Project Checkpoint 2: Parser

75 Points

2/28/2025

Attempt 1

Review Feedback
2/28/2025

Attempt 1 Score:
75

View Feedback

Anonymous Grading: **no**

Unlimited Attempts Allowed
2/12/2025 to 3/3/2025

Details

In this assignment, you will implement the parser for our language. This is the second step in the parsing process which takes the tokens emitted by the lexer and parses them into the AST, giving us a structured representation of our language.

Submission

You will submit a Zip file of the entire `src` folder (containing just the single `src` folder), which should contain both your implementation (`src/main/java`) and tests (`src/test/java`). On Windows, right-click on the src folder and "Compress to ZIP file" should have the expected behavior here.

- The first test submission is **Thursday, February 20**: Project Checkpoint 2: Parser (TEST 1 - Expr) (https://ufl.instructure.com/courses/523541/assignments/6419671) .
  - This covers only `expression` rules, with the exception of `object_expression` (which involves statements).
- The second test submission is **Monday, February 24**: Project Checkpoint 2: Parser (TEST 2 - All) (https://ufl.instructure.com/courses/523541/assignments/6451480) .
  - This covers all rules, statements included.
- The final submission is **Wednesday, February 26**.

Project Setup

See the following links for provided code and setup instructions:

- PlcProject (Parser Patch).zip (https://ufl.instructure.com/courses/523541/files/94912034?wrap=1) (https://ufl.instructure.com/courses/523541/files/94912034/download?download_frd=1) : Contains only new `parser` files and `Main.java` updates. These files should be added to your **existing project** (i.e. the Lexer) following the same folder structure as provided.
- Project Setup (IntelliJ & Gradle) (https://ufl.instructure.com/courses/523541/pages/project-setup-intellij-and-gradle)

Parser Overview

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 (see below).

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

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

first: Binary(+, 1, Binary(*, 2, 3))
second: Binary(+, Binary(*, 1, 2), 3)
```

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. The grammar for our language has been written in a way that supports this.

Crafting Interpreters

The **Parsing (https://www.craftinginterpreters.com/parsing-expressions.html)** 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 their language has more functionality, so make sure you only account for 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 language 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 (https://www.bottlecaps.de/rr/ui)** : Paste into "Edit Grammar", then "View Diagram".
  - **Important**: Escape characters (e.g. `\n` for newline) aren't supported by the tool and will display weirdly.

```
source ::= stmt*

stmt::= let_stmt | def_stmt | if_stmt | for_stmt | return_stmt | expression_or_assignment_stmt
let_stmt ::= 'LET' identifier ('=' expr)? ';'
def_stmt ::= 'DEF' identifier '(' (identifier (',' identifier)*)? ')' 'DO' stmt* 'END'
if_stmt ::= 'IF' expr 'DO' stmt* ('ELSE' stmt*)? 'END'
for_stmt ::= 'FOR' identifier 'IN' expr 'DO' stmt* 'END'
return_stmt ::= 'RETURN' expr? ';'
expression_or_assignment_stmt ::= expr ('=' expr)? ';'
```

```
expr ::= logical_expr
logical_expr ::= comparison_expr (('AND' | 'OR') comparison_expr)*
comparison_expr ::= additive_expr (('<' | '<=' | '>' | '>=' | '==' | '!=') additive_expr)*
additive_expr ::= multiplicative_expr (('+' | '-') multiplicative_expr)*
multiplicative_expr ::= secondary_expr (('*' | '/') secondary_expr)*

secondary_expr ::= primary_expr ('.' identifier ('(' (expr (',' expr)*)? ')')?)*
primary_expr ::= literal_expr | group_expr | object_expr | variable_or_function_expr
literal_expr ::= 'NIL' | 'TRUE' | 'FALSE' | integer | decimal | character | string
group_expr ::= '(' expr')'
object_expr ::= 'OBJECT' identifier? 'DO' let_stmt* def_stmt* 'END'
variable_or_function_expr ::= identifier ('(' (expr (',' expr)*)? ')')?

