P1 Due: Sun Oct 22, 2023, 11:59pm Scanner Project

Requirements

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 for testing it. Avoid generating a file with all tokens, or even a container with all tokens, just produce one token at a time upon a call.

The scanner could be implemented as

1. (70pt) Plain string reader or similar - read strings separated by spaces - assuming all tokens must be separated by spaces
2. (95pt) Tokens dont have to be separated by white spaces except when needed to prevent incorrect tokes
   * Eg. 5+2 doesnt need white spaces to figure this is integer followed by '+' followed by another number  
     (In option 1 above this would be an error because a token "5+5" doesnt match any definitions)
   * Eg. 5 2 which is number followed by number must have the white space or otherwise it would be just one number

In either case, additional 5pt for counting lines, for a total of 75 and 100.

You must have the **README.txt** file with your submission (in the submission directory) stating on the first line which option you are using: 1 or 2. If the information is missing, the project will be graded under option 1.

Implement a token as a triplet **{tokenID, tokenInstance, line#}** (or pair if not processing line numbers).

Dont forget **EOFtk** (end of file token) token at the end of processing

Implement the scanner in a separate file with basename "scanner"

For testing purposes, the scanner will be tested using a testing driver implemented in the the same scanner file or in a file with basename "testScanner". You need to implement your own tester and include as a part of the project. This tester will

* call the scanner, once for each single token, until it gets the EOFtk
* for each received token, it will display the token to the screen, one per line, including information (descriptive) on what token class, what token instance (if the token class includes more token instances) and what line (if applicable).

The scanner tester is just a function with just this code:

token=scanner(); // get one token  
while (true) {  
 //print descriptive token  followed by line number (if implemented) followed by token instance (string) if any then \n  
 if (token == EOFTk)  
 break;  
 token=scanner();  
}

Invocation

  P1 [***file***]

to read from stdin or file ***file***.f1 (the extension is implicit). Arguments are the same as P0. **Wrong invocations will not be graded.**

Grading Rubric

10 points for architecture and coding style regardless of implementation method  
5 points for stdin processing (or redirection)  
5 points for handling errors on invocation and in alphabet  
5 point for processing line numbers

The remaining points for properly recognizing/displaying tokens (50 option 1, 75 option 2)

Architecture

C or functional C++ must have (other languages appropriate differences)

* Types including token type in token.h
* Scanner in scanner.c and scanner.h
* Scanner tester testScanner() implemented in testScanner.c/testScanner.h  or with the above scanner file
* main.c processing the arguments then calling testScanner() function with interface and preparation as needed
  + main never calls the scanner itself except for potential preparation

P1 Lexical Definitions

* Case sensitive
* Each scanner error should display "LEXICAL 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
  + No other characters allowed and they should generate errors
* Identifiers
  + begin with a letter
  + continue with any number of letters, 8 significant total including the underscore
    - you may assume no identifier is longer than 8 characters (in testing, this is intended to make it easier not harder)
* Keywords (reserved, suggested individual tokens recognized as ID)  
  Note that the keywords start different from identifiers but the recommendation is not to find individual keywords in the automation (you can) but to recognize a generic "keyword" (start with letter and continue as long as letters) and then figure if indeed keyword (if not then error) and which keyword
  + start stop loop void var end scan print main cond then let func
* Operators and delimiters (single or double character, can produce individual tokens or as a group with an instance)
  + =  <=  >= > < ~ :  +  -  \*  /  %  . (  ) , { } ; [ ]  
    Note that <= >= +are double-char tokens w/o spaces between the characters
* Integers
  + any sequence of decimal digits, no sign, no decimal point, up to 8 significant
    - you may assume no number longer than 8 characters (in testing)
* Comments start with # and end with end with # and contain any characters (you may assume no white spaces inside as in #thisiscomment#)

P1 Implementation Suggestions

Representation

* 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

Option 1

* Suggestions for the string reader option #1 which assumes all tokens separated by spaces (string reader is the easiest approach to this option but other approaches can be taken)
  + Since scanf("%s",*data*) reads always up to a white space, each call to scanf() will return a complete token (or comment)
    - *data*can be C string or better C++ string
    - to determine which token, process the retrieved *data*element
      * if starts with letter then make sure it has the rest letters only, error otherwise, and it is an identifier but could be keyword
      * else if starts with a digit then it is an integer token (but check to make sure all are digits, error otherwise)
      * else if it starts with # then it is a comment (make sure ends with #)
      * else it must be operator/delimiter - inspect to check and classify
        + this could be easily done through an associative array
        + note some are single char and some are double char

