JOOS1W Compiler – Stage 1

Btara Truhandarien, Eric Lui, Trevor Kitt

# Overview

What has been completed of the compiler so far is organized into three primary stages: scanning, parsing and weeding.

The compiler is implemented in C++ with a heavy emphasis on modularization. Each stage of compilation is handled by dedicated classes that are constructed from a single main function that orchestrates the entire compilation procedure.

# Scanning

The responsibility of scanning the input files is handled by a single scanner object. This scanner contains a vector of DFA objects that are used to match tokens for the input text. Every DFA is designed to match tokens of a single type.

Every DFA begins in its start state. When the scanner reads a character from the input file, it loops over the vector of DFAs, calling the transition function for each DFA.

If the transition causes the DFA to enter an error state, the scanner flags the DFA to indicate that an error state has been reached and moves on to the next DFA. This DFA will not be used again until the scanner has finished generating the current token.

If the transition causes the DFA to enter an accepting state, the scanner takes note of the success and records the type of token that was found.

If the DFA enters a non-error, non-accepting state, no special note is taken.

The scanner will continue to read input and feed the input to the DFAs until every DFA has reported an error. Once all DFAs have entered an error state, the scanner constructs a token based on the final DFA to reach an accepting state. In this way, Simplified Maximal Munch is implemented.

After generating a token, the scanner resets all of the DFAs and removes their error flags. The final character read from the input file for the previous token (ie. the character that was not accepted by any DFA), will be fed to the newly reset DFAs as their first character to transition on.

When the compiler is invoked, it accepts any number of file names as parameters. For each file, the main function passes the file to the scanner object and has the scanner generate a vector of tokens for that file. The scanner reads every file independently and has no notion of the existence of multiple files.

The vectors of tokens generated by the scanner are stored in a map created by the main function. This map creates a mapping between the name of each input file and the vector of tokens that was generated for that file.

The scanner will throw an error if it encounters any non-ascii text, or if every DFA reaches an error state before any DFA has reached an accepting state.

The scanner will also throw an error if the bannedKeywordDfa reaches an accepting state. The bannedKeywordDfa checks the input for keywords that would be accepted in Java but are not accepted, and therefore banned, in joos1w.

# Parsing

The parser is implemented using a stack and a list of parser shift/reduce rules. It uses the LALR algorithm to parse the sequence of tokens generated by the scanner.

The grammar was developed with extensive reference to the online Java documentation as well as the specific details of the Joos language.

# Weeding

The weeder is designed around a set of individual weeds that each need to be checked. Every parse tree generated by the compiler is checked by the weeder for each of the weeds to ensure that the program behaves in a legal manner.

# Testing

The compiler was tested repeatedly throughout development. The assignment 1 tests were copied from the student environment and included in the git repository for this project. As the compiler was written, these tests were used to measure progress and check for bugs in the components that had been finished so far.

In addition to the assignment 1 tests, we created a number of our own tests to check for specific cases that did not seem to be represented in the assignment tests. These tests were added to the test folder containing the assignment tests in our repository.

We created a testing program to automate the process of running these tests and to provide results in a readable format.

# AST

The compiler will make use of an AST to simplify the parse tree generated by the parser. The AST is still in development, but the current design involves creating a class for each nonterminal in the AST.

These classes will have a meaningful name that should improve code readability for the rest of the parser.

Each class will contain a parent pointer and several child pointers. The class will have a child pointer for every possible type of child that that class could have. If a child is missing from a particular object, the pointer for that child will be initialized to NULL.

There will also be an epsilon class to represent non-terminals that are reduced to null.

The AST classes will eventually be tagged with attributes which can be used by attribute grammars throughout the remainder of the compiler.