CIS4650 W21 Checkpoint 2

Checkpoint 2 Report

Wafa Qazi(0932477) & Aqsa Pehlvi(1018401)

CIS*4650

Author Note

Page 1: Introduction, Techniques

Page 2: Design Process

Page 3: Summarizing Lessons, Assumptions and Limitations

Page 4: Possible Improvements, Contributions, Conclusion

CIS4650 W21 Checkpoint 2

Introduction

This report highlights the related techniques and design/implementation of our checkpoint 2 for CIS*4650. For this checkpoint we implemented a symbol table using our abstract syntax tree which validates input for our program by traversing the tree in post-order. The tree is able to report type definition errors such as mismatched types in expressions, undeclared/redefined identifiers, etc.

Related Techniques

For this checkpoint, we had two main tasks to complete. The first portion of the assignment we began by implementing our symbol table in SemanticAnalyzer.java. The first step of implementing our symbol table was to create different scopes/levels. We did this by incrementing levels everytime a new function declaration or while/if statement were visited. Once we finished creating different scopes we tested our code using different files to make sure that the levels were correct. After creating different scopes, we created a Hashmap and a new "NodeType" class. In the HashMap we store different definitions of declarations as we traverse the syntax tree. After creating our table, we began to populate the table with new entries. We did this by creating a "NodeType" every time we came across a simple declaration, array declaration or function declaration. These entries would store all the related information needed for us to maintain the symbol table and be able to check for type definitions.

CIS4650 W21 Checkpoint 2

Design Process

We began our design process by preparing ourselves with the relevant information needed to complete this checkpoint. To do this we reviewed class lectures and discussion posts on the course website. Once we had an idea of what we needed to do to complete the assignment, we began implementing the hashmap in our code. We chose to first set up an empty hashmap as well as empty nodes to make sure we were able to add items to the hashmap successfully. We chose to have one single hashmap that would be used instead of multiple for each scope that were linked together. We decided to use this design because it would be efficient in maintaining variables of the same name defined in different scopes. Once this portion was complete, we moved onto implementing levels and scopes. Once we had these in place we tested using print statements to make sure they were working and we could move on. Next, we began adding elements with their populated arraylists to the Hashmap. Finally, we added type checking to the compiler by using type inference and using the type checking rules provided in the lecture notes and specification. We did our type checking implementation in the visitor pattern according to post order traversal as suggested.

Summarizing Lessons

There are many worthwhile lessons we learned by completing this assignment. Some of the primary skills we gained from doing this project were how to implement and use a Hashmap that stores arraylists. This was a new concept for both of us so therefore we were able to gain some valuable experience using this structure while implementing our symbol table. Another

CIS4650 W21 Checkpoint 2

skill we acquired in this assignment was how to do type checking. Prior to this course neither of us had written compilers, therefore our compilers we were running code on would automatically test our code for valid return/function types. This was the first time we manually had to do this on our own and coming up with the logic to do it further improved our understanding of compilers as well as our overall programming skills.

Assumptions and Limitations

Assumptions:

- We assumed that functions of void type should not be able to return any expression or value. Likewise a function of type int cannot return null;
- We also assumed function parameters cannot be of type void.

Limitations:

- For function calls, the compiler cannot check if the values of the function call match the parameters of the function definition. But the compiler will check if the types are valid.

Possible Improvements

- Improvements could be made for how we are handling our function parameters upon definition so when functions are later called we can check to make sure the number of arguments and argument types match.

CIS4650 W21 Checkpoint 2

- How we handle variables in the table could also be organised a little better so recalling declarations can be done smoothly.

Contributions

Similar to the last assignment, we decided that the best way to work was simultaneously using liveshare. By scheduling designated times that we would both be able to work together, we were able to complete large portions of the assignment much more efficiently. Although we both contributed to the assignment, Wafa took lead on this assignment since she was more free and had a better understanding of it. Most of the assignment was completed together, however Wafa did work on the undefined/redefined portion of the symbol table individually.

Conclusion

In conclusion, we have learned a great deal about hashmaps, implementing symbol tables, and type checking from this phase of the compiler assignment. Completing this assignment has given us not only a better understanding of these core course concepts, but a better understanding of the Java language. We are looking forward to Checkpoint three and feel confident that we will be beginning with a strong starting position.