Programming Assignment 2: Implementing Syntactic Analysis

Sung Won An (sa579), Ju Hee Lee (jl2755), Thomas Lee (ttl33), Jonathan Park (jp882)

February 21, 2016

How to Run the Program

The “PA2.zip” file contains a pre-built compiler program. To run the program, one should first unzip the “PA2.zip” file. Then, one can run the program with “PA2/xic” script and the appropriate arguments.

We used Apache Ant to automate the build process and “PA2/xic-build” script has necessary commands to compile the program.

Summary

In this assignment, we generated the CUP parser which performs syntactic analysis on the lexer generated list of tokens from JFlex. The most challenging part of this assignment was trying to figure out how the lexer generated tokens is related to the terminals and the nonterminals used in the CUP file. For testing, we first ran our program using the provided test harness, and we came up with our own test cases as well. Throughout this assignment, we solidified our understanding of how LALR(1) parser works.

Specification

The following were specified throughout implementation because they were not specified in the programming assignment specifications.

1. If an array or multidimensional array is used either as a function return type or as a function argument, then it cannot have indices any of the brackets. For instance, int[3] is not allowed as a return type.
2. We do not allow the following array initialization {,} since there is no final element.

Design and Implementation

Architecture

Important classes:

- **Main.java**: This class instantiates the parser which uses the Scanner generated by the lexer to produce the tokens. It also instantiates the GlobalPrettyPrinter which creates the CodeWriterSExpPrinter that is used by all the classes in the AST. It begins by printing the largest object, Program, and works recursively down to print the entire S-Expression of the program.
- **xi.cup:** This is where all the grammar for parsing Xi is implemented. It lists all the terminals (which includes all the tokens generated by the lexer) and the nonterminals

which we had to associate with a Java class to build the AST. This file also specifies the precedence and associativity of each of the operators.

- **GlobalPrettyPrinter.java:** We created a global instance of the CodeWriterSExpPrinter which all classes in the AST can get by calling the getInstace() static function. This global instance is used by all the classes to call functions such as printAtom(), startList(), and endList(). The actual printing structure of the S-Expression was handled in each specific class of the /ast directory.
- **ast/***: This directory contains all of the classes that represent the nonterminals, and some wrapper classes for the terminals. Each class defines its own version of the prettyPrintNode() method, which is called recursively throughout the AST to print the complete S-Expression. The classes also contain the constructors which are called in the action code {: … :} in the CUP file.

To explain the abstract syntax tree in more detail, we created the following *key* interfaces:

- Expr: implemented by various kinds of expressions such as constant, binary expressions, unary expressions, function calls, identifiers, nested expressions, and array elements.
- NakedStmt: implemented by different kinds of statements such as if- and while-statements, function calls, and variable declarations.
- OpExpr: implemented by the unary and binary operators
- Type: implemented by different kinds of types such as primitive types and array types

We chose to create these interfaces for organization purpose and for the future. In the future, we plan to perform semantic analysis, which will require the hierarchical structure between related classes. All the subclasses of Expr represent expressions, and all the subclasses of NakedStmt represent statements.

Code Design

We used the singleton design pattern for the construction of the S-Expressions. In the main method, we created one instance of the S-Expression printer. From there, we created our own pretty print method called *prettyPrintNode()*, which is implemented by each class of the AST. To print the S-Expressions, we called prettyPrintNode() recursively while traversing the AST, inserting start and end brackets where needed. The abstract syntax tree was built using a tree structure. The classes did not inherit from a single root; instead, there were multiple “roots” of the tree. For instance, we made interfaces for Expressions and Statements, which were implemented by different classes. When we were implementing the printing of the S-Expressions, we sometimes ran into the design choice of recursion versus iteration. We usually implemented using recursion since it had cleaner code, but sometimes, iteration turned out to be simpler than thinking about all the different base cases of recursion.

Programming

Generally, we carried out the bottom-up implementation strategy. We first started by implementing the CUP file to get a better understanding of what needs to be done in this project. After we came up with the list of terminals and nonterminals, we started to build the AST. At the end, we implemented the Main method and error catching. Pair programming was very helpful for this assignment. Among the four of us, we split into groups of two to implement the grammar in the CUP file. When we built the AST, all four of us coded together since the hierarchy was very interconnected. It was hard to debug the grammar once the code compiled because even if a line number was specified, we had to figure out which grammar expression was causing the problem in parsing.

Testing

We divided and conquered testing by making individual tests to see if each non-terminal would reduce or be recognized correctly. We began with identifiers, expressions, arrays, and then built our way up to blocks of statements, function declarations, and then programs. We also actively parsed through Piazza posts to look for corner cases that we haven't considered yet, and consistently changed our grammar to account for those. If there were errors, they were all in small design choices and/or syntax peculiarities that we originally didn't consider.

Work Plan

We used pair programming to implement the grammar in the CUP file. Pairs of two worked on different parts, and we merged the results together. JuHee and Jonathan worked together to build the AST. Thomas and Jeff (Sung Won) worked together to implement the printing of the S-Expressions. We had to work in parallel because the S-Expressions are highly dependent on how we implemented the AST. It was hard for any person to begin testing until we had completed implementing the grammar, the AST, and the S-Expressions because we had no visible outputs until then. But once we had the S-Expressions printing out, we worked together to solidify our design choices so that we did not have differences in our design.

Known Problems

No known problems!

Comments

We spent about 50 hours on this assignment. The CUP file was hard to get started with because we were not familiar with the syntax, but once we understood how it works, it was pretty straightforward to implement the grammar. However, we had to do a fair amount of designing before building the AST. We were thinking about choices between super classes, abstract classes, interfaces, and the multiple hierarchies between the classes and so on. It was hard to test for this assignment because we could only start testing once we had the S-Expressions down. A good advice for this project would be - start out by implementing the

provided grammar in the PA2 assignment. I liked this assignment because it really got us to think about how LALR(1) parser works, and the differences between LL and LR parsers.