Midterm CS 315
Fall 2013
Instructor: Buğra Gedik

Name: Buğra Gedik
ID: -
Section: I/II

Q1 (10)	
Q2 (20)	
Q3 (20)	
Q4 (20)	
Q5 (30)	
Total (100)	

Exam time: *120 minutes*

a) (2pts) What is orthogonality in programming language design? Give an example to non-orthogonal design in Java.

Orthogonality means relatively small set of primitive constructs can be used in combination (with none or very few edge cases) to build data & control structures.

E.g. of non-orthogonal design in Java:
primitive types cannot be used as generic arguments, no ArrayList<int>

b) (2pts) What are pros and cons of dynamic typing compared to static typing.

Pros:

Flexibility
Less typing

Cons:

Error-prone
Run-time error checking (rather than compile-time)

c) (2pts) What is the fundamental difference between regular expressions and context free grammars? (single sentence please)

CFGs can represent recursion, REs can't.

d) (2pts) What kind of derivations are produced by LL parsers and LR parsers?

LL: leftmost derivation
LR: rightmost derivation in reverse

e) (2pts) What is a VM and what is a JIT compiler?

VM: Software based computer/machine that interprets typically platform independent code

JIT compiler: Run-time compiler within the VM that generates machine code

*Q) What did one regular expression say to another? A) .** – Anonymous

Q2) (20pts) Regular expressions

a) (5pts) Provide regular expressions for recognizing strings of 0s and 1s for which the number of 01 substrings and 10 substrings are equal.
Examples:

- '101' succeeds, because it has ONE '10' and ONE '01'
- '1001' succeeds, because it has ONE '10' and ONE '01'
- '1010' fails, because it has TWO '10's and only ONE '01'
- '10101' succeeds, because it has TWO '10's and TWO '01's

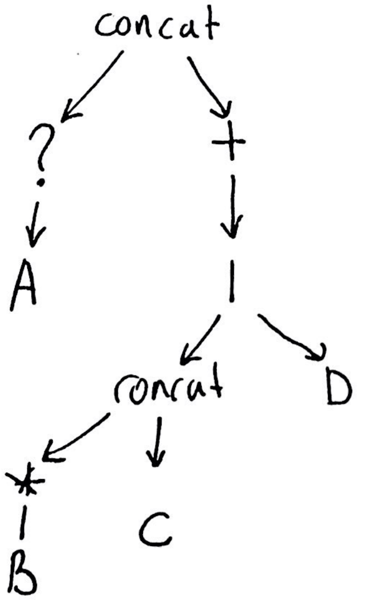
$(1+0+)*1+ \mid (0+1+)*0+$

$0+(1+0+)* \mid 1+(0+1+)*$

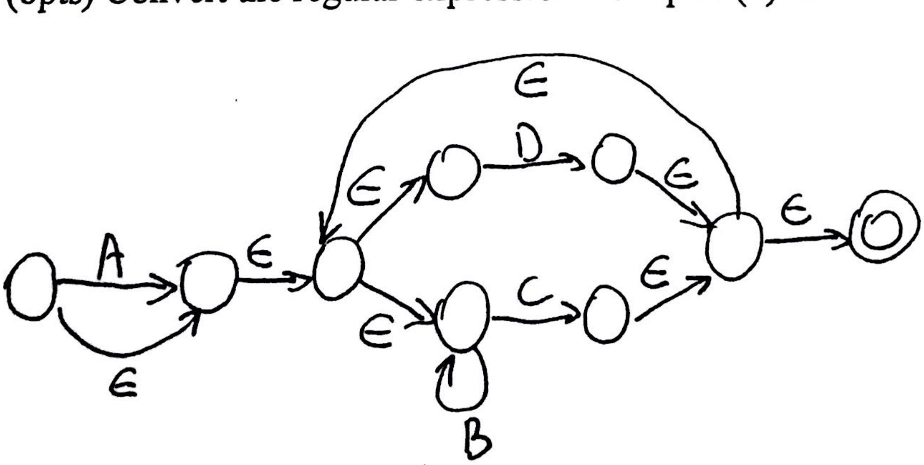
000110000101100

b) (7pts) Draw the parse tree for the regular expression A? (B*C|D)+

000 - 01 1 - 100



c) (8pts) Convert the regular expression from part (b) to an NFA.



"People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones." – Donald Knuth

Q3) (20pts) Grammars

Consider the following unambiguous grammar, where `ID` is a token, and the rest are non-terminals:

```
aexp -> aexp '&' texp | texp
texp -> uexp '%' texp | uexp
uexp -> ID
```

a) (5pts) While keeping the grammar unambiguous, add a third operator '#', which is a unary operator applied on the right (as in x#), and has higher precedence than the other two operators. Also, make changes to support parenthesized expressions via '(' and ')'.

aexp → aexp '&' texp | texp

texp → uexp '%' texp | uexp

$\boxed{\text{uexp → uexp '\#' | zexp}}$

zexp → '(' aexp ')' | ID

+/−

*//

**

b) (5pts) Write down the associativities and precedence levels of the operators. The available options for Associativity are {Left, Right, DoNotApply}. The available optios for Precedence are {High, Medium, Low}.

