Lexical Analysis Quiz

Question 1

Given two regular expressions `X` and `Y` and their corresponding NFAs. When creating the NFAs for `X|Y` and for `X*` using Thompson's construction, how many new states and epsilon transitions need to be added in each case?

Select one:

Two new states and two new epsilon transitions each.

Two new states and four new epsilon transitions for `X|Y` and two new states and three epsilon transitions for `X*`.

A new start state and two new epsilon transitions each.

Four new states and two epsilon transitions each.

Two new states and four epsilon transitions each.

The correct answer is: Two new states and four epsilon transitions each.

Question 2

What is the smallest language category that contains the language Java?

None

Context-free

Unrestricted

Regular

Context-sensitive

The correct answer is: Context-sensitive

Question 3

Which regular expression describes strings that can start with an `a` or a `b`, are followed by one or more `c`'s, and end in a `d`.

`a|b c+d`

`ac+|bd`

`(a|b)+cd`

`(a|b)(cd)+`

`(a|b)c+d`

The correct answer is: `(a|b)c+d`

Question 4

Suppose you are translating an arbitrary NFA into a DFA using the subset construction algorithm. Which NFA states are in the set of states that form the start state of the DFA?

The start state of the NFA and all the states anywhere in the NFA that can be reached with epsilon transitions.

Only the start state of the NFA.

All the states that can be reached with epsilon anywhere in the NFA.

The start state of the NFA and all the states that can be reached with a single epsilon transition from the start state of the NFA.

The start state of the NFA and all the states that can be reached with one or more epsilon transitions from the start state of the NFA.

The correct answer is: The start state of the NFA and all the states that can be reached with one or more epsilon transitions from the start state of the NFA.

Question 5

Which is the smallest language category that contains the language of nested Tiger comments.

Regular

Unrestricted

None

Context-free

Context-sensitive

The correct answer is: Context-free

HW1:

1. Consider the following format for floating point literals. Assume a fraction part consisting of a string of one or more decimal digits with an optional decimal point (period) that can be added immediately before or after *any* of the digits. In addition, an optional exponent can be appended to the fraction part, consisting of an `e` or `E`, followed by an optional sign, followed by one or more decimal digits. A floating point literal must contain either a decimal point, or an exponent, or both.
   Examples: `.0`, `0.`, `0.1`, `.01e-01`, `01E00`
   a. Give a regular expression for this construct. You may use ϵ.
      ((. D+ | D+ . D*)(epsilon | Exp Sign D+)) | D+ Exp Sign D+
   b. Use Thompson's construction to translate the regular expression into an NFA.
      start state s1;
      state s2, s4, s5, s6;
      final state s3, s7;

      <s1> . <s2>;
      <s1> D <s4>;
      <s2> D <s3>;
      <s3> D <s3>;
      <s3> Exp <s5>;
      <s4> . <s3>;

<s4> D <s4>;
<s4> Exp <s5>;
<s5> Sign <s6>;
<s6> D <s7>;
<s7> D <s7>;

where the syntax <s1>e<s2> means that there is an edge from s1 to s2 with label e.

Make sure that you don't accept integers!

2. Consider scanning literals for representing carbohydrates in chemistry. Such literals consist of one or more *parts*, each consisting of `C`, `H`, or `O`, optionally followed by a decimal integer greater than 1 (it can greater than 9). You may use `D0` to mean decimal digits 0-9, `D1` to mean 1-9, and `D2` to mean 2–9. (This is a computer science question, not a chemistry question. In particular, this specification allows bogus chemical formulas such as H_{42}.)
    a. Give a regular expression for these carbohydrate literals. You may use the iteration operators * and + and ?.
    ((C|H|O) (epsilon | D2 | D1D0+))+
    b. Give a DFA that recognizes exactly these literals (no ϵ edges). Don't forget to mark the start state and all accepting states.

start state s1;
final state s2, s4;
state s3;

<s1> C|H|O <s2>;
<s2> C|H|O <s2>;
<s2> 1 <s3>;
<s3> D0 <s4>;
<s2> D2 <s4>;
<s4> C|H|O <s2>;
<s4> D0 <s4>;

where the syntax <s1>e<s2> means that there is an edge from s1 to s2 with label e.

