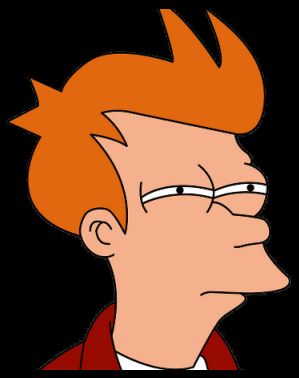
CS326 Project 3:
Programming in Java

Benjamin Cisneros

NOT SURE IF 'X' IS THIS 'X'



OR THAT 'X'...OR THAT 'Y'??

Description

The purpose of this assignment is to learn more about the implications of using *static* and *dynamic* scope. We will put the scoping rules taught in class into practice within a made-up hybrid language called **Jython**. In **Jython**, you will encounter situations where you need to address the binding of names to entities that could potentially represent functions, values, or other names. Here is an example of that

```
def f(n int) int {
  if n == 0 {
    return 1
  }
  return n * f(n - 1)
}

print f(5)
```

This is a function that uses recursion to compute the value of $k!$, that is, $k \times (n-1) \times (n-2) \times ... \times 1$. Even though the scoping rules may seem straightforward in the example above, it is not always simple to bind values and names in the order they appear. For example: When is the type of n resolved, and when is the invocation bound to the *factorial* function. This is something for you to think about... muahahahaha.

Collaboration

Students are encouraged to collaborate within the guidelines specified in the syllabus. However, it is important to emphasize that the primary objective of these assignments is to acquire the essential skills required for success in the field of computer science. Accordingly, students are expected to independently write their own code, ensuring that it is free from any form of plagiarism, whether from peers, past students, or current ones. **Therefore, to maintain academic integrity, all assignments will undergo cross-referencing across all sections of CS326 using the Measure Of Software Similarity (Moss) tool, which automatically assesses program similarities.** You may study together and discuss the assignments, but what you turn in must be your individual work. Assignment submissions will be checked for plagiarism using Moss. **Copying another student's program or sharing your program is a violation of academic integrity. Moss is not fooled by renaming variables, reformatting source code, or re-ordering functions.**

Background

For this programming assignment, we will be using Java to write the interpreter for **Jython**. Fortunately, I have already provided most of the class structure, so your task mainly involves completing a few methods. However, here is a brief overview of how object-oriented programming is expressed in Java syntactically. (Note that you should already be familiar with its semantic aspects from C++!)

- Classes are declared using the `class` keyword. Superclasses are specified using the `extends` keyword (like the colon in C++). For example

  ```
  class Name extends Expression {
    /* fields and meethods here */
  }
  ```

  declares a class called `Name` that is a subclass of `Expression`.

- In a class, both fields and methods can have one of three access levels: `private`, `protected`, or `public`. However, when the access level is not specified, fields can be accessed directly via properties — that is, we can use the name of the field to access it.

- If we want to use polymorphism in a class method, we do not have to declare the method with the `virtual` keyword. In Java, essentially *every* method is declared `virtual`.

- Methods can be defined within class declarations, similar to C++. However, in Java, we do not have method prototypes.

Remember that, unlike the previous programming assignment, you **CANNOT** modify the code based. In short, you **CANNOT** and **SHOULD NOT** modify th code base.

Jython

Jython is a small hybrid language that includes a handful of control structures. While its syntax bears a resemblance to C++ and Python, it is considerably more restricted, and there are notable differences, as we will soon discover. **Jython** specifically supports:

- Integers like `123`, booleans `true` and `false`, and strings.

- Basic arithmetic (`+`, `-`, `*`, and `/`).
- Numeric comparisons (`=`, `<>`, `>`, `>=`, `<`, and `<=`).
- Boolean operators (`and`, `or`, and `not`).
- Input (`print`).
- Simple control structures (`if`, `elif`, `while`, and `for`).
- Variables, declared with the `var` keyword and assigned using the `=` operator.
- User-defined n-ary functions declared with the `def` keyword.
- Finally, it adheres to the Pascal-style conventions for variable and function definitions.

