

- Here is the change log for this assignment write-up. I will try to be descriptive in my log messages.
- You can also subscribe to this page and receive Emails when changes are made.

In this assignment will you will construct the high level logic for a token scanner (aka a "lexer") and demonstrate its correctness. The name of your application must be `LUTHER`¹ (case sensitive).

Input

`LUTHER` will accept two command line arguments and produce a single output file. Since results are not going to `stdout`, **we won't worry about the usual `OUTPUT` project requirements.**

Argument	Value
1	Path to the scanning definition file
2	Path to a file of source to be tokenized by <code>LUTHER</code>
3	Path to an output file for storing the tokenized source

`LUTHER` should:

1. Read and store the data from the scanning definition file
2. Read the source file, character by character, and detect all tokens found; storing their details to the output file.

Scan Definition File

A scan definition file is line oriented, all fields are white space delimited.

The first line contains all the values of all ASCII characters permitted in the language, these are written as white space delimited strings.

Characters **NOT** in the sets `0-9`, `A-Z`, `a-w`, or `y-z` (notice the missing `x`) are **always** written in a hexadecimal escape sequence: `xHH` where `H` is an upper or lower case hexadecimal value.

Other characters may be written in escaped form (`xHH`) or as their visual glyph (for instance `A` for ASCII decimal code 65).

We will call this encoding "Alphabet Encoding."

The characters are in ascending ASCII order, but **there may be gaps** in the sequence, since not all ASCII characters are used for all languages.

The second and subsequent lines have two or three fields:

Field	Value
1	Path to a transition table file
2	The token id associated with the transition table file
3 (optional)	The <i>data</i> or <i>value</i> associated with the token id, written as a single string in alphabet encoding

If the optional field 3 is **not** provided, then the data or value associated with the token id is the sequence of source characters matched by the transition table of field 1.

Transition table files were produced by the NFAMATCH project, in NFAMATCH you *generated* them, `LUTHER` will need to *read them*. All the transition table files will have the appropriate number of columns for the alphabet provided on the first line of the scan definition file.

Your code is **not expected** to detect flaws or inconsistencies in the scanner definition file. Your code should abort with an exit status of 1 if a definition file cannot be opened or is devoid of data (all lines are empty lines). Otherwise you may assume the definition file is good to go.

Example Scanner Definition File

The following is a simple scanner definition file (clearly inadequate for a real programming language), but is a useful example to look at.

```
x0ax20x5C x6fpqrx73
```

```
wiki/noto.tt      pqrs
wiki/nots.tt      opqr
wiki/endsq.tt     endsq
wiki/twosmallwords.tt twosmallwords
wiki/whackamole.tt whack      x5c000x5cx20x5c000x5c
wiki/anyone.tt     IGNORE
```

This file defines an alphabet of the newline character (hex 0a), space character (hex 20), backslash ("whack") and o, p, q, r and s. There are six transition tables and token ids defined in the file. The `whack` token id has the optional third field that specifies alternative constant data for the token id (`\000\ \000\`). Each of the transition tables will have:

1. 8 transition columns, the first for the newline character, the second for the space character, the last for s
2. A starting state of 0
3. At least one accepting state
4. *Sequential and increasing state ids*, so line 3 of `twosmallwords.tt` holds state id 2
5. A state **without** a transition for a language character will have an E in the appropriate character column

Transition tables will **not** match an empty string.

You will want to store the transition table data in a data structure that permits easy "transition following" and inspection of the match status for the token type.

Coding

The fundamental task of a token scanner is to find sequential non-overlapping tokens in an input stream of characters, the following rules are critical:

1. A scanner reports tokens that are the **longest possible** match, regardless of which token id the match represents.
2. In the event that a sequence of input characters are matched by two or more transition tables (so none of the matches represent the longest token), the token id **that occurs first** in the scanner definition file "wins" the match and is reported.

