Checkpoint 1 Report

Scanner

With the given C1-Specification.tgz file and knowledge from the Warmup Assignment, a JFlex tool was used to create a **Scanner** .flex file for the C- language. The file generates a sequence of relevant tokens including keywords (bool, else, if, int, return, void, while), literals (false, true), symbols (+, -, *, /, <, <=, >, >=, ==, !=, ~, ||, &&, =, ;, ,, (,), [,], {, }), identifiers ([_a-zA-Z][_a-zA-Z0-9]*), and numbers (0-9) in which some of them were ignored (such as white spaces and comments) or reported for error recovery in the parser (such as invalid characters).

Parser

For the **Parser**, the file that handles the definition of rules and error recovery is cm.cup. Within this file, we essentially remade the given C- specs syntactic rules into the CUP's CFG form. Initially, our main focus was just to define the set of rules given to us. After defining all of the rules, we focused on simplifying them (similar to the given .cup file examples where we were able to merge small, simple rules into major rules). For example, the boolean and arithmetic rules were simplified by merging them directly into exp instead of having several different rules. We repeated a similar process for the other smaller rules. After simplifying our rules, we go back to the terminal and non-terminal declarations to group them up and organize them into the proper classes (defined from lecture 6, slide 7). After defining all of the rules, we added the Java code that'll run when certain rules are matched so we can handle the matched tokens and

use the information to help with the abstract syntax tree creation. The found information would be stored within the RESULT variable as its respective object. Afterwards, we create a multitude of object definition Java files in the abysn folder, this is where we'll define the objects missing from the lecture 6 slide 7 and define the objects accordingly. Next, we direct our focus towards the functions needed to print out the tree, for this we simply declare the visitor function prototypes in AbsynVisitor.java (in the absyn folder) and then define the visitor functions within ShowTreeVisitor.java (these functions handle the actual printing out of the absyn).

Error Recovery

Afterwards, we worked on **Error Recovery**. After checking that the parsing tree will print the abstract syntax tree with the given .cm files (that didn't contain any errors) we move on to error handling. For this portion, we go back to the rules and attempt to predict any possible errors and define Java code that will run when such scenarios come to pass. For example, a function declaration missing its left parenthesis int a params ({....}).

If an error is encountered, we generally just print the following message "Error message with Row number and Column Number" and then handle the error accordingly (usually substituting a null/NilExp value into a place that would have been missed had the error gone unnoticed).

By working through this process, we began to realize how closely connected error recovery and parsing is. During error recovery, we often encountered several errors that stemmed from small mistakes that were made earlier in the developmental process, specifically when setting up the files in the absyn folder. These mistakes

included incorrect variable types in the constructors and failing to define the necessary visit functions for the nodes in the syntax tree. Furthermore, there were also times where we had to fix the type declarations of our nonterminals in the grammar to get error recovery working properly.

Error recovery became more time-consuming than we initially anticipated because we had to address several issues related to the parser that were not handled before (mentioned in the paragraph above). These fixes added significant time to the developmental process.

Limitations/Assumptions

As for assumptions, limitations, as well as possible improvements. We assumed that error recovery would be straightforward, but it wasn't. The errors were connected, which meant that adding an error detector for one production in the grammar required you to add another error detector for another production. For improvements, we would like to improve error recovery by adding more error detectors for more production rules. A limitation would be that the current error recovery doesn't cover all of the possible errors. Another limitation would be that it could only sense errors before or after expressions, but not the expressions directly. This limitation limited the type of tests we could write for the assignment. Some of the errors would stop the creation of our syntax tree. We need to fix our error recovery for future checkpoints.

Contributions

All members of our team contributed to each part of the assignment. At first, we decided to split the assignment into three parts where Jessica would work on the

scanner’s .flex file and documentation, Justin would work on the parser’s grammar and printing within the .cup as well as the rules within the abysn files, and Arvind would work on the error recovery within the .cup file and server testing. Once we got more into the assignment, we realized the amount of work was disproportionate for the parser and error recovery. Thus, we decided to help each other out on how to fix those errors in those parts of the code. Jessica helped code the NilExp() rule in abysn, test 5, and parts of the error recovery while Arvind helped with .cup’s precedence, rule-making, tests 1-3. Justin created the CFG rules, created the terminal/non-terminal declarations, created the java code attached to CFG rules, created and defined the additional .java files in the absyn folder, helped to define the visitor functions, test 4, and helped with the error recovery.