Compilers - Checkpoint Three

Braydon Johnson, Neivin Mathew

April 5, 2017

1 Summary

Over the course of three checkpoints, a compiler for the C- language was constructed.

The compiler can read a C- program, build an abstract syntax tree, perform semantic analysis on the program and then finally convert it into intermediate assembly code targeted at the Tiny Machine architecture.

2 Compiler Construction Process

2.1 Initial Steps

To begin the project, we decided to use version control to keep track of our code and coordinate work between the group. Using a private repository on Github allowed us to collaborate effectively.

We also agreed to use Java as our language of choice for building the compiler. Using the powerful JFlex and CUP libraries, as well as the concepts of function overloading and class inheritance allowed us to focus more on the process of building the syntax tree, performing semantic analysis and generating intermediate code.

To begin the project, we used professor Fei Song's provided code as a starting point from `java_tiny.tgz`

2.2 Abstract Syntax Tree

Checkpoint One was built using an incremental process as suggested in the design document.

To build the scanning and parsing portions of the compiler, we began by analyzing the C- specification document. First, we built the scanner to recognize tokens for the language, and wrote all the grammar rules without simplifying them into ambiguous grammars.

Next, we followed the recommended syntax tree structures for the C- language to create the necessary classes required to represent the syntax tree. After writing all the required classes, we defined the methods to display the elements of the syntax tree.

With the structure of the tree completed, we added the embedded code into the parser to actually build the syntax tree. After the program was able to build the syntax tree successfully, we introduced new classes that would consume errors for their corresponding types. These enabled the parser to recover from syntax errors gracefully.

Finally, we simplified some of the grammar rules by using the CUP directives for precedence and associativities of mathematical operations.

2.3 Symbol Table and Semantic Analysis

The framework for semantic analysis was also built using an incremental process, similar to that which was used in checkpoint one.

First, we improved upon our initial design from checkpoint one. We improved our error recovery system and enabled the compiler to catch more errors and gracefully resume parsing from that point.

Next, we implemented an abstract structure for the symbol table that would be able to seamlessly create new scopes, add new symbols, retrieve symbols, and check for the existence of symbols in the program. Creating a logical layer of abstraction for the symbol table was imperative to the next step — implementing type checking.

Finally, we added type checking to the compiler. We used the concepts of function overloading and type coercion to break down the checking of types. Each method would check its corresponding syntax tree structure for type validity.

2.4 Code Generation

Code generation was also done using an incremental process.

First, we took some time to understand the TM Simulator instructions by reading and testing the sample assembly code and the corresponding C-code. This enabled us to learn how to translate C- code into assembly code by hand.

After understanding the TM Simulator instructions, we were able to recognize what actions the standard prelude, predefined input/output routines, and the finale were supposed to perform. This enabled us to generate the assembly code for the various structures of the syntax tree.

We began by generating the assembly code for expressions only. We then began to generate code for array accesses and and function calls. After this, we converted the control structures like conditionals, while loops and function calls into linear calling sequences.

2.5 Modifications to Old Code

For the final project, we improved a number of mistakes that were made in previous checkpoints.

Firstly, initially our program would simply print everything out to `stdout`, but now outputs the results to the proper files.

For the first checkpoint, we improved our syntax tree display, fixed minor token errors, some precedence errors and also greatly improved our error recovery for the compiler.

For the second checkpoint, we improved the printing of the symbol table and included the input and output functions that we had previously missed. Additionally, we adapted the symbol table to also store the offsets from the frame pointer and the addresses for the declared functions.

3 Retrospective

3.1 Lessons Learned

Over the course of the project, we understood the importance of building the project incrementally, since it allowed us to break down a large problem into smaller ones. It also allowed us to pinpoint where our errors were, since we knew the project was building successfully before adding new features that did not work.

Using Git for version control and collaboration also taught us the importance of version control, which gave us the ability to roll back mistakes we made and enabled group members to see what new parts had been added to the project since we last saw it.

Additionally, we also realized how to divide work between the group effectively so as to complete the project implementation on time, while leaving adequate time for testing.

4 Assumptions and Limitations

Some of the assumptions and limitations of the project are as follows:

- We assume that functions with `void` return types do not return anything. The function may have a `return` statement, but it cannot return an expression, even if it is declared to be `void`.

  The following code sample will produce an error:

  ```
  void void_function(void){
      void v;
  ```

```
    return v;
}
```

- We assume that function parameters will not be defined as void. Our compiler will produce an error statement for void variables, as mentioned above but will still add void variables to new scopes. The following function signature would be invalid:

```
int my_function(void x, void y)
```

- If a function is declared with an invalid return type, it is assigned a void return type so as to recover from the syntax error gracefully. The following invalid function declaration:

```
invalidtypehere function(void x, void y)
```

  will result in:

```
void function(void x, void y)
```

4.1 Improvements

Some possible improvements that could be made are as follows:

- The order of the function parameters should be checked and matched with the parameter types mentioned in the function signature.
- Variables that are defined as `void` must be dealt with more robustly. A possible solution is to disallow them altogether since they cannot

actually contain anything, and are simply a byproduct of the grammar rules.

- Currently, error messages for semantic errors only offer primitive information about the error. Improving detection and providing more detailed error messages would enhance the functionality of the compiler.

5 Contributions

5.1 Braydon Johnson

- Implementing code generation

5.2 Neivin Mathew

- Improving and adapting code from previous checkpoints
- Testing and documentation

6 Acknowledgments

For the third checkpoint, we built upon our existing code from the first and second checkpoint.

In Checkpoint One, we used the starter code provided by Professor Song in `java_tiny.tgz` and also followed the recommended syntax tree structure for the C- language from the course slides.