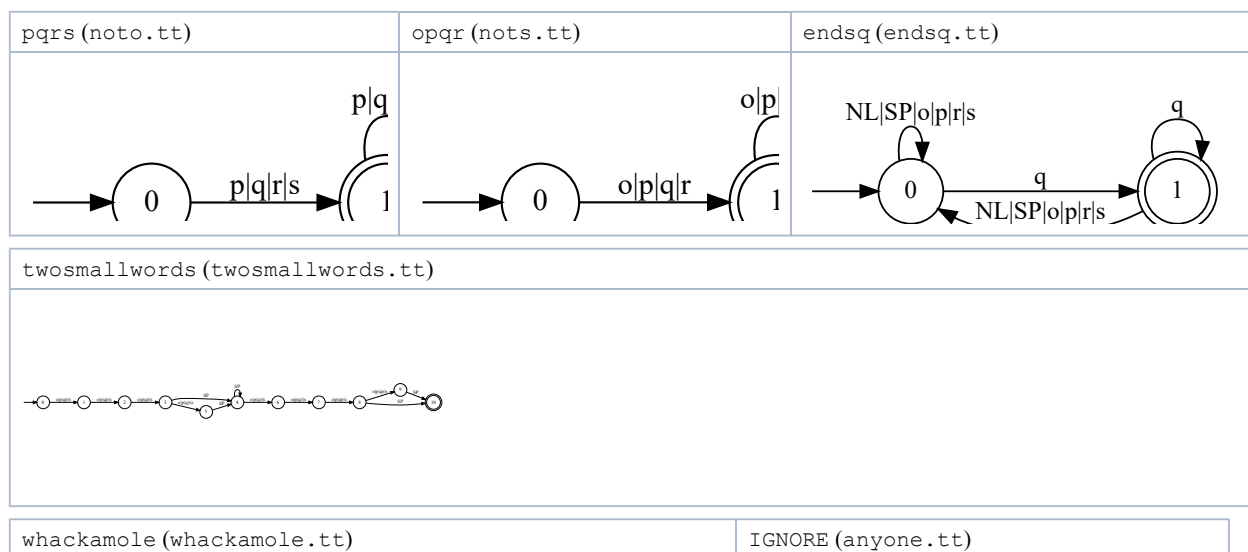
Output

LUTHER will report matched tokens to the file specified by the third command line argument. Four white space separated fields will be reported for each matched token id, and a line will contain the information for only one token id. The data to be written are:

1. The token id
2. The token value **using alphabet encoding**
3. The line number where the first character of the match occurs, reported as a decimal value
4. The character number where the first character of the match occurs, reported as a decimal value

Example(s)

Shown here are the DFAs stored in the `.tt` files of the **scanner definition file** example above (NL for the newline character, SP for space):





If a source file for this scanner is named `program.src` and LUTHER is invoked as follows, `tokens.dat` should be produced.

```
$ cat wiki/program.src
pqrpppppppppprrrr
rop rop rop \ rop \
p
q r s r
q
p q \pos rosp\ r
$ ./LUTHER wiki/scan.u wiki/program.src tokens.dat
:
$ cat tokens.dat
pqrs pqrpppppppppprrrr 1 1
IGNORE x0a 1 19
twosmallwords ropx20ropx20 2 1
twosmallwords ropx20x20x20ropx20 2 9
whack x5cooox5cx20x5cooox5c 2 19
endsq x20x0apx0aqx20rx20sx20rx0aqx0apx20q 2 31
IGNORE x20 6 4
whack x5cooox5cx20x5cooox5c 6 5
IGNORE x20 6 15
pqrs r 6 16
IGNORE x0a 6 17
```

Let's explain this output. This would be a good time to do some split screening or dual-windowing so that you can see the DFAs, `program.src` and `tokens.dat` in one visual panorama without a lot of scrolling. You can download these files as well if you prefer working with pencil and printed paper copies. Here is a slideshow that we'll go through in lecture.

