CPSC 4600 Project Language (PL) Compiler Technical Documentation

Roderick MacCrimmon
Spring 2019

1 Purpose

The purpose of this program, in its current state, is to perform the scanning stage of compilation for a single PL source file and perform basic error detection. The input file is parsed into tokens, which are stored in an output text file. When an invalid identifier, numeral or other character is detected, an error message is printed to the console along with the line in which the error occurred.

2 Design

Data Types

Symbol

- an enumeration type containing values for every token in the PL language including keywords, identifiers, special symbols, newline, etc.

Token

- A simple class with symbol, lexeme, and value data members. They symbol is one of the values described above, lexeme is the actual text contained in the source file, and value is an integer currently only used to store the value of numerals (integer literals).
- Only member functions are a default constructor, copy constructor and assignment operator

Classes

SymbolTable

- A hash table used to store identifier tokens. The table is initialized with all reserved words tokens to provide an easy method of determining if a scanned word is reserved. The hash function used is Contains a vector of c++ unique_ptrs for creating new tokens (unique_ptr used to avoid the need for explicit deletion) and the following operations:
- **contains**: check if an entry exists in the table for the given key
- **insert** insert a new element with they given key
- **get** retrieve the token with given key
- hashing is implemented by the private method **hash_fn** which uses the djb2 hash function found online at http://www.cse.yorku.ca/ oz/hash.html. Uses linear probing to handle collisions

Scanner

- Performs the lexical analysis on the input program. The constructor takes a reference to an input file stream object as input, from which it will read the input program, and a reference to a symbol table object into which identifier tokens will be inserted.

- **get_token**: the only public method, which returns the next token in the input file until it reaches the end of the file. All subsequent calls will return an end of file token.
- contains a large number of private methods for parsing the input text. Anything that begins with a letter is considered a word and must follow the pattern letter (letter | _ | digit*). Anything beginning with a digit is considered a numeral and must follow the pattern (digit)$^+$. Other symbols are considered individually.
- Tokens are also returned for invalid words, numerals and symbols and unrecognized characters.

Compiler

- The main "Administration" class which is responsible for creating, managing and using the other component objects. Currently stores scanner and symbol table objects. The constructor takes a reference to an input file stream object as input, which is used to create the scanner object.
- **scan**: there are two versions of this method that differ only in their output. One will write tokens to the provided output file stream in the format *symbol-name lexeme value* (space-separated) while the other will simply fill a vector of tokens. Only the first is currently used.
- Both scan methods make calls to the private **tokenize** method which repeatedly calls scanner's get_token method until an end of file token is returned. Any time an error token is returned, the rest of the line is ignored. A line count is also maintained by incrementing every time a newline token is seen in order to output error messages which identify where errors occurred

3 Limitations/Possible Extensions

Implementing the rest of the compiler will be the most important thing going forward, but there are some improvements that could be made to the scanner as well. Tokens currently take up more space than they need to by always having a lexeme and a value. For many tokens all we need is the symbol name for it to serve its purpose. While this is a pretty minor improvement, there could be multiple types of tokens inherited from a base class for word tokens, numeral tokens, etc. that only have as much info as needed. Since I already use a table of token pointers in the symbol table, incorporating this polymorphism would be fairly straightforward. Additionally, all lexemes could be stored in a buffer and only an index would be needed in the token rather than the full string.

One current issue with the scanner is that it does not enforce the PL limitation wherein only the first ten characters of an identifier are meaningful. While removing this limitation seems like an upgrade, it is not true to the PL language specification. This condition should be straightforward to implement but will require some additional testing.

Another improvement that could be made is to implement the scanner in a way that more closely resembles a finite-state automaton, rather than with the current conditional based approach I'm using. This would make rigorously describing it's output easier and likely be less prone to bugs, though maybe more difficult or tedious to implement.