Parsing

Question 1

Which styles of context-free grammars cause problems for LL parsing?

Question 1Select one:

Grammars that have a common left factor.

Grammars that are left-recursive.

Grammars that are left-recursive or have common left factors.

Grammars that are ambiguous, left-recursive, or have common left factors.

Grammars that are ambiguous.

The correct answer is: Grammars that are ambiguous, left-recursive, or have common left factors.

Question 2

Suppose we have the following two grammar rules

```
S -> id := E ;
E -> id + id
```

and we are parsing the input `x:=y+z;` . At which point in the parser trace do we apply the reduce operation for rule `E->id+id`?

When the entire input `id:=id+id;` has been pushed onto the stack and the input is empty.

When the stack contains `id:=` and `id+id` is next in the input.

When the stack contains `id:=id+id` and the next token in the input is the semicolon.

At any time when we find `id+id` anywhere on the stack.

When the stack contains `id:=E` and the semicolon is next in the input.

The correct answer is: When the stack contains `id:=id+id` and the next token in the input is the semicolon.

Question 3

A grammar with a dangling else is not in the LALR(1) grammar category, even though a parser generator can successfully fix the shift-reduce conflict on ELSE in favor of shift.

True

False

The correct answer is 'True'.

Question 4

Any grammar that can be parsed with a recursive-descent parser with one token lookahead can also be parsed with an LALR(1) parser generated by a parser generator.

True

False

The correct answer is 'False'.

Question 5

Suppose an LR(0) parser has a parse conflict for a given stack content with `X` at the top of the stack. An SLR parser may be able to resolve the parse conflict by only applying a reduce operation using grammar rule `N->X` if the lookahead token is in FOLLOW(`N`).

True

False

The correct answer is 'True'.

HW2:

Homework 2 Solutions
Context free grammars, LL and LR parsing
Consider the following simple context free grammars:
Grammar G_1:
$G \rightarrow A\$$
$A \rightarrow \epsilon$
$A \rightarrow bAb$
Grammar G_2:
$G \rightarrow A\$$
$A \rightarrow \epsilon$
$A \rightarrow Abb$
The start symbols are *G*, the nonterminals are *G* and *A*, and the terminal symbols are *b* and $ (end of file). Note that these grammars generate the same language: strings consisting of even numbers of *b* symbols (including zero of them).

a. Attempt to show a shift-reduce parse of the input string *bbbb* for a parser for grammar G_1. Show the contents of the stack, the input, and the actions (in the style of Figure 3.18 on Page 58 but without the subscripts for the parse states).

Indicate any conflicts and describe why they are conflicts. Is G_1 LR(1)? Is it LR(0)?

1. (a)

stack	input	action
	bbbb$	shift
b	bbb$	shift / reduce A->

G1 is not LR(1) since there is a conflict.
G1 is not LR(0) since it is not LR(1).

b. Attempt to show a shift-reduce parse of the input string *bbbb* for a parser for grammar G_2. Show the contents of the stack, the input, and the actions (in the style of Figure 3.18 on Page 58 but without the subscripts for the parse states).
Indicate any conflicts and describe why they are conflicts. Is G_2 LR(1)? Is it LR(0)?

(b)

stack	input	action
	bbbb$	reduce A->
A	bbbb$	shift
Ab	bbb$	shift
Abb	bb$	reduce A->Abb
A	bb$	shift
Ab	b$	shift
Abb	$	reduce A->Abb
A	$	accept

G2 is LR(1) since the parse worked.
G2 is LR(0) since we didn't need the lookahead for making parsing decisions (we know when to reduce by looking only at the stack).

Really, the arguments that G2 is LR(1)/LR(0) are kind of weak. The real way to show that would be to construct the parse tables. But since the language is that simple, and since we don't have too many options for parsing it, it works to argue from this example parse trace that it should work for any input.

BTW, in some books (e.g., the previous edition of the the 4101 textbook) and some material on the internet claim that a grammar with epsilon cannot be LR(0). G2 is a counter-example. An epsilon production must be the bottom-left leaf in a subtree of the concrete parse tree for an LR(0) parser to handle it.