First, some observations of the DFAs:

- The IGNORE DFA matches any single character, this prevents us from running into characters that *that aren't matched by any DFA*. As such IGNORE tokens are special cases and **we'll mostly ignore their behavior for the purposes of analysis**.
- The only DFA that matches across newline characters is `endsq`; which means all other DFAs must find their match on a single line. However, `endsq` cannot match across a whack (`\`).
- The only DFAs that permit space characters are `endsq`, `twolittlewords` and `whackamole`.

It would be good to work through some of these match results on your own, here are some **highlights to be sure you understand**:

- The first match found is of the entire first line (well, less the newline character). Both token ids `pqrs` and `opqr` match the entire line; the token is assigned the `pqrs` id because `pqrs` **comes before** `opqr` in the definition file `scan.u`.
- When (or at what character) was the decision made to make `pqrpppppppppprrrr` a token? Was it at the last `r` of the line? Was it at the newline terminating the first line? Was it at the first `o` of the second line? Or the first space character on the second line? The answer is **none of these** 😊 Here is a table showing where the matches, *starting with the first character of the first line*, fail for the various DFAs. The table also shows the last character of their longest previous match:

token id	line, character number	longest previous match
whack	1,1	none
endsq	2,19	1,12
twosmallwords	1,5	none
pqrs and opqr	1,19	1,18

The `endsq` DFA was actively matching up to the whack (`\`) on line two. **It wasn't until `\` was processed** that all the DFAs were in failed states and a "winning" token id could be chosen. Notice that although `endsq` had a failed DFA, it had previously seen an accepting match at character 12 (the last `q`) of the first line.

Token ids are not assigned based on which DFAs *did not fail*, because they *must all fail* before a decision is made! Token ids are assigned based on **match length (and definition file ordering in the case of match length ties)**.

- At which character positions in `program.src` were the next three tokenizing decisions made? We hope you are **not** tempted to say:
 - The newline at the end of the first line (IGNORE)
 - The space after the second `rop` of the second line (`twolittlewords`)
 - And the space after the fourth `rop` of the second line (another `twolittlewords`)

Don't confuse *at which character a tokenizing decision is made* with the *last character of a match* (or the *first*

unmatched character). While this may be the case, it is not always true and you must write LUTHER's logic to account for this.

After the first token was assigned to `pqrs`, **matching started fresh** on the first newline of the file. **Just like the first tokenizing decision**, the newline could not be assigned to `IGNORE` until the `\` character of the second line was processed because the `endsq` DFA matched everything from the newline (of the first line) to the space after the fourth `rop` of the second line. It was the only DFA still matching characters at this point, when it failed on the `\`, the longest match that could be made at the newline was the catch-all `IGNORE` DFA.

Then `twolittlewords` matched up to the space after the second `rop`, but it couldn't be tokenized until `endsq` failed at the `\` (**again!** 😊)

This happened one last time for the last `twosmallwords` token, in this case the "deciding character" turns out to be the first first *unmatched* character of the token, but as this example has shown **this is not always the case**.

The remaining tokenizing results are more straight forward to work through. Some clarifying details though:

1. Notice that the token data reported for `whack` token ids are from the optional third field of the scanner definition file.
2. All token values are written using **alphabet encoding**.
3. There is a lingering, unseen, white space character after the first `whack` token (line2). You can see this in the value of the following `endsq` token.
4. Note that when end of file is detected, the `endsq` DFA is still actively matching. Since it isn't in an accepting state after the last newline of the file, it must be abandoned.

