Scanner P1

P1

- 100 pts
- Submission command:
  `/accounts/classes/janikowc/submitProject/submit_cs4280_P1` *`SubmitFileOrDirectory`*
- Implement scanner for the lexical definitions
- The scanner is embedded and thus it will return one token every time it is called by a parser-directed translator
    - Since the parser is not available yet, we will use a tester program, see below
- The scanner could be implemented as
    1. Plain string reader assuming (75 max)
        - all tokens must be separated by spaces
        - lines not counted
        - implemented with switches/ifs/etc
    2. Module generated by lex (80 max)
    3. FSA table + driver (100)
- You must have the README 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 table data is and which routine is the driver. If this information is missing, the project will be graded at 75%.
- Implement a token as a triplet {tokenID, tokenInstance, line#} (if option with line numbers) . TokenID can be enumeration or symbolic constant, tokenInstance can be a string or can be some reference to a symbol table. The triplet can be struct.
- Dont forget EOFtk token
- For testing purposes, this project is to keep displaying tokens one at a time. Every token should be displayed to the screen, one token per line, listing line# (if available), some name identifying the token (not just the enumerated value), and the instance string of the token if any.
- Invocation:
  `testScanner [`*`file`*`]`
  to scan from stdin or file *`file`*`.fs17`
    - this is the same as P0
    - wrong invocations may not be graded
- Graded 20% for style
- 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 P0) then calling testScanner() function with interface and preparation as needed.

Lex Def

- All case sensitive

- 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 lexical errors
    - Each scanner error should display "Scanner Error:" followed by details including line number of available.
- Identifiers
  - begin with a letter and
  - continue with any number of letters or digits
  - you may assume no identifier is longer than 8
- Keywords (reserved, suggested individual tokens)
  - `Begin End Check Loop Void Var Return Input Output Program`
- Operators, delimiters, etc.
  - `=  <  <=  >  >=  !=  ==  :  +  -  *  /  &  %  .  (  )  ,  {  }  ;  [  ]`
- Integers
  - any sequence of decimal digits, no sign
  - you may assume no number longer than 8 digits
- Comments start with # and end with #

Suggestions

- Suggestions for the FA option
  - Have the scanner read the file through a filter. The filter will
    - skip over #...#
    - count lines
    - construct the string
    - return column number instead of the actual character read
  - represent the DFA as 2-d array of type integer
    - 0, 1, etc are numbers corresponding to the row for the next state
    - -1, -2, etc are different error cases
    - 1000, 1001, etc are final states for different tokens (e.g. 1000 is for ID, etc)
  - do not distinguish keywords and IDs in the automaton, but when ID detected check if a keyword (to be a keyword, the ID must match one of the keywords exactly and completely not partially so Var is a keyword but Vars is not)
- Suggestions for the string reader option (all tokens separated by spaces). The scanner function would need to:
  1. scanf (%s, data) reading up to white space so next token or comment (or EOF if fails)
  2. If (data[0] == '#') start skipping until you grab the first data after thematching #
  3. if (isletter(data[0] and stringContainsLettersDigits(&data[1]) then ID so check against keywords
  4. if(alldigits(data)) then it is integer
  5. If (strlen of data) == 1 then check for
     - if data[0] == '+' then plusToken
     - if data[0] is letter then IdToken
     - etc
  6. Etc

Token is a structure containing tokenID (can be enumerated or symbolic constants), instance (string) and line number (int). It may be passed on the interface or through global token variable. If global, it should probably be defined in scanner.c and prototyped as extern in scanner.h.

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.

Testing

Test the keyboard vs. file input but redirecting the input as in P0.

To test tokens, list all possible tokens separated by spaces and make sure you get them all followed by EOFtk. IF using more complex scanner allowing no spaces, test some tokens without spaces such as >> or Begin()

Lex Errors

On any error, we print as detailed message as possible and exit processing.Lexical errors are 3 kinds:

1. Characters not in alphabet (not listed among those making any tokens)
2. Invalid token, for example 23a is invalid if tokens need to be separated by WS (if not needed separated then it would be integer followed by identifier and thus not an error)
3. Source file cannot be open

Testing Detail

This section is non-exhaustive testing of P1

1. Create test files:
    1. `P1_test1.fs17` containing just one character (with standard \n at the end) :
       x
    2. `P1_test2.fs17` containing a list of all the tokens listed, all separated by a space or new line. For ids, use x12, for numbers, use 123. Add some identifiers that start with keywords, for example `Loop1` which is identifier
       ```
       x12 Begin End
       Check Loop Loop1
       //etc
       ```
    3. If WS not required, create another file where some tones are combined w/o WS (as long as the token combination doesnt create a new token)
       `P1_test3.fs17` containing a mix of tokens without spaces and with spaces.
       `x12()` // 3 tokens: identifier ( )
       `123xyz` // 2 tokens: number identifier
       `Loop1` // 1 token
       `Loop()` // 3 tokens
       Etc
    4. Test also with some extra comments, should not change the outputs
2. Run the invocations and check against predictions
    1. `$ P1 < P1_test1`
       Error
    2. `$ P1 < P1_test1.fs17`
       `IDTk x 1` // plus token, instance is "+" , line number is line if counting lines
       `EOFTk`
    3. `$ P1 P1_test1`
       As above
    4. `$ P1 P1_test2`
       Should output all listed tokens, one per line, ending with `EOFTk`

5. `$ P1 P1_test3`
   Should output the tokens you have in the file, splitting properly merged tokens