Overview

This project will be an extension on the previous project (building a parser). In this lab, you will implement a symbol table class, add support for new data types, implement the type checking of operations, and implement support for declaration statements.

Note: Please update functions implemented in your Project#1 if required. The core logic still remains the same, only update if there is a need.

Grammar

This is the updated grammar inline with the new requirements.

Tokens

VARIABLE	Represents variable identifiers and consists of alphabetic characters.
INTEGER	Represents integer literals.
FLOAT	Represents floating point literals.
CHAR	Represents character literals.
OPERATOR	Represents arithmetic operators, specifically $+, -, *, /$.
ASSIGN	Represents the assignment operator $=$.
SEMICOLON	Represents the semicolon ; used to denote the end of a statement.
PARENTHESIS	Represents open (and close) parentheses used for grouping.
Type	Represents the data type in declarations: 'int'|'float'|'char'.

Grammar

$$
\begin{aligned}
\text{Program} &\rightarrow \text{StatementList} \\
\text{StatementList} &\rightarrow \text{Statement StatementList} \mid \text{Statement} \\
\text{Statement} &\rightarrow \text{Declaration} \mid \text{Assignment} \mid \text{Expression} \\
\text{Declaration} &\rightarrow \text{Type VARIABLE ASSIGN Expression SEMICOLON} \\
&\quad \mid \text{Type VARIABLE SEMICOLON} \\
\text{Type} &\rightarrow \text{'int'} \mid \text{'float'} \mid \text{'char'} \\
\text{Assignment} &\rightarrow \text{VARIABLE ASSIGN Expression SEMICOLON} \\
\text{Expression} &\rightarrow \text{Expression OPERATOR Term} \mid \text{Term} \\
\text{Term} &\rightarrow \text{INTEGER} \mid \text{FLOAT} \mid \text{CHAR} \mid \text{VARIABLE} \mid (\text{Expression})
\end{aligned}
$$

Classes

Lexer

For this class, you will need to update the tokenize function to add the following new functionality:

- Construct regular expressions to support float data types
- Construct regular expressions to support multi character variable names
- Add support for character data types
- Add support for integer data types

Symbol table

This class is designed to store and manage identifiers

Attributes: table: A dictionary to hold identifiers as keys with their respective attributes (type and initialization status) as values.

Functions:

Function: add(identifier, type, initialized=False)
Purpose: Insert a new identifier with its attributes into the symbol table.
Input:

- identifier: The name of the variable to be added.
- type: The data type of the variable (e.g., INTEGER, FLOAT).
- initialized: Optional boolean flag indicating if the variable has been initialized (default is False).

Output: Updates the symbol table with the new identifier. Throws an exception if the identifier already exists to prevent duplicate declarations.
Implementation: Checks for the existence of identifier in the table. If absent, adds the identifier along with the type and initialized status to the table.

Function: is_initialized(identifier)
Purpose: Verify if a given identifier has been marked as initialized.
Input:

- The name of the variable to be checked.

Output: Returns True if the variable is initialized; otherwise, False.
Implementation: Looks up the identifier in the table and returns the initialized status.

Function: set_initialized(identifier)
Purpose: Mark an identifier's initialization status as True once it's assigned a value.
Input:

- identifier: The name of the variable to update.

Output: Sets the initialized flag of the variable to True. Throws an exception if the identifier does not exist.
Implementation: Modifies the initialized status of the identifier in the table.

Function: lookup(identifier)
Purpose: Retrieve the attributes of a given identifier, primarily its type.
Input:

- identifier: The variable name whose attributes are to be retrieved.

Output: Returns the data type of identifier. Throws an exception if the identifier is not found.
Implementation: Accesses the table and retrieves the type for the identifier.

Function: update(identifier, new_type)
Purpose: Update the type attribute of an existing identifier
Input:

- identifier: The name of the variable to update.
- new_type: The new type to be assigned to the variable.

Output: Changes the variable's type in the table. Throws an exception if the identifier does not exist.
Implementation: Updates the type attribute of the identifier to new_type in the table.

Type checker

The TypeChecker is a static class that provides methods to enforce type safety within the programming language, ensuring that the semantics of the language are adhered to during variable assignment and operations.

Functions:

Function: check_assignment(target_type, value_type
Purpose: To determine if a value of one type can be safely assigned to a variable of another type.
Input:

- target_type: The type of the variable receiving the assignment.
- value_type: The type of the value being assigned.

