt          →    DO EOL
                     block
                     LOOP (UNTIL expression)? terminator

continueStmt    →    CONTINUE EOL

exitStmt        →    EXIT EOL

ifStmt          →    IF expression then
                |    IF expression THEN EOL
                     block
                     (ELSEIF expression THEN EOL block)*
                     (ELSE EOL block)?
                     END IF terminator

forStmt         →    FOR IDENTIFIER EQUAL range (STEP expression)? EOL
                     block
                     NEXT (IDENTIFIER)? terminator

printStmt       →    PRINT expression terminator

quitStmt        →    QUIT terminator
```

```
returnStmt      →    RETURN expression? terminator

whileStmt       →    WHILE expression EOL
                     block
                     WEND terminator

expressionStmt  →    expression terminator
```

Expressions

Expressions produce values. Below are the grammar rules for expressions with the lowest precedence rules being first.

```
expression      →    assignment

assignment      →    coalesce
                     ((EQUAL | PLUS_EQUAL | MINUS_EQUAL | STAR_EQUAL | SLASH_EQUAL) coalesce)?

coalesce        →    ternary QUERY_QUERY ternary

ternary         →    logicalOr (QUERY expression COLON ternary)?

logicalOr       →    logicalXor (OR logicalXor)*

logicalXor      →    logicalAnd (XOR logicalAnd)*

logicalAnd      →    equality (AND equality)*

equality        →    is ((NOT_EQUAL | EQUAL) is)*

is              →    comparison IS comparison

comparison      →    bitwiseOr ((GREATER | GREATER_EQUAL | LESS | LESS_EQUAL) bitwiseOr)*
```

```
bitwiseOr       →    bitwiseXor (PIPE bitwiseXor)*

bitwiseXor      →    bitwiseAnd (CARET bitwiseAnd)*

bitwiseAnd      →    bitwiseShift (AMPERSAND bitwiseShift)*

bitwiseShift    →    addition ((LESS_LESS | GREATER_GREATER) addition)*

addition        →    multiplication ((PLUS | MINUS) multiplication)*

multiplication  →    unary ((SLASH | STAR | MOD) unary)*

unary           →    (NOT | MINUS) unary
                |    call

call            →    primary (LPAREN arguments? RPAREN | DOT IDENTIFIER)*

primary         →    BOOLEAN | NUMBER | STRING | NOTHING | SELF | IDENTIFIER
                |    (SUPER | SUPER DOT IDENTIFIER)
                |    LPAREN expression RPAREN
                |    LSQUARE arguments* RSQUARE
```

Helper Rules

To keep the above rules clearer, some of the grammar is split out into the following re-usable helper rules.

```
terminator       →    EOL | EOF

parameters       →    IDENTIFIER (COMMA IDENTIFIER)*

arguments        →    expression (COMMA expression)*

constructorDecl  →    CONSTRUCTOR LPAREN parameters? RPAREN EOL
                      block
                      END CONSTRUCTOR EOL

methodDecl       →    METHOD method END METHOD EOL

foreignMethodDecl →   FOREIGN METHOD method END METHOD EOL

staticMethodDecl →    STATIC METHOD method END METHOD EOL

method           →    IDENTIFIER LPAREN parameters? RPAREN EOL block

getter           →    GET IDENTIFIER EOL block END GETTER EOL
```

```
setter          →    SET IDENTIFIER LPAREN IDENTIFIER RPAREN EOL BLOCK EOL

range           →    expression (TO | DOWNTO) expression

then            →    THEN (breakStmt | continueStmt | exitStmt | returnStmt | expressionStmt) terminator
```