March 8,2016

Checkpoint 1 - Achievements

Brandon Tan
George Chapman-Brown

Achievements for this Checkpoint

As a whole, our group has successfully built a working scanner and parser to recognize the C-minus language. The scanner and parser is intended to convert an input string from a file containing a C-minus program into an abstract syntax tree which will be passed to a Semantic Analyzer component which will be implemented for the next checkpoint.

Implementation

Our Parser for C-Minus is implemented in Java. It produces a class-based AST for the C-minus language in 3 steps. Step one is Lexical analysis. Our parser uses a Jflex scanner. This uses regexes to express rules for parsing the grammar. The input is a String of Text in the C-Minus language, and the output is a series of tokens. The next step is done using CUP to generate a parser from a set of production rules provided. The parser generated takes the tokens generated by flex and turns them into an AST. The grammar rules are in a variation of Baccas-Naur form which has been simplified to take advantage of anti-ambiguity constructs provided by CUP. For instance, the number of grammar rules used for arithmetic expressions were reduced from four rules to three

by specifying that the multiplication and division symbols have higher precedence than addition and subtraction symbols.

Testing

Five different types of files were intended to be used as tests for the project. The first of the five files contained a completely valid C-minus program with each possible token appearing at least once, and is intended to test the base functionality. The second, third, and fourth file each contained the same program, but with a single error included. The second file tested the parser's ability to handle missing semicolons, since it is a common problem that occurs when writing C programs. The third file tested the parser's ability to handle errors within function calls. The fourth file tested the parser's ability to handle a bad function declaration. Finally, the fifth file tested the parser's ability to handle any combination of the previous three problems. Currently, the third and fifth file contain unrecoverable errors, and the others contain either correct code or recoverable errors.

Group Assessment

Brandon and George worked together to build the project. Most of the work was done in tandem, using Git to sync code changes frequently. George did most of the work defining the java Absyn code, and Brandon made most of the Cup and Flex code. Both people wrote the Documentation, and George wrote the Readme. Brandon wrote the test C-Minus programs.

Results

Our Program Generates an AST using C-Minus Files. The AST is expressed as a tree of abstract nodes, and nodes are checked and loaded using recursion. Each node of the tree is looked at in pre-order sequence, visiting each node of the AST for inspection. The AST is correct but may look strange due to a workaround that had to be done to handle arithmetic expressions. In order to preserve the type of an arithmetic expression token as an “Operation Expression (OpExp)”, non-operation expression tokens which arithmetic expressions reduce to (for instance, single numbers) would have to be converted to Operation Expressions by creating a new Operation expression for an operation “x + 0” where x is the non-operation expression token in question.

Error Recovery

In order to recover from errors, we implemented basic panic mode recovery. This works by searching for a closing statement after the error, and discarding the preceding line. Then the error statement is discarded, printing a warning state. Error tokens are treated as terminals, and when they are reduced, the output is discarded instead of creating the abstract syntax tree. These synch states use brackets, semicolons and curly braces to synchronize the discarded state without breaking the Abstract Syntax tree. The line number is kept throughout the program to report where errors occurred to the User, and different print statements tell the programmer what kind of error occurred.

Other errors, including invalid variable declarations, are regarded as fatal errors and stop the parsing prematurely.

Abstract Syntax Tree Implementation

The Abstract Syntax Tree is implemented using a series of inherited Java classes. Almost every class is inherited from the Class Absyn, which describes how to print each class, using recursion to follow the data to the outermost leaves. Absyn also includes a position variable for keeping track of where tokens came from. As Cup reduces tokens using the grammar rules, data is passed into a new class of the same kind as the left hand side. This class will contain all the data from the right hand in various forms. Often this involves holding Expression variables, for example in the case of an operation expression (5+7), you hold the Left hand expression, an operation integer, and a right hand expression. Then when you print an operation expression, you print the left hand side, operator, and then right hand side. These expressions are each printed recursively as well, following the entire tree. In the case of a Declaration List, a linked list is created to hold the old declaration list, (or create a new one if it's the first), and then loop through the list, adding the new declaration to the end of the list. When you print, you loop through them in reverse order, since the Parser is a bottom up parser.