Benjamin Grass, Cameron Russell, Andrey Yao
CS 2112

Summary:

Temporarily ignoring the actual implementation and the class hierarchy that would make this possible (which we describe later) we will use a binary tree to represent our AST. The root node for each rule will be the '-->' token whose left and right subtrees describe the condition of the rule and the command that executes of said condition is true respectively.

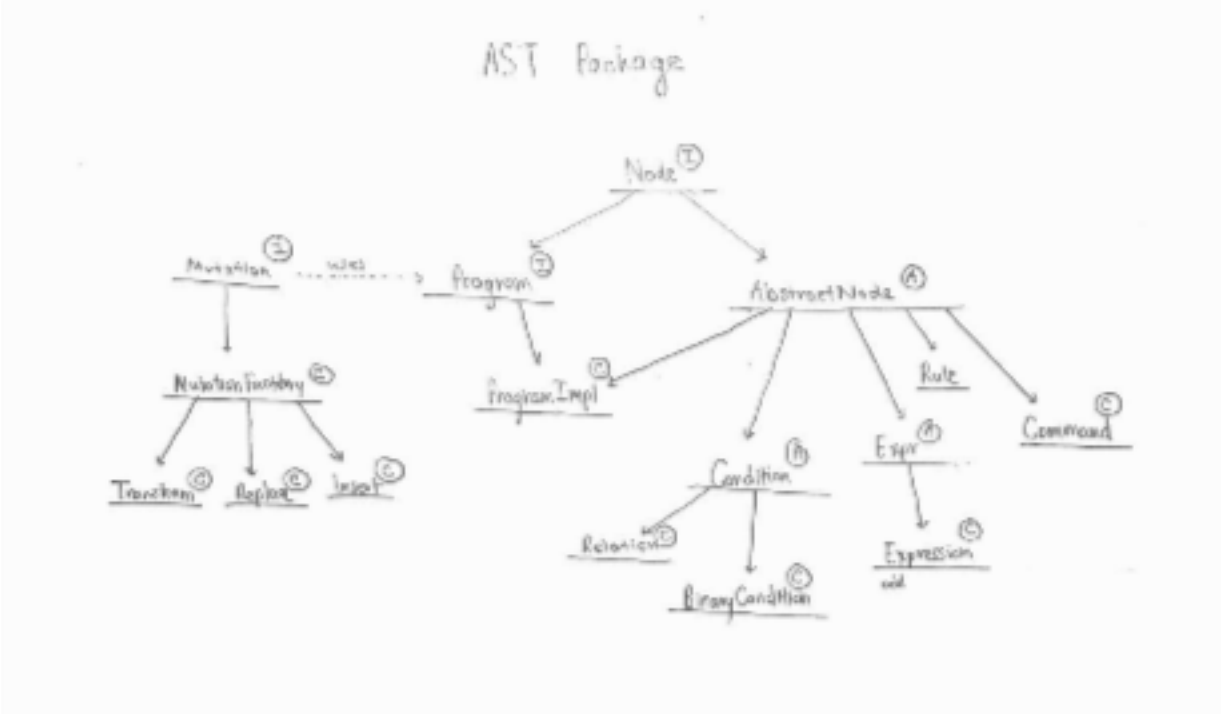
Specification:

The left side of the AST lends itself well to a binary tree. The right side of each rule, however, does not because there is no token separating instances of updates and the update-or-action. Additionally, the spec indicates that Program should contain the entire ruleset of the critter (because it extends Node, implying that it should have a tree structure, and since the language in the larger project spec indicates that Program has 0 or more rules). The problem being, again, that this does NOT lend itself to a binary tree structure because the program could, and likely will, have more than two rules.

Our solution to this was to make both the ProgramImpl class and the Command class (which we will be creating to represent the right) will extend AbstractNode as the others do, but instead of making use of the left and right children, they will contain an array of nodes. In the case of ProgramImpl this array will hold the root nodes of each rule. In the case of Command, the node will hold an array of n updates and also store the single update-or-action in an instance variable -- that way we could test the final update-or-action to see if it actually is an action (to determine if we need to loop over the critter's rules again) AND it solves the problem of not having a token to parse potentially many updates into a binary structure. The array in Command will point to the assignment operator of each update which will point to the relevant information in turn. **** WE FEEL LIKE THIS IS CLUNKY DESIGN, BUT WE CAN'T THINK OF AN ALTERNATIVE. CAN YOU PLEASE POINT US IN A BETTER DIRECTION? ****

getIndex will return a node at the supplied index in the preorder traversal of the subtree.

Class Hierarchy:



These are key topics to cover in your design overview document:

What are the key data structures you will use for this assignment? In particular, how will you represent an AST (abstract syntax tree – see §7.1) and what data structures will you use?

1. Binary Tree to represent our AST
    a. The specifics of parsing into an AST allude us at the moment but we are confident that each node needs only 2 children maximum to represent any given rule as an AST.
    b. A command node will be a leaf in the AST, containing a list of updates and at most one action. Each update will be a root node of another binary AST.

What are the key algorithms you will need? Which ones will be challenging to implement, and why?

1. In-Order-Traversal
    a. Once parsed into an AST the inorder traversal of each rule will be used in pretty printing.
2. The parser
    a. The parser turns the array of tokens into an AST. This algorithm will be the most challenging to implement, as it lies at the core of A4 and is responsible for the construction of a correct/accurate AST.
    b. As of right now, our plan is to search through the tokens and find the token with the highest level of precedence. This token will become a node in our AST and the tokens to

the right and to the left will be recursively called to our parser, creating left and right nodes until all tokens have become a node.

What will be your implementation strategy, and group how will you go about dividing responsibilities between the members?

Primary Tasks for A4:

•Alter Tokenizer to support end-of-line comments (**BEN, done already**)
• Implement the main method and command-line interface. (**done already?**)
• Design and implement a class hierarchy for representing abstract syntax trees. (**All**)
• Implement the Parser interface to generate abstract syntax trees. (**BEN**)
• Implement pretty-printing functionality as methods on AST nodes. (**ANDREY**)
• Implement a class or classes to perform fault injection (**CAMERON**)
•Written Questions (**done already**)

Of course, when someone gets stuck, we will all come together to help. Plus, we will do code reviews so we all understand the entire codebase, which will help in future assignments.

What will be your testing strategy to cover the wide range of possible inputs and the different kinds of functionality you are implementing?

We will be using fault injection to test a wide variety of possible inputs and the different kinds of functionality. To test invalid inputs, we will generate a string of tokens in random order and then run them through the parse tree.

Questions:

1. Why is there condition and binary condition?
2. Why is there a ParserFactory instead of just one single Parser object with different methods?
3. Is command line done?
4. Is it a bad design to have a mixture of binary tree and N-ary trees in the AST?