PLC Jott Interpreter

CSCI-344 Group Project

07/28/2019

1 Introduction

In this project you will create an interpreter for the programming language Jott. You must use Java to code this.

Jott is a very basic programming language. It will not do everything a normal programming language can do.

2 Detail of the Language

Jott is going to be very simple. This section will discuss each of the items you must implement. In a later section the grammar of the language will be laid out.

Each phase of the project will be laid out later in this document.

2.1 Mathematics

Jott will do basic integer/double mathematics; addition (+), subtraction (-) division (/), and multiplication (*).

Integer mathematical operations will only work with two integers as operands and return an integer answer. Integer mathematics with truncate and not round.

Examples:

- `3 + 2` becomes 5
- `3 - 2` becomes 1
- `3 *-2` becomes -6
- `5/2` becomes 2

Note: the spacing between operands and the operator is not required.

Double mathematical operations will only work with two doubles as operands and return a double answer.

Examples:

- `3.0 + 2.0` becomes 5.0
- `3.1 - 2.0` becomes 1.1
- `3.1 *-2.2` becomes -6.82
- `5.0/2.0` becomes 2.5

Note: the spacing between operands and the operator is not required.

Different typed operand expressions, such as `3 + 3.0`, are not allowed in this language; integer operation with a double operand. If this occurs it should be reported as an error.

Division by zero is not allowed and should be reported as an error.

Errors such as this will be validated in Phase 3, and should pass all Phase 2 tests.

There will be no chaining of mathematical operations. For example `3+2*3` is not allowed. This is to avoid order of operations issues.

2.2 Conditional Operations

Jott will do basic True/False conditional operations; greater than (>), greater than or equal (>=), less than (<), less than or equal (<=), equal to (==), and not equals (! =).

They follow the standard rules of other programming languages. The return type in Jott is Boolean.

Examples:

```
Boolean b = 5 > 2;
::print[b];
Boolean b2 = 5<= 2;
::print[b2];
```

Running of this code will output:

```
True
False
```

Note: the spacing between operands and the operator is not required.

Just like with math operation, the operands on either side of the operator must be of the same data type.

2.3 Strings

Strings must be contained in double quotes and cannot wrap lines. They must also contain only upper/lower case alphabetical characters, spaces, or numbers. No punctuation in strings.

Examples:

```
String s; s = "foo";
String y1; y1 = "bar";
String y2; y2 = "123  rf 17";
```

2.4 Variable Assignment and Usage

Jott allows for variable assignment and usage. Variables in Jott are just names for a provided value. Variables are types in Jott; Integer, Double, Boolean, or String; they are case-sensitive.

All variables must start with a lowercase letter and unique in the function scope. This will be validated in Phase 3.

Using a variable in an expression where its type does not match the expression is an error. Validated in Phase 3.

Example:

```
Integer x;
x = 5;
::print[ x + 3.2 ]
```

This is an error because `x` is an integer and `3.2` is a double. This error will be handled and reported in phase 3.

Variable existence will be validated in phase 3. Example:

```
Def foo[]:Integer{
    ::print[x];
}
```

Will pass phase 2 but not phase 3.

See the grammar below for the structure of variable assignment and usage.

Variables can only be declared at the start of a function and are scoped to that function. A variable can only be used if it has been declared first.

Example:

```
Def foo[]:Integer{
    Integer x;
    Double y;
    .... do something with x and y ....
}
```

This is not allowed:

```
Def foo[]:Integer{
    Integer x;
    .... do something  ....
    Double y;
    .... do something ....
}
```

New variables cannot be declared in while loops, if statements, and outisde of a function.

2.5 While Loops

Jott will have a basic while loop. The format is:

```
While[cond]{
    ... body ...
}
```

Details:

- cond is a conditional operation.
- the body will be any valid Jott code other than defining another function.
- nested loops are allowed.
- works like while loops in other languages.

Example:

```
Integer x;
x = 5;
While[ x > 0]{
    ::print[x];
    x = x - 1;
}
```

Running of this code will output:

```
5
4
3
2
1
```

Note: exact spacing as shown in the format above in not required. The following for example is allowed:

```
While [ cond]{
    ... body ...
}

While[cond ] {
    ... body ...
}

While[cond]
{
    ... body ...}
```

A return statement in a while loop is not considered a valid return path from a function as the while loop does not have to run at all.

