Final Report

Wafa Qazi(0932477) & Aqsa Pehlvi(1018401)

CIS*4650

Introduction

This report highlights the related techniques and design/implementation of our checkpoint 3 for CIS*4650. For this checkpoint we utilized our previous implementations to generate assembly code that we can test on the TM Simulator provided by Professor Fei Song.

Design Process

Checkpoint 1

For checkpoint 1, our design process was based solely on the professors recommended design for implementation. We began by simplifying the grammar rules given to us, and then implemented each rule based on its dependencies of other rules. After that we began building on the given tree structure classes for our C- language, which was the first part of the process of designing our syntax tree output. During the design process we ran into some issues, specifically with implementing cases with epsilon. We ended up finding a solution to these issues with trial and error, as well as referring to the sample parser given to us. Over time we were able to perfect our syntax tree design by adding extensive error recovery and improving how our output looked.

Checkpoint 2

For checkpoint 2, our design process began with how we wanted to represent our symbol table (hashmap). We had to make sure our design would be able to handle variable declarations of different types and contained within different scopes. 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. Over time we added to our symbol table implementation to be able to receive addresses of variables stored in order to complete code generation.

Checkpoint 3

For the final checkpoint, we began our design process by preparing ourselves with the relevant information needed to complete this checkpoint. We began our preparation by reviewing the lecture slides and recorded lecture footage. After gaining a good grasp on assembly language and memory allocation, we took time to get used to the TMSimulator provided by the professor and ran the provided generated assembly code files to help us understand the flow of the various instructions. Afterwards we were able to implement the emitting routines needed to generate the code and also then generate the code for the standard prelude, input/output routines and the finale. Then we started with a very basic dummy program that just generated the code for an empty main function and we started maintaining the global offset for our program. Then as we got a good grasp of the various offsets, we began generating assembly code for expressions and assignments. Then incrementally we implemented the code generation for control structures if-then-else and while. After we had completed these we were able to generate fac.cm then we moved on to calling structures and inner-blocks and were able to generate code for gcd.cm.

Finally we added some error checking so that if the C- program contains any semantic errors no code would be generated.

Summarizing Lessons

Checkpoint 1

From finishing checkpoint one, we learned two main lessons. The main lessons learned were how to use JFlex(more in-depth then the warm up assignment) as well as how to use the CUP tool. Some smaller lessons we also learned from this assignment include adding grammar rules in BCNF, error recovery and techniques to resolve ambiguities such as precedence directives. Context-free grammars, parsing techniques like left-most derivation, and creating syntax trees are also lessons we gained by completing this phase of the compiler.

Checkpoint 2

During checkpoint two, we learned how to implement and use a Hashmap that stores arraylists. Since neither of us had experience with linked hash tables, this was a great opportunity for us to gain experience with them. Another valuable lesson gained from this was how to do type-checking. This important lesson was used in our checkpoint three as well when we updated grammars to handle errors.

Checkpoint 3

From checkpoint three, we learned all about turing machines and assembly language. Since this phase of the assignment required us to turn a C program into assembly code, we needed to first understand how to write assembly code. Some smaller lessons that we learned were how to write a prelude/encore, memory allocating in machine language, and offsetting.

Assumptions and Limitations

Assumptions:

- We assumed that function parameters cannot be of type void.
- We also assumed that functions of type int must always return a value and void doesn't return anything.
- We also assumed that the programs will be tested within C- specification
- We assumed the the -s and -c flags will be used together to avoid compiling code with semantic errors

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.
- The Compiler does not handle any array values or array parameter functionality thus far. The code cannot be generated properly for these expressions.

Possible Improvements

For overall improvements of our compiler we could add the functionality to be able to check function call parameters with their related parameter types in the function definition. Our generation implementation could also improve by adding the ability for our code to generate assembly instructions for arrays and array parameters and also checking for out-of-bound exceptions. Also overtime we could possibly add the functionality to be able to generate code optimization, this would generate more efficient and concise assembly code.

Modifications to Old Code & Issues We Encountered

When beginning the design implementation of our final checkpoint we realized we had to refactor and design much of our old code. Over time we had to add to our visitor pattern for maintaining offsets and we had to implement a more effective recursive pattern to be able to check for addresses and such. When using our symbol table in the code generation file our algorithm for finding and adding variable declarations needed to be factored into the new implementation because we were using levels for scoping organizations. But our final implementation level wasn't relevant so we took it out initially but had to add it back in. Many of the issues we faced during the final part was related to how we were handling the recursive pattern. Because this was so closely related to how offsets were being handled we had to do various error checks if our offsets were off.

Contributions

As mentioned in our checkpoint one and two reports, we initially began this assignment attempting to work on the code individually when it best suited our own individual schedules. However, soon into checkpoint one we quickly learned that coding simultaneously using Liveshare actually resulted in us being more productive and getting work done more efficiently. From that point on we did the majority of the assignment together during pre-allotted time slots, often setting aside 5-6 hours each day to code together until we reached our goals. However,

during the last checkpoint, Aqsa was having a harder time understanding certain concepts and had a more hectic schedule, so Wafa completed a good chunk of checkpoint three individually.

Conclusion

In conclusion, we gained invaluable knowledge through completing this assignment of constructing a compiler. From this checkpoint, we learned about assembly language and turing machines, offsetting, and memory allocation. However, there was a great deal more we learned through completing the other phases of this project. From checkpoint one we learned a great deal about abstract syntax trees and parsing. From checkpoint two we gained valuable information on symbol tables and type checking. The entirety of this project was done using CUP and java. Since both of us had very little/no experience with these languages, we can now safely say these two languages are ones we now have an intermediate grasp on. Completing this assignment gave us so many useful skills that we will definitely use in future compiler design projects as well as many other types of coding projects.