# P1: Scanner
## *Due March 11, 2018 at 11:59pm*

- Implement scanner for the provided lexical definitions (see below)
- 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
- The scanner should be implemented as FSA table + driver
- You must have a README.txt file with your submission stating where the FSA table is and which function is the driver.
- Implement a token as a triplet {tokenID, tokenInstance, line#}
- Don't 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.
    - 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]
```

    - Wrong invocations may not be graded
    - Don't confuse executable name with file name or function name
- You must have
    - types including token type in token.h
    - implement scanner in scanner.cpp and scanner.h
    - implement the tester in another file testScanner.cpp and testScanner.h
    - main.cpp processes the arguments (as P0) then calls testScanner() function with interface and preparation as needed
    - include timer.h and print time to screen

## Lexical Definitions

- All case sensitive
- Alphabet
    - all English letters (upper and lower), digits, plus the extra characters as shown below, plus WS
    - No other characters allowed and they should generate lexical errors
    - Each scanner error should display "Scanner Error:" followed by details including line number
- Identifiers

    - begin with a *lower case* letter and
    - continue with any number of letters or digits
    - you may assume no identifier is longer than 8 characters
- Keywords (reserved, suggested individual tokens)
    - `start end iter void var return read print program if then let`
- Operators and delimiters group.
    - `=  <  >  :   +  -  *  /   #  . (  ) , { } ; [ ]`
- Integers
    - any sequence of decimal digits, no sign
    - you may assume no number longer than 8 characters
- Comments start with ! and end with !

## P1 Suggestions

- Token is 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
    - 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}; // string is comprised of the characters in the token, int is the line#
```

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

## Testing

This section is non-exhaustive testing of P1

1. Create test files:

    1. P1_test1.txt containing just one character (with standard \n at the end) :
       x
    2. P1_test2.txt containing a list of all the tokens listed, all separated by a space or new line. For ids, use x, x1, and x12, also iter1, and var1; for numbers, use 1, 12, and 23.
       x x1 x12 iter iter1 var var1
       //etc
    3. 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.
       x x+ x1 x-x1 x+x1+x12
       //etc
    4. Test also with some extra comments and/or blank lines, should 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
       Identifier x 1
       EOFTk
    3. $ scanner P1_test2
       Should output all listed tokens, one per line, ending with EOFTk
    4. $ scanner P1_test3
       Should output the tokens you have in the file, splitting properly merged tokens

## Grading

- Programming and architectural style: 20%
    - Includes naming conventions and organization described on first page
- Compiles, runs, produces output to screen: 10%
- Prints line numbers in output: 5%
- Properly handles merged tokens (with no white space): 20%
- Correct output (disregarding line number and merged token errors): 45%
    - Includes recognizing keywords correctly
    - Includes printing EOFTk