2.6 If Statements

Jott will have a basic if statement. The format is:

```
If[cond]{
    ... body ...
}
Elseif[cond]{
    ... body ...
}
Else{
    ... body ...
}
```

Note: The same rules apply here with spacing and newlines as they do in while loops.

Details:

- The if is required.

- There can be zero or more elseif that must come after the if but before the else
- There can me only at most one else; there can be none.
- cond is any statement that evaluates to a True or False result.
- nested ifs are allowed.

Examples:

```
Integer x;
Integer y;
x = 5;
If[x == 5]{
    ::print["Yes"];
}
Else{
    ::print["No"];
}


y = 7;

If[y < 6]{
    ::print[4];
}
Elif[ y >= 7]{
    ::print[3];
}
Else{
    ::print[11];
}

If[x + y > 8]{
    ::print[x + y];
}
```

Running of this code would output:

```
Yes
3
12
```

A if statement can only be considered as a valid return path from a function if it has an else statement and all if/elif/else have a return statement.

2.7 Functions

Jott will allow for function definitions and calls.

Format of function definitions in Jott:

```
Def name[ varName:varType, ... ]:returnType{
   ... body ...
   Return ...;
}
```

Details:

- def: states that you are defining a function.
- name: will stand for the name of the function. It will follow the same rules as variable name. The name must be unique; this will be checked and reported in phase 3. A function and a variable can have the same name.
- parameters are enclosed in square brackets, [and]. They will be a comma separated list.
- parameters will be defined in the form `name:type`. Name will follow the same rules as a variable. Type will be any type that a variable can be.
- the body will be any valid Jott code other than defining another function.
- return will return a value from the function. The type of the return must match the return type of the function. There must be a return path from a function with a non-Void return. While loops with a return do not count as a valid return path; as the loop does not have to run. An if statement with and else AND a return in all parts of the if statement are returnable; otherwise they are not. This will be validated in phase 3.
- function with no return will have a return type of `Void`. `Void` is not a valid variable type; proper function return will be validated in phase 3.
- similar to ifs and whiles, the exact spacing above is not require. For example the following is valid:

```
Def name[varName:varType, ... ] :returnType {
line1;line2;Return ...;}
```

Functions must be defined before they can be used in the code. For example if function `foo` calls function `bar` then `bar` must be defined before `foo`. This is validated in phase 3.

Format of a function call:

```
::name[param1, param2, ...]
```

Details:

- name: is the name of the function being called.
- param1, param2, ... are the arguments to the function. The number and type of the arguments must match the function definition; validated in phase 3.

Examples of using functions:

```
Def foo[ x:Integer,y:Double]:String
{
    If[x>5]{
        Return "foo";
    }
    Elif[y<3.2]{
        Return "bar";
```

```
    }

    Return "foobar";
}

bar[s:String]:Void{ ::print[s];}


String x;
::print[::foo[5]];
::print[::foo[3.2]];
x = ::foo[11];
::print[x];
::bar[::foo[5]];
```

2.8 Builtin Functions

Jott will have a few builtin functions. These functions will not be defined in the source file. They are built into the language; an example of this would be `print` in Python.

No other function can be named the same as a builtin function; this will be an error. Validated in phase 3.

They will be called just like any other function.

2.8.1 `print` Statement

Jott will have the ability to print data to the screen and go to the next line. It will return nothing. The parameter to "print" can be any data type in Jott; except Void. It will:

- take in an expression that will be evaluated and printed to the screen.
- it adds a newline after it prints.
- it will return nothing

Examples:

```
::print[3];
::print[3.2];
::print["foobar"];
::print[ 3 + 4 ];
```

Running of this code will output:

```
3
3.2
foobar
7
```

2.8.2 `concat` Statement

Jott will have the ability to concatenate two string together; only strings. It will:

- take in two Strings
- will return a new String that is the second param concatenated to the end of the first param.

Examples:

```
String s;
String s1;
String s2;
s = ::concat["foo", "bar"];
::print[s];
s1 = "foo";
s2 = ::concat["bar", s1];
::print[s2];
::print[::concat["foo", ::concat["bar", "baz"]];
```

Running of this code will output:

```
foobar
barfoo
foobarbaz
```

NOTE: notice the functions can be used as a parameter of another function. This is allowed as long as the return type of the function matches the type of the function parameter.

2.8.3 `length` Statement

Jott will have the ability to determine the length of a string. It will:

- take in a String; validated in phase 3.
- return the length of the string; as an Integer.

