Scanner

/accounts/classes/janikowc/submitProject/submit\_cs4280\_P1 *SubmitFileOrDirectory*

Implement scanner for the provided lexical definitions.

The scanner is embedded and thus it will return one token every time it is called. Since the parser is not available yet, we will use a tester program to call the scanner.

The scanner could be implemented as

1. Plain string reader - read strings separated by spaces - (**70 points**) assuming
   * all tokens must be separated by spaces
   * lines may not be counted (**75 if counted**)
   * comments may be without spaces (will be tested without spaces in this option)
2. Module generated by lex (**75 points**)
3. FSA table + driver (**100 points**)

* You must have the README.txt file with your submission stating on the first line which option you are using: 1, 2, or 3, and if 3 then include information where the FSA table is and which function is the driver. If this information is missing, the project will be graded under option 1
* Implement a token as a triplet {tokenID, tokenInstance, line#} (if option with line numbers)
* Dont forget EOFtk token
* Implement the scanner in a separate file with basename "scanner"
* For testing purposes, the scanner will be tested using a testing driver implemented in file with basename "testScanner". You need to implement your own tester and include as a part of the project. This tester will ask the scanner for one token at a time and display the token to the screen, one per line, including information (descriptive) on what token class, what token instance, and what line, if applicable.
* Invocation:

   scanner [*file*]

to read from stdin or file *file*.sp2020

* + Arguments are the same as P0
  + Wrong invocations may not be graded
  + Dont confuse executable name with file name with function name
* Graded **20 points** for style regardless of implementation method
* You must have (C++ can be accordingly different)
  + types including token type in token.h
  + implement scanner in scanner.c and scanner.h
  + implement the tester in another file testScanner.c and testScanner.h
  + main.c processing the arguments (as in P0) then calling testScanner() function with interface and preparation as needed.

[Previous](https://umsl.instructure.com/courses/64310/modules/items/1043248)[Next](https://umsl.instructure.com/courses/64310/modules/items/1043251)

P1 Lexical Definitions

* Case sensitive
* Each scanner error should display "SCANNER ERROR:" followed by details including the line number if counted
* Alphabet
  + all English letters (upper and lower), digits, plus the extra characters as seen below, plus WS
  + No other characters allowed and they should generate errors
* Identifiers
  + begin with a  letter and
  + continue with any number of letters or digits
  + you may assume no identifier is longer than 8 characters (in testing)
* Keywords (reserved, suggested individual tokens)
  + begin end loop void var return in out program iffy then let data
* Operators and delimiters group (all single character except ==)
  + =  <  > == :   +  -  \*  /   %  . (  ) , { } ; [ ]
* Integers
  + any sequence of decimal digits, no sign, no decimal point
  + you may assume no number longer than 8 characters (in testing)
* Comments start with @ andEndWithWithoutSpaces@

P1 Suggestions

* Token is a triplet **{tokenID, tokenInstance, line#}** (if option with line numbers) .
  + **tokenID** can be enumeration (better) or symbolic constant (worse)
  + **tokenInstance** can be a string or can be some reference to a string table
  + the triplet can be struct
* Suggestions for the string reader option #1
  + Implement scanner as 'scanf("%s",data)' and then processing data as below
    - if starts with lower case letter then it is identifier but check against keywords
    - if starts with a digit then it is an integer token
    - then it must be operator or delimiter, you may use one group or look what it is and split various tokens
      * this could be easily done through an associative array
* Suggestions for the FA option
  + File can be opened and lookahead character can be set explicitly before the first call to the scanner for the first token
  + Have the scanner not read directly from the file but from a filter. The filter would count lines, skip over spaces and comments, construct string of characters for the current token, and return the column number in the table corresponding to the character
  + Represent the 2-d array for the FSA as array of integers
    - 0, 1, etc would be states/rows
    - -1, -2, etc could be different errors
    - 1001, 1002, etc could be final states recognizing different tokens
  + Recognize keywords as identifiers in the automaton, then do table lookup
* To print tokens I would suggest an array of strings describing the tokens, listed in the same order as the tokenID enumeration. For example:

enum tokenID {IDENT\_tk, NUM\_tk, KW\_tk, etc};  
 string tokenNames[] ={"Identifier", "Number", "Keyword", etc};  
 struct token { tokenID, string, int};

Then printing tokenNames[tokenID] will print the token description.

P1 Testing (update 3/30)

This section is non-exhaustive testing of P1

1. Create test files:
   1. **P1\_test1.sp2020** containing just one character (with standard \n at the end) :  
      X
   2. **P1\_test2.sp2020** containing a list of all the tokens listed, all separated by a space or new line.  
      x  xy xyz x1 x2   
      1 12 23 12345  
      begin end loop void var return in out program iffy then let data  
      =  <  > == :   +  -  \*  /   %  . (  ) , { } ; [ ]
   3. If WS not required, create another file where some token from above are combined w/o WS (as long as the token combination doesnt create a new token)  
      **P1\_test3.sp2020** containing a mix of tokens without spaces and with spaces.  
      x  x+2  x123>x1234  2%3  2 = = 3 2==3  
      // the last one is == token
   4. Test also with some extra comments, should not change the outputs  
      // add @comment@ to any file between any tokens
   5. **P1\_test4.sp2020** containing  
      2x // should be error but number followed by ID is fine  
      2 ^ x // ^ should throw an error
2. Run the invocations and check against predictions
   1. $ scanner P1\_test  
      // Program error file not found
   2. $ scanner < P1\_test1  
      // System error file not found, testing kb input redirection
   3. $ scanner < P1\_test1.sp2020  
      IDTk X 1  
      EOFTk
   4. $ scanner P1\_test1  
      // As above
   5. $ scanner P1\_test2  
      Should output all listed tokens, one per line, ending with EOFTk
   6. $ scanner P1\_test3  
      Should output the tokens you have in the file, splitting properly merged tokens
   7. $ scanner P1\_test4  
      SCANNER ERROR: ...