## P1: Scanner

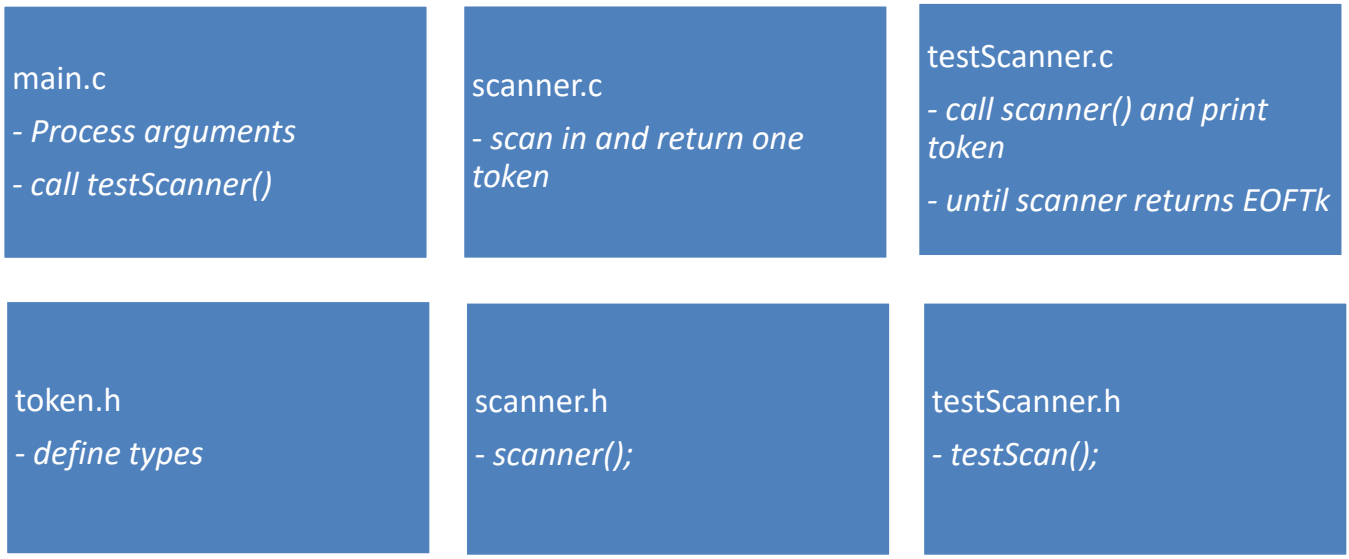
***Due Feb. 27 at 11:59pm***

- Implement scanner for the provided lexical definitions (see next page)
- The scanner is embedded and thus it will return one token every time it is called by a parser
  - Since the parser is not available yet, we will use a tester program
- 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.
  - The tester will ask 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.
- Invocation:

```
scanner [file]
```

  - As in Project P0, a filename may be specified or the information can be read from stdin
  - Wrong invocations may not be graded
  - ***Program must compile and run on clark.rnet.missouri.edu***

- Additional requirements:
  - Types, including token type, must be defined in token.h
  - You must implement scanner in scanner.c (.cpp) and scanner.h
  - You must implement the tester in another file testScanner.c (.cpp) and testScanner.h
  - Your main.c (.cpp) processes the arguments (as done for P0) then calls testScanner() function with interface and preparation as needed




- ***Important***: You have two options for your submission as follows

  1. Plain string reader - read strings separated by spaces - (70 points) assume:
     - all tokens must be separated by spaces and no token has invalid characters

        - lines may not be counted (+5 points if counted)
        - comments may be without spaces if it helps

    2. FSA table + driver (100 points)

- Note that the *maximum* number of points you can earn for option #1 is 75 (out of the possible 100), see rubric on last page.
- ***You must have a README.txt file with your submission stating on the first line which option you are using: #1 String Reader or #2 FSA Table***.
- If #1, then on the next line state if lines are counted and used in tokens.
- If #2, then on the next line state the location of your table in the code and whether lines are counted.