Examples:

```
Integer i;
i = ::length["foo"];
::print[i];
```

Running of this code will output:

```
3
```

2.9 Comments

Jott will allow for line comments. Comments will start with `#`. Anything after the `#` and until the newline; newline is to be considered part of the comment.

Examples:

```
#this is a comment
::print[ "hello" ]; #after this is a comment
# no matter what is here this is a comment !123.;;;
```

2.10 `main` Function

All Jott programs must have a main function. This function is responsible for running the program; think on the `main` function in C.

It will take in no command line arguments and return nothing.

Format of the `main` function:

```
Def main[]:Void {
    ... body ...
}
```

Details:

- will take in no params.
- will return nothing.
- no other function can be named main.
- just like other functions exact spacing as shown above is not required.

You must validate its existence and format in phase 3.

Example:

```
def foo[x:Integer]:Void{
    ::print[x];
}

def main[]:Void{
    ::foo[5];
    ::foo[11];
}
```

Running of this code will output:

```
5
11
```

and will exit the program.

3 The Grammar of Jott

This section will outline the grammar of the Jott programming language. Anything that violates this grammar should be reported as an error. You can modify this grammar, but the new grammar must be equivalent.

```
<program> ->  <function_def>*<EOF>

<function_def> -> Def <id>[func_def_params]:<function_return>{<f_body>}
<func_def_params> -> <id>:<type><function_def_params_t>* | ϵ
<func_def_params_t> -> ,<id>:<type>

<body_stmt> -> <if_stmt> | <while_loop> |
```

```
                <asmt> | <func_call>;
<return_stmt> -> Return <expr>; | ϵ
<body> -> <body_stmt>*<return_stmt>
<f_body> -> <var_dec>*<body>

<if_stmt> -> If[<expr>]{<body>}<elseif_lst>*<else>
<else> -> Else{<body>} | ϵ
<elseif> -> Elseif[<expr>]{<body>}

<while_loop> -> While[<expr>]{<body>}

<func_call> -> ::<id>[<params>]

<params> -> <expr><params_t>* | ϵ
<params_t> -> ,<expr>

<type> -> Double | Integer | String | Boolean
<function_return> -> <type> | Void
<var_dec> -> <type><id>;
<asmt> -> <id>=<expr>;
<bool> -> True | False

<operand> -> <id> | <num> | <func_call> | -<num>
<expr> -> <operand> | <operand><relop><operand> |
          <operand><mathop><operand> | <string_literal> |
          <bool>
```

Please note the difference between $\star$ (Kleene star) and $*$ (multiplication).

`num` can be a integer or a double, as defined in the DFA. Bool can be either `True` or `False`. `str_literal` is a string as defined in the DFA.

Any line starting with `#` is a comment line and should be ignored by the interpreter; but still factors into the line numbers.

Like other programming languages that use block notation (Java and C) spacing and new lines are not important. The following are equivalent:

```
::print[x];
::print[ x ];
::print[x ] ;
::print[ x];
```

Multiple statements, separated by a semicolon and any number of spaces, are allowed:

