CSc 453 — Assignment 2

Christian Collberg
University of Arizona, Department of Computer Science

January 28, 2026

1 Introduction

This assignment has two parts. In the first part you will answer some theory questions, similar to those you saw in the in-class exercises. In the second part you will write a lexical analyzer for Luca.

2 Theory Questions

Each question is worth 4 points.

Some of the problems require you to submit your answer in the form of a Graphviz file. You can learn about Graphviz by watching lecture **misc-1** in `https://ligerlabs.org/lectures.html`.

2.1 Problem 1

Consider this grammar for floating point numbers:

```
float  → + float1 | - float1 | float1
float1 → digit float1 | float2
float2 → . float3
float3 → digit float4 | digit
float4 → digit float4 | float5
float5 → E float6
float6 → + float7 | - float7 | float7
float7 → digit float7 | digit
```

Show a derivation for the number "`-1.23E+4`".

Enter your answer in the file `problem1.json`. The first few steps have been provided for you.

2.2 Problem 2

In computer security it's important to avoid buffers with constant length since they can lead to buffer overflows. Construct a regular expression that can search through your C codebase to look for buffer declarations that could possibly be problematic.

The basic structure you're looking for is this:

```
char buffer_name [integer_constant] ;
```

`buffer_name` should be one of

```
"buf"
"BUF"
"buffer"
"input"
```

Whitespace (space or tab) can be inserted anywhere,' like this for example

`␣␣␣char␣␣␣␣␣␣buffer_name␣␣␣␣[␣␣␣␣␣␣integer_constant␣␣␣␣␣␣]␣␣␣␣␣;`

We assume the entire declaration occurs on one line. However, there can be other code on the same line. We will test your regular expression using `egrep`, like this:

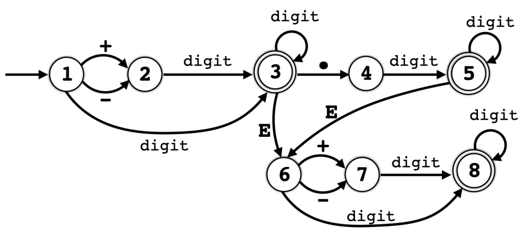
```
> cat problem2_test.c
   /* comment here! */ char BUF [4] ; int buff_ptr=0;
blah blah
more blah
char array[10];
char BUF[LENGTH] ;;;;
char    buffer    [   42    ] ;;;
super blah
char input[67];

> egrep '...' problem1_test.c
   /* comment here! */ char BUF [4] ; int buff_ptr=0;
char    buffer    [   42    ] ;;;
char input[67];
```

Enter your answer in the file `problem2.json`.

2.3 Problem 3

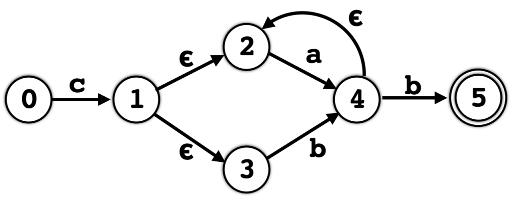
Consider this DFA:



Show the sequence of transitions that the DFA goes through when matching the string `"+1.23E+45"`. Enter your answer in the file `problem3.json`.

2.4 Problem 4

Compute `epsilon-closure` for all the nodes in this NFA:



Enter your answer in the file `problem4.json`. State 0 is the start state.

2.5 Problem 5

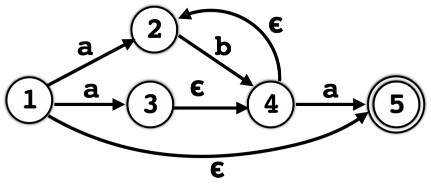
Use Thompson's construction to construct the NFA from this regular expression:

```
a*(b|c)
```

Enter your answer in the Graphviz file `problem5.gv`.

2.6 Problem 6

Consider this NFA:



Use the subset construction to construct the corresponding DFA. Enter your answer in the Graphviz file `problem6.gv`. State 1 is the start state.