	Associativity	Precedence
%	Right	Medium
#	DoNotApply	High
&	Left	Low

c) (5pts) Give a rightmost derivation for the following:

`x % y % (w & u)#`

aexp ⇒ [redacted] texp

⇒ uexp '%' texp

⇒ zexp '%' texp

⇒ ID '%' texp

⇒ ID '%' uexp '%' texp

⇒ ID '%' zexp '%' texp

⇒ ID '%' ID '%' texp

⇒ ID '%' ID '%' uexp

⇒ ID '%' ID '%' uexp '#'

⇒ ID '%' ID '%' zexp '#'

⇒ ID '%' ID '%' '(' aexp ')' '#'

⇒ ID '%' ID '%' '(' aexp '&' texp ')' '#'

⇒ ID '%' ID '%' '(' texp '&' texp ')' '#

⇒ ID '%' ID '%' '(' uexp '&' texp ')' '#

⇒ ID '%' ID '%' '(' zexp '&' texp ')' '#

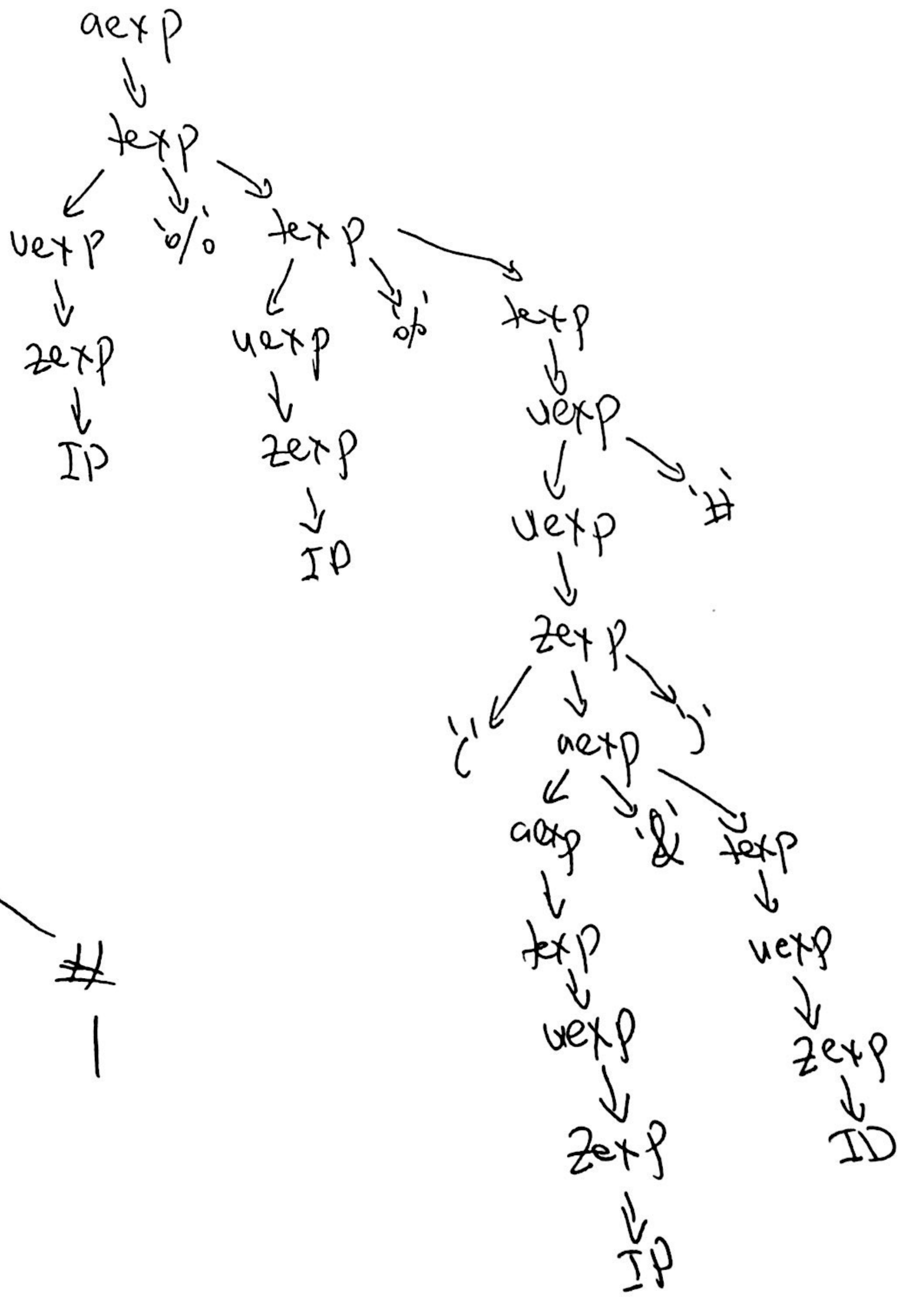
⇒ ID '%' ID '%' '(' ID '&' texp ')'