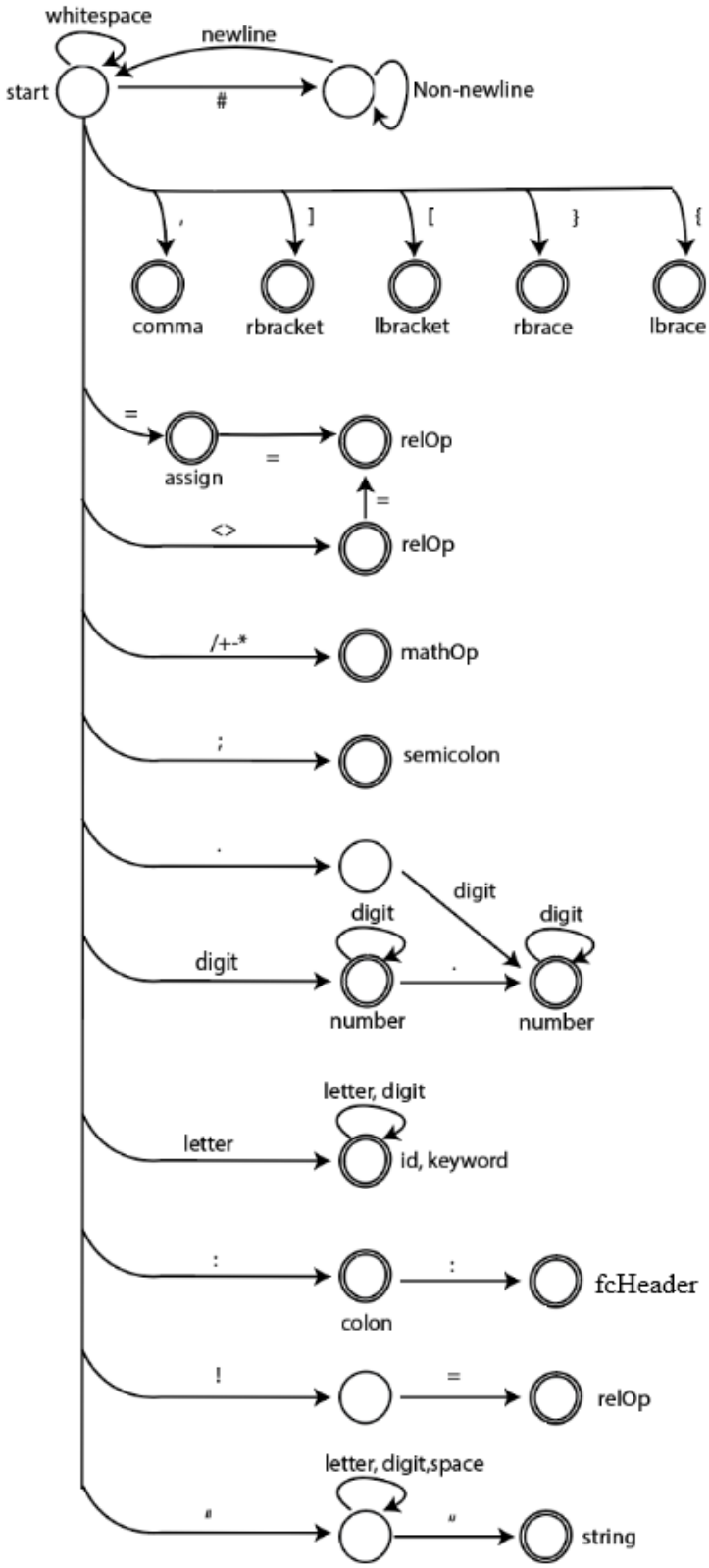
```
String hello;String world;hello = "Hello";world = " World";
String hello;String world; hello = "Hello";   world = " World";
```

Multi-line statements are also allowed in Jott:

```
String foo =
    ::concat[ "bar", "baz" ];
```

4 DFA for Scanning Jott

Here is the DFA to help you tokenize Jott for Phase 1..



5 Error Handling

Your translator must handle errors in the syntax and semantics. If a syntax/semantic error is found the Jott translator will not translate to the new language.

When handling an error (violations of the grammar or semantic errors), you will report a the proper type of error (Syntax/Semanitc), the reason for the error, and the line of code that caused the error; including the line number of the error (blank lines count as a line).

Errors will be reported to `System.err`.

Example runs with errors, and their messages, have been included with this write-up.

6 Running the Program

In phase 4, you will write a main class; `Jott.java`.

Executing the program will be done `java Jott input.jott`

`input.jott` is just a generic name for any program written in Jott; Jott program files typically end in `.jott`, but do not have to.

The output will be the Jott file exectutes with any output going to `Sysytem.out`.

The output must match exactly the provided samples.

There must be four distinct phases in the program; scanning (tokenizing), parsing (build parse tree), semantic analysis (building an AST), and execution. You cannot move onto the next phase without completing the prior phase without error.

NOTE: The parse tree and AST can be the same tree.

This will be placed in a Java class called `Jott`.

7 Example Jott Programs and Outputs

Example Jott programs and their C/Java/Python counterparts will be provided.

8 Other Constraints

You are not allowed to use any third part packages not built into Java on the CS machines. The code must compile on the CS machines. Failure to compile on the CS machine will result in a zero for that phase.

You must handle all errors and/or exceptions. Any errors and/or exceptions thrown and not handled will result in heavy penalties.

The program crashing at anytime will result in heavy penalties.

Failure of your program to run with the provided samples will result in heavy penalties for this phase. Test with the samples! But be aware the samples are not all encompassing and your program will be tested with other tests. So make some of your own.

9 Submission

Zip all source files for your program into a zip file called `phaseX.zip` (where X is the number of the phase) and submit to the proper myCourses' Assignment box. Include a `README` explaining how to build your project.

