VYPa Compiler

Name: Pablo López
VUT login: xlopezp00
Team members: 1 (Me)

Before starting with the project documentation, I wanted to comment that I have only been able to implement the following parts of the compiler: Lexer, parser, symbol table and semantic analysis. Due to lack of time I have not been able to develop the remaining parts.

How to run it:

First, you situate in the project directory: using cd vypa_compiler/
Then execute: make vypcomp
Finally: ./vypcomp tests/—-   any file in tests/directory

make clean to delete the output files and make vypcomp to recompile.

Errors that I could not resolve:

Adding to the symbol table declaration of the following form: int x, y, z;

Test 4, I had to modified the name of the function "void Shape …" :

```
class Shape : Object {
  int id;
  void initializeShape() { print("constructor of Shape"); }
  string toString() { return "instance of Shape " + (string)(this.id); }
}
```

Semantical analysis (could not resolve it because I run out of time):

```
Unknown node type encountered: 12
Unknown node type encountered: 22
Unknown node type encountered: 16
```

How the lexer works:

The lexer identifies and categorizes the different components of the VYPlanguage. These components, also called tokens, include keywords, identifiers, literals, operators, and symbols. The lexer also handles comments and whitespace, ensuring they are ignored during tokenization.

Key Features and Functions
Keywords and Reserved Words:
The lexer matches predefined reserved words such as class, int, string, if, and return. When these words are encountered in the source code, they are converted into specific tokens

(e.g., CLASS, IF, RETURN). This ensures the parser can recognize them during syntax analysis.

Operators and Symbols:
The lexer supports a range of operators (+, -, *, /, ==, etc.) and symbols (;, {, }, :), which are essential for the syntax of the language. Each operator or symbol is matched and converted into its corresponding token.

Identifiers:
Identifiers, such as variable and function names, are recognized as sequences of letters, digits, and underscores starting with a letter or underscore. These are converted into IDENTIFIER tokens and stored with their associated string values.

Literals:
Integer Literals: Numbers are matched using regular expressions, ensuring they are properly parsed and stored as INTEGER_LITERAL tokens with their numeric values.
String Literals: Strings enclosed in double quotes are identified and handled, including escape sequences like \n and \t. These are converted into STRING_LITERAL tokens.

Comments:
Line Comments: Lines beginning with // are ignored by the lexer.
Block Comments: Text enclosed within /* */ is also ignored, regardless of its content. This ensures clean tokenization without interference from commented-out code.

Error Handling:
If an unrecognized character or sequence is encountered, the lexer outputs a lexical error message and attempts to continue tokenizing the remaining input. This feature aids in debugging and provides meaningful feedback for developers.

Miscellaneous:
Whitespace (spaces, tabs, and newlines) is ignored to streamline tokenization, focusing solely on syntactically significant elements.

Integration with the Compiler
The tokens generated by this lexer are passed to the parser, which processes them according to the grammar rules defined in Bison. This modular approach ensures that the lexical and syntactic analyses are cleanly separated, enhancing the maintainability and scalability of the compiler.

This lexer implementation exemplifies a robust and efficient approach to tokenizing a programming language, addressing both functional and usability concerns in the compilation process.

How the parser works:

Yacc/Bison parser for a simple object-oriented language that processes class definitions, function declarations, and various statements. It constructs an Abstract Syntax Tree (AST) during the parsing process and handles semantic analysis, such as symbol table management.

Key Components:
Token Definitions:

Terminal Tokens: Defined using %token for keywords, identifiers, literals, and operators (e.g., CLASS, INT, STRING, IDENTIFIER, etc.).
Union: %union defines types for the token values, including integers, strings, and AST nodes.
Grammar Rules:

Program Structure: A program can consist of class definitions, function definitions, or both. The AST root is built based on these elements.
Class Definitions: Classes are defined with optional inheritance, and their members (attributes and methods) are processed.
Function Definitions: Functions are defined with parameters and a block of code. The function name and signature are checked against the symbol table.
Declarations: Variables can be declared with or without initialization, and are added to the symbol table.
Statements: Includes if, while, return, and assignment statements, along with blocks of statements.
AST Construction:

Each grammar rule builds an AST node, which represents various constructs like classes, functions, variables, expressions, etc.
Special AST nodes are created for different elements such as if, while, function calls, and member accesses.
Symbol Table Management:

Symbol Table: A table that stores information about variables, functions, and classes. Each symbol is checked before being declared to avoid duplicates.
Function and Parameter Handling: Functions and their parameters are added to the symbol table and checked for previous declarations.
Error Handling:

Error Reporting: Syntax errors (e.g., duplicate class or variable declarations) trigger yyerror() messages, notifying the user of the issue.
Semantic Functions:

Functions like add_symbol are used to add symbols to the symbol table and ensure no conflicts.
create*Node functions create various types of AST nodes for classes, functions, and expressions.

How the AST works:

The primary purpose of the Abstract Syntax Tree is to represent the hierarchical structure of the program's source code. It is used during the parsing phase of the compilation process to break down the program into components that are easier to manipulate, analyze, and optimize. The AST also serves as an intermediate representation between the parsing stage and the code generation or interpretation stages in a compiler or interpreter.

The AST allows the compiler to process language constructs efficiently, such as:

- Expressions (e.g., mathematical operations, function calls)
- Statements (e.g., variable assignments, conditional statements)
- Declarations (e.g., variable and function declarations)

The AST is composed of various node types, each representing a different syntactic element of the program. Each node contains specific data relevant to that construct (e.g., operator type, variable name, function arguments). Below are the key concepts and node types defined in the provided code:

Operator Enums
These enums define the types of operators used in the language:

Binary Operators (BinaryOperator): These operators work on two operands (e.g., a + b). Examples include:

- OP_ASSIGN: Assignment (=)
- OP_ADD: Addition (+)
- OP_SUB: Subtraction (-)
- OP_MUL: Multiplication (*)
- OP_DIV: Division (/)
- OP_LT: Less than (<)
- OP_GT: Greater than (>)
- OP_LE: Less than or equal (<=)
- OP_GE: Greater than or equal (>=)
- OP_EQ: Equal to (==)
- OP_NE: Not equal to (!=)

Unary Operators (UnaryOperator): These operators work on a single operand (e.g., -a). Examples include:

- OP_NEG: Negation (-)
- OP_NOT: Logical NOT (!)

Node Types for Language Constructs
These node types represent various constructs in the source language, each corresponding to a syntactic element in the AST:

- AST_PROGRAM: The root node of the AST, representing the entire program. It contains a list of classes and functions.

- AST_CLASS: Represents a class declaration. It includes the class's name, its parent class (if any), and a list of members (e.g., variables, functions).
- AST_FUNCTION: Represents a function declaration. It includes the function's name, return type, parameters, and body.
- AST_DECLARATION: Represents a variable or attribute declaration. It contains the type and name of the variable, along with an optional initialization expression.
- AST_ASSIGNMENT: Represents an assignment statement where a value is assigned to a variable.
- AST_BLOCK: Represents a block of code, which contains a list of statements or sentences.

  .
  . (Not all included)
  .
- AST_PRINT: Represents a print statement that outputs values.

Node Structures:

Each AST node contains a base structure that is shared across all nodes, along with specialized structures for each node type. The base structure includes the type of the node (which is essential for identifying the node in the tree) and a pointer to the next node in a list (if applicable).

Generic AST Node (ASTNode)

- type: Specifies the type of the node (e.g., AST_CLASS, AST_FUNCTION).
- next: A pointer to the next node in the list (if the node is part of a linked list of nodes, such as a list of statements or function parameters).

Specific Node Types

- ASTProgramNode: Represents the program as a whole. Contains lists of classes and functions.
- ASTClassNode: Represents a class declaration, including the class name, its parent class (if any), and the list of class members.
- ASTFunctionNode: Represents a function declaration, including the function name, return type, parameters, and body.
- ASTDeclarationNode: Represents a variable or attribute declaration, with the type, name, and optional initialization.
- ASTBinaryOpNode: Represents a binary operation, with left and right operands and an operator.
- ASTUnaryOpNode: Represents a unary operation, with a single operand and an operator.