A scanner and parser for **Jython** have already been pre-written for your convenience. They are in the files `Lexer.java` and `Parser.java`. Check them out for the full details of the syntax of the language. Here is an overview of the grammar:

Lexer

```
number
 : digit+ ('.' digit+)?

string
 : '\"' character* '\"'

character
 : letter | digit | symbol

boolean
 : 'True' | 'False'

identifier
 : letter (letter | digit)*

identifier_list
 : identifier (',' identifier)*

letter
 : 'A' | 'B' | 'C' | 'D' | 'E' | 'F' |
 | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' |
 | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' |
 | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' |
 | 'Y' | 'Z' |
 | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' |
 | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' |
 | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' |
 | 's' | 't' | 'u' | 'v' | 'w' | 'x' |
 | 'y' | 'z' |

digit
```

```
 : '0' | '1' | '2' | '3' | '4'
 | '5' | '6' | '7' | '8' | '9'

boolean
 : True | False

symbol
 : ' ' | '!' | '@' | '#' | '$'  | '%'
 | '^' | '&' | '*' | '(' | ')'  | '-'
 | '_' | '=' | '+' | '{' | '}'  | '['
 | ']' | ':' | ';' | '"' | '\'' | '<'
 | '>' | ',' | '.' | '?' | '/'  | '|'
```

Parser

```
script
 : statement* EOF

statement
 : simple_stmt
 | compound_stmt ';'?

simple_stmt
 : expression_stmt
 | declaration_stmt
 | assignment_stmt
 | print_stmt
 | pass_stmt
 | import_stmt

compound_stmt
 : if_stmt
 | while_stmt
 | for_stmt
 | def_stmt
 | class_stmt
 | try_stmt
 | return_stmt

expression_stmt
 : expression

declaration_stmt
 : 'var' identifier type ('=' expression)?

type
 : value_type
 | array_type
 | class_type
```

```
 | 'void'

value_type
 : 'int'
 | 'real'
 | 'string'
 | 'bool'

array_type
 : '[' number ']' value_type

class_type
 : identifier

assignment_stmt
 : identifier assign_op expression ';'?

assign_op
 : '='
 | '+='
 | '-='
 | '*='
 | '/='
 | '%='
 | '**='

print_stmt
 : 'print' expression ';'?

pass_stmt
 : 'pass' ';'?

import_stmt
 : 'import' identifier ';'?
 | 'from' identifier 'import' identifier ';'?

if_stmt
 : 'if' expression block ('elif' expression block)? ('else' block)?

while_stmt
 : 'while' expression ':' block

for_stmt
 : 'for' identifier type 'in' expression block

def_stmt
 : 'def' identifier '(' parameter_list? ')' type block
```

```
class_stmt
 : 'class' identifier block

try_stmt
 : 'try' block ('except' expression block)* ('finally' block)?

return_stmt
 : 'return' expression?

block
 : ':' statement+

parameter_list
 : identifier type (',' identifier type )*

expression
 : or_expr

or_expr
 : and_expr ('or' and_expr)*

and_expr
 : not_expr ('and' not_expr)*

not_expr
 : 'not' not_expr
 | comparison

comparison
 : bit_or_expr (comp_op bit_or_expr)*

comp_op
 : '=='
 | '<'
 | '>'
 | '!='
 | '<='
 | '>='
 | 'is' 'not'?

bit_or_expr
 : xor_expr ('|' xor_expr)*

xor_expr
 : bit_and_expr ('^' bit_and_expr)*

bit_and_expr
 : shift_expr ('&' shift_expr)*
```

```
shift_expr
 : arith_expr (('<<' | '>>') arith_expr)*

arith_expr
 : term (('+' | '-') term)*

term
 : factor (('*' | '/' | '%') factor)*

factor
 : ('+' | '-' | '~') factor
 | power

power
 : atom trailer* ('**' factor)?

atom
 : number
 | string
 | boolean
 | identifier
 | 'None'
 | '[' argument_list? ']'
 | '(' expression ')'
 ;

trailer
 : '(' argument_list? ')'
 | '[' expression ']'
 | '.' identifier

argument_list
 : expression (',' expression)*
```

Let us highlight some noteworthy aspects of **Jython** that you should keep in mind:

- Declared variables must always be initialized. They are assigned an initial value using a statement like the following:

  ```
  var x int = 5
  ```

  and afterwards, they can be reassigned using just the `=` operator without the `var` keyword.