Failing to follow these instructions will result in a zero for that phase.

Emailed Submissions will not be accepted! If it does not make the dropbox it will not get accepted. Be aware it takes time to upload a large amount of code. Do not wait until the last second to start the upload. Submission time is when the submission is 100% complete; not when you start submitting it.

Only one group member needs to submit. Only the last submission is kept and graded.

Questions will not be answered within the late window.

10 Grading

This project will be graded as follows:

- (15%) Phase 1: Tokenizer
- (30%) Phase 2: Parser
- (30%) Phase 3: Semantic Analysis / AST
- (25%) Phase 4: Interpretation

Peer evals are required and will be graded. Please follow the insstructions on the peer evals. Failure to do so will result in a zero for the perr eval.

This project is 30% of your final grade. It is broken down:

- (27%) Project
- (3%) Peer Eval

11 Phase 1 Details

In phase 1 you will be making a tokenizer for the Jott files. The tokenizer will read a Jott file and output an `ArrayList<Token>` representing the tokens in the provided file.

The tokenization will be based on the DFA provided in Section 4.

You will create a class call `JottTokenizer`. This class will have:

- a function defined as:

  ```
  public static ArrayList<Token> tokenize(String filename)
  ```

  This function will take in the absolute/relative path of the file to parse. It will return an ArrayList of tokens. The `Token` class and `TokenType` enum have been provided.

If there is an error, you are to report it to `System.err` and return `NULL`.

Helper functions are allowed and encouraged.

Example Jott source file:

```
#this is an example
Def main[]:Void{
 ::print[5];
 #this is a comment
 ::print[ "foo bar" ];
}
```

This will result in the following list of tokens:

```
Def main [ ] : Void { :: print [ 5 ] ; :: print [ "foo bar" ] ; }
```

Note the space are there to separate the tokens and are not actually tokens themselves.

Example of an program with an error reported during this phase:

```
#this is an example
Def main[]:Integer{
    ::print[5;
    If[ x ! 5 ]{
        ::print[ 10 ];
    }
}
```

This would report the error:

```
Syntax Error
Invalid token "!". "!" expects following "="
filename.jott:4
```

The character after the `!` is a space. There is no valid transition from `!` using a space and it is not in an accepting state.

Notice it did not pick up on the missing `]` in this phase. It is only tokenizing and not checking structure. This will be handled in phase 2.

Notice it also did not pick up the wrong return type for main, or the missing return in a returnable function. This will be handled in phase 3;

Error messages in this phase should report the error type and where the error occurred in the file. Sample format:

```
Syntax Error:
<Message>
<filename>:<line_number>
```

There is a provided tester for this phase that your code must work with.

12 Phase 2 Details

In phase 2 you will make a parser based on the grammar listed in Section 3. The goal is to build a parse tree.

You will create a class called `JottParser`. This class will have:

- a function defined as:

  ```
  public static JottTree  parse(ArrayList<Token> tokens)
  ```

  This function will take in an ArrayList of tokens created by a JottTokenizer and return the root of the tree represented by those tokens. If there is an error creating the tree this function will report the error to `System.err` and return `null`.
  `JottTree` is a provided Interface and described below.

JottTree Interface (more details in the provided file):

```
public Interface JottTree{
    public String convertToJott();
    public boolean validateTree();
    public void execute();
}
```

In this Phase you will be implementing the `convertToJott()`. This function will output the tree as Jott code. This is used for testing purposes to validate that your parse tree was created properly.

This interface can be used to subclass all of your nodes in the parse tree. It is highly recommended that you make multiple nodes for each left hand side of a grammar rule; with a few exceptions. For instance a `FunctDefNode`, `AsmtNode`, etc.

Note that in this phase it may produce semantically invalid Jott code; this will be validated in Phase 3;

As spacing is not required in Jott it may be hard to read the file after it is created. You are welcome to add spacing and newline for readability.

Let's look at the example from Phase 1.

Example Jott source file:

```
#this is an example
Def main[]:Void{
 ::print[5];
 #this is a comment
 ::print[ "foo bar" ];
}
```

This will result in the following list of tokens:

```
Def main [ ] : Void { :: print [ 5 ] ; :: print [ "foo bar" ] ; }
```

Notice the return is missing. That is not validated in this phase.

HINT: Make tree nodes for most of the left-hand items in the grammar. Items such as braces, semicolons, etc also do not have to be in the final tree. They just really need to be verified for existence.

