CSC-223
Assignment 5
Work in a Pair

DEC Parser

In this assignment, you will implement a parser for the DEC language, a simple programming language designed for this course. This is the next component in our compiler development series: an earlier assignment was implementing a tokenizer that converts raw text into tokens, and the last assignment involved building an Abstract Syntax Tree (AST) structure. Now, we will connect these components by creating a parser that transforms tokens into a meaningful AST representation of input source code text.

What You Need to Do: Continuing From the Last Assignment

You may continue to work with your partner via the same repository or a new repository for this assignment; the same repository is recommended. Please see the “Git and Github Collaboration” section in the first assignment for details.

What You Need to Do: Adding to the Project Structure

As shown in Figure 1, our Parser (and tests) will be defined in the Parser folder ***and*** in the Parser namespace.

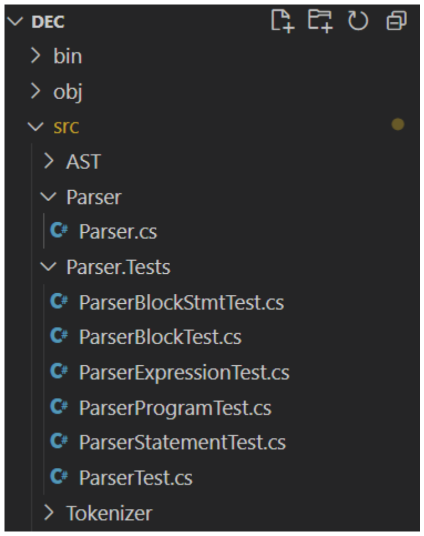

Figure 1: The desired project structure including the parser.

Background: Parsing

Parsing is a crucial step in language processing that occurs after lexical analysis (tokenization). While the tokenizer you previously built converts raw text into meaningful tokens, the parser takes these tokens and organizes them into a hierarchical structure according to the grammar rules of the language. The resulting structure will populate the AST classes you implemented in the previous assignment, representing the syntactic structure of the program and serving as the foundation for further processing, such as semantic analysis, optimization, and code generation.

Background: Grammar and Syntax

Programming languages follow syntactic rules that define valid program structure, just as natural languages have grammar rules. A formal grammar precisely specifies how language elements combine to form valid programs. Formal grammars are essential because they provide unambiguous syntax definitions, enable automated program parsing, help compiler developers analyze language features, and establish foundations for proving program correctness.

Background: Backus-Naur Form (BNF)

BNF is a notation technique used to describe the syntax of programming languages. It consists of:

- ***Terminal symbols*** are the basic building blocks of the language that cannot be broken down further by the rules of the grammar. They are called "terminal" because they terminate the derivation process—no further substitution rules can be applied to them. In a programming language context, terminal symbols are the actual tokens that would be produced by your lexical analyzer and may include keywords (e.g., `return`), operators (e.g., `+`, `:=`), punctuation (e.g., `{`, `}`), literals (e.g., numbers and strings), and identifiers (variables). We often also refer to them as terminal symbols since they are represented as leaves in an abstract syntax tree.

- ***Nonterminal symbols*** are abstract syntactic categories that can be replaced by applying the production rules. They represent higher-level concepts in the language (e.g., `expr`, `stmt`).

- ***Production rules*** define how nonterminal symbols can be transformed or "rewritten" into other symbols (both terminals and nonterminals) to generate valid sequences in the language.

Consider an example of BNF notation for our DEC language:

```
expr ::= variable | literal | '(' expr binary_op expr ')'
```

This rule says an `expr` (expression) can be a variable, *or* a literal, *or* a parenthesized expression with a binary operator. The vertical bar `|` represents alternatives, and symbols in quotes are literals that must appear exactly as shown: `'('` and `')'`.

