Metadata

Benjamin Grass (bdg83), Cameron Russell (cfr59), Andrew Yao (awy32)
CS2112 A4: Parsing and Fault Injection

Summary (50 – 300 words)

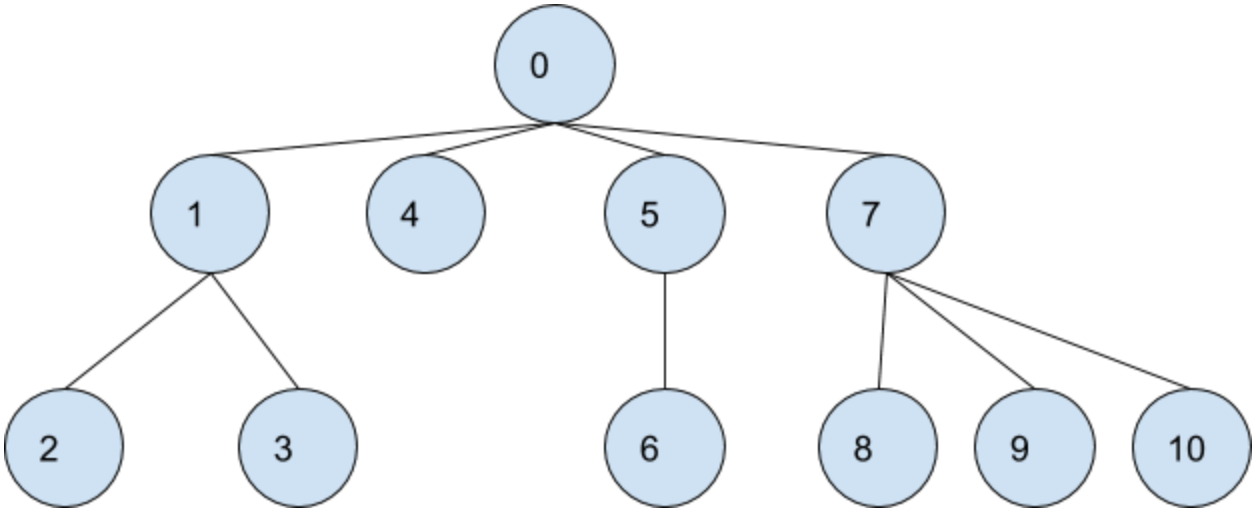
In designing our project we had some fundamental misunderstandings regarding how parsing worked, a major issue we faced over the course of this project was clarifying what exactly was going on during parsing and adjusting our plans around this new knowledge.

We made Expr into a class of its own because there is no difference between the actual node representation of a term and a factor in the grammar, each has two children of type Expr and holds an operator that describes the operation to be performed. Additionally, we split the factors in the grammar into two classes (Factor and Unary) each of which extends Expr. This separation allowed us to more easily group common functionality because Factors (like numbers) are purely terminal symbols, while Unaries (like mem[expr] and sensor[expr]) have 1 Expr child. Factor and Unary both extend Expr so they can be returned from a call like parseFactor called by parseTerm called by parseExpr and not violate the expected return type, Expr.

We are unaware of any *major* problems but we are missing some functionalities that would make our critter program more versatile; we only support three types of mutations: swap, replace, and transform, which limits the possible program types of new critters, but we intend on implementing more mutation types after submission.

Specification (10 – 500 words)

The nodeAt method did not specify a traversal method for the ordering of the nodes, so we decided to implement a preOrder order that looks like this:



We implemented this by recursively traversing the tree and adding its nodes to an ArrayList in preOrder order. Then, we return the node at the index in the ArrayList specified by the value passed to nodeAt. We chose this representation because it ensures nlogn runtime, which is decently quick, and was simple to implement.

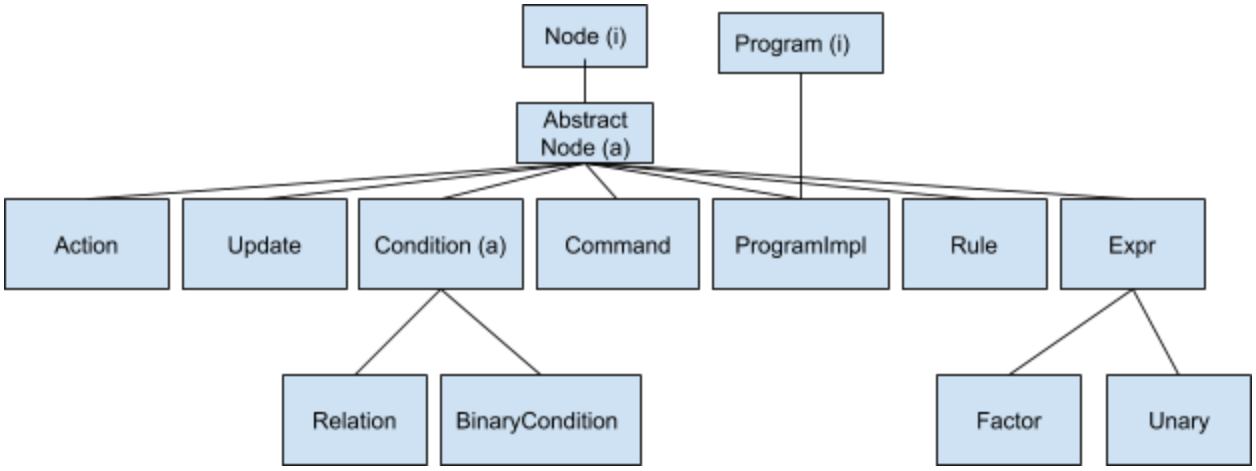
Additionally we decided that we would construct our AST using an n-ary tree instead of a binary tree with exception for program and command nodes in order to more uniformly represent the function of a Node. Each node's children is stored in an internal array called children and the methods to add, view and return a Nodes children are the same across all implementations of the Node interface. We chose this because we felt like a uniform representation of Nodes was more important than the convenience of simply having left and right Node pointers as we would have had we gone with a binary tree.

