# Mechanics of Programming Lexical Analysis

# CSCI243 Project 1

9/28/2014

## 1 Revisions

9/28/2014 - First version.

## 2 Due Date

Sunday, October 12, by 11:59:59pm.

## 3 Overview

Language processing programs (such as compilers, interpreters, and assemblers) face a number of challenging tasks. They must examine the original source code being processed, must break it up into its component pieces (called *lexical analysis*), verify that those pieces form legal statements in the language (called *syntax analysis*), and then generate an equivalent version of the program that can be executed directly by the computer (called *code generation*).

Lexical analysis is typically performed by a program called a *scanner*. The scanner reads the input character by character, determining what type of element each represents, and converting that information into a form that is easier for the rest of the language processing software to deal with. Each of these converted items is called a *token*; using tokens makes it easier for the syntax analyzer (called a *parser*) to examine the input elements and verify their legality.

For this assignment, you will implement a scanner for a small subset of the C language: identifiers, decimal integers, octal integers, block-style comments, and basic arithmetic operators. Scanners can be implemented in many different ways. For this assignment, you will implement your scanner using a data structure known as a *transition matrix*.

### 3.1 Token Description

In order to separate the input into tokens, a scanner must have descriptions of the tokens. For the subset of C you will be processing, tokens have the following descriptions:

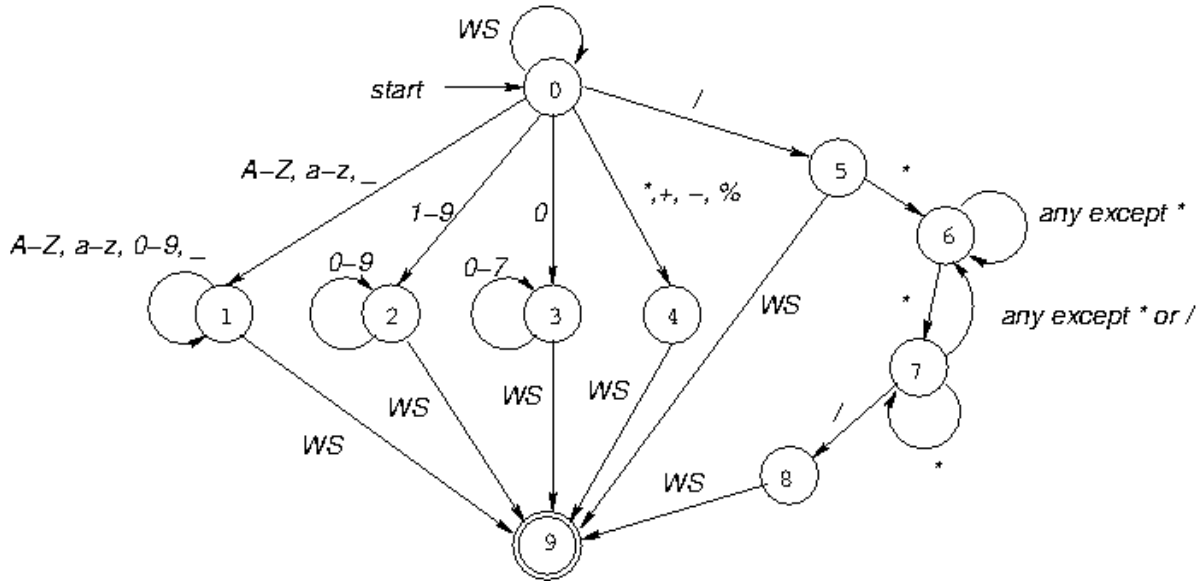
| *Token* | *Specification* |
|---|---|
| identifier | alphabetic or underscore character, followed by zero or more alphanumeric or underscore characters |
| decimal integer | digit 1 through 9, followed by zero or more digits 0 through 9 |
| octal integer | digit 0, followed by zero or more digit characters 0 through 7 |
| block-style comment | `/*` followed by any number of characters and `*/` |
| arithmetic operator | `+`, `-`, `*`, `/`, and `%` characters |

### 3.2 Describing the Operation of the Scanner

One way of representing the actions of a scanner when processing tokens such as these is a *finite-state machine* (FSM) consisting of a set of *states*, a set of *transitions* between those states, and a set of *actions* to take when particular transitions are followed. An FSM is typically represented graphically as a *state diagram*, in which each FSM state is a graph node and each transition between states is a directed path from one node to another. Each path is commonly labeled with the event that causes the transition. The originating state for such a transition is often called the *current* state; the destination state is similarly called the *next* state.

