Implementation Project: Checkpoint #2

C-Minus Compiler

by Matt Breckon and Dean Way

Build, Compile, and Run

Makefile

To build and compile the project, run:

```
$ make
```

This Makefile builds the scanner file c-.flex using the JFlex lexer/scanner generator for Java, builds the parser file c-.cup using the CUP LALR(1) parser generator for Java along with the abstract syntax tree Java files found in the directory /absyn and the symbol table Java files found in the directory /symb, and finally compiles the C-Minus Main Java file.

To scan and parse C-Minus source code with optional arguments, run:

```
$ java -classpath ./java/cup.jar:. C- [-a] [-s] [file_name.cm]
```

To run the test suite of sample C-Minus programs, run:

```
$ make test
```

To run the test suite of sample C-Minus programs with the abstract syntax tree, symbol tables, or both, run:

(1) `$ make test_tree`

(2) `$ make test_table`

(3) `$ make test_all`

Development Process

Continuation

To start off checkpoint 2, refactoring was needed in order to allow the “-s” argument on compiler execution. Initially, Cminus.java looked for exactly 1 or 2 arguments, just the filename or the argument “-a” and the filename. The original implementation in the first checkpoint hardcoded checking the number of arguments; if the argument size was 2, then it verified if the first argument was “-a”. Now the program loops through all arguments checking if the arguments passed are valid and appends them to a list to be passed to the parser; the last argument of the loop is then assigned to string filename while verifying the file is of type “.cm”.

The refactoring of this portion of the code was developed by Matt.

Symbol Table

The design of the symbol table and its symbols was modelled similarly to the class structures of the abstract syntax tree. A singleton class SymbolTable was designed to represent a compiled program’s symbol table. This SymbolTable object is comprised of a stack of hashmaps. The stack represents scope levels where the SymbolTable class provides two methods, newScope() and leaveScope(), to push and pop the hashmaps onto and off of the stack. Note: the Java collection Deque was used as the stack due to the collection Stack class not inherently providing an iterator from the top of the stack to the bottom.

Each hashmap stores declared functions and variables for the given scope level based on the position of the stack. To avoid collision of functions and variables with the same name within the same scope, a list of two strings is used as the key for the hashmap: the first string being the identifier name and the second string being the type FUNC or INT to differentiate functions and variables.

The SymbolTable class provides the methods addSymbol() and validSymbolInScope() to add symbols to the table based on the current scope level and a boolean method to validate if a symbol has been declared within the current scope.

The symbol table was developed in collaboration between Matt and Dean. The implementation of displaying and printing the symbol table was developed by Dean.

Symbols

To represent the symbols, an abstract class Symbol was created to hold the symbol's id and address in memory. Two classes, SymbolFunction and SymbolInt, extend the abstract class Symbol to hold their respective attributes. SymbolFunction contains a list of respective function parameters and is of type FUNC; SymbolInt is of type INT. The class SymbolArray extends the class SymbolInt as int is the only legal type in C-Minus for arrays; this class holds the size of the int array.

The symbols and their respective classes were developed in collaboration between Matt and Dean.

Type Checking

Within the class SymbolTable, the method addSymbol() looks at the current scope level for conflicting variable declarations. It checks if the key of the symbol to be added is null within the current hashmap; if the variable is null at the given key mapping then the symbol is added to the table in the current scope. Otherwise, an error is output that a variable has been re-declared within the current scope.

The class SymbolTable also contains a method sameType() designed to compare two symbols. If either symbol is null, or if both symbols do not have matching id, type, and class, then the method returns false. The following block of code within the method checks if the compared symbols are functions. If the symbols are functions, then the method runs through all of the parameters for both symbols and compares them.

The implementation of type checking was developed by Dean.

Assumptions and Discrepancies

Running through the small programs listed in checkpoint 2's marking scheme, we noticed two major conflicts between these programs and how our compiler is designed. The first conflict being that these programs have variables of type void and attempt to utilize these variables. Our compiler is designed according to the specifications of the C-Minus text which states that, "in a variable declaration, only the type specifier **int** can be used." Therefore, the type-checking programs 1-3 within the marking scheme will flag invalid type declarations. Furthermore, type-checking programs 3-4 declare functions with identifiers containing digits. According to the C-Minus text, an identifier is comprised of a letter, followed by zero or more letters; C-Minus identifiers cannot contain digits.

Test Programs

Directly from checkpoint 1, in order to demonstrate the scanner, parser, and abstract syntax tree for the C-Minus language, five sample programs were developed in the /programs directory. These programs were named from 1.cm to 5.cm. Additionally, **gcd.cm** was included from the C-Minus text to perform Euclid's algorithm on two variables, x and y.

For checkpoint 2, an additional program was written named 6.cm. This program was written to compile without errors and was designed to test multiple scope levels of the compiler's symbol table. Variables x and y were declared globally, along with three functions foo(), x(), and main(). Function foo() contains a local parameter z; it declares variable x within its local scope and assigns to it. It then proceeds to declare a variable y within the local scope of an if statement followed by an assignment. Function x() is designed to confirm the global declaration of int x() does not conflict with the global int x; this function assigns a value locally to the global variable y. Function main() locally declares x and y followed by assigning a value to the local variable x. It then calls the two previous functions passing the local x variable as a parameter to the foo() function.

In addition, files 7.cm to 9.cm are pulled directly from checkpoint 2's marking scheme.