The `binary_op` nonterminal defines what constitutes a valid binary operator: any one of these seven symbols shown below.

```
binary_op ::= '+' | '-' | '*' | '/' | '//' | '%' | '**'
```

When implementing a parser, we follow these grammar rules to determine how to build the corresponding AST structure. ***Each production rule typically corresponds to a parsing method in your code.***

The DEC Language Specification

Our declarative language (DEC) is a simple language consisting of block, assignment, and return statements along with expressions. DEC uses the following syntax:

For statements:

```
stmt        ::= assign_stmt | return_stmt | block_stmt
assign_stmt ::= variable ':=' expr
return_stmt ::= 'return' expr
block_stmt  ::= '{' stmt* '}'
```

For expressions:

```
expr       ::= '(' variable ')'
             | '(' literal ')'
             | '(' expr binary_op expr ')'
binary_op  ::= '+' | '-' | '*' | '/' | '//' | '%' | '**'
variable   ::= [a-z]+
literal    ::= integer | float
integer    ::= [0-9]+
float      ::= [0-9]+ '.' [0-9]+
```

This grammar above contains a few extensions to basic BNF notation to provide a more concise way to express repetition patterns. The notation `stmt*` is interpreted as "zero or more statements," indicating that a block can contain any number of statements (including none). The `+` symbol in `[a-z]+` means "one or more occurrences" of lowercase letters. Thus variables in DEC consist of one or more lowercase alphabetic characters.

An Example

Figure 2 presents a complete DEC program with nested blocks and variable scoping. Note that the DEC code does not permit comments and each instruction must be on separate lines, including the beginning and ending of a block.

```
{
  x := (5)
  y := (10)
  z := (x + y)
  {
    a := (z * 2)
    return (a)
  }
  return (z)
}
```

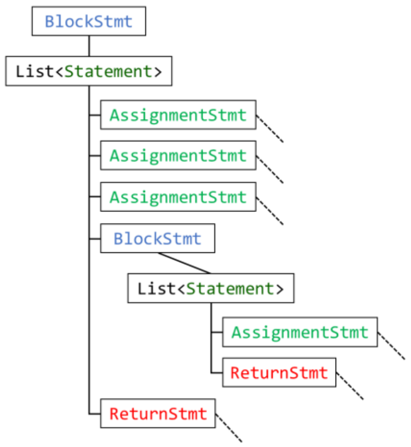

Figure 2: A sample DEC program (left) and its corresponding object-based representation using our AST from the last assignment (right).

This example showcases key features of DEC: variable assignment, arithmetic operations, nested scopes (with variables from outer scopes being accessible in inner scopes), and return statements.

1. The program starts with a main block enclosed in `{ }` (required for all programs).
2. Three variables are defined in the outer scope: `x`, `y`, and `z`.
3. A nested block is created, introducing a new scope.
   - Within this nested scope, `a` is defined and would be initialized to 30 since `z` from the parent scope is accessible.
   - The nested block returns `a` (30); this return would thus exit the program.

4. Finally, the main block has an unreachable return statement.

What You Need to Do: Parser Implementation

Your task will be to parse input DEC program code: tokenizing the text and creating an AST representation of the code as shown in Figure 2.

The entry point to the parser will be the static method `Parse` method in the `Parser` class:

```
public static class Parser
{
    public static AST.BlockStmt Parse(string program)
```

Recall that a static method belongs to the class itself rather than to any specific instance (object) of the class. This means that a user will not need to create an instance of the `Parser` class to parse a DEC program. Since static methods cannot access instance-specific data, this is acceptable because our parser does not require any internal information to be stored. Additionally, it is important to remember that static methods can only call other static members of the class, including methods and fields. Therefore, all supporting methods must also be declared as static.

In greater detail, the `Parse` method serves as the main entry point for parsing a DEC program. It takes a complete program text as a string parameter, converts it to individual lines of code, and returns an AST with an `AST.BlockStmt` root node. During execution, `Parse` creates a new `SymbolTable` and passes it to the `ParseBlockStmt` method to maintain variable scoping. The method performs initial validation to ensure the program follows basic structural requirements—specifically, checking that the program starts with an opening brace `{` and ends with a closing brace `}`. If these syntax requirements are not met, it throws a `ParseException` with a descriptive error message, helping developers identify and fix issues in their DEC code. This method encapsulates the entire parsing process, delegating the details of statement and expression parsing to more specialized methods while maintaining responsibility for the overall program structure validation.

Implementation strategy. Parsing can be complex, therefore a regimented approach will help to ensure a robust and efficient parser. We will implement all methods listed in Table 1, we recommend you work from the "bottom up" with respect to the AST. That is, we should implement ***and test*** (with the tests provided) in the following order: expressions, statements, blocks, and finally the top-level `Parse`. By following this strategy, you can test each layer of functionality before moving on to the next. The tests provided will validate your implementation at each step.