- The **if**, **while**, and **for** constructs do not require parentheses around their conditional expressions. For example, here is a program that repeats a string n number of times:

  ```
  def repeatStr(str string, n int) string {
    var s string = ""
    while n > 0 {
      s += str
  ```

```
    n -= 1
  }
  return s
}

print repeatStr("hello", 5)
```

- A standard **if** statement (without an else clause) appears as you would anticipate:

```
if x < 10 {
  print true
}
```

  However, in the case of an **if-else-if** construct, you utilize the keyword **elif** instead of **else if**, followed by the condition and the code block. Remember to terminate the second clause with the **done** keyword:

```
if x < 10 {
  print true
} elif x > 10 {
  print false
}
```

- Each function is n-ary, meaning it takes n number of arguments, and return values are defined by assigning a value to a variable. Here is an example of a function that computes the factorial of its argument:

```
def f(n int) int {
  if n == 0 {
    return 1
  }
  return n * f(n - 1)
}
```

  Once a function is defined, you invoke it using its name. For example, using the previously defined function, we can print the factorial of 5 like this:

```
print f(5)
```

  where `print` is a keyword that instructs the translator to print a value on the console.

- A **for** statement follows a slightly unconventional syntax, one that is closer to Python. For example, you use a range expression to specify the number of iterations a loop must execute.

```
var x [3]int = [1, 3, 9]
for i int in x {
  print i
}
```

  The loop mentioned above iterates and displays numbers stored in `x`. Keep in mind that a **for** loop can only be used with arrays.

- Finally, no more semicolons! A statements **does not** end with a semicolon.

Translator

You may have noticed that the code provided for this assignment is quite lengthy. This can seem overwhelming, especially when you are working with code you did not write or do not fully understand yet. However, the upside is that you will not have to build everything from the ground up. I suggest jumping into the exercises and focusing on what you need to do rather than trying to grasp every detail of the code all at once. (It is often best to understand what you need to accomplish before diving in!) As always, feel free to reach out if you have any questions. Below is an overview of how the interpreter works.

1. The scanner (`Lexer.java`) reads in the source file (or a single statement) and breaks it up into tokens.
2. The parser (`Parser.java`) combines these tokens according to the grammar rules. As a result, an Abstract Syntax Tree (AST) is created. This represents our program (or statement), containing only essential information that is crucial for the binding process.
3. Each node within the Abstract Syntax Tree (AST) serves as a reference to an object belonging to a subclass of AST. The primary subclasses consist of `Statement` for statements and `Expression` for expressions. The common attributes that these subclasses share are an **array** list of nodes that contains references to differents types of AST nodes, and a **type** that is used when determining the type (i.e., *int*, *string*, or *boolean*) of a local declaration, variable, or expression. Lastly, every `Statement` and `Expression` subclass contains an *heyFry* method designed to visit each node in the AST, and a *addChild* method designed to add nodes to a subtree in the AST.
4. The highest-level node within the AST (which must be of the `Statement` type) is often made of one or more expressions.

For this assignment, you are required to finish implementing the `Symboltable.java`, and `Semantic Analysis.java` Review the provided files and directories to understand the number of statements and expressions present in **Jython**. Dive deep into their contents to grasp the intricate correlations between each of these files.

Control Structures

Jython has the following control constructs: **if**, **elif**, **while**, and **for**. In practice, however, there are effectively only three distinct types, as an **elif** is essentially an **if** statement with a *null* else block when the **else** clause is not used. A statement in Jython contains a reference to nodes that are added to the array list of nodes in the Abstract Syntax Tree (AST) (see the `Node` class). In contrast, a block represents a sequence of statements constructed by the parser, which combines individual statements and appends them to form a list. The **if** statement behaves similarly to the **while** statement. Specifically, the condition of the **if** block must be evaluated first. Once the condition is evaluated, the **if** block is executed. The **else** block, if present, is only evaluated when the **if** condition is false. This differs slightly from the **while** loop, where only the loop condition and the loop body are evaluated repeatedly.