In a state diagram, each state represents a step along the path from an initial (*start*) state to a final (*accepting*) state. Arrival at the accepting state indicates that a particular type of token has been *recognized* and will be returned to the calling program. Typically, accepting states are indicated in a state diagram as nodes which have a different appearance from the other nodes.

In the case of an FSM for a scanner, the labels on the paths are typically the character(s) which cause that transition to be taken. Here is an example state diagram for an FSM that recognizes the tokens described above (the label *WS* on a path indicates that any whitespace character - typically, space, tab, and newline - causes this transition):



In this example state diagram, there are ten states numbered 0 through 9. State 9 is represented as a node having two circles; this is the accepting state for the diagram. Transitions from one state to another occur based on characters that have been read into the scanner.

The start state for this diagram is state #0; every time the scanner is invoked, it enters this state and begins reading characters one at a time. For each character, the FSM will follow a transition arc determined by that character. In this example, when the scanner is in state #0, whitespace characters (typically space, tab, and newline) keep the scanner in that state, alphabetic or underscore characters cause a transition to state #1, the digit character '`0`' causes a transition to state #3, and so on.

Transitions in a FSM will typically have actions associated with them, such as "save this character", "convert this integer from character form to integer form", "look up this identifier in the symbol table", and so on. These are often added to the label on the arcs representing transitions in the state diagram; for instance, the transition from state #0 to state #1 might be labeled "`A-Z,a-z,_/save`", to indicate that the `save` operation should be performed on the character that caused this transition.

For simplicity, the state diagram shown above doesn't indicate any actions; presumably, there would be additional information provided with the diagram to indicate what actions should be performed with each transition. An example of such additional information might be "for the transition from state #0 to state #0 and all transitions into state #9, discard the character that caused the transition; for all other transitions, save the character in a buffer for later processing".

### 3.3 Representing the State Diagram

The graph describing the states and transitions in our FSM is an abstract representation. In order to use it within a program implementing the scanner, we must use a more concrete representation. One such representation is a *transition matrix* (TM).

As the name implies, a TM is a two-dimensional array that contains information describing the transitions made by our scanner. Each row in the matrix represents a state in the FSM, while each column represents a class of input characters (alphabetics, whitespace, etc.). Each cell (the intersection of a state with a character class) represents the transition taken from that state when an element of that character class is read in. For example, here is a simple TM representing the FSM shown above:

| State | Character Class | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | WS | A-Z, a-z, _ | 0 | 1-7 | 8, 9 | / | * | +, -, % |
| 0 | 0 | 1 | 3 | 2 | 2 | 5 | 4 | 4 |
| 1 | 9 | 1 | 1 | 1 | 1 | - | - | - |
| 2 | 9 | - | 2 | 2 | 2 | - | - | - |
| 3 | 9 | - | 3 | 3 | - | - | - | - |
| 4 | 9 | - | - | - | - | - | - | - |
| 5 | 9 | - | - | - | - | - | 6 | - |
| 6 | 6 | 6 | 6 | 6 | 6 | 6 | 7 | 6 |
| 7 | 6 | 6 | 6 | 6 | 6 | 8 | 7 | 8 |
| 8 | 9 | - | - | - | - | - | - | - |
| 9 | - | - | - | - | - | - | - | - |

Several entries in this matrix are indicated as "`-`", which means that the state diagram contains no transition from this state when a member of that character class is encountered. Typically, these state/character combinations represent error conditions, such as encountering an alphabetic character as part of a decimal integer.

In order to simplify processing of the TM, we often designate a special state as an error handling state. In this state, we collect input until we are able to "resynchronize" the FSM (for instance, by reading characters until a specific type of character has been read in, such as a whitespace character or a semicolon character). This state may have an explicit number, or may not. Commonly, any unspecified transitions will be treated as transitions to this error state.

