Final Exam: May 17 Name: _____

You will have 3 hours for this exam, although you should not need that long. This exam is open-book and open-note (including DyKnow, and the copy of the textbook on Moodle, but not tools such as Kojo, JFLAP, or Logisim). Please take some time to check your work. If you need extra space, write on the back. You must show your work to receive any partial credit. There are a total of 40 points on this exam.

1. (10 points) Let R be the regular expression $(a^* \mid (ba))^*b$ over the alphabet $\{a, b\}$.

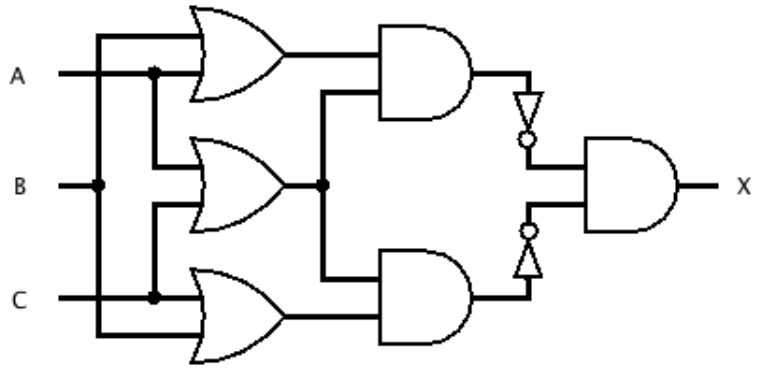
   (a) Which of the following strings will be matched by R?

   ε a b $abba$ $baab$ $aabbaab$ $abababa$ $bababab$

   (b) State a simple condition that describes the strings matched by R:

   (c) Give a deterministic finite automaton that accepts the same set of strings as matched by R:

2. (10 points) Consider the following circuit:



(a) Give a truth table describing the output X in terms of the inputs A, B, and C:

(b) What is the gate delay of this circuit (assume AND, OR, and NOT gates all have a delay of 1)?

(c) Draw an equivalent circuit with fewer gates *and* a shorter gate delay. Do not use any gates with more than two inputs:

3. (5 points) Here is the simplified turtle drawing language from Exam 1:

```
sealed trait Drawing
case class Rect(w: Double, h: Double) extends Drawing
case class Beside(d1: Drawing, d2: Drawing) extends Drawing
case class Above(d1: Drawing, d2: Drawing) extends Drawing
```

Recall that `Rect(w, h)` produces a rectangle extending from $(0, 0)$ to (w, h), `Beside(d1, d2)` shifts `d2` to the right so that its origin is on the right-hand edge of the bounding box of `d1`, and `Above(d1, d2)` shifts `d2` up so that its origin is on the top edge of the bounding box of `d1`.

Furthermore, here is a type of trees containing integer values only at the leaves:

```
sealed trait Tree
case class Leaf(n: Int) extends Tree
case class Node(left: Tree, right: Tree) extends Tree
```

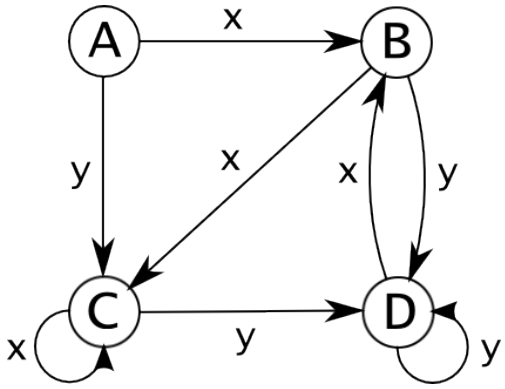
If we had a function `total(t: Tree): Int` that added up all of the numbers in the leaves of `t`, then we could write the following function to render the tree as a drawing:

```
def render(t: Tree): Drawing = t match {
  case Leaf(n) => Rect(n, 1)
  case Node(left, right) => Above(Rect(total(t), 1),
                                  Beside(render(left),
                                         render(right))
}
```

(a) Write the desired function `total(t: Tree): Int` that adds up all the numbers in `t`. For example, the function call `total(Node(Node(Leaf(1), Leaf(2)), Leaf(3)))` should return `6`:

(b) What drawing is produced by `render(Node(Node(Leaf(1), Leaf(2)), Leaf(3)))`? Use a scale where 1 unit is roughly 1 cm.

4. (10 points) Consider the following finite state machine (start and final states are not relevant for this problem, so none are marked):



(a) Finish filling in the following table, where the entry for row i, column j is the shortest (possibly empty) string of input which will take the machine from state i to state j. If there are several shortest strings, list them all. If there is no string going from i to j, put a dash (—).

	A	B	C	D
A	ε	x		
B	—			
C				
D				

(b) Give a regular expression which describes *all* of the input strings (of any length) that will take the machine from D to itself:

5. (5 points) Computer programmers often need to handle unknown values: for example, functions may throw an exception, or go into an infinite loop, rather than return a value. One way to deal with this is to use a "ternary," or "three-valued," logic.[1] The database query language SQL supports a NULL value to represent an unknown field in a database record. Correspondingly, the ternary logic used to evaluate conditions in SQL has the third possibility UNKNOWN in addition to TRUE and FALSE.

   Here is a Scala implementation of ternary logic:

```
sealed trait Ternary
case object TRUE extends Ternary
case object FALSE extends Ternary
case object UNKNOWN extends Ternary

def And(p: Ternary, q: Ternary): Ternary = (p, q) match {
    case (TRUE, TRUE) => TRUE
    case (FALSE, _) => FALSE
    case (_, FALSE) => FALSE
    case _ => UNKNOWN
}

def Or(p: Ternary, q: Ternary): Ternary = (p, q) match {
    case (TRUE, _) => TRUE
    case (_, TRUE) => TRUE
    case (FALSE, FALSE) => FALSE
    case _ => UNKNOWN
}

def Not(p: Ternary): Ternary = p match {
    case TRUE => FALSE
    case FALSE => TRUE
    case _ => UNKNOWN
}
```

   (a) Evaluate `Not(And(FALSE, UNKNOWN))`:

   (b) Evaluate `Or(Not(FALSE), Not(UNKNOWN))`:

   (c) Does the De Morgan law `Not(And(p,q)) == Or(Not(p),Not(q))` hold for all values of `p` and `q`?

   (d) Does the Excluded Middle law `Or(p,Not(p)) == TRUE` hold for all values of `p`?

[1] Interestingly, one of the early investigators of three-valued logic was Jan Łukasiewicz, who also introduced the "Polish" prefix notation for expressions. The ternary logic seen here is due to Stephen Kleene, who also invented regular expressions.