In our implementation our Command class represents a single node that stores an array of Updates to execute if the condition evaluates to true and an additional Action field that either contains an action null. This way, later on when we code the evaluation of these AST's, we can simply execute all the updates, and either execute an Action and end the critters turn or make another pass through the rules depending on whether or not that Action field is null.

In terms of mutations, specifically the replace mutation, the specs were ambiguous as to whether a node can select and replace itself. In our design, a node will never be replaced by itself, as this defeats the purpose of a mutation. However, if there is no other node in the AST of the same Category, then an unmutated program will be returned.

Design and Implementation (150–1000 words)

We represented the AST using a collection of classes that implement the Node interface and extend the AbstractNode abstract class to break down the various types of Nodes that may be inserted into the tree and facilitate parsing. At the most basic level, each of these classes function as Nodes so we pulled out all the shared functionality and put it in AbstractNode. When considering each of these nodes, we needed to construct a class hierarchy that lent itself to representing the grammar as accurately as possible, this means allowing classes to extend one another when they might be returned in parsing its parent. For example, in parsing a Condition you might run into a BinaryCondition such as 'OR' or 'AND' or a Relation like '>', '<', '=' etc. Our class hierarchy was as follows:



By constructing our hierarchy in this way we made parsing easier for ourselves because we could call parseExpression() and ultimately return a Factor or a Unary when the nested method calls that deal with parsing hit a base case. One of the invariants of each class is that is contains children of the correct types which we enforce using the constructors for each Node which take parameters of the proper types. This is also checked in the actual parsing method and once added, children cannot be set as we did not implement a setChildren method. We chose to split up the potential terminal symbols that extend Expr because they serve different purposes, Factors represent nodes with no children like numbers and MEMSUGAR while Unaries represent Nodes with 1 child like mem[expr], sensor[expr] and a negative expression.

The parsing algorithm is made up of a series of methods each designed to parse a specific element of the grammar. First, a program is created and parseProgram() is called. This, in turn, calls parseRule(), which calls parseCondition() which calls parseConjunction() which calls parseRelation() which calls parseExpression() which calls parseTerm() which calls parseFactor(). At this point, assuming a terminal symbol is parsed, we start climbing back up the call stack each time knowing what *should* come next or in other words, what kind of token we expect. We can use consume to skip over those expected tokens, creating a node along the way, and parse the expected kind of value on the right just as we did during the first pass through. An interesting continuation of this algorithm is in parsing a list of updates of size n. Since there are no tokens in between updates to separate them, we initially thought that parsing them would be particularly difficult but in reality no changes needed to be made to our algorithm. When parsing the right side of an update, the expression, the parseExpression method will terminate on the last terminal symbol without continuing onto the next token, even if there isn't a separation between them. Even though that may seem obvious now, it was not apparent to our group at first and we struggled with the idea of parsing a list of updates because there was no symbol denoting when one started and another began.

We used an n-ary tree to represent the AST because, as previously discussed, we felt that the slight added convenience of having left and right child node pointers was outweighed by the uniformity that an n-ary implementation allows. If we had chosen a binary tree we would have

needed to make sloppy special cases for program and command which we felt detracted from the overall design.

We implemented this project with a largely top down approach as we carefully designed the class hierarchy first in order to allow us to write the simplest code possible as we parsed and manipulated the AST. I think this was ultimately a good idea, as it vastly improved the quality of our code throughout the project and made each of our tasks far easier. Each group member implemented the parts of the code assigned to them in the Design Doc as well helping teammates.

Testing (150 – 1000 words)

Besides just using all the test cases the course staff gave us we used the following additional tests for each of the packages. We implemented a method in MetaTest.java that randomly generates an arbitrarily large valid program. This allows us to test large amounts of varying programs to ensure all edge case scenarios are covered. Additionally, we tested the mutation classes by repeatedly mutating a program and then checking to make sure the program is valid at the end of all the mutations. Moreover, we tested the pretty printing by parsing in a program and then parsing its string representation, making sure the resulting outputs are equal.

Work plan

Because A4 had three primary components, parsing, pretty printing, and fault injection, it lent itself well to being split among three group members. However, because checking the accuracy of fault injection is dependent on pretty printing, which is dependent on parsing, this work strategy was a mistake on our end. We quickly figured out this work plan did not allow us to test incrementally; rather testing was only possible at the very end, so in future assignments, we will certainly need to improve our work strategy by starting earlier and not subdividing the code into three distinct parts.

Known problems (1–500 words)

We currently have no known bugs. But if we do, they are not bugs, they are special features.

Comments (1–200 words)

Most of the time we spent on this assignment was dedicated to designing for a couple of reasons. First of all, this assignment posed a particularly difficult design problem as we struggled to design a system whose class hierarchy and parsing methods adhered to the grammar we were given. This required a lot of thought and careful planning on our end which took a lot of time in the beginning of this process. Second of all, we struggled with git throughout the entire project

up until the last two days which at times prevented us from writing code switching our focus from implementation to shoring up our design while we tried to fix git.

In terms of what I wish we would have been told, I think a little bit of a deeper introduction to AST's before the assignment would have been helpful. The lecture on the topic was very good, but I was still left a little uncertain as to some of the specifics.