Another item of note in this matrix is that some characters which are similar (such as the digit characters) are split into separate classes. This must be done any time some characters from a set must be treated differently in some situations. In the case of the digit characters, an initial '`0`' indicates an octal integer, so it must be separated from the other digits; similarly, octal integers cannot contain the characters '`8`' and '`9`', so those must be separated. (This is also the reason for separating '`/`' and '`*`' from the rest of the arithmetic operators.)

We also typically want to represent the actions associated with transitions in the FSM. This can be shown in many ways; in a description such as the one above, we might indicate a transition to state *nn* with an action of *act* as "*nn*/*act*".

Here is a modified version of the TM from above that handles errors through transitions to a new state #99 and indicates whether the scanner should perform "save" (`s`, record the character in a buffer) or "discard" (`d`, not record the character) actions for each transition.

| State | Character Class | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | `WS` | `A-Z, a-z, _` | `0` | `1-7` | `8, 9` | `/` | `*` | `+, -, %` |
| 0 | 0/d | 1/s | 3/s | 2/s | 2/s | 5/s | 4/s | 4/s |
| 1 | 9/d | 1/s | 1/s | 1/s | 1/s | 99/d | 99/d | 99/d |
| 2 | 9/d | 99/d | 2/s | 2/s | 2/s | 99/d | 99/d | 99/d |
| 3 | 9/d | 99/d | 3/s | 3/s | 99/d | 99/d | 99/d | 99/d |
| 4 | 9/d | 99/d | 99/d | 99/d | 99/d | 99/d | 99/d | 99/d |
| 5 | 9/d | 99/d | 99/d | 99/d | 99/d | 99/d | 6/s | 99/d |
| 6 | 6/s | 6/s | 6/s | 6/s | 6/s | 6/s | 7/s | 6/s |
| 7 | 6/s | 6/s | 6/s | 6/s | 6/s | 8/s | 7/s | 8/s |
| 8 | 9/d | 99/d | 99/d | 99/d | 99/d | 99/d | 99/d | 99/d |
| 9 | 0/d | 0/d | 0/d | 0/d | 0/d | 0/d | 0/d | 0/d |

Note, though, that even this TM is not complete, as we have not indicated what to do with unexpected input (e.g., other characters, input that isn't a valid character) or reaching end-of-file while processing tokens.

## 4 Assignment

For this assignment, you will write a program named `tokenize` which uses a scanner implemented using a transition matrix (as described above). However, the transition matrix your scanner uses is not fixed; instead, you will read it in from a file whose name is provided on the command line, and use it to recognize tokens in characters read from the standard input. All input to your program will be ASCII[1] characters.

Character classes will be identified by number, as follows (names in the "Class Name" column are C preprocessor macro names defined in a header file we are supplying to you):

| # | *Class Name* | *What this Class Represents* |
|---|---|---|
| 0 | `CC_WS` | whitespace characters (space and tab) |
| 1 | `CC_NEWLINE` | the newline character (usually treated as a whitespace character) |
| 2 | `CC_ALPHA` | alphabetic characters and the underscore character |
| 3 | `CC_DIG_0` | the digit character 0 |
| 4 | `CC_DIG_1_7` | the digit characters 1 through 7 |
| 5 | `CC_DIG_8_9` | the digit characters 8 and 9 |
| 6 | `CC_CHAR_SLASH` | the division operator |
| 7 | `CC_CHAR_STAR` | the multiplication operator |
| 8 | `CC_ARITH_OP` | other arithmetic operators ( `+`, `-`, `%`) |
| 9 | `CC_OTHER` | any other ASCII character |
| 10 | `CC_EOF` | end-of-file on the standard input (often treated as whitespace) |
| 11 | `CC_ERROR` | anything that isn't an ASCII character |

1. American Standard Code for Information Interchange, a standard encoding scheme for representing character information, used on many computer systems.

(You can find information about the ASCII character set in many programming textbooks, through web searches, or by executing the command "`man ascii`" on the CS Ubuntu®[2] computer systems.)

### 4.1 Supplied Files

We are providing some files for your use when building and testing your scanner. Retrieve them using this command:

```
get  csci243  project1
```

This will create the following files in your working directory:

- `header.mak` - this can be used with the `gmakemake` program on the CS computer systems to create a `Makefile` to simplify management of your program.
- `classes.h` - This header file defines macros for all the character classes your scanner will process.
- `tm.1` - An example transition matrix source file.
- `input.1` - A file containing sample input to your program, for use with the `tm.1` TM file.
- `output.1` - A file containing example output from your program when it is given the `tm.1` and `input.1` files as input data.