  .
  . (Not all included)
  .
- ASTTypeCastNode: Represents a type cast expression

Functions for Node Creation

The following functions are provided to create nodes for the AST. Each function initializes a node of a specific type with the relevant data:

Node Creation Functions:

- createProgramNode: Creates a program node with lists of classes and functions.
- createClassNode: Creates a class node with a name, optional parent class, and a list of members.
- createFunctionNode: Creates a function node with a name, return type, parameters, and body.
- createDeclarationNode: Creates a declaration node for a variable or attribute.
- createUnaryOpNode: Creates a unary operation node (e.g., -a).
  .
  . (Not all included)
  .
- createTypeCastNode: Creates a node representing a type cast operation.

Helper Functions:

- appendNode: Appends a node to a list of nodes

The AST structure provides a systematic way to represent the syntactic and semantic components of a programming language. By organizing the source code into a hierarchical tree of nodes, the AST allows for efficient analysis, optimization, and transformation of the code. This structure is essential in compilers and interpreters, enabling various stages such as syntax checking, semantic analysis, code generation, and optimization. The modular and flexible design makes it easy to extend and adapt the AST for additional language features as needed.

How the Symbol table works:

The symbol table stores information about symbols, which include variables, functions, and classes. Each symbol's details are stored in a Symbol structure, while the SymbolTable structure holds all symbols and tracks their count.

Symbol Structure:

- name: The name of the symbol (e.g., variable or function name).
- type: The symbol's type (e.g., int, float, class).
- defined: A boolean indicating if the symbol is defined.
- is_function: A boolean indicating if the symbol is a function.
- is_class: A boolean indicating if the symbol is a class.
- is_object: A boolean indicating if the symbol is an object (instance of a class).
- For functions:
-
- func.parameters: The function's parameters.
- func.param_count: The number of parameters.
- For classes:
-
- class.methods: The class's methods.

- class.attributes: The class's attributes.
- class.method_count: The number of methods.
- class.attr_count: The number of attributes.
- class.parentClass: The parent class name, if any.

Symbol Table Structure:
- symbols: An array of Symbol structures.
- symbol_count: The number of symbols in the table.

init_symbol_table
Initializes the symbol table by resetting the symbol count and clearing the symbols array.

add_symbol
Adds a new symbol to the table, including function or class details.

find_symbol
Searches for a symbol by name in the table.

print_symbol_table
Prints all symbols in the symbol table.

free_symbol_table
Frees memory used by the symbol table.

Helper Functions for Class and Method Extraction
- extractAttributesFromClassBody: Extracts attributes from a class body in the AST.
- extractMethodsFromClassBody: Extracts methods from a class body in the AST.
- countAttributes: Counts the number of attributes in a class body.
- countMethods: Counts the number of methods in a class body.
- getMemberType: Gets the type of a class member (method or attribute).

The Symbol Table plays a crucial role in managing symbols within a compiler or interpreter, supporting efficient symbol lookup, type checking, and scope management. This header file provides essential functions to initialize, add, find, and print symbols, as well as manage class structures and members.

How the Semantic Analysis works:

The semantic analysis phase checks that the program's constructs are semantically correct. It verifies that variables and functions are properly declared and used, checks for type compatibility, and ensures that the program adheres to the language's semantics.

getNodeType
Gets the type of a given AST node, using the symbol table to resolve symbols such as variables and functions.

performSemanticAnalysis

Performs the semantic analysis on the entire AST, verifying that all declarations are valid and checking for any semantic errors in the program.

check_variable_declaration
Checks whether a variable has been declared in the symbol table. Ensures that a variable is not used before being declared.

check_function_redefinition
Checks if a function is being redefined in the symbol table. Ensures that functions are not re-declared with the same name.

check_types_compatibility
Checks if two types are compatible with each other, typically used when performing operations on variables. It ensures that operations like addition or comparison can be legally performed on the given types.
check_parameter_type
Checks whether the type of a function parameter matches the expected type, ensuring the correct type is passed to the function.

Semantic analysis ensures the correctness of the program's meaning by checking variable declarations, function definitions, and type compatibility. The functions provided in this header file facilitate these checks, ensuring that the code adheres to the semantic rules of the language. This phase helps catch errors early, preventing issues during runtime or further stages of compilation.