3 A lexical analyzer for Luca

Your task is to write a lexical analyzer for the language Luca. Your program should be named `luca_lex`. `luca_lex` takes a Luca source program as input and produces a list of tokens as output, like this:

```
$ luca_lex prog1.luc > prog1.luc.out
```

The result is a list of tokens in XML format:

```
PROGRAM␣P;
␣␣␣CONST␣C␣:␣BOOLEAN␣=␣true;
BEGIN
␣␣␣WRITE␣"Hello!";
␣␣␣WRITE␣666;
␣␣␣WRITE␣6.66;
␣␣␣--␣Here's␣comment!
END.
```

⇒

```
<block>
   <TOKEN kind="PROGRAM" line="1"/>
   <TOKEN kind="IDENT" line="1" value="P"/>
   <TOKEN kind="SEMICOLON" line="1"/>
   <TOKEN kind="CONST" line="2"/>
   <TOKEN kind="IDENT" line="2" value="C"/>
   <TOKEN kind="COLON" line="2"/>
   <TOKEN kind="IDENT" line="2" value="BOOLEAN"/>
   <TOKEN kind="EQ" line="2"/>
   <TOKEN kind="IDENT" line="2" value="true"/>
   <TOKEN kind="SEMICOLON" line="2"/>
   <TOKEN kind="BEGIN" line="3"/>
   <TOKEN kind="WRITE" line="4"/>
   <TOKEN kind="STRINGLIT" line="4" value="Hello!"/>
   <TOKEN kind="SEMICOLON" line="4"/>
   <TOKEN kind="WRITE" line="5"/>
   <TOKEN kind="INTLIT" line="5" value="666"/>
   <TOKEN kind="SEMICOLON" line="5"/>
   <TOKEN kind="WRITE" line="6"/>
   <TOKEN kind="REALLIT" line="6" value="6.66"/>
   <TOKEN kind="SEMICOLON" line="6"/>
   <TOKEN kind="END" line="8"/>
   <TOKEN kind="PERIOD" line="8"/>
   <TOKEN kind="EOF" line="9"/>
</block>
```

3.1 Getting Started

This assignment can be done in teams of one or two.

To get started, follow these instructions:

1. Unzip `assignment-2.zip` which gives you these files:

```
assignment-2/assignment-2.pdf
assignment-2/validator.sh
assignment-2/firstname_lastname/lexer/Lex.java
assignment-2/firstname_lastname/lexer/Char.java
assignment-2/firstname_lastname/lexer/Input.java
assignment-2/firstname_lastname/lexer/Token.java
assignment-2/firstname_lastname/lexer/LexTable.java
assignment-2/firstname_lastname/tests/*.luc
assignment-2/firstname_lastname/test_results/*.out
assignment-2/firstname_lastname/collaboration.json
assignment-2/firstname_lastname/survey.json
```

2. Then, assuming your name is Pat Jones, execute these commands:

```
> cd assignment-2
> mv firstname_lastname pat_jones
```

   If you are working in a group of two, pick one of your names.

3. Now, go ahead and solve the problems, entering your answers in the template files in `pat_jones/*`. Your main work should be modifying the Java source files in `assignment-1/pat_jones/lexer`.

4. Skeleton Java code has been provided for you in `assignment-2/firstname_lastname/lexer/*.java`.

Don't show your code to anyone outside your team, don't read anyone else's code, don't discuss the details of your code with anyone. If you need help with the assignment see the TA or the instructor.

3.2 Grading

21 test cases have been provided for you in `assignment-2/firstname_lastname/tests/`. The expected output is given in `assignment-2/firstname_lastname/test_results`.

Each test case will give you 0 points if you get it wrong, 3 points if you get it right. There is no partial credit. You can earn an additional 13 points from more complicated secret test cases.

You can see some of the test cases in Appendix B.

3.3 Output Format

The output of `luca_lex` is a text file in *XML* format. There are essentially two kinds of entries: those that represent literals and identifiers contain an argument `value`, the rest do not:

```
<block>
   <TOKEN kind="PROGRAM" line="1"/>
   <TOKEN kind="EQ" line="2"/>
   <TOKEN kind="IDENT" line="1" value="P"/>
   <TOKEN kind="STRINGLIT" line="4" value="Hello!"/>
   <TOKEN kind="INTLIT" line="5" value="666"/>
   <TOKEN kind="REALLIT" line="6" value="6.66"/>
   <TOKEN kind="CHARLIT" line="6" value="C"/>
   <TOKEN kind="EOF" line="8"/>
</block>
```

The last token generated is always `EOF`. A complete list of tokens is given in Appendix A.