*Do not modify the* `classes.h` *file!!!!* You will need it to compile your program, but you will not be submitting it to `try`.

You can test your program by running commands such as these:

```
$ ./tokenize tm.1 < input.1 > myoutput
$ diff myoutput output.1
```

The output from the `diff` command will indicate places where your output doesn't match the correct output.

### 4.2 Program Operation

Your program will be invoked this way:

```
./tokenize  tmfile
```

This invokes your scanner with the name of a transition matrix description file (*tmfile*) as its only command-line argument. Your scanner will open this file, process its contents, and close the file; after this, it will read characters from the standard input, accepting or rejecting tokens until all input has been processed (i.e., an attempt to read another character returns an end-of-file indicator). A more complete description of this process is given below.

The basic operation of your programs is:

- verify the command-line argument

2. Ubuntu is a registered trademark of Canonical Ltd.

- read the indicated file and build the transition matrix
- process characters from the standard input

See below for more details on these steps.

### 4.2.1 Format of the transition matrix description file

A transition matrix description file begins with three lines of text that describe the basic characteristics of the matrix, in this order:

```
states  Ns
start  S
accept  A
```

where *Ns* is a positive integer representing the number of states in the matrix, *S* is the number of the start state, and *A* is the number of the accepting state. Following these three lines will be a series of zero or more lines which describe the transitions and actions for individual states. Each of these lines begins with a state number, which is followed by one or more transition specifiers having the format "*c*/*kx*", where *c* is the number of the character class causing this transition, *k* is the state to which the transition goes, and *x* is an action to be taken when this transition occurs.

You are guaranteed the following about the contents of this file:

- all fields on all lines of the matrix description file will have one or more whitespace characters between them;
- the first three lines (states, start, accept) will always be present and in the order shown above;
- there will be at least one state in the matrix, and the number given on the `states` line will be a non-negative integer;
- there will be no more than 75 states in the matrix
- every state number in the file will be valid (e.g., if there are 10 states, valid state numbers are in the range 0..9); this includes:
  - state numbers for `start` and `accept` lines
  - state numbers at the beginning of transition lines
  - next state numbers (*k*) in transition specifiers
- all other fields in transition specifiers will be valid:
  - every character class number (*c*) will be in the range 0 through 11
  - each character *x* will be either `s` to save the character causing the transition, or `d` to discard the character - no other actions will occur.

Not every state will have a transition line in the TM file, not every transition for a given state will be specified, transitions for a state may appear in any order, and transition lines may occur in any order. Any transition that is not specified should be treated as a transition to the error state, with an action of `d` (discard the character that caused the transition).

Here is an example legal TM description file:

```
states 10
start  0
accept 9
2  10/9d  1/9d  4/2s  3/2s  5/2s  0/9d
3  0/9d  1/9d  10/d 3/3s  4/3s  5/9d
0  0/0d  1/0d  2/1s  3/3s  4/2s  5/2s  6/5s  7/4s  8/4s  10/9d
4  0/9d  1/9d  10/9d
5  0/9d  1/9d  7/6s  10/9d
1  0/9d  1/9d  2/1s  3/1s  4/1s  5/1s  10/9d
6  0/6s  1/6s  2/6s  3/6s  4/6s  5/6s  6/6s  7/7s  8/6s  9/6s  10/9d
7  0/6s  1/6s  2/6s  3/6s  4/6s  5/6s  6/8s  7/7s  8/6s  9/6s  10/9d
8  0/9d  1/9d  10/9d
```

In this example, we do have transition lines for each state, but they are not in order by state number, and on some of them the transition specifiers are not ordered by character class number. Some transitions are not specified; e.g., for state #4, transitions for character classes 0, 1, and 10 are specified, but all others would default to going to the error state with an action of "discard".

As in the example transition matrices shown above, use state #99 for your error state.

#### 4.2.2 Verification of the command-line argument

Your program may be invoked without the name of a TM file on the command line. If this happens, you should print this usage message to the standard error output:

```
usage: ./tokenize tm_file
```

and the program should exit.

