CPSC 4600 Technical Documentation

Brett Cassidy & George Chelariu

SCANNER & PARSER

Design

The scanner started as the program which Shahadat provided in his slides. We worked upon that foundation to include not only some missing symbols but also remove certain symbols that are not part of PL.

We chose to a vector for the symbol table to hold any of the tokens. The tokens have a hash value to help prevent any collision and the hash function also reflects that by creating at least a semi-unique hash value for every value in the vector.

Upon detecting that two symbols would have the same hash value, the hash value for the symbol it tries to insert gets added onto itself until there is an empty slot for it.

In order to distinguish between different special characters, we set up a long list of IF statements recognizing every special character. Special characters that have more than one character (such as “->”) consult the look ahead character and are also at the beginning of the special character recognizing function. This is done to take care of the most complex symbols first. Every other token is simple, just creating a token with appropriate name, value, and lexeme before we just return it.

We got rid of the prototype that converts the enumerated symbol number (from 256 to 283) into the symbol name (NUM, DOT, PLUS, etc.) and put it in the token class. The getSymbol function uses a helper function called myName to call spellS(Symbol) which then calls getSymbol(). This is done to have less function calls by the user.

The maximum number of errors the scanner will allow before it quits is 10. We felt this is enough to show the scanner recognizes where the errors are without taking up too much space in the terminal. Also, the scanner only detects one error per line as it does not need to worry about how many errors are in the line, just which lines.

The administrator counts the lines of the input file as it’s scanning tokens. Whenever the token is a new line character, the administrator advances the total number of lines counted thus far.

Each token is clearly shown in the output file, where it shows its name, value, and lexeme (if applicable). Empty lines show only the line number. This is done to make the output file look similar to the input file and make the program output easier to read.

The maximum number of errors in total is 10 for both the parser and scanner combined.

Our parser does not need an output file, it only determines if the input matches the PL grammar, which we have developed the parser to recognize. Much like the scanner, the parser shows the line number of the error as well as the token that it's having trouble with.

The parser is set up with a large amount of function calls, each of which follow the PL grammar. Error handling is done with a bunch of Boolean functions and the error functions are split up into three separate ones. This is done to handle different cases (IDs, nums, etc..)

Classes

1. Token
    a. Used as a holder object to control the information that all the other components can read as needed
2. Administration
    a. Is the main controller for the parser and the scanner, passes the relevant information, and acts like a bridge to the two components. It also puts all tokens retrieved from the scanner to be put into a vector for the parser to use.
3. Symtable
    a. A table designed to hold variable information and knowledge of the declared variables (to be used at a later time).
4. Scanner
    a. Looks through the whole PL program, converts everything to token class objects to be manipulated, and looks for invalid Tokens.
5. Parser
    a. Takes Tokens one at a time then follows through a bunch of recursive functions to follow through the grammar rules of the language.
    b. Uses two vectors to hold the follow set. One uses Lexemes for things like "read" "end" etc and the other holds names for things like NUM, ID, etc.

c. Has a Token holder to hold the Token that we are currently handling
d. Has two strings: currentName and currentLex. These are from before and are just handles for the information from the input Token.
e. Has a pointer to the Admin so that it can report errors and getTokens a lot easier

Functions

1. Token
    a. String myName() is a helper function to return the Token type (ID NUM etc…) and is mainly used to help readability.
    b. String spellS returns the spelled out symbol of the token.
2. Administration
    a. Int scan() sets up the environment to call the scanner by setting up a loop and calling getToken a lot. If any of the tokens are bad ones then it calls error and passes the relevant info to error. The new line token calls NewLine. If good tokens are found then the scanner puts them in the output file and also puts them in a Vector for the parser to go through.
    b. Void NewLine counts lines for error handling only called when a newLine Token is found. It makes the error boolean true so we can count errors again if found.
    c. Void error outputs Scanner errors to the screen with the line number and error type. It also makes the error boolean false to allow only one error per line.
    d. Int parse calls the parser to start parsing via the work function, and it also returns how many errors were found by the parser
    e. Token get is how the parser gets tokens: through the administrator that then sends them through the parser
3. Symtable
    a. Int hashfn(string) is our hashing function to make the hash table work properly. This is done by adding the ASCII value of the first 10 letters, underscores, and numbers together modulus by the size of the hash table. For example the ID *ab* and *ba* have the same values. Returns hash value.
    b. Void LoadResvd() is called in the constructor and just pre loads all the reserved words to the symtable.

    c. Int search uses the hash function to look for certain Lexemes and stops when it finds an empty space. Returns the Lexeme spot or -1 if it's not there.
    d. Int getOC and bool full are both to return how many elements there are in the symtable.
    e. Parse error() takes errors from the Parser and passes the info through the error function
4. Scanner
    a. The scanner constructor constructs a scanner with an input file pointer, a symbol table pointer that is initialized immediately to the reserved words, and a look ahead character assigned to a space.
    b. getToken consults the look ahead character and decides what to recognize based on that. The look ahead character is advanced if we encounter spaces or tabs. Also, if we detect the comment character ($) we recognize it as a comment, and return a new line token.
    c. bool isX, where X is either a space, an alphabetic character, a number, a special character, or a comment. These functions detect what the aforementioned look ahead character is dealing with.
    d. Token recognizeX, where X is either a name (ID), a number, a special character, or a comment. Each function returns an appropriate token based on what kind of token it is. The recognizeComment function is void, and simply advances the pointer until the new line character.
5. Parser
    a. work(admin&) is the main function that does the parsing. It knows what the admin is to go through that to get tokens and to report errors through it without needing to return out of the parse tree to report them
    b. Void Verror() is an error reporter. It tells the administrator that there's an error so that it can handle it. The error function then returns to a lower level since it does not add anything to the follow set. However, the parent might add things to its follow set.
    c. Another void error() *without the V* function handles the error then returns the depth of the recovery Token by first searching through Lex.
    d. Void error2nd() is the same as error but does not report the error to the administrator. It instead searches through Name first and tries to find the token value that is of least depth before returning it.

e. stop() helper function goes through the vectors to find the depth.
f. stop2nd() is the same as stop but uses the lexeme before the name. These work by taking the current Token and searching through stopLex and stopName to find if the current Token is in the follow set and then returning the value +1 if it was found in stoplex or $(value + 1)(-1)$ if it was found in stopName. We do this so we can return one variable that finds both the cases.
Using this, we can then find if the error is in a parent's direct follow set by seeing if both the numbers above are in between the size of the two vectors at the start of the function, with stopName.size() being turned into a negative number
g. All the 27 functions, from Expression to Factor, are layers of the recursion.

Limitations

1. Scanner
    a. No error output but can get confused as the hash table only uses the first 10 characters to determine the hash
    b. Identifies Keywords as ID's that might cause future issues
2. Parser
    a. In the test file in line 30 if you remove a terminal symbol, it will cycle through and find "true" as that is an ID will return up to statement part as assign statement handles that instead of realising that it is a keyword and is handled differently there will be fixed with the next part of the assignment
        i. This is because it reads the keyword as an ID and Id is a very base case that needs to go up to statement part