Line numbers are allowed to be "approximately" correct, i.e. they should be ± 1 from the "correct" line.

Be careful to get the output syntactically correct! Avoid extra spaces between `<` and `TOKEN` and `/` and `>`, for example. You can check the validity of the XML you generate using the `xmllint` program:

```
> cat f.out
<block>
   <TOKEN kind="REALLIT" line="10" value="1.2E+99"/ >
   <TOKEN kind="EOF" line="11"/>
</block>
> xmllint f.out
f.out:2: parser error : attributes construct error
   <TOKEN kind="REALLIT" line="10" value="1.2E+99"/ >
                                                   ^
f.out:2: parser error : Couldn't find end of Start Tag TOKEN line 2
   <TOKEN kind="REALLIT" line="10" value="1.2E+99"/ >
```

3.4 Luca Lexical Rules

- Luca line comments start with a `--`-sign and extend to the end of the line. They are not included in the output of `luca_lex`.
- Luca structured comments start with a `(*` and must end with `*)`. They are not allowed to be nested. They are not included in the output of `luca_lex`.
- Luca is case-sensitive.
- Strings start and end with a `"`-character and cannot contain a `"`-character. They cannot extend past the end of a line. There are no escape characters, so there is no way to have a `"` inside a string.
- Character literals start and end with a `'`-character and must contain exactly one character (not a `'`).
- Identifiers consist of letters and digits, and must start with a letter.
- Integer literals consist of a sequence of digits.
- Real literals have the syntax

$$((\texttt{digit}*.\texttt{digit}+)|(\texttt{digit}+.\texttt{digit}*))(\texttt{E}(\backslash+|-)?\texttt{digit}+)?$$

  Examples of valid floating-point numbers:

  ```
  0.5    .5    5.    5.0    5.E-6   100.587E99
  ```

- Control characters other than tabs and newlines are not allowed in Luca source files.

3.5 Luca Lexical Errors

Instead of printing error messages, `luca_lex` generates these special *error tokens*:

```
ERROR_UNTERMINATED_STRING
ERROR_REALLIT
ERROR_ILLEGAL_CHARACTER
ERROR_UNTERMINATED_COMMENT
ERROR_UNTERMINATED_CHAR
ERROR_EMPTY_CHAR
```

Error messages can then be generated by the parser instead.

Here's an example output when lexing a program consisting only of the empty character `''`:

```
<block>
   <TOKEN kind="ERROR_EMPTY_CHAR" line="1"/>
   <TOKEN kind="EOF" line="2"/>
</block>
```

You don't have to do any error recovery. We won't test any of your output that appears after an error token.

3.6 Implementation Notes

- **This assignment should be coded in Java**.
- Make sure that your `Makefile` is working properly. The TA will do the following, and nothing else:

  ```
  $ make
  $ luca_lex test1.luc > test1.luc.out
  ```

  In other words, you must provide a shell script called `luca_lex` that calls Java with the appropriate parameters.
- **You cannot use `lex` or any other similar lexical analysis generator for this assignment, either directly or indirectly.** I expect you to build the finite state machine by hand, and code it by hand.
- **You cannot use any AI resources for this assignment.**
- **You should build your scanner the way we've discussed in class, using a table-driven finite state automaton.** If you try anything else you will get 0 points.

4 Submission

To submit, follow these instructions:

1. Verify that your submission contains a working `Makefile` and *all* the files necessary to build the lexeer. Your program must compile out of the box. The graders will *not* try to debug your program or your makefile for you.
2. Fill out `collaboration.json` with information about your group.
3. Optionally fill out `survey.json` to give me feedback about the assignment.
4. Package up your answers and verify that they look OK:

```
> zip -r pat_jones.zip pat_jones
> assignment-2/validator.sh pat_jones.zip
==== This is the LigerLabs assignment validator ====
          ...         ...         ...         ...
==== 6: Checking that all problems have been answered.
==== 7: Checking that all answer files have valid structure.
```

