BARZ Compiler: An Overview of the Abstract Syntax Tree Generation

Ismael Talib Ridha Barzani
Student ID: 40188139
Department of Electrical and Computer Engineering
Concordia University - Gina Cody School
Montreal, Quebec, Canada
ismaelmergasori@gmail.com

***Abstract*—Generating a useful Abstract Syntax Tree (AST) is usually the final step in the development of the front-end phase of a compiler. The AST serves as an intermediate representation that captures the syntactic structure of the code. A rudimentary syntax analyzer usually generates a parse tree which suffers from the inclusion of semantically meaningless symbols and the dispersion of semantic information induced due to the removal of ambiguities and left recursions as the initial grammar is transformed to an unambiguous LL(1) grammar. Using syntax-directed translation, the parser can be complemented with the generation of a useful and efficient AST data structure. This tree-based structure replaces the linear token sequence and provides a representation for subsequent compiler phases to perform further analysis, augmentation, and transformation. In this report, I present the changes and augmentations done to the LL(1) grammar and the previous `Parser` implementation. Firstly, I list introduced semantic actions to the LL(1) grammar and give a brief description of each action. Secondly, the design and implementation of the new version of the parser are described. Lastly, used tools or libraries are listed.**

***Index Terms*—Compiler Design, Abstract Syntax Tree, Attribute LL(1) Grammar**

I. Introduction

Modern compiler design benefits from a clear division between syntax and semantics. In this project, the traditional LL(1) parser has been augmented using syntax-directed translation to construct an Abstract Syntax Tree (AST) directly during parsing. Rather than generating a verbose parse tree that includes many semantically irrelevant tokens, the parser now builds a compact AST that captures only the meaningful syntactic constructs. The integration is achieved by modifying the grammar to include semantic actions (each marked by a preceding "_" symbol) that invoke AST node creation and manipulation routines. The provided code demonstrates this integration: the `ASTNode` and `AST` classes form the backbone of the AST structure, while the parser calls a generic condition-directed function `ast.performAction()` to execute embedded semantic actions. This approach not only simplifies further semantic analysis and code generation but also enables visualization through Graphviz dot output, thus facilitating debugging and comprehension of the compiler's front-end.

This is submitted as part of COMP-6421 Assignment #3.

II. Attribute Grammar Specifications

The attribute grammar is obtained by inserting semantic action symbols in the LL(1) grammar. A brief description of each of the actions is listed below.

A. Program Structure Actions

- `_createRoot`: Establishes the foundation of the entire AST by creating the program root node with three main branches - a list for all class declarations, a list for global function definitions, and a list for class implementations. This forms the top-level structure that will contain all program elements.
- `_addToProgram`: Organizes program components by placing each declared class, function definition, or class implementation into its appropriate collection in the program tree, maintaining clear separation of concerns between different program element types.

B. Class-Related Actions

- `_createClassId`: Records a class's identity by storing its name in a dedicated node, which will later become part of the class declaration structure.
- `_createInheritanceList`: Prepares a container to track a class's parent classes for inheritance relationships, enabling the object-oriented hierarchy.
- `_addInheritanceId`: Builds the inheritance hierarchy by adding a parent class name to the current class's inheritance list.
- `_makeInheritanceList`: Finalizes the inheritance relationships by processing all parent class identifiers into a complete inheritance hierarchy.

- `_createClass`: Assembles a complete class declaration by combining its name, inheritance relationships, and member declarations into a cohesive class definition node.

C. Implementation-Related Actions

- `_createImplementationId`: Identifies which class is being implemented with a node representing the class name.
- `_addImplementationFunction`: Extends a class's behavior by adding a method implementation to the implementation block.
- `_createImplementation`: Connects a class with its functional behavior by creating a node that associates the class name with its implemented methods.

D. Function-Related Actions

- `_createFunctionId`: Records a function's identity by storing its name in a dedicated node.
- `_setConstructor`: Marks a function as a special class constructor method, which will handle object initialization.
- `_createFunctionSignature`: Defines a function's interface by combining its name, parameter list, and return type into a signature.
- `_createConstructorSignature`: Creates a specialized signature for class constructors with parameters but no explicit return type.
- `_createFunctionBody`: Forms the implementation part of a function by gathering statements into a cohesive body.
- `_createFunction`: Combines a function's signature (interface) with its body (implementation) to create a complete function definition.
- `_createFunctionDeclaration`: Creates a member function declaration for class interfaces without implementation details.

E. Member-Related Actions

- `_setVisibility`: Controls access to class members by specifying their visibility as either public or private.
- `_addMember`: Expands a class definition by adding a new member (method or attribute) with its visibility specification.
- `_createAttribute`: Defines a class-level variable that will store object state information.

F. Variable-Related Actions

- `_setVariableId`: Assigns an identifier name to a variable being declared.
- `_setVariableType`: Specifies what kind of data a variable will store by assigning its type.
- `_createVariable`: Combines a variable's name and type information to form a complete variable declaration.
- `_createLocalVariable`: Creates a function-scoped variable that exists only within its containing function.
- `_setTypeInt`, `_setTypeFloat`, `_setTypeCustom`, `_setTypeVoid`: Specify particular data types for variables or return values, handling primitive types and custom class types.
- `_addArrayDimension`, `_addDynamicArrayDimension`: Build array type information by adding size specifications or indicating dynamic sizing.
- `_processArraySize`: Integrates array dimensions with the base type to create a complete array type specification.