Table 1: Methods to be implemented in the `Parser` class.

Return Type	Method	Arguments	Description	Consumes	Exceptions
Expressions					
`AST.ExpressionNode`	`ParseExpression`	`List<Token>`	Parses an expression enclosed in parentheses.	Consumes `(` and eventually `)`.	`ParseException` if the expression syntax is invalid: starts with a `(` and ends with a `)`.
`AST.ExpressionNode`	`ParseExpressionContent`	`List<Tokenizer.Token>`	Parses the content of an expression.	Consumes the expression one token at a time.	`ParseException` if the expression syntax is invalid.
`AST.ExpressionNode`	`HandleSingleToken`	`Tokenizer.Token`	Handles a single token expression (variable or int / float literal).		`ParseException` if the token is invalid.
`AST.ExpressionNode`	`CreateBinaryOperatorNode`	`string op,` `AST.ExpressionNode l,` `AST.ExpressionNode r`	Creates the appropriate binary operator node based on the operator.		`ParseException` if the operator is invalid.
`AST.VariableNode`	`ParseVariableNode`	`string`	Validates and creates a variable node.		`ParseException` if the variable name is invalid.
Individual Statements					
`AST.AssignmentStmt`	`ParseAssignmentStmt`	`List<Tokenizer.Token>,` `SymbolTable`	Parses an assignment statement and adds the variable as a key to the symbol table (with a `null` value).		`ParseException` if the assignment operator is invalid.
`AST.ReturnStmt`	`ParseReturnStatement`	`List<Tokenizer.Token>`	Parses a return statement.		`ParseException` if the return statement contains an empty expression.
`AST.Statement`	`ParseStatement`	`List<Tokenizer.Token>,` `SymbolTable`	Determines the type of statement and delegates to the appropriate parsing method: assignment or return.		`ParseException` if an unknown statement is encountered.
Blocks					
`void`	`ParseStmtList`	`List<string> lines,` `BlockStmt`	Parses a list of statements within a block. The list of statements may include a new block that should be handled recursively.	Consumes all statements until an end to the block: `}`.	`ParseException` if the program ends unexpectedly or invalid character is encountered.
`AST.BlockStmt`	`ParseBlockStmt`	`List<string> lines,` `SymbolTable`	Initiates parsing of a block. Should be called recursively as needed.	Consumes `{` and eventually `}`.	`ParseException` if the block does not begin with `{` and end with `}`.

Recursion. Recursion is the perfect mirror to the recursive structure of programming language grammars, making it the natural paradigm for implementing parsers. In a language like DEC, expressions can contain other expressions, and blocks can contain other blocks, creating a fundamentally nested structure. This recursive nature is elegantly captured in production rules like `expr ::= '(' expr binary_op expr ')'` where the definition of an expression includes references to itself. When implementing a parser, this recursive grammar structure translates directly into recursive method calls. The `ParseExpression` method might call `ParseExpressionContent` and subsequently calls `ParseExpression` when encountering a nested expression, thus creating a call stack that mirrors the depth of expression nesting in the source code.

This recursive approach allows parsers to handle arbitrarily complex nesting without predefined limits. For example, when parsing a complex expression like `(x + (y * (z - 5)))`, the parser recursively descends into each level of parentheses, building the appropriate AST nodes at each level before returning up the call chain. Similarly, when parsing nested blocks, the `ParseBlockStmt->ParseStmtList ParseBlockStmt` indirect recursive chain can handle blocks within blocks. This elegant symmetry between language grammar and parsing implementation makes recursion not just a useful technique, but the most natural and expressive way to implement a parser that faithfully represents the structure of the language it processes.

What You Need to Do: Testing

All tests have been provided for this assignment; see Figure 1. You should follow the recommendations in the "Parser Implementation" section above.

What You Need to Do: Commenting

Comment code consistent with the standards stated in the first assignment: when needed, use an LLM to generate comments then refine as needed.

Submitting: Source Code

You will demonstrate your source code to the instructor, in person. Be ready to demonstrate successful execution of all unit tests and describe all tests and code you were required to implement.