Compilers - Checkpoint Two

Braydon Johnson, Neivin Mathew

March 22, 2017

1 Summary

For the second checkpoint, we implemented type checking for programs written in the C- language.

At this point, the compiler can recognize different scopes in C- programs and can build a symbol table for the entire program, along with recognizing symbols in more specific scopes. The symbol table enables the compiler to recognize when variables are used without declaration and if they are redefined within the same scope. Additionally, the compiler can display the different variable scopes in a program.

The compiler can also check types of expressions in C- programs. It has the ability to ensure that array ranges are integers, assignments and other operations are valid, and checking function return types.

Additionally, we also improved upon our initial design from Checkpoint One by fixing various bugs, and enhancing the error recovery of the parser. We also simplified some of our grammar rules by using the precedence and associativity directives defined in CUP.

2 Design Process

2.1 Implementation

The compiler for checkpoint two was 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 powerful 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.

2.2 Lessons Learned

In this checkpoint, 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 impor-

tance 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.

3 Assumptions and Limitations

A few assumptions were made while building the compiler. The project has a few limitations which are listed below:

- 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 main(void){
    void v;
    return v;
}
```

- For function calls, the compiler does not have the ability to check if the arguments to the function call match the parameter types that were defined in the function definition. However, the compiler will check the validity of the types and try to perform a lookup in the symbol table for the arguments.

4 Potential Improvements

5 Contributions

5.1 Braydon Johnson

- Refactoring grammar rules
- Improving error checking and recovery
- Writing tests and testing the compiler

5.2 Neivin Mathew

- Implementing type checking functionality
- Writing documentation

6 Acknowledgments