CSC-223
Assignment 4
Work in a Pair

# Abstract Syntax Tree and Builders for DEC

Abstract Syntax Trees (ASTs) serve as a fundamental bridge between the textual representation of code and its logical structure thus enabling a wide range of language processing tasks. By transforming linear text into a hierarchical tree structure that captures the semantic relationships between language elements, ASTs make it possible to analyze, transform, and execute code with precision. The Builder pattern enhances AST construction by separating the complex process of creating tree nodes from their representation. Rather than focusing on step-by-step construction, the primary value of a Builder lies in its ability to provide different construction strategies (like debugging or testing variants) without changing how clients interact with the AST. This clean separation allows language tools to adapt to various needs while maintaining a consistent interface, ultimately creating more flexible and maintainable compiler or interpreter implementations.

This assignment asks us to implement the core components of a language processor by creating classes that support an Abstract Syntax Tree (AST) along with the necessary code to build these structures.

## 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: AST Class Hierarchy

Abstract Syntax Trees provide a structured representation of code that captures its logical organization while abstracting away syntactic details, making it possible to analyze and transform programs systematically.

***Expression AST Example.*** We introduce this idea by example. Consider an assignment statement in our language and its corresponding expression tree (Figure 1a) and abstract syntax tree (Figure 1b).

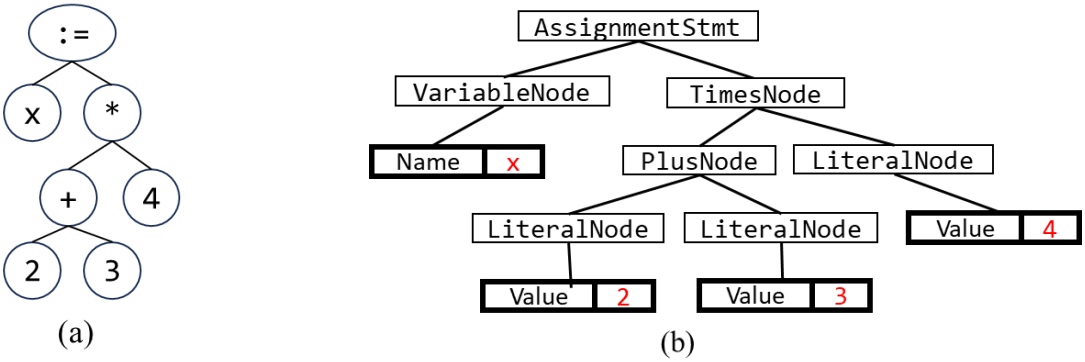



Figure 1: The (a) expression tree and (b) abstract syntax tree corresponding to `x:= ((2 + 3) * 4)` in our DEC language.

The statement `x := (2 + 3) * 4` would be represented in an expression tree (Figure 1a) similarly to how we would represent it as an AST (see Figure 1b). The AST would start with an `AssignmentStmt` at the root, with left child being a `VariableNode` for `x` and a `TimesNode` as the right child. As is clear in the right-hand side of the assignment statement, the expression is more than one operation: two

expressions are being multiplied, a `PlusNode` (with two `LiteralNode` children representing `2` and `3`) and another `LiteralNode` representing `4`.

Observe in this hierarchical structure, the relative positioning of nodes inherently captures the order of operations, eliminating the need for explicit parentheses—the `PlusNode` being a child of the `TimesNode` automatically ensures the addition happens before multiplication, preserving the semantics of the original expression without relying on textual syntax.

Our task is to implement expression classes for the AST as described in Table 1.

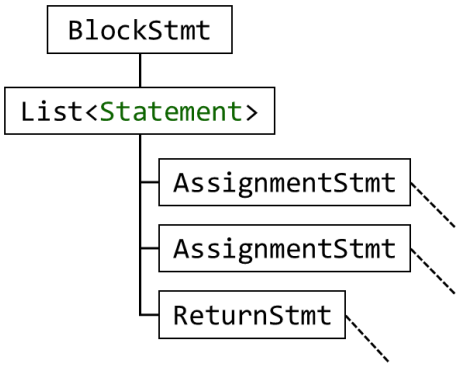
Table 1: Abstract AST Classes and their Concrete class children for representing expressions.

| Abstract Class | Child Classes |
|---|---|
| `ExpressionNode` | `Operator`, `LiteralNode`, `VariableNode` |
| `Operator` | `BinaryOperator` |
| `BinaryOperator` | `PlusNode`, `MinusNode`, `TimesNode`, `FloatDivNode`, `IntDivNode`, `ModulusNode`, `ExponentiationNode` |

***Statement AST Example.*** ASTs can extend far beyond representing simple expressions to capture entire programs. They can represent both value-producing expressions and action-performing constructs like assignments and control flow. Indeed, the previous example (Figure 1) explored an assignment statement, not just an expression. DEC offers a few types of statements that are explored in Figure 2.

```
{
    a := (11)
    b := (4)
    return (a // b)
}
```

(a)




(b)

Figure 2: (a) A code block in DEC consisting of two assignment statements and a return statement along with (b) the corresponding, incomplete, AST.