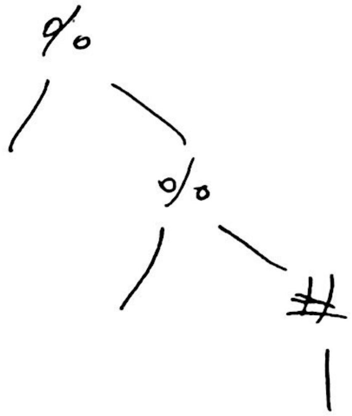
⇒ ID '%' ID '%' '(' ID '&' uexp ')'

⇒ ID '%' ID '%' '(' ID '&' zexp

⇒ ID '%' ID '%' '(' ID '&' ID'

d) (5pts) Draw the parse tree for the expression from part c).





"My definition of an expert in any field is a person who knows enough about what's ... to be scared." – P. J. Plauger

Q4) (20pts) Bottom-up parsing

Consider the below grammar:

```
regex  -> regex '|' term
        | term
term   -> term factor
        | factor
factor -> factor '*'
        | exp
exp    -> CHAR
        | '(' regex ')'
```

For the input `(A|B)*C` draw the bottom-up parsing steps for an LR parser. At each step, show the partial trees. Number the steps.

1) (

2) (A

3) (CHAR
A

4) (exp
CHAR
...

5) (factor
exp
char

6) (term
factor
...

7) (regex
term
...

8) (regex |
term
...

9) (regex | B
term
...

10) (regex | CHAR
term B
...

11) (regex | exp
term CHAR
... ...

12) (regex | factor
term exp
... ...

13) (regex | term
term factor
... ...

14) (regex
regex | term
...

15) (regex)
regex | term
...

16) exp
(regex)
...

17) factor
exp
...

18) factor *
exp
...

19) factor
factor *
...

20) term
factor
...

21) term C
factor
...

22) term CHAR
factor C
...

23) term exp
factor CHAR
...

24) term factor
factor exp
...

25) term
term factor
...

26) regex
term

"...ing complexity is the essence of computer programming." – Brian Kernighan

Q5) (30pts) Left factoring and LL parsing

Consider the following grammar:

```
1. SetLiteral → '{' SetLiteralBody '}'
2. SetLiteralBody → ε
3. SetLiteralBody → NonEmptyBody
4. NonEmptyBody → SetElement
5. NonEmptyBody → SetElement ',' NonEmptyBody
6. SetElement → ID
7. SetElement → SetLiteral
```

a) (6pts) Left factor this grammar (i.e., factor out the shared RHS prefix from the rules 4 and 5). Write each rule on a separate line with a unique number (i.e., do not use | to fold two rules into one). Name the new non-terminal you introduce as Tail.

```
1. SetLiteral → '{' SetLiteralBody '}'
2. SetLiteralBody → ε
3. SetLiteralBody → NonEmptyBody
4. NonEmptyBody → SetElement Tail
5. Tail → ε
6. Tail → ',' NonEmptyBody
7. SetElement → ID
8. SetElement → SetLiteral
```

b) (6pts) For each rule (there should be 8 of them), compute the PREDICT() function. That is, find the set of look ahead characters for which the rule can be applied.

PREDICT(1): '{' PREDICT(2): '}' PREDICT(3): ID, '{' PREDICT(4): ID, '{'

PREDICT(5): '}' PREDICT(6): ',' PREDICT(7): ID PREDICT(8): '{'

c) (6pts) Create an LL parse table for the new grammar.

	'{'	'}'	','	ID
SetLiteral	1	—	—	—
SetLiteralBody	3	2	—	3
NonEmptyBody	4	—	—	4
Tail	—	5	6	—
SetElement	8	—	—	7

d) (8pts) Parse the following input using the parse table.

Stack	*Input*	*Rule*

SetElement Tail }	A,{B},{}}	7
ID Tail }	A,{B},{}}	Eat A
Tail }	,{B},{}}	6
, NonEmptyBody }	,{B},{}}	Eat ,
NonEmptyBody }	{B},{}}	4
SetElement Tail }	{B},{}}	8
SetLiteral Tail }	{B},{}}	1
{ SetLiteralBody } Tail }	{B},{}}	Eat {
SetLiteralBody } Tail }	B},{}}	3
NonEmptyBody } Tail }	B},{}}	4
SetElement Tail } Tail }	B},{}}	7
ID Tail } Tail }	B},{}}	Eat B
Tail } Tail }	},{}}	5
} Tail }	},{}}	Eat }
Tail }	,{}}	6
, NonEmptyBody }	,{}}	Eat ,
NonEmptyBody }	{}}	4
SetElement Tail }	{}}	8
SetLiteral Tail }	{}}	1
{ SetLiteralBody } Tail }	{}}	Eat }
SetLiteralBody } Tail }	}}	2
} Tail }	}}	Eat }
Tail }	}	5
}	}	Eat }
DONE	DONE	DONE

e) (4pts) Draw the parse tree