

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

In this assignment will you will *close the loop* on our scanner (lexer) implementation. Your application will compile regular expressions into an NFA and write an input `scan.u` file compatible with LUTHER. The name of your application must be **WRECK** (case sensitive).

WRECK will take a command line argument that will be **very similar** to the input for LUTHER. WRECK will produce a `scan.u` file as output as well as several `.nfa` files that could be processed by (a slightly modified) NFAMATCH.¹

Input

Argument	Description	Example Value
1	(Input) lexing config file	<code>scan.lut</code>
2	(Output) scanner config file	<code>scan.u</code>

The format of `scan.lut` is identical to `scan.u` (defined in LUTHER) **except that** in the second and subsequent lines the first field is a **regular expression** instead of an optimized transition table definition file.

Coding

WRECK's first job is to choose a λ character² that does not exist in the `scan.lut` alphabet contained on the first line of the file. Since we are using **alphabet encoding**, this doesn't even have to be a printable character.

WRECK's biggest job is to convert regular expressions to NFAs, and then writing those NFAs to disk. Much of this logic has been implemented in your LGA group code base. You are **allowed to use your shared LGA code in this assignment**. Here are the steps your code should take for each regular expression and token id in `scan.lut`.

1. Using `llre.cfg`, your group's top-down LL(1) parser routines and the "silly lexer" written in `lga-silly-lexing.pdf`, parse each

regular expression.

WRECK should have an exit status of 2 if a syntax error is encountered for any regular expression in `scan.lut`; the final state of `scan.u` and NFA files is ignored.

2. Using either the completed *concrete syntax tree* (the "raw" parse tree) or by implementing *syntax directed translation* as the parse tree is constructed, derive an RE expression³ tree as described in lga-re-sdt.pdf. This **abstract syntax tree** (AST) will have internal nodes for only ALT, SEQ, * and (possibly) + and range; it will have leaf nodes that are either character values, λ leaves, or . (dot) leaf. **It's the student's choice to use + nodes (or convert them to SEQ and * subexpressions) and range nodes (or convert them to ALT subexpressions).**
3. Using the pseudo code solutions of lga-re-nfagen.pdf, convert the RE AST into a λ square matrix L and a transition table T .

WRECK should have an exit status of 3 if a semantic error is encountered for any regular expression in `scan.lut`; the final state of `scan.u` and NFA files is ignored.

The only possible semantic error⁴ is the *dash* range operator with inverted (descending ASCII code) arguments.

4. Write the NFA defined by L and T and your chosen λ character to disk using the NFA file format of NFAMATCH **except that** all character values are expressed in the **alphabet encoding** of LUTHER. Recall that while the alphabet line of `scan.u` files may have character values globbed together, the **NFA format requires white space delimited values** everywhere.

Output

WRECK will generate two forms of output, the `scan.u` file and NFA files.

scan.u

WRECK will store into `scan.u` (the output filename provided in the second command line argument) the data from `scan.lut` **except** that instead of the regular expression of the first field, it will store `tokenid.tt`. Where `tokenid` is the second field of the same line.

For example, if the `scan.lut` file contains

```
x0ax20x5C abcde
```

```
(\s|\\|b|c|d)*a      lasta
(\s|a|c|d)*b          lastb
(\s|\\|a|b|d)*c       lastc
d+                    dee      D
.                      IGNORE
```

Then the generated `scan.u` file would be

```
x0a x20 x5C x61x62x63x64x65
lasta.tt      lasta
lastb.tt      lastb
lastc.tt      lastc
dee.tt        dee      D
IGNORE.tt     IGNORE
```

I've lined the columns up nicely to make this write up easier to read, you don't have to be nearly as organized in your output: just use white space delimited fields.

Also, note that while the first line has been formatted differently, it contains **the same information** as the original `scan.lut` and it's formatting is valid according to the definition in LUTHER. This is perfectly acceptable.

tokenid.nfa

In the example above, the WRECK run would have also generated an NFA definition file for each regular expression. **The NFA file must be named `tokenid.nfa` to match the `tokenid.tt` value written in `scan.u`.**

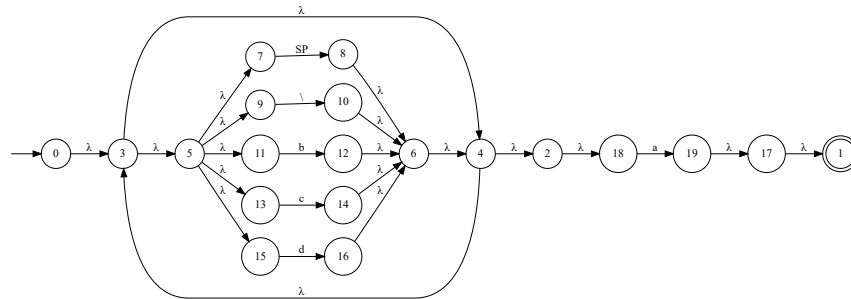
For instance, the contents of `lasta.nfa` **might be**

lasta.nfa	NFA visualization
-----------	-------------------

```

20 Z x0a x20 x5c a b
c d e
- 7 8 x20
- 9 10 x5c
- 11 12 b
- 13 14 c
- 15 16 d
- 18 19 a
- 6 4 Z
- 5 9 Z
- 17 1 Z
- 10 6 Z
- 16 6 Z
- 19 17 Z
- 5 11 Z
- 5 7 Z
- 12 6 Z
- 5 13 Z
- 5 15 Z
- 2 18 Z
- 4 3 Z
- 4 2 Z
- 8 6 Z
- 0 3 Z
- 14 6 Z
- 3 4 Z
- 3 5 Z
+ 1 1

```



"**Might**" because the choice of λ is near arbitrary, and since this is an **NFA** the number of representations is unbounded.

The format for the NFA file is the same as in NFAMATCH except that all the **white space delimited character values** use **alphabet encoding**.

The **tokenid.nfa** file(s) should be written in the WRECK process's invocation directory (its *current working directory*).

grader.sh

I am providing to students the same tarball the grader will use for testing your WRECK. 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 wreck-student.*
wreck-student.tar.bz2
$ mkdir ~/tmp
$ cd ~/tmp
$ tar xjf ../wreck-student.tar.bz2
:
$ ls
wreck/
```

Second, set the COMPGRADING environmental variable with:

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

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

```
$ cd ~/compilers/wreckingBar
$ ls WRECK
WRECK
$ ~/tmp/wreck/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. truncated data files
- c. semantic errors
- d. syntax errors
- e. and the difference between your WRECK 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 or directory along side your WRECK 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 WRECK 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:

- 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.)

- Your archive contains only the essential files, **don't provide unneeded files**
- Don't provide files *that have been provided to you* (such as `cblock.nfa`); that would just be silly.
- 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 60 points.

Requirements	Points	Notes
Meets compilers course project requirements	10	
exit status 1 on IO errors: <i>inassessible files</i> (<code>scan.lut</code> , any output file cannot be opened for writing) and <i>no data</i> (<code>scan.lut</code> empty)	10	
exit status 3 on semantic errors	5	
exit status 2 on syntax errors	5	
generates correct <code>scan.u</code>	10	
generates correct <code>tokenid.nfa</code> NFA files	20	

1. NFAMATCH needs to know about **alphabet encoding**. (1)
2. Be sure to not choose NUL 0x00. (2)
3. AKA *abstract syntax tree*. (3)
4. Can you think of another? If so, let me know! (4)

Assignments/WRECK (last edited 2022-04-12 18:55:15 by khellman)