It is important to remember to evaluate the next statement to ensure that the translator processes the statement (or statements) that follow a **block**, an **if**/**elif**, a **while**, or a **for** statement. Here is an example of an **if** statement

```
var x int = 10
var y int = 5
if  x < 20 {
  print true
} else {
  print false
  if y > 10 {
    print "y is greater than 10"
  }
}
```

and its corresponding AST representation

```
+-Program
   |-Var
   |  |-Name: x
   |  |-IntType: int
   |  +-IntLiteral: 10
   |-Var
   |  |-Name: y
   |  |-IntType: int
   |  +-IntLiteral: 5
   +-IfElse
      |-BinaryOperation
      |  |-Name: x
      |  +-IntLiteral: 20
      |-Print
      |  +-BoolLiteral: true
      |-Print
      |  +-BoolLiteral: false
      '-IfElse
         |-BinaryOperation
         |  |-Name: y
         |  +-IntLiteral: 10
         +-Print
            +-StringLiteral: y is greater than 10
```

Variables

There is a `SymbolTable` class defined in scope directory There is also a "global" Symboltable object `table` that will be used to implement a single, global scope. We will also use this `table` to correctly implement static/dynamic scoping. Here is an example:

```
var x int = 10
x = 5
k = x      // ERROR 1
var x = 40 // ERROR 2
```

Take note of two important aspects. Firstly, declarations are **always** initialized. Failing to provide an value to a declaration will result in a syntax error during translation. This initialization is

necessary for the translator to determine the type of the variable. Additionally, it is crucial to generate informative error messages in cases where an identifier is declared twice (i.e., two `var` declarations) or when it is assigned a value before being declared. For example, `ERROR 1` indicates that variable `k` was never declared, while `ERROR 2` indicates that variable `x` already exists, and therefore, we cannot declare another variable with the same name.

What about getting variable reference to work? For example, in

```
var x int = 10  // line 1
x = x + 10      // line 2
print x         // line 3
```

In line 2, there are two references to the variable `x`, and one reference in line 3. How are the references to `x` in line 2 resolved? How can we be sure that adding 10 to `x` is valid? To resolve both references, you will need to use the methods provided by the `Symboltable` class (we will cover this in more detail shortly).

Scopes

This is the challenging aspect of this assignment. Your task is to ensure that static scoping functions correctly with **block**, **if/elif**, **while**, and **for** statements in your **Jython** trasnlator. This can be achieved by enhancing the `Symboltable` class with additional data structures and introducing two new methods, namely *openScope*() and *closeScope*(), which perform the expected actions. Do not forget that you also have to deal with a global scope.

Even if you choose not to follow all the steps exactly as I recommend, I highly advise testing your code frequently as you make progress. The key to the success of a substantial change like this one is to break it down into small, manageable steps and ensure that everything functions correctly before moving forward — test, test, and test before moving on to something else!

- I have already added these methods to the `Symboltable` class. At this point, these methods have no functional purpose other than printing a message when we open and close a scope. Note that we call each of these methods in our `main` method to open and close the global scope. You need to make sure that they are called when you enter and leave a **def** (function) or a **block** (with curly braces).
- The `entries` field in `Symboltable` is hashtable of strings to *Object*s, wheren an *Object* can be anything. The rationale behind utilizing a hashtable is to enable the reuse of the same name (i.e., a string) in various scopes and associate it with distinct values (i.e., different *Object*s). You will have to write the implementations for *lookup*(), *bind*(), and *rebind*(). The *lookup*() and *rebind*() methods should access the `entries` table for the given identifier, while the *bind*() method should add a new value to the `entries` table of the corresponding identifier. Remember that it is an **error** to bind a variable to a value if the variable already exists in the current scope. For example, if you attempt to rebind a variable, meaning you intend to assign its declaration to it, it should already exist in `entries` table. Therefore, you cannot update the value of that variable; in other words, you cannot change the declaraction of this variable since it has already been assigned.
- The entries in the `Symboltable` are stored in a hashtable, mapping strings (variable names) to objects (their values). Each time a new scope is created, a new symbol table is initialized, with its parent pointer set to reference the enclosing scope (the current scope). To manage