//these rules correspond to lexer tokens
token ::= identifier | number | character | string | operator
identifier ::= [A-Za-z_] [A-Za-z0-9_-]*
number ::= [+-]? [0-9]+ ('.' [0-9]+)? ('e' [+-]? [0-9]+)?
character ::= ['] ([^'\n\r\\] | escape) [']
string ::= '"' ([^"\n\r\\] | escape)* '"'
escape ::= '\' [bnrt'"\]
operator ::= [<>!=] '='? | [^A-Za-z_0-9'" \b\n\r\t]
```

Notice that each rule corresponds to one of the provided `parse` methods. As in the lexer, you should ensure rules that "delegate" to other rules (e.g. `stmt`/`primary_expr`) do not change the state of the token stream; it's only job is to delegate to the proper rule. Maintaining this separation of concerns will reduce the likelihood of bugs in your lexer.

AST Representation

The following table lists all AST types and example inputs. **Pay particularly attention to error behavior**: A grammar can often be ambiguous whether input should be considered an error (and if so where) or simply lexed/parsed in a different way. For example:

- `LET = 5;`, *as a statement*, could be considered an invalid `LET` statement, but could also be parsed as an assignment to the variable `LET`. Our language will use the former to provide more descriptive error messages to users (namely, missing variable name).
- `LET`, *as an expression*, could be considered invalid as the keyword is also used for `LET` statements, but could also be parsed as the variable `LET`. Our language will use the later as for simplicity we do not have a concept of "reserved words" that limits which identifiers are allowed to be used.

If an error should be thrown, this should use the provided `ParseException` class. This class takes a `String message` - this should be a descriptive message to help identify the issue on your side; tests will not use this message.

AST Type	Description	Examples
`Source`	Represents a full source file containing multiple statements.	`first; second; third;`

`Stmt`	Represents a statement, which performs a side effect as opposed to evaluating to a value like expressions.	`stmt; //Ast.Stmt.Expression`

`Stmt.Let`	Represents a `LET` statement, which defines a variable.	`LET name;`
		`LET name = value;`

`Stmt.Def`	Represents a `DEF` statement, which defines a function.	`DEF name() DO END`
		`DEF name(parameter) DO END`

`Stmt.If`	Represents an `IF` statement. An absent `ELSE` branch is represented as an empty list, same as it were present without statements.	`IF cond DO then; END`
		`IF cond DO then; ELSE else; END`

`Stmt.For`	Represents a `FOR` statement.	`FOR name IN expr DO stmt; END`
`Stmt.Return`	Represents a `RETURN` statement.	`RETURN;`

`Stmt.Expression`	Represents an expression as a statement. For parsing, any expression here is allowed and these will be validated later.	`variable;`
		`function();`

`Stmt.Assignment`	Represents an assignment statement. For parsing, any expression here is allowed and these will be validated later.	`variable = value;`
		`object.property = value;`

`Expr`	Represents an expression in our language, which evaluates to a value.	`expr //Ast.Expr.Variable`

`Expr.Literal`	Represents a literal expression, one of:
	• Nil (`NIL`): Uses `null`.
	• Boolean (`TRUE`/`FALSE`): Uses the `Boolean` class.
	• Integer (e.g. `1`): Uses the [`BigInteger` class (https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/math/BigInteger.html)](https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/math/BigInteger.html), which supports arbitrary precision.
	  ◦ Hint: This one is a bit weird because `new BigInteger(String)` doesn't support exponents. Maybe there's a way to utilize `BigDecimal` for this?
	  ◦ Note: Negative exponents *may* not be representable as integers, so any non-integer value in this case will just be returned as a
		`NIL`
		`TRUE`
		`1`
		`1.0`
		`'c'`
		`"string"`
		`"Hello,\nWorld!"`

	`BigDecimal`. We will exclude this test case from grading as it was an oversight on the original specification. • Decimal (e.g. `1.0`): Uses the `BigDecimal` class (https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/math/BigDecimal.html), which supports arbitrary precision. ◦ Hint: see `new BigDecimal(String)`. • Character (e.g. `'c'`): Uses the `Character` class. You will need to remove the surrounding quotes and replace any escape characters (see String below). • String (e.g. `"string"`): Uses the `String` class. You will need to remove the surrounding quotes and replace any escape characters. ◦ Hint: There's a finite number of escape characters, so `String.replace` is a reasonable option - just pay close attention to the order that replacements happen! ◦ Note: String replacing may cause quirks with inputs containing backslashes along with multiple escapes (e.g `"\\b"` vs `"\\\b"`). We will exclude any test involving backslashes next to other escape sequences from grading as it was an oversight on the original specification. ▪ This bug has been in PLC for *years*	
`Expr.Group`	Represents a group expression, which can change operator precedence.	`(expr)`
`Expr.Binary`	Represents a binary expression (logical, comparison, additive, multiplicative). These are represented as separate rules in our grammar to handle precedence, but can be a single AST class because precedence is instead represented by nesting in the AST (see overview above for details). • Note: Logical expressions use `AND`/`OR` as operators (e.g. Python) - it doesn't just have to be an actual operator symbol!	`1 + 2` `1 * 2` `1 + 2 * 3 // 1 + (2 * 3)` `1 * 2 + 3 // (1 * 2) + 3`

`Expr.Variable`	Represents a variable access.	`variable`
`Expr.Property`	Represents a property access on an object (called the "receiver").	`receiver.property`

`Expr.Function`	Represents a function invocation.	`function()`
		`function(argument)`

`Expr.Method`	Represents a method invocation on an object (called the "receiver").	`receiver.method()`

`Expr.ObjectExpr`	Represents an object literal, which defines an object with fields/methods.
	• Note: This AST uniquely has the `Expr` suffix to avoid confusion with Java's `Object`. This isn't the only way, but works fine for our purposes.
		`OBJECT DO LET field; END`
		`OBJECT DO DEF method() DO END END`

Changelog

2/15	• Updated **PlcProject (Parser Patch).zip (https://ufl.instructure.com/courses/523541/files/94912034?wrap=1)** **(https://ufl.instructure.com/courses/523541/files/94912034/download?download_frd=1)** with fixes for incorrect tests: ◦ testLetStmt Initialization missing `new Ast.Expr.Variable("expr")` value. ◦ testExpressionStmt Function missing `new Token(Token.Type.OPERATOR, ";")` semicolon token. ◦ testAssignmentStmt Variable having name `expr` instead of `variable`. ◦ testBinaryExpr Lower Precedence having operator `+` instead of `*` (inner expression). ◦ testObjectExpr Field having an extraneous `new Token(Token.Type.OPERATOR, ")")` parenthesis token.
2/25	• Defined behavior for negative integer exponents (which may result in decimals); will be excluded from grading. • Defined behavior for backslashes with string replacement, will be excluded from grading. • Note: These were both discussed in lecture previously and have been on test submissions.