5. Upload `pat_jones.zip` to d2l. If you are working in a group of two, you should only submit once.

A Luca Tokens

token	token_name
+	PLUS
-	MINUS
*	STAR
/	SLASH
%	PERCENT
:=	COLONEQ
!	BANG
:	COLON
,	COMMA
[	LBRACK
]	RBRACK
(	LPAREN
)	RPAREN
.	PERIOD
;	SEMICOLON
^	CARET
@	ATCHAR
‘	BACKQUOTE
AND	AND
OR	OR
=	EQ
>=	GE
>	GT
<	LT
<=	LE
#	NE
ISA	ISA
NARROW	NARROW
TRUNC	TRUNC
FLOAT	FLOAT
NOT	NOT
PROGRAM	PROGRAM
PROCEDURE	PROCEDURE
VAR	VAR
BEGIN	BEGIN
END	END

token	token_name
FOR	FOR
NEW	NEW
TYPE	TYPE
WRITE	WRITE
READ	READ
WRITELN	WRITELN
ENDFOR	ENDFOR
EXTENDS	EXTENDS
REF	REF
ENUM	ENUM
CONST	CONST
ARRAY	ARRAY
RECORD	RECORD
METHOD	METHOD
CLASS	CLASS
OF	OF
IN	IN
TO	TO
DO	DO
BY	BY
IF	IF
THEN	THEN
ELSE	ELSE
ENDIF	ENDIF
LOOP	LOOP
ENDLOOP	ENDLOOP
EXIT	EXIT
WHILE	WHILE
REPEAT	REPEAT
UNTIL	UNTIL
ENDDO	ENDDO
integer literal	INTLIT
real literal	REALLIT
string literal	STRINGLIT
char literal	CHARLIT
identifier	IDENT
	EOF

B Examples

```
[␣]␣(␣)
[]()
```

⇒

```
<block>
   <TOKEN kind="LBRACK" line="1"/>
   <TOKEN kind="RBRACK" line="1"/>
   <TOKEN kind="LPAREN" line="1"/>
   <TOKEN kind="RPAREN" line="1"/>
   <TOKEN kind="LBRACK" line="2"/>
   <TOKEN kind="RBRACK" line="2"/>
   <TOKEN kind="LPAREN" line="2"/>
   <TOKEN kind="RPAREN" line="2"/>
   <TOKEN kind="EOF" line="3"/>
</block>
```

```
:
:=
```

⇒

```
<block>
   <TOKEN kind="COLON" line="1"/>
   <TOKEN kind="COLONEQ" line="2"/>
   <TOKEN kind="EOF" line="3"/>
</block>
```

```
.12
34.
45.67
8.E5
.9E5
1.2E5
1.2e5
1.2e-5
1.2E+5
1.2E+99
```

⇒

```
<block>
   <TOKEN kind="REALLIT" line="1" value=".12"/>
   <TOKEN kind="REALLIT" line="2" value="34."/>
   <TOKEN kind="REALLIT" line="3" value="45.67"/>
   <TOKEN kind="REALLIT" line="4" value="8.E5"/>
   <TOKEN kind="REALLIT" line="5" value=".9E5"/>
   <TOKEN kind="REALLIT" line="6" value="1.2E5"/>
   <TOKEN kind="REALLIT" line="7" value="1.2e5"/>
   <TOKEN kind="REALLIT" line="8" value="1.2e-5"/>
   <TOKEN kind="REALLIT" line="9" value="1.2E+5"/>
   <TOKEN kind="REALLIT" line="10" value="1.2E+99"/>
   <TOKEN kind="EOF" line="11"/>
</block>
```

```
1
12
123
1234
```

⇒

```
<block>
   <TOKEN kind="INTLIT" line="1" value="1"/>
   <TOKEN kind="INTLIT" line="2" value="12"/>
   <TOKEN kind="INTLIT" line="3" value="123"/>
   <TOKEN kind="INTLIT" line="4" value="1234"/>
   <TOKEN kind="EOF" line="5"/>
</block>
```

```
"foo"
""
'a'
```

⇒

```
<block>
   <TOKEN kind="STRINGLIT" line="1" value="foo"/>
   <TOKEN kind="STRINGLIT" line="2" value=""/>
   <TOKEN kind="CHARLIT" line="3" value="a"/>
   <TOKEN kind="EOF" line="4"/>
</block>
```

```
--␣this␣is␣a␣comment␣--
-␣--␣and␣here␣is␣another␣one
```