#### 4.2.3 Reading and building the TM

To open the TM file, use the `fopen()` function from the standard i/o library. Assuming that `argv[1]` is the command-line argument, the following code will open the file and verify that the open succeeded; if the open fails, the file name and an explanation of the failure will be printed to the standard error output, and the program will exit:

```
FILE *fp;

fp = fopen( argv[1], "r" );
if( fp == NULL ) {
    // something went wrong
    perror( argv[1] );
    exit( 1 );
}
```

The variable `fp` can then be used in calls to input functions when you want to read from the file; for instance,

```
char *ptr, buf[256];

while( (ptr = fgets(buf, 256, fp)) != NULL ) {
    // process this line from the file
}
```

Parsing the lines in the matrix file can be done in several ways, but we recommend using the string library to assist you. One function in particular will be helpful here:

```
char *strtok( char *str, const char *delim );
```

This function "tokenizes" the string supplied as its first parameter, using the characters in its second parameter as delimiters between tokens. Each call to the function returns one token from the string. There are many examples of the use of `strtok()` available online; see the Ubuntu manual page with "`man strtok`", or see `www.cplusplus.com` for a simple example.

Parsing the first three lines is relatively easy, as they each contain a single word (which tells you the parameter being specified) followed by an integer (which can be converted using your own integer conversion routine, or using a C library function). Once you know the number of states in the matrix, you know the dimensions of the matrix data structure (one row for each state, and one column for each of the 12 character classes).

Parsing the transition lines is a bit more involved: you do not know how many of them there will be; you do not know the order in which they will appear; and each line will contain anywhere from one to twelve transition specifiers. Fortunately, you do know that the first thing on each line is the number of the state to which this line applies; once you have converted that, you have the row index into the matrix. Each transition specifier has the same format (described above); these can be processed easily using the `sscanf()` function from the stdio libary.

Once you have built the transition matrix, you should write its contents to the standard output. Print the two header lines

```
Scanning.using.the.following.matrix:
.....0....1....2....3....4....5....6....7....8....9...10...11
```

(where the character "`.`" represents a single space character) followed by one line for each state, using the following generic format:

```
nn..kkx..kkx..kkx..kkx..kkx..kkx..kkx..kkx..kkx..kkx..kkx..kkx
```

where *nn* is the state number (printed in a two-digit field), each *k* is the state to which the scanner moves when it encounters this character class (also printed in a two-digit field), and each *c* is a single character `s` or `d` indicating the action taken during this transition. As the "`.`" characters indicate, fields on these lines are separated by two space characters.

Here is example output for the matrix described earlier in this section:

```
Scanning using the following matrix:
     0    1    2    3    4    5    6    7    8    9   10   11
 0   0d   0d   1s   3s   2s   2s   5s   4s   4s  99d   9d  99d
 1   9d   9d   1s   1s   1s   1s  99d  99d  99d  99d   9d  99d
 2   9d   9d  99d   2s   2s   2s  99d  99d  99d  99d   9d  99d
 3   9d   9d  99d   3s   3s   9d  99d  99d  99d  99d   9d  99d
 4   9d   9d  99d  99d  99d  99d  99d  99d  99d  99d   9d  99d
 5   9d   9d  99d  99d  99d  99d  99d   6s  99d  99d   9d  99d
 6   6s   6s   6s   6s   6s   6s   6s   7s   6s   6s   9d  99d
 7   6s   6s   6s   6s   6s   6s   8s   7s   6s   6s   9d  99d
 8   9d   9d  99d  99d  99d  99d  99d  99d  99d  99d   9d  99d
 9  99d  99d  99d  99d  99d  99d  99d  99d  99d  99d   9d  99d
```

You may represent the nodes in your matrix in any way that seems appropriate. For each combination of current state and input character class, you will need to keep track of the next state and the action to be taken when that transition occurs; this suggests that a two-dimensional array of a `struct` might be a useful representation to consider.

#### 4.2.4 Processing characters from the standard input

The second form of input your program must process is a series of characters read from the standard input. Your program will read these characters and use the transition matrix to recognize tokens in the input.

While you are allowed (and encouraged) to use library parsing and conversion functions to process the transition matrix description file, you are not allowed to use them to process the standard input. Characters must be read one at a time from the standard input using `getchar()` or an equivalent operation.

