Implementation Project: Checkpoint #1

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 finally compiles the C-Minus Main Java file.

To scan and parse C-Minus source code with optional abstract syntax tree -a, run:

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

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

```
$ make test
```

Development Process

Foundation

To start the project off, git was used for version control and collaboration between the two of us. The first commit consisted of working with base code provided by Fei Song. This provided a basis of example code for a scanner and parser to generate an abstract syntax tree for the Tiny language. From here, JFlex and CUP were needed to generate the scanner and parser for the project. Both of us had previously installed JFlex but this was our first experience with CUP. The CUP binary was obtained online and placed within the

directory /java of the project. The Makefile was then modified to support the /java/cup.jar file within the Java classpath. From here, files such as tiny.flex and tiny.cup were renamed to support the current project as c-.flex and c-.cup.

This foundation of the project was developed by Matt.

Scanner

The second major commit to the project 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.

The scanner was developed in collaboration between Matt and Dean.

Parser

Once the scanner was setup to support the lexical conventions of C-Minus, the next stage was developing the grammar of the language. Using the provided Backus Normal Form 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 assignment takes right precedence. The relational operators take left precedence in this order: less than, less than or equal to, greater than, greater than or equal to, equal to, and not equal to. The additive and multiplicative operators take left precedence in this order: plus, minus, times, and over. 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.

The skeleton for the grammar based on the syntax and semantics of C-Minus was designed by Matt and the functionality of the grammar was developed by Dean.

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, 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.

The abstract syntax tree classes were developed in collaboration and the abstract syntax tree's functionality was developed by Dean.

Test Programs

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.

File 1.cm consists of one function largest() to determine the greatest value of three passed variables and return the result; this program was designed to compile with no errors.

File 2.cm consists of one function printIntegers() that takes an array and prints each integer of the array to stdout; this program was designed to compile with errors (1) variable "size" passed to function of type void instead of int and (2) misspelling of type void as "vod" in main() function declaration.

File 3.cm consists of one function eq() to compare two int variables, returning 0 if they are equal and 1 if they are not; this program was designed to compile with (1) relational operator typo of "!==" intead of "!=" and (2) the expression call to function output() has no trailing semicolon.

File 4.cm consists solely of the main() function that creates an int array and adds the integers together; this program was designed to compile with errors (1) main() function trailing compound-statement missing parentheses and (2) array a[] declared without size.

File 5.cm consists of a completely broken program; this program was designed with invalid assignment, invalid declaration, invalid returns, invalid expressions with a trailing

semicolon, invalid function declaration of main(), invalid array type as parameter, invalid call to main(), invalid call of function function(), and invalid return type.

The five sample files were developed by Matt.