this, we use a stack called `scopeVariables`, where the current scope is always at the top, and its parent scope sits directly below it. When a new binding is created, you insert the variable's name and its corresponding value into the entries table of the current scope. When a scope is closed, you simply return to its parent scope — the enclosing scope of the current one.

I understand that this might appear challenging initially, but it was even more challenging for me to create this assignment (although it was quite enjoyable!). My goal is for you to grasp that seemingly simple tasks can be quite complex to implement. I also aim to provide you with insight into the intricacies involved in designing and implementing a programming language.

SymbolTable

The symbol table is a prime example of a structure that requires global management in a top-down parsing translator. In this assignment, you will implement symbol table processing for the **Jython** language. This will involve traversing the Abstract Syntax Tree (AST) using the HeyFry pattern — a technique created by none other than Dr. Vasko during his younger, more powerful years — along with the optional implementation of a symbol table manager.

The symbol table is globally accessible. Each entry in the symbol table contains a string and a value, where value is an *Object*. For example, for a declaration of the form

```
var x int = 10
```

The name `x` should be bound to its declaration, represented as an AST node of type `Var`. However, since **Jython** is a dynamic language, we will bind the variable to its type, which in this case is a node of type `IntType`. It is important to note that different types of declarations store different information about their names. For this assignment, a variable declaration may store its type, as well as whether it is a local, instance, or global variable. In contrast, a function declaration might store its argument types and the function's definition. Consider the following snippet of code:

```
def square(x int) int {
  return x * x
}
```

We need to store both the function declaration and definition. Since this is a dynamic language, simply storing the return type and parameter types would not be enough for the interpreter to correctly handle function calls during execution.

We can summarize a symbol table and its operations using the following bullet points:

- A map (table) from names to information about them
- Each symbol table entry is a binding
- A declaration adds a binding to the map
- A use of a name looks up binding in the map
- Report a type error if none found

Static, Nesting Scoping

A scope defines the region in which a variable is accessible and links variable uses to their declarations. In static scoping (also known as lexical scoping), symbol references (i.e., identifiers) are resolved based on the structure of the source code. Here are some key rules to keep in mind:

- Scopes may be nested (i.e., contained within another)
- Blocks may (or may not) defined new scopes
- There is typically a global scope at the top level
- Statements/expressions may reference symbols of
  - the current scope
  - any parent scope (may depend on lexical ordering), or
  - the global scope
- In general, we cannot reference symbols defined in sibling/child scopes.

How do we resolve names? Solution

- Have one symbol table per scope
- Each scope's symbol table referes to its lexically enclosing scope's symbol table
- The root is the global scope's symbol table
- Look up declaration of name starting with the nearest symbol table, proceed to enclosing symbol tables if not found locally.

Here is a simple example:

```
def foo(n int) void {
  def a int = 0
  while a < n {
      def x int = ...
      ...
      if a > 5 {
        def y int = x
        ...
      }
  }
}
```

Name	Value
n	*
a	*

Name	Value
x	*

Name	Value
y	*

Hey Fry!!!! Over here....

We will consider the **HeyFry** pattern for this assignment, which consits of something called "multiple dispatch".

Multiple Dispatch. Recall, a virtual function in C^{++} enables a method to be customized based on the actual run-time type of the receiving object. (If you are unsure about how virtual functions work, there are plenty of online tutorials available for you to read or watch.) For example, in Java

```
A obj = new B();
obj.foo();
```

will call the *foo* method on the instance *obj*. However, even though *obj* is declared to be of type *A*, its actual run-time type is that of type *B*, which, in the above example, must be a subtype/subclass of type *A*. In essence, the dispatch for *foo* is determined entirely by the actual run-time type of *obj*, irrespective of the run-time type of the supplied parameters. Most compilers locate the appropriate definition of *foo* based on the declared types of the supplied parameters. However, this resolution occurs at compile-time, when the actual types of the parameters may be unknown. Despite the run-time types of the parameters potentially being more refined, this detail does not impact the dispatch.