Option 2

* This assumes implementing FSA (in principle could be done with deeply nested if/else but not advised)
  + File can be opened and lookahead character can be set explicitly if needed before the first call to the scanner before the first token
  + Have the scanner not read directly from the file but through a filter
    - When the scanner needs a character, it asks the filer for one char
    - The filter would count lines, skip over spaces and comments, append the character to the current token instance and return the column number in the table corresponding to the character
  + After designing FSA, represent the 2-d array representation for the FSA as array of integers
    - 0, 1, etc would be states (using rows if states counted from 0)
    - -1, -2, etc. could be different errors
    - 1001, 1002, etc could be final states recognizing different tokens
  + Merge keywords with IDs in the automaton, then do a table lookup if ID is detected
* Comments
  + Can be handled in the filter when the scanner reads from the source (skipping over)
  + Can be discarded out if the string starts with # (and ends with #)
  + Can be designed in the FSA but a bit more complex

To print tokens I would suggest an array of strings describing the tokens, listed in the same order as the tokenID enumeration (assuming enumerated from 0). 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.

### Architecture

As discussed in the notes on Lexical Analysis, a scanner module (our P1) can be implemented architecturally two different ways regardless of the option.

1. The scanner is invoked once and it reads the entire input and generates a file or a container of tokens. The next module reads tokens from the file or the container at which time the scanner is not invoked any more.
2. The scanner is invoked every time a token is needed by the next module and reads from the input file only as much as needed.

Below is again some explanation of the two architectural ways - assuming option 1 as illustration. In practice it should be the second way but for a small project either architectural organization would work out.

#### **Scanner Invoked Once**

The scanner reads the entire input and generates a file or container of tokens (with whatever is your representation of a token). As such the scanner is invoked once. I would suggest call it prescanner(). Then, whenever the next module needs a token, it will read one at a time from the file or the container. Call that one the scanner() to avoid confusion.

For option 1 the prescanner would then be like this

prescanner()   
string data  
container tokens such as vector  
open input file (or keybord)  
cin >> data from the file  
while (not end of file)  
 examine data and decide what token it is (or error), create the token  
 put the token on the container tokens  
  cin >> data from the file  
when done generate the eoftoken and put on the container

Then the actual scanner() function would be

scanner()  
  read the first element from the container  
  delete it from the container and return it

#### **Embedded Scanner Invoked Every Time a Token is Needed**

In this case prescanner is not needed. The scanner could be,

scanner()   
// assume input file is already open, must be opened prior to this call  
string data  
cin >> data from the open file  
if input failed upon eof then generate and return the eoftoken  
else example the data, decide which token it is (or error) and return the token

# **P1 Testing**

This section is non-exhaustive testing of P1

**1.** Create test files:

1. **P1\_test1.f1** containing just one character (with standard \n at the end) :  
   2
2. **P1\_test2.f1** containing a list of all the tokens listed, all separated by a space or new line.  
   x  xy xyz hello   
   1 12 23 12345  
   =  <=  >= > < ~  :   +  -  \*  /  %  . (  ) , { } ; [ ]  
   start stop loop void var end scan print main cond then let func
3. If WS not required in your approach, create another file where some tokens from above are combined w/o WS (as long as the token combination doesnt create a new token)  
   **P1\_test3.f1** containing a mix of tokens without spaces and with spaces.  
   x  x+22  xyz=123  2%3  2>3 2>=3 2~3  -(2-3)  
   Note the last three will recognize two numbers separated by respectively > >~ and ~
4. Test also with some added comments, should not change the outputs  
   Add #comment# to any file between any tokens, should not change the output
5. **Some other tests cases**  
   2var  
   //if you dont require spaces, could be error or integer followed by keyword either is fine  
   2 ^ x  
   //^ should throw an error, not in the alphabet

**2.** Run the invocations and check against predictions

1. $ P1 P1\_test  
   // Program error file not found
2. $ P1 < P1\_test1  
   // System error file not found, testing kb input redirection
3. $ P1 < P1\_test1.f1  
   NumTk 2 1  
   EOFTk
4. $ P1 P1\_test1  
   // as above
5. $ P1 P1\_test2  
   //Should output all listed tokens, one per line, ending with EOFTk
6. $ P1 P1\_test3  
   //if not requiring WS should output the tokens you have in the file, splitting properly tokens
7. $ P1 P1\_test4  
   LEXICAL ERROR: [some proper message]