Hints and Testing

These conceptual keystones might be helpful in your algorithm development:

1. Any DFA, for any particular starting character of the source, can be in one of two states: `MATCHING` or `WILL_NOT_MATCH`.²
2. Any DFA, for any particular starting character of the source, should always remember the details of it's longest accepting (previous) match. This longest match may not always exist, but when it does it can only be replaced by a *longer accepting match*.³
3. When a DFA enters the `WILL_NOT_MATCH` state, it's longest match *is not* invalidated.

grader.sh

I am providing to students the same tarball the grader will use for testing your LUTHER. Here is how to use it:

First, download this tarball to your Mines Linux account ("alamode" machines!) and unroll it in a temporary directory.

```
$ ls luther-student.*
luther-student.tar.bz2
$ mkdir ~/tmp
$ cd ~/tmp
$ tar xjf ../luther-student.tar.bz2
:
$ ls
luther/
```

Second, set the `COMPGRADING` environmental variable with:

```
$ source ~khellman/COMPGRADING/setup.sh ~khellman/COMPGRADING
```

Now go to the directory holding your LUTHER and execute the `grader.sh` script from the `luther-student.tar.bz2` resource.

```
$ cd ~/compilers/LexLuther
$ ls LUTHER
LUTHER
$ ~/tmp/luther/grader.sh
:
:
:
```

You will need to read any messages from the script carefully, and you may need to hit `ENTER` several times throughout its course. This script checks for:

- a. missing data files
- b. malformed data files (of various types, including empty `program.src` and invalid characters)
- c. and the difference between your LUTHER results and expected results.

The latter test results are displayed on the terminal screen, but they whip by pretty quickly so you may need to scroll up and make sure you see them all.

When there is a discrepancy in expected results the script either:

- points you to a data file along side your LUTHER where more failure details are available,
- or all the details are displayed on screen.

Before the `grader.sh` terminates, it will show a summary of some specific rubric line results for the programming project.

The grader will use this very same script, along with some additional testing data to check your submitted work. If your LUTHER flies through without a hitch, you can likely suspect a good grade for the assignment.

Submit Your Work

Partners

You may complete this assignment in a group of two or three other students from the course (you may also work solo if you choose). Exams may have questions specific to course programming assignments — so be sure your group is truly working as a team and have a solid understanding of your submission's design and algorithms.

If you decide to complete this assignment as a group, you must tell your instructor one week before the assignment due date. Only one of you should submit the assignment for grading. You will all be able to see the grading result from your own logins.

Checking and Submitting Your Work


Here are some things to double check your submission against:

- i. Your archive does not raise errors when "unrolling" with the `${COMPGRADING}/explode-subm` script.

```
tmpdir@alamode $ ls
my-program.tar.bz2
tmpdir@alamode $ ${COMPGRADING}/explode-subm -P PROGRAM my-program.tar.bz2 ./here
```

(Where PROGRAM is the required application name for this assignment.)

- ii. Your archive contains only the essential files, **don't provide unneeded files**
- iii. Don't provide files *that have been provided to you* (such as `cblock.nfa`); that would just be silly.
- iv. Make sure the `grader.sh` script for the assignment runs to completion.

When you are happy with your work, log into  the course website and submit your project archive file for grading.

Rubric

This work is worth 70 points.

Requirements	Points	Notes
Meets compilers course project requirements	10	
exit status 1 on inaccessible file name (<code>scan.u</code> , <code>program.src</code> , <code>token.tt</code> , output file cannot be opened for writing)	5	
exit status 1 on no data (<code>scan.u</code> empty)	5	
exit status 0 on empty <code>program.src</code> , output file should exist with zero length (truncated)	5	Clearly this would be considered a "user invocation" error
exit status 1 on invalid character input	5	
reconcile correct token id by transition table ordering on match ties on same line	10	
correct token stream generated for non-overlapping matches on same line	10	
correct matching over newline(s)	10	
correct token stream and first character location generated for more complex matches	10	

1. For obvious reasons 😊 (1)
2. Or you could think of it as "alive" or "dead", "good" or "bad", "working" or "done" (2)
3. Technically, not 100% true, but the absence of this hint leads to a more complicated algorithm. (3)