AST traversal. We are building an abstract syntax tree (AST) for our translator, with various node types all derived from AST. To differentiate the nodes' behaviors, we override methods in their class definitions (in fact, *addChild*() is one example). Additionally, we need to traverse the AST for tasks like printing, name resolution, or type resolution. Each task could be achieved by refining tree traversal in extensions of a shared superclass. Consequently, the action taken on a node depends on its type and the traversal method. With multiple dispatch, both refinements can participate in method dispatch simultaneously.

We will utilize Fry to handle the aforementioned scenario. What we will do is ensure that each node in the AST calls the appropriate code by arranging for them to

- Accept a call from a class that extends Fry and that performs the traversal
- Call Fry back using a method in that class that is customized to a particular AST node

Suppose we want to traverse our AST to print out the content of each node. The traversal class would be a particular kind of Fry action, like this

```
class PrintNodes implements Fry {
  public void overHere(StringLiteral s) {
     println "String literal value is " + s.value;
  }
  public void overHere(IntLiteral i) {
     println "Integer literal value is " + i.value;
  }
  ...
}
```

Each of the methods above knows how to print its own particular kind of node. Each node in the AST, in turn, accepts a call to Fry from a specific class that extends/implements a Fry action, like this

```
class StringLiteral extends Literal {
  ...
  @Override
  public void fryCall(Fry f) {
```

```
    f.overHere(this);
  }
}
```

Now, to print the value of a `StringLiteral`, we can do the following:

```
StringLiteral str = new StringLiteral("hello");
PrintNodes printer = new PrintNodes();
str.fryCall(printer);
```

Please trace the call to become familiar with the HeyFry pattern. Only by tracing the sequence of calls will you be able to understand how we can traverse each node in the AST.

Type Checking

Recall, a *type* is a set of values along with a set of operations that can be performed on those values. A *type system* is a collection of rules that assign types to expressions, statements, and the overall program, determining which operations are valid for which types. It provides a formal framework for checking type correctness, typically by specifying rules about the structure of expressions and how types interact.

We need to implement type checking to ensure that operations are applied to the correct number of arguments and that those arguments have the appropriate types. *Correct type* may refer to the exact type specified, or it could indicate that a predefined implicit coercion will be applied. Let us examine a `BinaryOperation` as an example.

The function `overHere` is used to perform semantic analysis on a binary operation. It checks the types of the operands of the binary operation, ensures they are compatible, and sets the resulting type based on the operator. The function receives a `BinaryOperation` node (node) which represents a binary expression with:

- `node.operator`: The operator (e.g. +, -, *, etc.).
- `node.left`: The left operand of the binary operation.
- `node.right`: The right operand of the binary operation.

The method `fryCall` is invoked on the left operand (node.left). This evaluates the left operand and updates the `currentExprType` variable with its type. The left operand's type is then stored in `lType`. Similarly, the method `fryCall` is invoked on the right operand (node.right), which evaluates the right operand and updates `currentExprType`. The right operand's type is stored in `rType`. If either `lType` (left operand type) or `rType` (right operand type) is null, an error is thrown. This ensures that both operands have valid types. What you need to do is complete the swith statement.

- Switch on the Operator: The switch statement processes each possible operator case, applying specific checks based on the operator and operand types.
- Case 1: *, /, -, **
  - For multiplication, division, subtraction, and exponentiation operators, both operands must be either integers or real numbers (IntType or RealType).
  - If either operand is not of a numerical type, an error is thrown.

- The resulting type is set to IntType if both operands are integers; otherwise, it is set to RealType.

