CIS*4650 Checkpoint 1 – Documentation

Overview

This document covers the first checkpoint (partial implementation) of a parser for the C Minus programming language (C-). For this checkpoint there were three deliverables; a scanner, parser, and parse tree visualizer. The scanner for the C- language was made with the help of the "JFlex" lexer tool. The parser was created using the "cup" parser generator tool. Both tools were written for use with the Java programming language. The final deliverable is a program that traverses the parse tree for a given C- program and prints out the corresponding nodes.

Assumptions

When building this programming language parser, a couple of assumptions needed to be made. I needed to assume that the context-free grammar outlined in the C- specification file was valid in handling all possible programming structures in a C- program. I also needed to assume that the abstract syntax tree classes outlined in the lecture notes would correspond to nodes in the grammar and could thereby handle all C- structures.

Limitations and Possible Improvements

This parser implementation handles all possible structures defined in the C- programming language which is essentially a simplified version of C. It is limited in that it can only handle and continue parsing without aborting for the errors outlined in Parser - Error Handling. Possible improvements would include implementing more extensive and robust error handling for each of the different program structures, continuing parsing whenever possible after encountering errors.

Design Process

This partial compiler implementation was orchestrated as a group of modules. The JFlex tool is used to create our scanner. The parser created with the help of cup will use this scanner as a helper when reading C- symbols from a program file. The parser then creates the syntax nodes using the C- code grammar. Once the parsing has been completed, we can traverse the resulting abstract syntax tree.

Scanner

To build the scanner we simply define all the possible token classes available in the C- programming language using regular expressions. Many of these token classes will be taken verbatim from the C- specification. Using a lexer builder tool we can develop a scanner to feed us a sequence of C- tokens from a program file.

Parser

To build the parser we use a parser generator tool. With this tool we can define a context-free grammar for C- using the grammar outlined in the C- spec. We use embedded code to instantiate classes that correspond to each variable and terminal in the grammar.

Abstract Syntax Tree

Using the code embedded in the parser we can build a tree of nodes beginning from the start variable when using the parser on a given C- program. We can utilize the visitor design pattern which allows us to add new syntax tree nodes with very little additional code and traverse them depending on their structure.

Implementation Process

Scanner

To implement the scanner we modify the regular expressions given in the C- spec and port them over to a **cminus.flex** JFlex file. In addition, we create token classes for comments, whitespace, and error. In terms of C- these tokens have no meaning so they must be captured by the lexer but omitted when the time comes to generate symbols for the C- language. We include a special JFlex tag **%cup** to signify a cup-jflex integration and the tool creates a Lexer.java class that we wrap and use in our **Scanner.java** class.

Parser

To build the parser we define all of the terminals and non-terminals from the grammar in the cup specification file. To build this parser incrementally we can define all the terminals in our cup file as each of the token classes specified in the JFlex file and all the non-terminals as simple **Exp** types to begin with. Again, we use the C- specification as a reference. Next we define our grammar in cup syntax but without instantiating any **absyn.java** classes. We apply precedence rules as needed to avoid any shift/reduce and reduce/reduce conflicts. Once we confirm this parser works by testing it against the sample C- programs we can define the rest of the abstract syntax tree nodes as specified in the CIS*4650 lecture slides. According to the grammar rules specified in the cup file we can now write the embedded code necessary to create the syntax tree. We map the syntax nodes to their corresponding productions, instantiating new nodes when necessary. We can confirm a working grammar by parsing sample C programs and examining the outputted tree. Finally, we use the built-in precedence rules from cup to simplify our grammar.

Grammar Simplification

1. Merge rule 2 and 3.
2. Copy rules 14-17 into rule 13.
3. Remove rules 19, 21, 23, 25, 27. Copy their terminal values to their parent rules.

The following precedence rules are used to resolve shift/reduce and reduce/reduce conflicts:

precedence left error;
precedence left INT, VOID;
precedence right ASSIGN, ELSE;
precedence nonassoc LT, LTE, GT, GTE, EQ, NOTEQ;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;

Error Handling

For this checkpoint the focus of error handling was on the major components/structures in C-. This included invalid declaration sequences, expression sequences, and expressions with binary operations. Additionally, I handled missing semicolons for variable declarations, missing closing braces for compound statements, invalid types for variable and function declarations, and invalid argument lists. In cup this means introducing new rules with the error symbol in place of the expected valid terminal or non-terminal. To attempt to handle the errors gracefully I print the error to stderr with the type of symbol, error message, and line number of where the error took place. I attempt to continue parsing the C- program by completing the correct structure whenever possible or introducing a new error symbol specific to the missing/invalid type (ex. NilDec, NameTy.ERROR).

Abstract Syntax Tree

Every concrete abstract syntax Java class located in **abysn/** implements an accept method for a visitor class. In this method the syntax node delegates display functionality to the **AbstractVisitor** class passed in as a parameter. The syntax node passes its own context (using

this) to the visitor class such that it can display the node according to its own definition. For example, an **IfExp** node would display its structure differently than a **VarDec** node. A concrete visitor class **ShowTreeVisitor** will implement the **AbstractVisitor** class and take on the responsibility of defining how different nodes are displayed. This way new syntax nodes can be added easily as they just need to extend Absyn.java and call *AbstractVisitor.visit()* in *accept()* and have a visit implementation specified in the visitor class.

Lessons Learned

Regular expressions complemented by the use of a stack can be suitable for smaller languages (as in the Warmup Assignment) but as we get more complex languages with many variations of structures we need to consider using context-free grammars which are nicely abstracted away from their implementations by tools like cup and yacc.