Compiler Design 4600 -Part 2 -Scanner and Parser Construction

Jesse Bevans, Kevin Watchuk,
Ryan Wenman, Thomas Richardson

Design:

An overview of Scanner:

Main opens the files and initializes the other classes. Administration deals with reading from the input file, and outputting to both the console and output file. Administration works on the same file pointer as the Scanner keeping track of the new lines while it continues to call for tokens from scanner. The scanner's only public method is getToken which looks into the input file and determines what the next token is. Following the rules laid out. It identifies symbols and numbers handing them back with no issue, in the case of identifiers or keywords however it identifies the lexeme. After it identifies the lexeme the scanner hands the lexeme to the symbol table to see if it is already contained within the hashtable. If the lexeme maps to a token within the hashtable then that token is returned and handed back to the scanner which then hands the token back to the caller, however if the token isn't in the hashtable then a new token with the relevant information is inserted into the hashtable by calls from the scanner. With each getToken call administration is keeping track of the errors and the error count.

An overview of Parser:

Since we only have a single addition of the Parser class we can refer to the class description below for its overview.

Purpose of classes:

Token:

Token is the base class for all token types, it is unable to be created in and of itself but has useful functions defined to be called by classes derived from it. It has a variable sname of type Symbol (which is the enumerator) to determine at basics what token is being processed.

Scanner:

The scanner creates and hands back tokens through the getToken() function. All private functions are for proper token identification and recognition.

NameToken:

NameToken is derived from token and is used for all keywords and identifiers. When being created they are checked against entries in the symbol table to check if they already exist or not. The added attributes are position to identify position of token in the hashtable as well as lexeme to identify the lexeme of the token.

Symbol:

The purpose of Symbol is to hold the enumerator being used to determine the sname of every token.

SymbolTable:

Implements a hash table with linear probing to maintain pointers to Tokens. Will be extended to be used in the future stages of the project.

Search(string lex) : Searches for a token that has a lexeme that matches the given lexeme. Returns a null pointer if not found

Insert(NameToken* tok) : Inserts a given name token into the table. If it already exists, returns that pointer from the table. If the table is full and the token does not exist in the table, returns a token that indicates the table is full, so that scanning can abort.

Hash collisions are dealt with by linear probing. Tested with strings ‘call’ and ‘aB’ which hash to the same value, ‘aB’ is shown to be correctly inserted after ‘call’.

Administration:

Administration reads the input file and output file . As well it handles the error messaging and outputs them to the console.it handles the overall communication between main the scanner. Using the function scan, we can output the error message and typing, as well as handle max errors and line cutting.

NumberToken:

NumberToken is derived from Token. It is the type of token used to deal with numbers. The only added field is value which is an interview used to determine whether the value of the provided numeral.

SymToken:

SymToken is derived from Token. It has no added fields. It exists for symbols as symbols can be derived from the enumerator as discussed in class. The reason this class exists is because ABC Token cannot be instantiated.

Main:

Our main file is defined as our driver program, it is named main instead of plc.cpp to present a target for our makefile.

Our driver takes in two input arguments; one for input, one for output. If either of these arguments are not present then an error message is displayed in the terminal and the program exits. This program is designed as a skeleton to implement the scanner, symbol table, and administration. These classes are created as objects and controlled through the driver program. Once the scanner completes and the input file has been scanned to the symbol table the objects are destroyed to prevent memory faults.

Parser:

When the parser is first constructed it is passed the current scanner as a parameter as a reference to allow efficient communication between the two pieces of code. The parser class also has member variable called laToken which contains a pointer to the current token passed to the parser from the scanner. The Parser class has a single public method called parse(). Which initially checks to see if the laToken is NULL which indicates the necessity of the first call to getToken which is the public member function of Scanner to pass tokens to the parser. Once laToken has a value then parse() recursively descends into an implicit parse tree to check syntax. When laToken equals the End of File token parse() returns 0 for a successful pass through the target program or it returns the amount of errors encountered in parsing.

The implicit parse tree is created by recursively calling the functions that have been defined to represent the various production rules of the PL language grammar.

For testing/demonstration of this part of the project, the parser prints the name of each function as it is entered, as well as the name of a token when it is read, and when it is 'matched' with its terminal position in the language.

Error reporting/recovery is currently handled as Panic-mode error recovery. When an unexpected token is seen, the parser returns to a function that could possibly parse it as a following token

Limitations:

Scanner :

1) We do not explicitly give a warning with regards to identifier lexemes that get cut due to a 10 character limit
2) We need to have an input and output file, however only the input file needs to be an existing file, the output file can be made on execution.
3) There is a limit currently to many variables the user can make, currently that is 290 distinct variables (the hash table is 307 and there is 17 keywords).

Parser :

1) Due to current lack of ID type checking, variable/constant/procedure names cannot be differentiated. This causes a problem in the Factor function where a variable name should parse differently than a constant name, but for now an ID will always parse as a variable name.
2) Error reporting and recovery works, but needs improvement. Administration was not properly re-implemented, so errors are not counted and limited.
3) We plan on improving the parser before implementing the final part of the project, communication and scheduling got in the way of a well written project.