Checkpoint 2 Report

Lance Paje, Pasan Undugodage

**Related Techniques and Design Process**

For SemanticAnalyzer.java we borrowed most of the architecture from ShowTreeVisitor.java in the sense that we used SemanticAnalyzer.java as a class that defines visit methods. SemanticAnalyzer.java is also where we store and traverse the hashmap. This hashmap is used to create and print the symbol table as well as to assist in type checking dynamically. The symbol table adds symbols whenever a class that inherits from an abstract class Dec is used.

We manage the symbol table using helper functions that help with node insertions, node deletions and looking up nodes with specific characteristics. The function for node insertion takes in the name of the declaration, the declaration itself and its level before creating the nodetype and inserting it into the symbol table. The function for node deletion takes in a level and then scours the hashmap for nodetypes containing that level and deleting them. We have two lookup functions inside SemanticAnalyzer.java: one that takes in a level and returns if there are nodetypes in that level and another for looking if there's a specific nodetype that matches a given name and type combination.

We added a variable of type Dec called dtype to every class that inherits Exp. This is used for checking if the declaration being used was already present in the symbol table. We paid special attention to AssignExp, OpExp and CallExp since these three can add declarations onto existing declarations and in turn create complex statements. We also paid attention to ReturnExp and CallExp since they had special

cases that needed to be checked. Finally VarExp was the backbone for ensuring it created the correct dtypes as our structure directly copied their dtypes from VarExp.

**Lessons Gained**

At first we overcomplicated the matter of how we should go about printing the symbol table as we were trying to keep track of the levels inside the scope of the declarations but then we decided to add the entering/exiting scope statements in DeclarationList, IfExp, WhileExp and FunctionDec. This greatly simplified the process of printing the symbol tree but had extra entering/exiting scope statements that didn't contain any symbols. Later on we moved to printing the statements after buffering them so that the declarations would appear at the end of the block.

During the development for the error checking portion of the assignment, we ended up creating a lot of boilerplate code through helper functions as we had to keep looking for similar elements in similar cases.This was mostly due to the edge cases in ReturnExp and CallExp needing similar access. We also came across that the error checking for IntExp won't be used in practice as the parser would pick up if the number being passed in wasn't a number. VarExp was key to getting correct as it tended to be a leaf node in the AST to begin with.

Testing our error checking was difficult again as all parts needed to work first before any real testing could begin. This entailed dtype being set in most classes that inherit from Exp. In some cases another error checker needed to work first before another could operate. For instance the checks on VarExp needed to work to ensure that the dtypes made were correct and in turn the checks in AssignExp and OpExp would work properly.

**Assumptions and limitations**

Since the C minus specification file requires only two types to be defined, INT and VOID, we made assumptions that all files passed through our parser would only contain these variables. We also assumed that only INT functions should have return statements as a result. Conversely OpExp assumes that it should only have variables of type int on both its left and right side. AssignExp on the other hand only checks if both the left hand side and right hand side are of the same type.

Our Parser still works under the assumption that only a single file is being processed and that external libraries aren't being used in the files for parsing. Outside of the use of the functions input() and output(), our parser will look for existing function declarations in the symbol table. Furthermore on the function declarations, we assume that all functions were defined first before any call to them is made.

In terms of the declarations, we assumed that none of them would share the same name, especially between SimpleDec and ArrayDec as this could cause issues with assignment. For example, if there was an array and a variable both name "a" and "a" was set to a number, an error would pop up in trying to set the array to an integer.

As for limitations, most of the work done was on the file SematicAnalyzer.java. This led to merge conflicts happening often and required one of us to either drop our changes to the file or figure out how to incorporate the other person's code. This in turn slowed down our progress and more hours were wasted in syncing up work.

**Possible Improvements**

Some possible improvements we could make to our parser are: more static type checking, reduction in boilerplate by using more helper functions, and figuring out a

better way to distribute work. To begin with, the tiny.cup file was able to replace the error checking of the IntExp. The tiny.cup file could have additional rules appended to non terminals that create instances of VarExp and deal with cases of mistyping there.

For the reduction in boilerplate, many of the existing helper code needs to traverse the symbol table, as a specific kind lookup is needed to grab a specific node. A new generalized helper function that can give a nodetype at a given level or name would help in reducing boilerplate and increase readability of the source code. On the other hand another way to solve this problem is to generalize what a lookup function outputs in order to only have one function traverse the symbol table and many smaller functions that extract the specific data necessary from the NodeType.

As stated previously we had trouble in distributing the work properly and ended up getting in each other's way. A solution to this problem could be that we assign work based on the class as opposed to the objective we set out to do. For example, instead of dividing the work based on who's printing the symbol table and who's implementing error checks we instead give one person the classes that inherit from Dec and another person the classes that inherit from Exp.

**Distribution of Work**

For this checkpoint the distribution of work was split between the objective goals of this assignment: printing out the symbol table and performing error checks. Lance did the work on most of the variable declarations and printing out the symbol table while Pasan did work in regards to the error checking. However we did not anticipate for the error checking portion of this code to be work intensive, with all of its edge cases, and offloaded some of the work on Lance.

Aside from the main objectives of this assignment Lance made small changes to the CM file in order to allow for the -s option to print out the symbol table, updated the makefile to also compile the SemanticAnalyzer.java file, and made new test files for this Checkpoint.

Pasan created many of the functions in SemanticAnalyzer.java, including ones responsible for creating, populating and deleting the symbol table, cleaning up the print statements for error checking and printing out the symbol table and finally outputting errors to stderr.

Again work was split for the error checks due to the edge cases of each exp class needing to be addressed.