CS164 Midterm 1

Fall 2004

- Please read all instructions (including these) carefully.
- **Write your name, login, and circle the time of your section.**
- Read each question carefully and think about what's being asked. Ask the proctor if you don't understand anything about any of the questions. If you make any non-trivial assumptions, state them clearly.
- Some questions span multiple pages. Be sure to answer every part of each question.
- You will have 1 hour and 20 minutes to work on the exam. The exam is closed book, but you may refer to your one side of one page of handwritten notes.
- Solutions will be graded on correctness and clarity, so make sure your answers are neat and coherent. Remember that if you can't read it, it's wrong!
- Each question has a relatively simple and straightforward solution. We might deduct points if your solution is far more complicated than necessary. Partial answers will be graded for partial credit.
- Write all of your answers in the space provided on the exam, and clearly mark your answers. Use the backs of the exam pages for scratch work. Do not use any extra scratch paper. Do not unstaple the exam.
- Turn off your cell phone.
- Good luck!

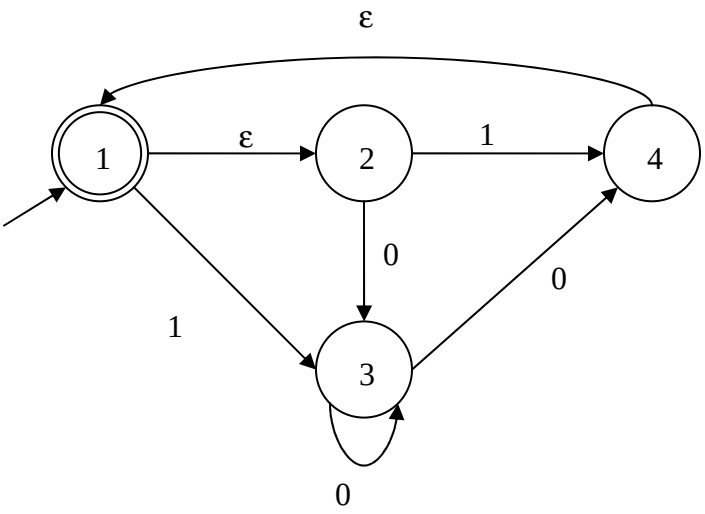
NAME and LOGIN:

Your SSN or Student ID:

Your section: 10:00 11:00 2:00 4:00

Q1 (30 pts)	Q2 (30 pts)	Q3 (30 pts)	Q4 (10 pts)	Total (100 pts)

1. Regular Expressions/Finite Automata



Part a). Give a string that is accepted by the NFA above, and a string that is not accepted.

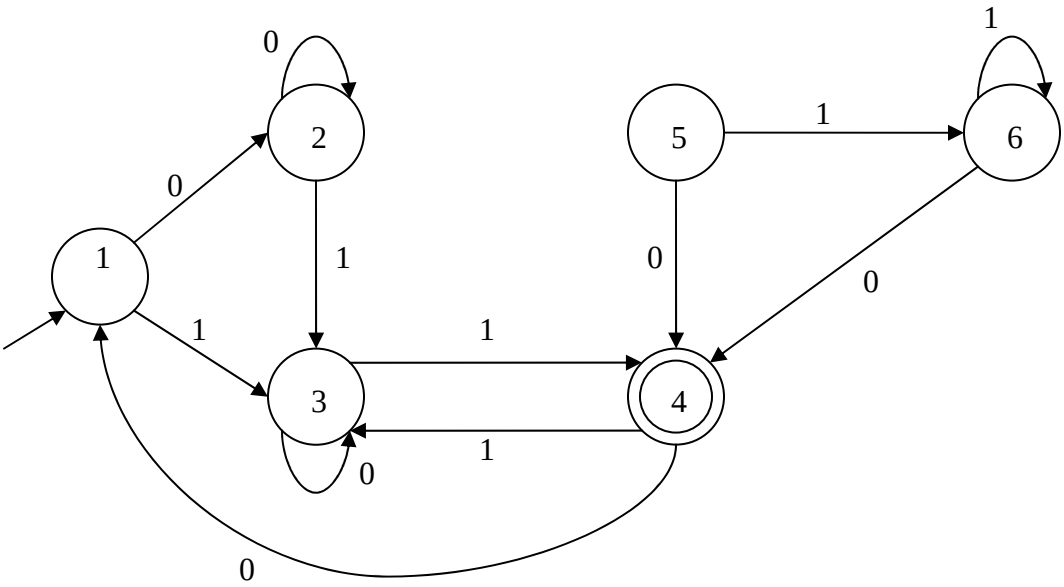
Accepted:

Not accepted:

Part b). Convert the NFA above to a DFA.

Part c). Give a regular expression that describes the set of strings accepted by this NFA.

Part d). Minimize the DFA below.

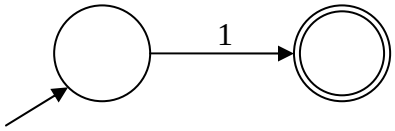


Part e). Given a particular expression *A* and an NFA that recognizes *L*(*A*), we would like to construct an NFA that recognizes *L*(*A**). We will use the following construction for *A**:

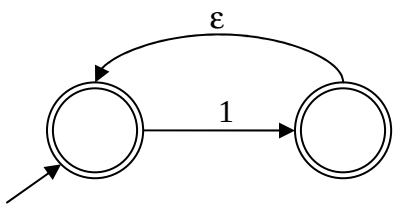
> Make the start state of *A* an accepting state of *A**. For each accepting state *s* of *A*, add an epsilon transition from *s* to the start of state *A*.