G. Statement-Related Actions

- `_addBodyStatement`: Builds up a function's implementation by adding a statement to its body.
- `_createExpressionStatement`: Transforms an expression (like an assignment or function call) into a complete statement.
- `_createIfStatement`: Constructs a conditional execution structure with a condition, a then-branch, and an else-branch.
- `_createWhileStatement`: Forms a loop structure that repeatedly executes a body while a condition remains true.
- `_createReadStatement`: Creates an input operation that reads a value into a variable.
- `_createWriteStatement`: Creates an output operation that displays an expression's value.
- `_createReturnStatement`: Builds a statement that specifies a function's return value and terminates execution.
- `_createBlock`: Groups multiple statements together into a single compound statement block with its own scope.
- `_createSingleStatement`: Wraps an individual statement to be used where a block might be expected.
- `_addStatement`: Incorporates a statement into a growing block of statements.

H. Expression-Related Actions

- `_pushIdentifier`, `_setSelfIdentifier`: Track references to variables or the current object instance in expressions.
- `_pushFloatLiteral`, `_pushIntLiteral`: Handle numeric constant values in expressions.
- `_setRelop`: Specifies a comparison operator (equal, not equal, less than, etc.) for building conditional expressions.

- `_createRelationalExpr`: Forms a comparison between two expressions using a relational operator.
- `_pushAddOp`, `_processAddOp`: Handle addition, subtraction, and logical OR operations in expressions.
- `_pushMultOp`, `_processMultOp`: Handle multiplication, division, and logical AND operations in expressions.
- `_pushSign`, `_applySign`: Process positive and negative signs applied to expressions.
- `_finishExpression`, `_finishArithExpr`, `_finishTerm`, `_finishFactor`: Complete the construction of expression components at different precedence levels.
- `_handleParenExpr`: Manages expressions within parentheses to enforce evaluation order.
- `_createNotExpr`: Forms a logical negation expression that inverts a boolean value.

I. Array and Function Call Actions

- `_createArrayIndex`: Builds an expression that represents accessing a specific element in an array.
- `_processArrayAccess`: Handles array indexing operations when accessing array elements.
- `_createFunctionCall`: Assembles a function invocation with its name and argument list.
- `_addActualParam`: Builds a function call's argument list by adding parameter expressions.
- `_continueParamList`: Manages the process of adding multiple parameters to a parameter list.
- `_setParamId`, `_setParamType`: Record a formal parameter's name and type information.
- `_createFormalParam`: Constructs a complete formal parameter declaration with name and type.
- `_addFormalParam`: Incorporates a parameter into a function's formal parameter list.

J. Object-Oriented Actions

- `_processDotAccess`: Handles object member access using dot notation (object.member).
- `_pushDotIdentifier`: Records the member name in a dot notation access.
- `_processFunctionReturn`: Manages chained method calls where a method's return value is accessed (obj.method().field).
- `_createAssignment`: Forms an operation that assigns a value to a variable or object field.
- `_setAssignOperator`: Specifies the assignment operator for variable modification.

III. Design Decisions

Several key design decisions underpin the architecture of the AST generation and parser augmentation:

- **AST Node Structure and Management:** The `ASTNode` class is designed to represent individual nodes in the AST. It features pointers to the leftmost child, leftmost sibling, right sibling, and parent—providing a flexible structure that supports various tree traversal strategies. Each node is assigned a unique number (tracked by a static counter), which aids in debugging and visualization. This pointer-based design enables efficient adoption of children and sibling nodes via methods such as `adoptChildren()` and `makeSiblings()`.
- **Centralized AST Construction:** The `AST` class acts as both a factory and a manager for the AST. It encapsulates common operations including node creation (`createNode()`), assembly of family nodes (`makeFamily()` for both binary and variadic cases), and tree output through the `writeToFile()` method. By isolating AST-related logic in this module, the overall system gains modularity and ease of maintenance. The use of a semantic stack (implemented as a vector) in the `AST` class further aids in the orderly construction of the tree by temporarily holding nodes during the parsing process.
- **Integration of Semantic Actions in the Parser:** The parser has been enhanced to recognize when a symbol on the parse stack represents a semantic action rather than a standard terminal or non-terminal. When such an action is encountered, the parser calls `ast.performAction(x, currentLexeme)` to trigger the corresponding AST manipulation. This tight coupling of syntax and semantics allows for the AST to be constructed concurrently with parsing, eliminating the need for a separate post-processing phase. Additionally, by updating the derivation string and managing the semantic stack concurrently, the parser maintains both syntactic derivations and semantic structure coherently.
- **Handling Attribute Migration:** To support attribute migration during parsing, a secondary semantic stack is used. This design choice simplifies the management of AST node attributes as subtrees are built and later adopted into higher-level nodes. The approach ensures that each semantic action has immediate access to the subtrees it needs to process, facilitating the creation of composite nodes via the `makeFamily()` routines.
- **Graphviz Integration for Visualization:** The decision to output the final AST in Graphviz dot format (via the `writeToFile()` method in the `AST` class) was driven by the need for a visual representation of the tree. This visualization aids in verifying the correctness of the AST structure

and helps in debugging complex syntax-directed translations.

IV. Conclusion

Integrating syntax-directed translation into the LL(1) parser has resulted in a streamlined and semantically rich AST generation process. By embedding semantic actions directly within the grammar and leveraging a dedicated AST data structure, the parser is now capable of dynamically constructing an AST that filters out syntactically extraneous details. The `ASTNode` and `AST` classes provide a robust and flexible infrastructure for representing complex program constructs, while the use of a semantic stack ensures orderly attribute migration and subtree assembly. Additionally, the ability to output the AST as a Graphviz dot file adds a valuable visual dimension to the debugging and verification process. Overall, the design decisions made in this project not only enhance the efficiency and clarity of the compiler's front-end but also establish a strong foundation for further semantic analysis and code generation phases.