- Case 2: +
  - The + operator has special rules.
  - It does not allow boolean operands.
  - If both operands are strings, the result is a string (StringType).
  - If only one operand is a string, an error is thrown.
  - If either operand is an array, an error is thrown.
  - If both operands are integers or real numbers, the result is either IntType or RealType, depending on the operands.
- Case 3: and, or
  - The logical `and` and `or` operators only work with boolean operands.
  - If either operand is not of type BoolType, an error is thrown.
  - If both operands are booleans, the result is a BoolType.
- Case 4: <, >, <=, >=
  - Relational operators only work with numerical types (either IntType or RealType).
  - If either operand is not a number, an error is thrown.
  - The result of a relational operation is always a boolean (BoolType).
- Case 5: ==, <>
  - The equality (==) and inequality (<>) operators require the operands to be of the same type, with exceptions for comparing integers to real numbers.
  - If the operands are of different types, an error is thrown.
  - The result of an equality/inequality operation is always a boolean (BoolType).
- Default case: If the operator does not match any of the predefined cases, a generic runtime error is thrown.

Let us now look at the `UnaryOperation` node:

1. Check for Numerical Types (Integer or Real):
   - If the currentExprType is either an integer (IntType) or a real number (RealType), the method checks if the operator is either "+" or "-" (the only valid unary operators for numerical types).
   - If the operator is neither "+" nor "-", it throws a RuntimeException indicating an invalid operator for a numerical expression. This prevents using operators like * or / in unary expressions where only + and - are allowed.
2. Check for Boolean Type:

- If the currentExprType is a boolean type, the method checks if the operator is "not" (the only valid unary operator for booleans).
- If the operator is not "not", it throws a RuntimeException indicating that the operator is invalid for a boolean expression.

3. Handle Incompatible Operator Types:
    - If the type of the expression is neither numerical nor boolean (e.g., a string, object, or unknown type), the method throws a RuntimeException indicating that the operator is incompatible with the expression type.

Finally, let us look at the `Assignment` node. Again, the switch block handles different assignment operators by validating whether the operands' types are compatible.

- Case 1: -= , *= , /=
    - These operators are typically used for arithmetic operations. Both operands (the variable vType and the expression eType) must be either integers (IntType) or real numbers (RealType).
    - If either operand is not a numerical type, an error is thrown.
    - If both operands are integers, the resulting type is IntType. If one or both operands are real numbers, the resulting type is RealType.
- Case 2: +=
    - The += operator checks for compatibility between the operands.
    - If the left-hand operand (vType) is a StringType, the right-hand operand (eType) can be an IntType, RealType, BoolType, or StringType.
    - If the right operand is one of these types, the resulting type of the assignment will be StringType.
- Case 3: **= , %=
    - These operators (likely exponentiation and modulo) require the left operand (vType) to be an integer type (IntType).
    - The right-hand operand (eType) should also be an integer or real type.
    - If the types do not match, an error is thrown. The resulting type is IntType if both operands are integers; otherwise, it will be RealType.
- Case 4: =
    - This case handles the assignment operator =.
    - If the right-hand side (eType) is an array type, it checks if the left-hand side (vType) is also an array type. If both are array types, it compares their types using the compareArrayTypes method. If one is an array and the other is not, it throws an error.
    - If both sides are not arrays, it ensures that the types of the left-hand side and right-hand side are compatible (either both integer types, both real types, both boolean types, or both string types). If not, it throws an error.

Resolving variables/functions

Name node

The method is designed to check if the identifier in the Name node corresponds to a function or a variable.

1. If it corresponds to a function, the **currentFunction** is updated.
    - Find a function definition in the functionTable using the identifier (node.identifier) provided in the node.
    - functionTable.lookup() is a method that searches for a function by its name (represented by node.identifier).
    - If a function is found (i.e., function is not null), the currentFunction is set to the function found in the functionTable.
2. If it corresponds to a variable, the **currentExprType** is updated.
    - If no function is found (i.e., function is null), the code proceeds to look for a variable with the same name in the variableTable.
    - variableTable.lookup() searches for a variable by its identifier (node.identifier), and the result is stored in currentExprType.
    - If the variable is also not found (i.e., currentExprType is null), a RuntimeException is thrown.
3. If neither a function nor a variable is found, a RuntimeException is thrown to indicate that the identifier was never declared.