Here is an example construction. The regular expression is 1. We use the construction above to build a NFA to recognize 1*.

NFA for 1



Using the algorithm, we get the NFA to the right for recognizing 1*



Circle YES if the construction will give an NFA that recognizes *L*(*A**) for any *A*, and give a brief explanation. Circle NO if the construction does not always work, and give a counter example by showing a regular expression along with an input on which the construction fails to behave as desired.

YES NO

2. First/Follow/LL(1)

This question concerns the context-free grammar given below (where capital letters denote non-terminals, small letters denote terminals).

$E \rightarrow Z\ X\ b \mid Z\ a$
$X \rightarrow d\ Z \mid \epsilon$
$Z \rightarrow a\ X\ X \mid X$

Part a).

Fill in the following table with First and Follow sets for the grammar.

α	*FIRST* (α)	*FOLLOW* (α)
E		
X		
Z		
$Z\ X\ b$		
$Z\ a$		
$d\ Z$		
$a\ X\ X$		
X		
ϵ		

Part b).

In the *LL*(1) parsing table below, fill in the row corresponding to the non-terminal A:

	a	*b*	*d*	$
Z				

Part c).

Is the grammar *LL*(1)? Justify your answer.

Part d).

Suppose that in the above grammar, you replace $\in$ with *c*, so the grammar becomes

$E \rightarrow Z\ X\ b \mid Z\ a$
$X \rightarrow d\ Z \mid c$
$Z \rightarrow a\ X\ X \mid X$

Now, you are parsing the string **a c c d c b** with a non-deterministic LR parser. For each of the following stack/input configurations, state whether or not they can lead to a successful parse (YES/NO will suffice).

a c c | d c b

a X X | d c b

a X X d Z | b

a X X X | b

Z X | b

3. Decaf Lexer Specification

Background. This question asks you to improve your PA2 Decaf lexer by *adding new lexeme definitions*. We will add new lexemes not in order to extend the Decaf language, but instead to improve your lexer's poor error reporting manners. Currently, whenever there is a lexical error, your PA2 lexer happily throws an exception, instead of telling the programmer what specific kind of lexical error was encountered. We will add lexemes that correspond to three common lexical errors, so that the lexer can print a useful error message.

The problem. Your goal is to extend your lexer so that it recognizes three kinds of malformed string literals; when there string literals are matched, the lexer must print an appropriate error message and ignore the matched lexeme.

Use the regular expression syntax from PA2. As a reference, the `dlex` code on the next page defines a correct *string literal* lexeme, for strings as defined in PA2. Your code can refer to the definitions on the next page.

- *Undetermined string literals*. Match string literals that contain newline or end-of-file before closing double quote.

  Note that the malformed string literal ends at the newline. That is,

  ```
  print(“adadasda);
  print(“adadadfa”);
  ```

  will produce token sequence PRINT, LPAR, PRINT, LPAR, STRINGLITERAL, RPAR, SEMICOLON.

  Also note that a string literal that has a newline immediately after a backslash should be treated as an unterminated string literal with a bad escaped character (see the last bullet on this page).

- *Bad string literals*. Match a string literal that includes a bad “escaped” character; i.e. a backslash followed by something other than an `n`, a single quote, a double quote, a space, or another backslash.

- *Bad escaped character and unterminated string literals*. Match a string literal that has a newline immediately after a backslash. For example, given:

  ```
  “very bad string \
  abc
  ```

  the lexer should report an unterminated string literal with a bad escaped character on line 1, and an identifier on line 2.

Turn over.

Correct PA2 string literal:

```
legalascii = ' ' | '!' | ('#' - '[') | (']' - '~')
stringchar = legalascii | ('\\' + ('\"' | '\\' | \n\ | ' '));
stringlit = '\"' + stringchar*0 + '\"';
if ( stringlit) { return STRINGLITERAL; }
```

Write your `dlex` code here:

The following ASCII table may also come handy:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	“	#	$	%	&	‘	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	d	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

4. Improving your Lexer Generator

Background. In this question, you will improve your PA2 lexer generator. As you may remember, one annoying thing in PA2 was that you had to enter the dlex specification in Java syntax, so that we didn't have to develop a parser for regular expressions. In this question we'll develop a proper parser for regular expressions. Don't panic, you will only have to specify the tokens, the grammar, and show an example AST.

The problem. You are to specify the tokens, the grammar and the AST for a parser that accepts regular expressions and outputs their ASTs. Regular expressions must follow the syntax we used in the lectures. Here are a few such regular expressions:

- a
- a | b
- (foo)*
- foo*.bar.baz+
- foo
- f.o.o
- foo | bar+
- foo*+

Notes: the "." Operator is mandatory, except in strings, (that is, *f.o.o.* is the same as *foo*, but (a | b)(c | d) is illegal, it must be (a | b).(c | d)). The priority of operators is "*" and "+" are highest, then ".", then "|".

Turn over.

Part a). Design the lexer for the language or regular expressions. That is, show which tokens should the lexer recognize. For each token, show lexeme (using the dlex syntax of PA2). Also, when applicable, show the attributes of the token.

TOKEN	LEXEME	ATTRIBUTES

Part b). Show a grammar for the language of regular expressions. The grammar must be unambiguous and must observe the priorities of operators. The terminals of the grammar should be the tokens you defined in Part a).

Part c). Draw an AST for each of the following four regular expressions. Define AST nodes as you see fit.

foo	*f.o.o*	*foo|bar+*	*foo*+*