**Homework Summary**

For this part of the Ruby Interpreter Collection of Assignments we will be modifying our existing Parser so that instead of printing parser output, it will build an **AST** (abstract syntax tree) that our interpreter can use to “run” programs written in the Tiny Language.

**Details**

I have given you all of the files necessary to complete the assignment. You will need to modify **Parser.rb** so that it builds the AST while it is parsing a file. I have included an **AST.rb** file that defines an AST node object that can be used to build this tree. It also has a method that can print the tree in list format (convenient since our interpreter will be written in a list-based language!).

I have done some of the work for you in Parser, so that you can use that as an example of how you should proceed. At the end of this document are screenshots showing what correct output should look like given some input files that I will also provide to you for debugging.

**Note**: We are building an **AST** and **NOT** a parse tree. What is the difference anyway!? The difference is that a

**Parse Tree**

parse tree would look much like the examples we went through in class when we compared parse trees to derivations. They would INCLUDE every rule in our class as root nodes or sub nodes in the tree. However, an

**AST (Abstract Syntax Tree)**

AST **ONLY** includes the actual lexemes that are needed to be interpreted. If you take a look at the compiler series article that I mentioned in a Canvas announcement, they illustrate that in one of the series articles.

For our assignment, we will always start with a program node as the root of the whole tree. That is the only “not concrete” thing that should be part of our tree.

**Rubric**

**(-20 pts)** Program won’t run

**(-5 pts)** Each extraneous symbol that is added to the tree

**(25 pts)** AST is properly formed

**Extra Credit (20 pts)**

Because of the way our grammar is defined (and because we want to produce a tree in list format) our tree **WILL** be nested correctly **BUT** our operands will be in reverse order. Since it is something that is consistent, it is an easy problem to handle in our interpreter (we already know it will be have this way, so just read in the operands in reverse order).

However, it is possible to produce the tree in list form with the operands in correct order. In order to do this, you will need to modify the **AST.rb** class. You will need to:

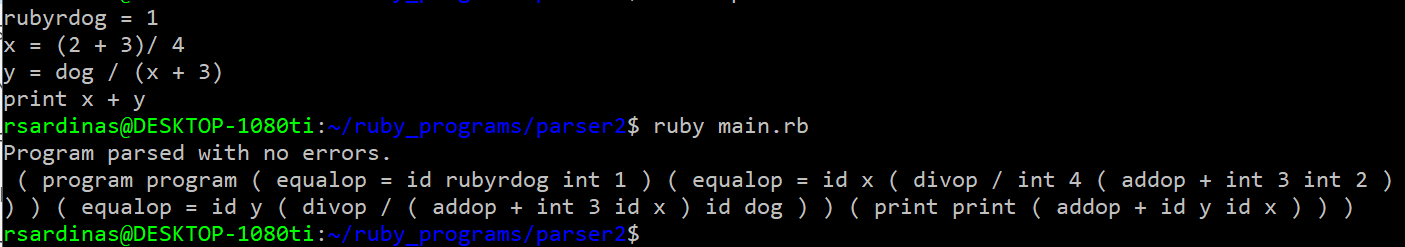
1. add a method in **AST.rb** that allows you to swap the first child of a node
2. instead of calling **addChild** everywhere in **Parser.rb**, there are at least 2 instances where you will need to call your new custom function instead (maybe called **addAsFirstChild**).

**Deliverables (That means this is what you turn in)**

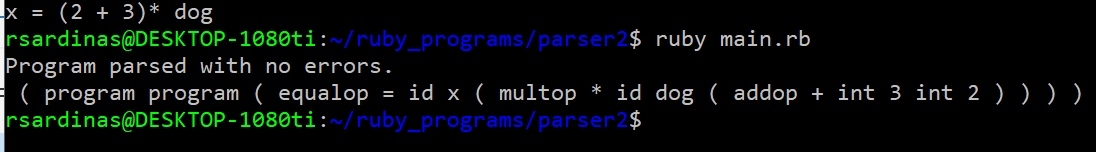
1. Parser.rb
2. AST.rb (ONLY if you attempted the extra credit)

**Screenshots**

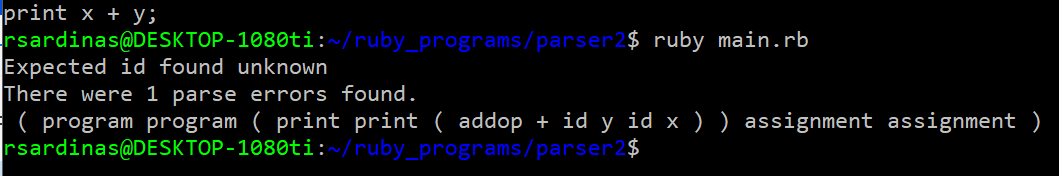
Using input1.txt



Using input3.txt



Using input5.txt (not written according to grammar specifications)



Using input4.txt (not written according to grammar specifications)