## Lexical Definitions

- All case sensitive
- Alphabet:
    - All upper- and lower-case English letters, digits, plus the extra characters as shown below, plus WS
    - *No other characters are allowed and they should generate errors*
    - Each scanner error must display "SCANNER ERROR:" followed by input string and line number
- Identifiers
    - begin with a ***lower-case*** letter (a-z) and
    - continue with ***one or more*** letters or digits (no underscores)
        - e.g. d3, aDD8, z920, bfa, are all valid and a, A3, 382, are all invalid identifiers
- Keywords
    - **`Again If Assign Move Show Flip Name Home Do Spot Place Here There`**
- Operators and delimiters group
    - **`& + / % . { } << <-`**
- Numbers
    - any sequence of decimal digits (0-9), no sign, no decimal point
- Comments start with ***** and end with *****

## P1 Suggestions

- Don't forget **EOFtk** token
- Implement token as a triplet {tokenID, tokenInstance, line#}
    - TokenID can be enumeration (better) or symbolic constant (worse) (see below)
    - tokenInstance can be a string or can be some reference to a string table
    - The triplet can be a struct
- Suggestions for the String Reader option #1:
    - Implement scanner as 'scanf("%s",data)' and then process data as below
    - If starts with a upper-case letter then it is a potential keyword, but check against keyword list
    - If starts with a digit then it is an number token
    - If starts with a valid character, but not a letter or digit, then it must be operator or delimiter, you may use one group or look what it is and split various tokens
        - This could be done through an associative array
- Suggestions for the FSA Table option #2:

- 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


Filter1:

- Track line number
- Map character to FSA Table column number
- Skip over comments

Filter2:

- If token is a candidate keyword, then check to see if it is in keyword list

- 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 candidate keywords in the automaton, then do table lookup

- Suggest an array of strings describing the tokens, listed in the same order as the tokenID enumeration, to print tokens. For example:

```
enum tokenID {IDENT_tk, NUM_tk, KW_tk, etc};
string tokenNames[] ={"Identifier", "Number", "Keyword", etc};
struct token {tokenID, string, int}; // string is comprised of the characters in the token, int is the line #
```

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

## Testing

This section is a sample subset of testing of P1

1. Create test files:
   1. P1_test1.txt containing just one character (with standard line return at the end):
      4

      Try changing the 4 to each valid and invalid character and retest each.
   2. P1_test2.txt containing a list with one or more of each of the token types, all separated by a space or new line:

x1 xA2 If jfa 301 var1 + &<
//etc

Try changing characters to make strings that are invalid and test each for correct error message.

3. If WS is not required, create another file where some token from above are combined w/o WS (as long as the token combination doesn't create a new token)
   P1_test3.txt containing a mix of tokens without spaces and with spaces:
   x3+  xA  If1<-+y1+z12
   //etc
4. Test also with some extra comments and/or blank lines, be sure these do not change the outputs

2. Run the invocations and check against predictions
   1. $ scanner P1_test

      Program error file not found

   2. $ scanner P1_test1.txt

      Number 4 1
      EOFTk

   3. $ scanner P1_test2

      Identifier x1 1
      Identifier xA2 1
      Keyword If 1
      Identifier jfa 1
      Number 301 1
      Identifier var1  1
      Operator + 1
      Operator & < 1
      Operator < 1
      EOFTk

   4. $ scanner P1_test3
      Should output the tokens you have in the file, properly splitting merged tokens

## Grading

- Programming and architectural style: 20 points
  - Includes naming conventions and organization (as described on first page)
  - README with required information (as described on second page)
- Uses FSA table + driver: 15 points
- Properly handles merged tokens (with no white space): 10 points

- Prints line numbers in output: 5 points
- Correct output (disregarding line number and merged token errors): 50 points
    - Includes recognizing keywords correctly
    - Includes proper printout of one token per line and token type, token value, and line number
    - Includes printing EOFTk
- Late submissions accepted for up to 3 days with a 10% penalty per day

*Programs missing makefile or that won't properly compile on clark.rnet.missouri.edu will not be graded.*