(c) G1 is not LL(1) since b predicts both A-> and A->bAb.
G2 is not LL(1) since it is left recursive.

Of the language classes we have discussed in class, what is the smallest category into which L(G1) fits? Justify your answer. [Hint: This is a trick question!]

(d) L(G1) is a regular language. It can be described with the regular expression (bb)*.

2. Context free grammars, LL parsing

Consider the following grammar:

E → E + T | T
T→ id | id() | id(L)
L → E, L | E

The nonterminals are E, T, and L. The terminals are “+”, id, “(”, “)”, “,”. The start symbol is E.

Modify the grammar such that it can be parsed by an LL(1) parser.

Show Nullable, FIRST, and FOLLOW and derive the LL(1) parse table for the modified grammar.
Left-factoring and eliminating left recursion yields

1. E -> T E'
2. E' -> + T E'
3. E' ->
4. T -> id T'
5. T' ->
6. T' -> (T"
7. T" ->)
8. T" -> L)
9. L -> E L'
10. L' -> ; L
11. L' ->

b)

	FIRST	FOLLOW	NULLABLE	+	id	(	)	;	$
E	id	$;)	n		1				
E'	+	$;)	y	2			3	3	3
T	id	+ $;)	n		4				
T'	(	+ $;)	y	5		6	5	5	5
T"	) id	+ $;)	n		8		7		
L	id	)	n		9				
L'	;	)	y				11	10	

(c) The parse tree is easy to draw, but hard in ASCII, so I leave

this as exercise again.

Lexical Analysis
Study online at https://quizlet.com/_b70dcy
1. Lexical Analysis: the main task of the _____ is to convert the lexemes into tokens
2. Regular Expressions (RE): The notation for describing a set of the character string.
Ex. if its printf accepts it but if it's Printf it does not
3. What is the output of the lexical analysis?: Tokens
4. What happens if the lexical analyzer finds any invalid tokens?: generates error message.
5. What is the flow of lexical analyzer with parser?: Source code ' Lexical Analyzer ' Syntax Analyzer/Parser
6. When the lexical Analyzer finds any tokens it sends them to _____?: the Parser
7. what happens If token are "()" "," or ":" ?: They get Represented as integer code.
8. For every _____ there is a predefined rule called _____ which identifies if the _____ valid or not.: Lexeme, patterns, tokens
9. What are lexemes?: String of input characters matched for a token.
In other words, everything in the source code is a lexeme.
10. What are Tokens?: Data structure containing token type and value.
In other words, the lexical analyzer turns the lexeme into tokens.
11. What are the most common tokens for Lexical Analyzers?: Identifiers, Keywords, Operators, Special Symbols, Constants
(IKOSC)
12. What are alphabets?: Represent any finite set of symbols
13. What are the three components of alphabets: 1. Binary
2. Hexadecimal
3. English language characters
14. What are strings?: Any finite sequence of alphabets
15. What letter is used for denote empty string?: Epsilon
16. What are some of the common special symbols?: Arithmetic symbols (+,-, *..)
assignment(=)
punctuation(comma , ; , .)

Lexical Analysis
Study online at https://quizlet.com/_b70dcy
17. What happens to all the extra space after lexical analysis?: It gets removed
18. What does the syntax look like of a transformation, given:
a = b + c * 10: <id, pointer to the symbol for a>
<op, '='>
<id, pinter to the symbol for b>
etc...

19. Whenever the scanner comes to a token it cannot understand it yields an error called a _____: Lexical error
20. given the code:
Printf("the value of a is %d",a);: There would be a lexical error for the lexemes for Printf would result in a token that could not be found in the grammar rule.
21. What are Syntax errors?: Error that happens shortly after the lexical error is detected.
22. Syntax Errors require an error recovery solution or a _____: Panic Mode Recovery.
23. What are the 4 possible actions of error recovery?: 1. Deleting the successive character (ex. printff -> printf)
2. Inserting a missing character
(ex. prntf -> printf)
3. Replacing an incorrect character with the correct character (ex. Printf -> printf)
4. Transposing two adjacent characters (ex. rpintf -> printf)
2 / 2