Also in this phase you will implement `convertToJott`.

When `convertToJott` is called the output should be:

```
Def main[]:Void{print[5];print["foo bar"];}
```

Notice the comments and spacing is gone, but it is basically the same code. It is fine if the spacing is added back, but not required.

Example invalid Jott source file:

```
#this is an example
Def main[]:Void{
 ::print[5];
 #this is a comment
 ::print[ "foo bar" ]
}
```

Notice the semicolon is missing.

This would report:

```
Syntax Error
Missing semicolon
filename.jott:5
```

Example invalid Jott source file:

```
#this is an example
Def 123foo[]:Integer{
 ::print[5];
 #this is a comment
 ::print[ "foo bar" ]
}
```

Notice the function name is invalid. There is another error; missing semicolon after the second print. And a third error, missing return. Your program will report the first error it sees and stop.

This would report:

```
Syntax Error
Expected id but got number for function name
filename.jott:2
```

Example invalid Jott source file:

```
#this is an example
Def foo[]:Integer{
 Integer x =;
 ::print[5];
 #this is a comment
```

```
 ::print[ "foo bar" ]
}
```

Notice the value after the equals is missing. Removing the = or adding a value after the equals would make this valid.

This would report:

```
Syntax Error
Assignment missing right side expression
filename.jott:3
```

Notice all of these are issues with typing the code and not the meaning.

Example valid Jott source file for parsing but not phase 3:

```
#this is an example
def foo[]:Integer{
 Integer x;
 Double y;
 x = 3;
 y = 3.2 + x;
 ::print[5];
 #this is a comment
 ::print[ "foo bar" ]
}
```

Valid for parsing because `Double <id> = <num> + <id>;` is valid syntax. Phase 3: semantic analysis will report the data type issues.

Error messages in this phase should report the error type and where the error occurred in the file. Sample format:

```
Syntax Error:
<Message>
<filename>:<line_number>
```

There is a provided tester for this phase that your code must work with.

13 Phase 3 Details

In this phase you will implement the `validateTree` function and the main class `Jott.java`.

The `validateTree` function will determine if the parse tree follows all the required semantic rules.

For example:

```
Integer i; i = 5; //valid
Integer i; i = 5.5; //invalid
Integer y; y = i; //valid
::foo[ y ]; // invalid if foo is expecting a non-integer
::foo[]; // invalid if foo expects params

foo[]:Integer{ //invalid missing return
    ::print[5];
}
```

Other examples (not all inclusive) of semantic errors:

- Function call to a yet defined function.
  - An error should be reported similar to:

    ```
    Semantic Error
    Call to unknown function <function_name>
    <filename>.jott:<linenumber>
    ```

- Missing/incorrectly defined main function.
- Invalid type being assigned into a variable.
- Invalid type being passed into a function param.
- Missing return statement for a non-void function. If statements will make this tricky.
- Use of undefined variable/function;
- Uninitialized variable being used.

HINT: If you structure the parse tree with various types of nodes in Phase 2, this will become easy. For instance if you have a `NumberNode` that knows it is a double value, a table of variables and their type, and implement an `AssignmentNode`. The `AssignmentNode` can ask its children, right child `NumberNode` and left child `IdNode`, their types and if they are different report the error.

In this phase the missing return shown in phase 2 should be reported. An error should be report similar to:

```
Semantic Error:
Missing return for non-Void function main
filename.jott:2
```

Error messages in this phase should report the error type and where the error occurred in the file. Sample format:

```
Semantic Error:
<Message>
<filename>:<line_number>
```

14 Phase 4 Details

In this phase you will implement the `execute` function; this is the code interpretation phase.

Example Jott source file:

```
#this is an example
Def main[]:Void{
 Integer x;
 x = 5;
 ::print[x];
 #this is a comment
 ::print[ "foo bar" ];
}
```

This will display when executed:

```
5
foo bar
```

15 Sample Code and Testers

Samples and testers will be provided for each phase. Your program/code must work with these samples and testers. If they fail to work with the samples and tester there will be heavy penalties.

Samples and testers will not be all encompassing. Create and test with your own test cases. Testers are really just there for you to have the proper structure to ensure testing works when grading.

16 `Hello World` program

When learning a new language typically "Hello World" is one of the first program taught. In Jott "Hello World" will look like (contained in a file called `hello_world.jott`):

```
Def main[]:Void{
    ::print["Hello World"];
}
```

Running this program will result in `Hello World` geing printed to the screen.