CPSC 4660 PL Scanner

Steven Deutekom
Ricky Bueckert
University of Lethbridge

March 11, 2020

1 Purpose of Software

The software we are building is a PL compiler that will compile files in the PL language so that they can be run on a stack machine. The current phase is the implementation of the Parser with error checking and recovery. This program will take text files and parse them to see if they are syntactically correct pl programs. Currently it outputs the tokens that were parsed to a file and identifies syntax errors. The scanner built in the previous portion of the project is used to collect tokens and pass them to the scanner. Current test files are available to test each function in the recursive decent implementation of the parser. They also test several different errors to ensure proper error recovery. Debugging output can also be enabled to see what recursive descent function has been called and when tokens are matched.

2 Design

This section decribes each class, its function, and how it connects to other classes. More detailed documentation of the specific functions and members of each class is available in the **ReferenceDoc.pdf**. The **New Additions** start in the Administration section.

2.1 Scanner

The Scanner class is the main class for this part of the project. It is responsible for converting a string of characters into Tokens. It collects groups of characters in the text based on the first character of the group and returns a token using the longest valid lexeme that can be made from the group. The groups are Identifiers and keywords, Integers, And various symbols, as defined by the PL language.

Once a group of characters is classified a token is created and it is returned to the process that called `Scanner::getToken()`. If the lexeme is a name it is added to the Symbol table and the token is updated to discover if the lexeme is a keyword or an Identifier.

If the scanner discovers a character that is not in the alphabet an bad character error token is returned. If the lexeme is a number and it is larger than a valid integer then a bad number error token is returned. If the symbol table is full then the scanner returns a full table error. All of these errors do not stop the scanning process or change the way subsequent lexemes are scanned.

2.2 Token

The token class holds information about a lexeme. In our implementation we have given it 3 fields. First a Symbol to identify what kind of token it is. Second there is the lexeme that the token was created from. Finally we have an integer value that stores the value of an integer token or the position of an identifier or keyword in the symbol table. The token class is only responsible for holding data and has public methods

to access and mutate its data members. For debugging purposes it also has a method to return a string representation of itself that can be printed.

2.3 Symbol

Symbol is an enum that collects all the possible token types. These are used to classify and identify different tokens. In the same file there is also a table that maps token types to a string identifer so symbols can be printed in a readable way.

2.4 Administration

The Administration class is responsible for connecting all the other classes. It holds information for input and output files. It also is responsible for calling the scanner to get tokens and keeping track of line numbers. The class outputs each scanned token to a file while it runs. It also keeps track of the number of errors per line and makes sure that only one error is printed per line. It also keeps track of the total number of lines with errors. After 10 lines have encountered errors the parsing is suspended.

2.5 Parser

The paser class is responsible for ensuring the syntax of the input program is correct. There is only one public function in the parser `parse()`. Calling this function begins the recursive descent parsing of the input file. Through the administration class the next token is obtained from the scanner. One lookahead token is maintained at all times. This token is used to decide what grammar rules to pursue while parsing is taking place.

Each function in the parser follows a rule in the grammar of pl. These functions either call other functions representing more rules as well as match tokens. If the lookahead token does not match up with the next expected terminal symbol then an error is thrown. The parser calls the administration class to print the error contents. Then in order to recover from the error the parser looks for the next token that will put the parser back in a stable state so it can continue parsing. In order to do this, the parser must pass a set of tokens that it expects to see to the recursive functions. This way it knows when to stop advancing after an error. Theses sets are explained in more detail in the grammar section.

If the parser is free of errors there is currently no output. Debugging information that displays the parser functions that are called and what tokens are matched. When errors are encountered it is possible to see what rule was being parsed.

2.6 Grammar

In order to properly perform error recovery the parser must know what terminal symbols are expected to put a the parser back in a stable state. The set of expected tokens is called a stopset. These stopsets are built by the first sets that were built from the pl grammar.

In the grammar header we explicitly build all the necessary first sets and store them in a map. We maintain an enum with non-terminal symbols to index this map and access the first sets. This map allows us build stopsets easily by unioning these sets together and passing them around during the parsing. Along with this map there are some helper functions that allow easy union a set and easily check a set for membership. The membership function is not stricly necessary, and is not being used yet, but we hope it will improve the readability when checking to see if a symbol is in a stopset.