As your program recognizes characters that comprise a token, those characters should be saved in a buffer (as indicated by the `s` transition character). Once the token has been recognized, those characters will be printed out (see below).

When you begin looking for a token, your scanner will enter the initial state specified by the matrix description file (which is not necessarily state #0!). As you read characters one by one, each character should be classified (whitespace, alphabetic, etc.). Using the current state as the row index and the character class as the column index into your matrix, your scanner will perform whatever action is given for that combination, and will transition into whatever next state is specified.

As your scanner processes the input, it should print out the sequence of state transitions it makes (beginning with the start state); when it reaches the accepting state, it stops reading characters and returns its success/failure indicator to the rest of the program. If the scanner moves into the error state, stop printing state transitions, and read (and discard) characters until a whitespace character has been read. After the scanner returns the success/failure indication to the calling routine, that routine should indicate this in the output (as described below). All output produced for a single call to the scanner (both

output from the scanner and output from the calling routine) should appear on one line, with output for separate calls occurring on separate lines.

When a token has been recognized by the scanner, print "`recognized 'token'`" after the sequence of state transitions, where *token* is the sequence of characters that were matched. If the scanner rejects the input, print "`rejected`" and if EOF was reached before any characters were collected for the token (e.g., if the last few characters in the input were whitespace), print "`EOF`".

Consider this input data given to your program (where ★ represents a single space character, ➞ a newline character, and ✖ the end of input):

```
abc★123★44xyz★/*★hello★*/➞✖
```

Assuming that the transition matrix matches the state diagram shown above, your program would produce the following output during scanning:

```
0 1 1 1 9 recognized 'abc'
0 2 2 2 9 recognized '123'
0 2 2 99 rejected
0 5 6 6 6 6 6 6 6 7 8 9 recognized '/* hello */'
0 9 EOF
```

Remember that EOF is a recognized character class (`CC_EOF`); this is why the final line of output shows a transition from 0 to 9 before the EOF message is printed.

### 4.3 Program structure

Conceptually, a scanner is a supporting function within a larger program. The program will call the scanner when it needs to examine the next input token; the scanner processes input characters until it either recognizes a token or determines that the input doesn't form a valid token, at which time it returns information to the calling routine about its findings (such as "I recognized an [X] token" or "the input wasn't a valid token" or "there were no more input characters").

A minimal modular design for this program might include at least the following functions:

- a `main()` function, which repeatedly calls the scanner to obtain tokens;
- a matrix creation function, which handles processing of the transition matrix description file;
- a classification function, which takes an input character parameter and returns a character class; and
- a scanner function, which performs the steps necessary to read and recognize one token.

You will need to create and submit at least one C source file for this assignment:

- `tokenize.c` - This is where you implement your `main()` routine. You may also implement other "helper" functions here, although a better organization would be to place each module in separate source files.

You may create any number of additional C source and header files, which should be submitted along with your `tokenize.c` file.

## 5 Submission

Remember that your `main()` function must be in a file named `tokenize.c`. You may use any names you wish for other `.c` and/or `.h` files you submit.

Submit your code using the following command:

```
try grd-243 project1-1 tokenize.c other-files
```

where *other-files* are any additional `.c` and/or `.h` files you are submitting. You may also submit the following files if you wish:

- if you wish to communicate additional information about your solution to your instructor or grader, a plain-text file named `README` or `README.txt` (do *not* submit other formats!)
- if you are using a revision control system which doesn't put revision log information into the files you are submitting, a plain-text file named `revisions.txt` which contains your log information

Remember that the `classes.h` file should not be submitted (if you do submit it, `try` will reject your submission).

## 6 Grading

Your program will be graded based on the following criteria:

- (40%) TM input handling: This includes opening the file, reading the TM data, building the TM, closing the file, and correctly handling any errors that might occur (e.g., no file was specified, the specified file couldn't be opened, etc.).
- (40%) Token processing: Correct processing of the standard input, including correctly recognizing valid tokens and rejecting invalid tokens.
- (10%) Programming Style: Are you following a consistent and reasonable coding style? Is your code reasonably structured?
- (10%) Documentation: Is your code properly commented and documented? Please check with your individual instructor regarding any additional requirements (e.g. version control).

A program that cannot be compiled and linked by `try` will not be accepted, and will receive a grade of '0'.