Implementation Project: Finale

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 into assembly code suitable for the TMSimulator.

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

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

To run the test suite of sample C-Minus programs compiling to .tm assembly, run:

```
$ make test
```

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

(1) `$ make test_tree`

(2) `$ make test_table`

Recap: Checkpoint 1 & 2

Foundation

To start the project off, git was used for version control and collaboration between the two of us. More specifically, a hidden repository on Github allowed us easy collaboration. To begin the project, we both decided on the Java language to utilize the JFlex lexer/scanner and the Java CUP parser. Given sample code for a scanner and parser to generate an abstract syntax tree for the Tiny language, we then had a basis to learn from and start our own project working towards a compiler for the C-Minus language.

Scanner

The first major task was to flesh out the terminals of the C-Minus language. Terminals consisting of keywords, special symbols, and additional tokens from the lexical conventions of C-Minus were incorporated into the declarations of the c-.cup file and then added to the lexical rules of the c-.flex file. Building the lexical rules of the scanner proved to be a straightforward task as the lexical conventions of C-Minus were mostly an expansion of the lexical conventions of the sample scanner for the Tiny language. Worth noting, the C-Minus tokenization of identifiers is much more primitive to C as the scanner only tokenizes identifiers consisting of one or more letters compared to C's identifiers consisting of one letter followed by zero or more letters, digits, or underscores.

Parser

With the scanner working to support the lexical conventions of C-Minus, the next stage was developing the grammar of the language. Using the provided Backus Normal Form

grammar from the spec for C-Minus allowed for the development of a skeleton for the parser. The grammar section of the c-.cup file was design to identically match the BNF grammar for C-Minus. This portion also proved to be fairly straightforward in implementing using CUP's syntax.

From here, precedence was added to expression operators. This allowed for resolution of shift reduce problems. A return statement is nonassociative while an assignment takes right precedence. Furthermore, the relational operators follow left precedence. The additive and multiplicative operators also take left precedence. Lastly, the else token takes left precedence in order to resolve the shift reduce conflict with an if statement that does not include the else statement.

Abstract Syntax Tree

While the parser was being developer, the abstract syntax tree was designed in parallel within the /absyn directory. Conceptually, the tree consists of nodes representing non terminals and terminals. These nodes were implemented with abstract classes for declarations, parameters, expressions, and statements. From here, classes were developed extending these abstract classes to represent the left hand and right hand sides of C-Minus' BNF grammar. As the scanner and parser read a C-Minus source file from input, the parser builds the abstract syntax tree based on the developed grammar for the language.

Symbol Table

The design of the symbol table and its symbols were modelled similarly to the class structures of the abstract syntax tree. A 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 deprecated 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.

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.

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 throws an Exception. The method haveMatchingParameters() is utilized to compare an existing declared symbol with a used symbol. If both symbols have identical parameters, then the C-Minus function was used correctly and the method returns true.

The type checking itself was done through navigation of the tree using the previously mentioned methods and through analysis of specific cases such as use of a function that returns void as an index of an array.

Finale: Checkpoint 3

Intermediate Code

Given the scope of the project and how minimal the language of C-Minus is, we opted to not implement intermediate code in order to generate our assembly code. We decided on this route as we wanted reduce the workload required in building an intermediate code generator and a scanner/parser to convert that intermediate code into assembly.

Furthermore, code optimization is also not within the scope of this project of which having intermediate code generation will contribute towards.

Code Generation

To begin the code generation, the first thing we needed to tackle was modifying our tree of symbols. Initially, each symbol node contained relevant information about the given node such as whether it was a variable, array, or function. In the previous checkpoint, these nodes also contained an integer variable named address to represent its address in memory. These nodes needed to be modified to allow the address to be set and retrieved as they were initially blank in the previous implementation. On top of this, the symbol table needed to track the current offset from the frame pointer and the global offset. To expand on this, a “current function” variable was used to track the exact function the symbol table is currently working in while traversing. The symbol table was also modified to maintain a unique stack of temporary variables found at each scope.

From here, the rest of the code generation came down to analyzing the tree traversal for key blocks of code such as declarations, functions, expressions, and statements. Utilizing temporary variables for expressions, stack frames for blocks such as functions, and maintaining the local offset of each stack frame allowed us to begin generating a starting point for assembly code. The next focus was correctly loading variables through assignment operations based on their scope and subsequent offset; with this working correctly it allowed for expressions to become the next focus.

Retrospect

Issues and Assumptions

Throughout the course of the project, the occasional brick wall was met during the development process. Along with this, assumptions had to be made to further the project. While working towards checkpoint 2, sample C-Minus programs were provided with the marking scheme that utilized digits within function names. Within the C-Minus spec, identifiers, such as function names, can only consist of one or more letters as opposed to C-like languages that allow identifiers to consist of one letter and zero or more letters, digits, or underscores.

Another discrepancy that specifically happened during checkpoint 2 was within provided sample programs while type checking. Within the C-Minus spec, variables cannot be declared of type void. Our implementation followed this spec and essentially made type checking irrelevant when it came to variables as our parser would catch void declaration of a variable before reaching the stages of the symbol table. However, type checking was still needed in order to check for use of void functions within expressions and indexing arrays with void functions.

During the code generation phase of the project, an annoying issue arose. This being we did not immediately think that the TMSimulator used to simulate the assembly code was designed to parse with specific padding and formatting. The address required exactly three columns, comments on a line had to follow exactly one space and one tab, and so

on. This was an easy fix; however, it was a clearly an annoying problem that diverted our attention for longer than it should have due to us not immediately realizing the problem.

Testing

Test Programs

Working through checkpoint 1 through 3, it was ideal to create sample C-Minus programs in order to target specific functionality of the compiler. The functionality being both components of the compiler working as intended and the compiler identifying specific errors to catch within incorrect C-Minus programs. As previously mentioned under the build, compile, and run section, we've provided easy ways to test the compiler, abstract syntax tree, and symbol tables on a range of provided sample programs.

Within the /programs directory, a total of 11 C-Minus programs can be found. The first 5, named 1.cm through 5.cm, consist of specific tests involving the scanner, parser, and abstract syntax tree. The intended focus was for the first program to scan and parse into an abstract syntax tree successfully and the following programs to contain various lexical and/or syntactical errors to be flagged by the compiler. Along with this, the program gcd.cm was also included as sourced within its header.

The following 4 programs, named 6.cm through 9.cm, consist of specific tests involving the symbol table and type checking. The first program compiles without error and is intended to demonstrate the different layers of variable and function scope. The next 3

programs were specifically designed to compile with errors testing various aspects of type checking and the use of undeclared variables and functions.

Lastly, final program 0.cm was included as a general editable file as we progressed throughout the project. This allowed us to maintain our previous files untouched while freely testing whatever came to mind when we needed it.

Conclusion

Parting Words

Overall, this project has been an extremely interesting one. It's truly helped demystify some of the things that lie under the hood of languages developers take for granted every day. Going into this originally, both of us had minimal experience in a previous course using Lex & Yacc developing a very primitive compiler taking "config" files and turning them into Java GUI-based dialog boxes. This previous project had a much less satisfying final result compared to actually working on a compiler for a language, even if that language was as basic and imaginary as C-Minus.