⇒

```
<block>
   <TOKEN kind="MINUS" line="2"/>
   <TOKEN kind="EOF" line="3"/>
</block>
```

```
(*␣comment␣here␣*)
(␣*␣*␣)
(*␣(␣*␣comment␣here␣*␣)
*6␣and␣here␣*)
*)
```

⇒

```
<block>
   <TOKEN kind="LPAREN" line="2"/>
   <TOKEN kind="STAR" line="2"/>
   <TOKEN kind="STAR" line="2"/>
   <TOKEN kind="RPAREN" line="2"/>
   <TOKEN kind="STAR" line="5"/>
   <TOKEN kind="RPAREN" line="5"/>
   <TOKEN kind="EOF" line="6"/>
</block>
```

```
''
```

$\Longrightarrow$

```
<block>
   <TOKEN kind="ERROR_EMPTY_CHAR" line="1"/>
   <TOKEN kind="EOF" line="2"/>
</block>
```

```
?
```

$\Longrightarrow$

```
<block>
   <TOKEN kind="ERROR_ILLEGAL_CHARACTER" line="1"/>
   <TOKEN kind="EOF" line="2"/>
</block>
```

```
.5E-x
```

$\Longrightarrow$

```
<block>
   <TOKEN kind="ERROR_REALLIT" line="1"/>
   <TOKEN kind="EOF" line="2"/>
</block>
```

```
.5E
```

$\Longrightarrow$

```
<block>
   <TOKEN kind="ERROR_REALLIT" line="1"/>
   <TOKEN kind="EOF" line="2"/>
</block>
```

```
'
```

$\Longrightarrow$

```
<block>
   <TOKEN kind="ERROR_UNTERMINATED_CHAR" line="1"/>
   <TOKEN kind="EOF" line="2"/>
</block>
```

```
(*␣fooo
```

$\Longrightarrow$

```
<block>
   <TOKEN kind="ERROR_UNTERMINATED_COMMENT" line="1"/>
   <TOKEN kind="EOF" line="2"/>
</block>
```

```
(*␣fooo␣*␣)
lots␣of␣stuff
here␣42


blah
```

$\Longrightarrow$

```
<block>
   <TOKEN kind="ERROR_UNTERMINATED_COMMENT" line="1"/>
   <TOKEN kind="EOF" line="7"/>
</block>
```

```
"blah␣blah␣blah
42
```

$\Longrightarrow$

```
<block>
   <TOKEN kind="ERROR_UNTERMINATED_STRING" line="1"/>
   <TOKEN kind="INTLIT" line="2" value="42"/>
   <TOKEN kind="EOF" line="3"/>
</block>
```

```
"blah␣blah␣blah
```

$\Longrightarrow$

```
<block>
   <TOKEN kind="ERROR_UNTERMINATED_STRING" line="1"/>
   <TOKEN kind="EOF" line="2"/>
</block>
```

C Optional Extensions

If you want to challenge yourself some more add one or more of these extensions. Submit extensive test cases for each extension and a `README` file that describes what you did. There is no extra credit, just the satisfaction of a job well done!

D Perfect hashing

Use perfect hashing for the keywords. You may use `gperf`.

D.1 Nested comments

Add Modula-2-style nested comments:

```
(*
   This is a comment.
   (*
      This is a comment within a comment.
   *)
*)
```

It is a static error for comments to be unbalanced. That is, every `(*` must have a matching `*)`. Nested comments are not included in the output of `luca_lex`.

D.2 C-style escape-characters

Allow C-style escape-characters within string literals. You should support at least `\n`, `\"`, and `\t`.

D.3 Hexadecimal, octal, and binary integer constants

Allow hexadecimal, octal, and binary integer constants. Hexadecimal literals consist of the prefix `0x` followed by a sequence of hexadecimal digits (`0-9`, `A-F`, `a-f`). Upper and lower case hexadecimal digits may be freely mixed. Octal constants consist of the prefix `0o` followed by a sequence of octal digits (`0-7`). Binary constants consist of the prefix `0b` followed by a sequence of binary digits (`0,1`).

In every case, the output of `luca_lex` should be the corresponding decimal representation:

```
0xfF ⇒ (INTLIT 255)     0o66 ⇒ (INTLIT 54)     0b11001 ⇒ (INTLIT 25)
```