Output: Returns True if the assignment is type-safe; otherwise, False.
Implementation: Compares target_type with value_type and determines whether such an assignment is permissible within the language's rules.

Function: result_type_of_op(left_type, op, right_type)
Purpose: Ascertain the result type of an operation given operand types and the operator.
Input:

- left_type: The type of the left operand.
- op: The operator (e.g., +, -, *, /).
- right_type: The type of the right operand.

Output: The expected result type if the operation is valid. Raises a TypeError if the operation is incompatible with the provided types.
Implementation: Utilizes a set of rules that define valid operations for each type pairing and returns the resulting type accordingly.

Function: check_op(left_type, op, right_type)
Purpose: Validate the applicability of an operation based on operand types and the operator.
Input:

- left_type: The type of the left operand.
- op: The operator.
- right_type: The type of the right operand.

Output: True if the operation is semantically valid. Otherwise, raises a TypeError.
Imlementation Leverages result_type_of_op to assess operation validity. If valid, it returns True; if not, it raises a TypeError.

Parser

Function: parse_declaration
Purpose: parse_declaration is responsible for parsing variable declaration statements, which may include variable initialization. The method handles the intricacies of determining whether a variable is being declared and potentially assigned a value in one statement.
Input:

- The method relies on the internal state of the Parser object, which includes a stream of tokens (the program's lexical components) and the current position within that stream. The tokens are previously generated by the Lexer and are now to be syntactically analyzed.

Output: This method returns a Node object representing the syntax tree node for the declaration statement. The Node encapsulates whether it is a plain declaration or a declaration with an assignment (initialization).
Implementation:

1. **Token Consumption:** The method starts by consuming a token that should correspond to a type specifier (e.g., int, float). It then consumes the next token, which should be an identifier representing the variable's name.
2. **Optional Initialization:** It inspects the subsequent token to determine if it's an assignment operator (=). If an assignment is detected, the method proceeds to consume the operator and parse the following expression which represents the value to be assigned to the variable.

3. **Symbol Table Update:** After the semicolon (;) token is consumed, indicating the end of the declaration statement, the method updates the symbol table, recording the variable and its type.
4. **Node Construction:** Depending on whether an initializer was encountered and parsed:
    - If an initializer is present, it constructs a Node of type DECLARATION_WITH_ASSIGNMENT, attaching the type, variable, and initializer expression as children nodes.
    - Without an initializer, it constructs a Node of type DECLARATION, attaching only the type and variable as children nodes.
5. **Return AST Node**: The newly constructed Node is returned, representing the declaration statement within the AST.

The parse_declaration method is a critical component that ensures the parsed code is semantically represented as a tree structure, reflecting the declarations and initializations that will eventually be translated into machine actions or interpreted as program operations.

Testing

There are six additional unit tests in this project that aim to verify your implementation of your symbol table, new data types, type checking of operations, and parse declaration function

Run the python tester file by executing the following command to test your parser implementation.

```
Python3 test_parser.py
```

Here is an overview of these unit tests:

1. **Test_symbol_table_operations**: Verifies the symbol table's ability to maintain the association between variable names and their data types. It ensures that new symbols can be added, existing symbols can be updated and retrieved, and errors are thrown for lookup failures.

2. **Test_lexer_floats_chars:** Ensures that the lexer correctly identifies and categorizes floating-point numbers, integer literals, and character literals. This test checks the lexer's ability to tokenize a sequence of characters representing various data types.

3. **Test_parser_declaration:** Checks the parser's ability to handle variable declarations without initialization. The parser should correctly update the symbol table with the declared variables and identify any undeclared variables or type mismatches in assignments.

4. **Test_type_mismatch:** Focuses on the parser's ability to detect and reject assignments where the value being assigned does not match the variable's declared type, thereby enforcing type safety.

5. **Test_type_checker:** Evaluates the TypeChecker class to ensure it only allows valid operations between compatible data types. This includes checking assignments and arithmetic operations, ensuring that operations like adding a character to a number are correctly identified as errors.

6. **Test_char_assignment_and_invalid_op:** Tests the parsing and evaluation of character assignments and ensures that exceptions are raised for invalid operations involving characters, such as attempting to add a numeric value to a character.

Grading of Test Cases

The grading of the test cases in this project is done based on the number of test cases passed.

There are six test cases that you need to pass for this project, on top of the ten test cases from the previous project.

Grading: Similar to Project 1 Rubric, specifics to be decided.