The example in Figure 2 contains a block (indicated with `{` and `}`) with two assignment statements and a return statement performing integer division. In the AST implementation, the entire block would be represented by a `BlockStmt` object containing three statements in its list. The first two are `AssignmentStmt` objects, each with a `VariableNode` (`a` or `b`) on the left and a `LiteralNode` (`11` or `4`) on the right. The final statement is a `ReturnStmt` containing an `IntDivNode` with two `VariableNode` operands.

Our next task is to implement statement classes for the AST as described in Table 2.

Table 2: Abstract AST Classes and their Concrete class children for representing expressions.

| Abstract Class | Child Classes |
|---|---|
| `Statement` | `BlockStmt`, `AssignmentStmt`, `ReturnStmt` |

***Unparsing.*** The implementation of the top-level abstract classes (`ExpressionNode` and `Statement`) *must* define an abstract `Unparse` method.

```
public abstract string Unparse(int level = 0);
```

The `Unparse` method in AST classes serves as a critical reverse-engineering tool that converts the intermediate, hierarchical representation back to a readable version of the source code. It supports debugging by verifying correct AST construction (this programming assignment) and enabling code transformation by externalizing modified ASTs (future programming assignments). Each AST class implements its own `Unparse` method. Nodes that contain other nodes (like `BlockStmt` or `BinaryOperator`) call `Unparse` on their child nodes, then add their own syntax (like braces or operators) to create the complete source code representation. This recursive approach elegantly mirrors the AST structure itself, making verification of parser correctness relatively straightforward.

Unparsing also facilitates 'pretty-printing' of source code. When unparsing, each AST node receives a level value indicating its nesting depth for managing indentation. Container nodes like `BlockStmt` use this level to indent their opening and closing curly braces, then increment `level` when calling `Unparse` on their children. This recursive approach ensures nested statements appear with progressively deeper indentation, making the generated code visually represent its logical structure. Remember the `GetIndentation` utility method you wrote in assignment 0? Use it to create readable code that preserves the hierarchical relationship of program elements.

## What You Need to Do: Builders

Now that we have discussed and (hopefully) you implemented the AST classes, our goal will be to build ASTs. The process of taking source code and converting it into an AST is referred to as parsing. Our goal in this assignment is not to build a parser (that is the next assignment); instead, our goal in this assignment is to implement tools that the parser will use.

***Problem.*** A parser can create an Abstract Syntax Tree (AST), but this means the parser performs two separate functions: analyzing the program's structure and constructing the AST. If a user only needs to verify whether a program has valid structure, building the AST becomes unnecessary. It would be beneficial to separate these processes so users can avoid the construction step when it is not needed.

**Solution.** The solution is to separate structural analysis from AST creation by implementing a dedicated Builder class. Instead of directly creating nodes, the structural analyzer would request the Builder to construct and return nodes when needed. This approach allows flexibility through different builder implementations:

1. For AST construction, use a `DefaultBuilder` that creates nodes as originally performed.
2. For structure validation only, use a `NullBuilder` that returns `null` instead of creating nodes.
3. For debugging, use a `DebugBuilder` that adds diagnostic output during node construction.

This separation of concerns lets users choose the appropriate implementation based on their specific needs.

The `NullBuilder` has been provided for you; it will be your responsibility to implement the `DefaultBuilder` and `DebugBuilder`. The implementation of the `DebugBuilder` is to output information to the `Console` and return the appropriate objects using the `DefaultBuilder` base class.

## What You Need to Do: Adding to the Project Structure

As shown in Figure 3, our AST and Builder source code will be situated in the AST folder ***and*** in the AST namespace.

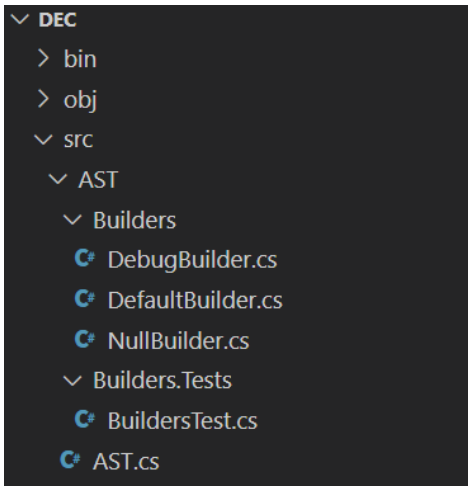



Figure 3: The desired project structure including the AST folder, implementation, and builders.

## What You Need to Do: Testing

All tests should be xUnit tests and adhere to the norms of `xUnit` including using `[Fact]` and `[Theory]`.

***New to CSC-223!*** LLMs can be helpful tools for generating testing code for your assignments. When using an LLM to create tests, include prompts like "write thorough tests for all methods", "make sure to test with duplicate data as well" to ensure comprehensive coverage, and "use `[Theory]` and `InlineData` to avoid redundancy in testing code". ***However, be warned that generated code isn't always perfect—your professor has encountered several cases where tests were written incorrectly for the corresponding code.*** Always review each test carefully to verify it is testing what you intend and functioning properly with your implementation. It is also fair to ask the LLM to see if any cases have been missed in the tests that it generates. In fact, it is advised that you ask an LLM to 'check its work'; do not accept a first answer.

Even though you may use an LLM for testing, you will explicitly be asked about test coverage.

## 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.