Function node

1. Check for function name duplication: If the function name is already used, an error is thrown.
    - Attempts to find an entry in the functionTable that matches the identifier (name) of the function being defined.
    - If the function is found (i.e., lookup() returns a non-null value), a RuntimeException is thrown with a message indicating that the function name already exists, signaling an error because function names must be unique within the scope.
2. Bind function name to its definition: The function is added to the functionTable so it can be referenced later.
    - The functionTable.bind() method associates the function name (given by node.name.identifier) with the actual function definition (node). The node here represents the function's complete definition, including its parameters, body, etc.
3. Open a new scope: A new scope is created for the function's local variables.
4. Process parameters: Each parameter of the function is processed (e.g., validated, bound, etc.).
5. Process function body: Each statement in the function body is processed.
6. Close the scope: The local scope is closed once the function's body has been processed, completing the function's definition.

Call node

The method checks and validates a function call by:

1. Processing the qualifier (if applicable).
    - The node.qualifier refers to the object or entity on which the function is being invoked (e.g., in someObject.someMethod(), someObject is the qualifier).
    - The fryCall(this) method call performs some processing on the qualifier, passing the current context (this) as an argument.
2. If **currentFunction** is null, meaning the function being called is not recognized, the method throws a runtime exception indicating a semantic error.
3. Collecting the types of the arguments passed.
    - A new ArrayList of Type is created to hold the types of the arguments passed to the function.
4. Ensuring the number of arguments matches the number of parameters.
    - For each argument in the node.arguments list (representing the function's arguments):
        - arg.fryCall(this) is called, which processes the argument (similar to how the qualifier was processed).
        - The currentExprType (the type of the evaluated argument) is added to the ArrayList of Type.
5. Checking that the argument types match the expected parameter types (including arrays).
    - Check if the number of arguments matches the number of parameters .
    - If the sizes do not match, an error is thrown, indicating a mismatch in the number of arguments for the function call.
    - Loop over each argument and parameter type:
        - If both the argument type (aType) and the parameter type (pType) are array types, it delegates the comparison to a helper function compareArrayTypes().
        - If they are not both array types, the code checks whether the argument type matches the parameter type. It ensures that the argument and parameter are of the same type (int, real, bool, or string). If not, it throws an error, indicating the type mismatch.
6. Setting the result type based on the return type of the function.
    - After all argument types are validated, the **currentExprType** is set to the return type of the function being called. This indicates what type of result is expected after the function is executed.
7. Throwing errors if the validation fails at any point, such as if the function is not found or there is a type mismatch.

Make Like a Tree and _____

Remember that the AST is built using a list (i.e., a dynamic array, which is similar to a vector in C++). Again, refer to the AST as you "walk" down the tree and to make sure that you visit every node! Here is the generated AST for the piece of code given in the Introduction section.

```
+-Program
   +-Function
      |-Name: factorial
      |-Parameter
      |  |-Name: n
      |  +-IntType: int
      |-IntType: int
      +-IfElse
         |-BinaryOperation
         |  |-Name: n
         |  +-IntLiteral: 0
         |-Return
         |  +-IntLiteral: 1
         +-Return
            +-BinaryOperation
               |-Name: n
               +-Call
                  |-Name: factorial
                  +-BinaryOperation
                     |-Name: n
                     +-IntLiteral: 1
```

Contents of main

Now let us look at the entry point of the program. In main, we do the following:

1. Prompt for a read a line of code
2. Create tokens from the line of code
3. Create the AST (namely a statement)
4. Execute the statement
5. Repeat until the user enters the command `#quit`

Work List

- Finish the methods *bind*(), *rebind*(), and *lookup*() in `Symboltable.java`
- Finish the implementation of the class `SemanticAnalysis.java`

Specifications

- Document your code.

- Use variables with meaningful names.
- Make sure your code is error proof.
- Do not modify the provided code (if any).
- All of the user input and output is done in main.
- You may assume valid input.
- No global variables (unless specified)

Submission

Submit the source file to Canvas by the deadline.

Sample Run

Refer to the link provided on Canvas.