Initial_____

Université d'Ottawa
Faculté de génie

École d'ingénierie et de technologie de l'information

uOttawa

L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Information Technology and Engineering

COURSE:	SEG2101 Software Construction	**PROF:**	Jiying Zhao
SEMESTER:	WINTER 2006	**DATE:** **TIME:**	April 28, 2006 09:30 – 12:30

FINAL EXAMINATION

NAME	
STUDENT NUMBER	

Final Exam

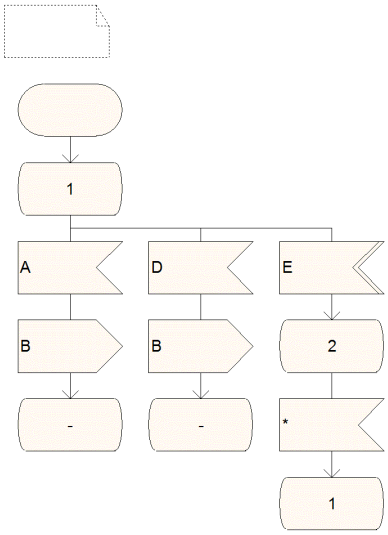
1. There are three (3) types of questions in this examination.

Part 1	**Multiple choice**	**16 marks**	
Part 2	**Short answer**	**52 marks**	
Part 3	**Problem solving**	**18 marks**	
Total		**86 marks**	

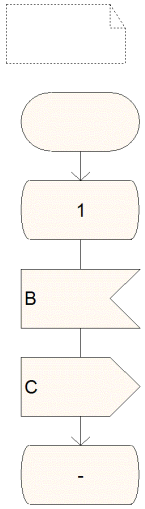
2. The space allocated for each question is limited. In case of necessity you may use the other side of the pages to continue.
3. Initial all the pages.
4. Submit back all the pages.

- **Multiple choice questions [2 marks each, please circle the best answer]:**

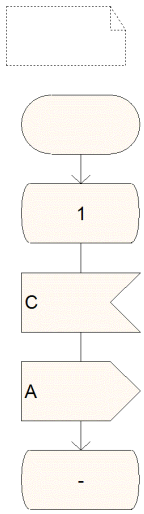
process final1_1



process final1_2



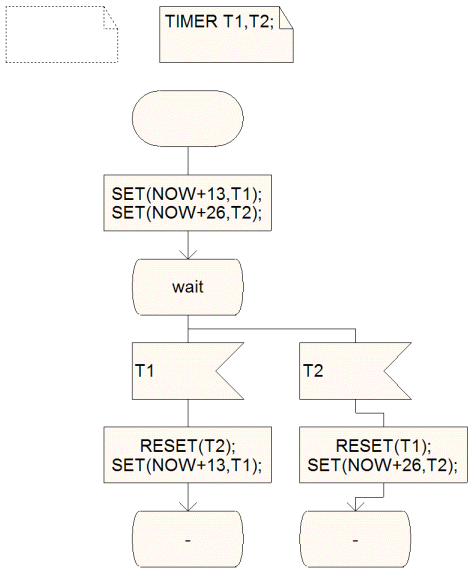
process final1_3



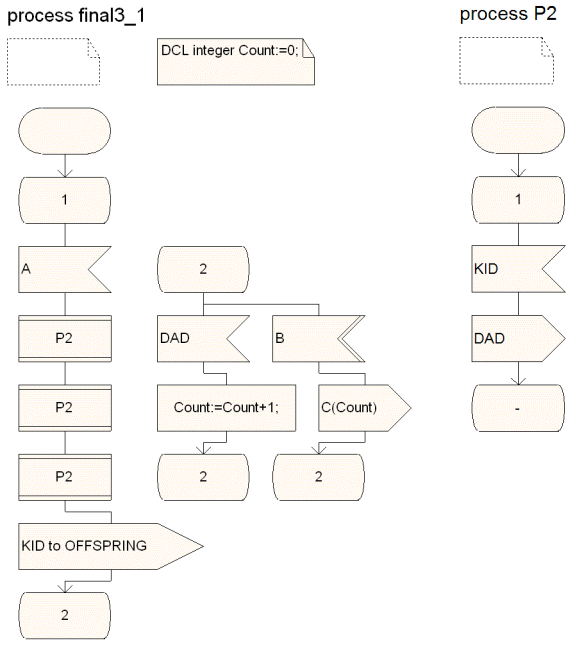
1. Refer to the diagram above. Signal D and E are from the environment. Which of the following **is not** correct?
   a) Suppose final1_1, final1_2, and final1_3 have been just initialized. When there is no signal sent from the environment, final1_1, final1_2, and final1_3 will not be able to output any signals.
   b) Suppose final1_1, final1_2, and final1_3 stay at state 1 and are not generating any signal. When signal D is sent from the environment, final1_1, final1_2, and final1_3 will start to output signals.
   c) When final1_1, final1_2, and final1_3 start generating signals and there is no further signal sent from the environment, the three processes will not stop and will generate signals endlessly.
   d) Whenever signal E is sent from the environment, the three processes will always stop generating signals.
   e) At least one of the above (a,b,c,d) is wrong.

2. Refer to process final 2. In 100 time unites, how many T2 signal(s) will be generated?
   a) 0
   b) 1
   c) 2
   d) 3

process final2



3. Refer to the diagram below (final3_1 and P2). Signals A and B are from the environment. Sinal C is to the environment. Long time after inputting signal A, the user inputs signal B. What value will be returned by the signal C (the integer value carried by C)?
   a) 0 b) 1
   c) 2 d) 3



4. Refer to the grammar below.

```
S --> NP VP
NP --> N
NP --> Det N
Det --> the
N --> man
N --> dog
N --> cat
VP --> V NP
V --> bites
V --> catches
```

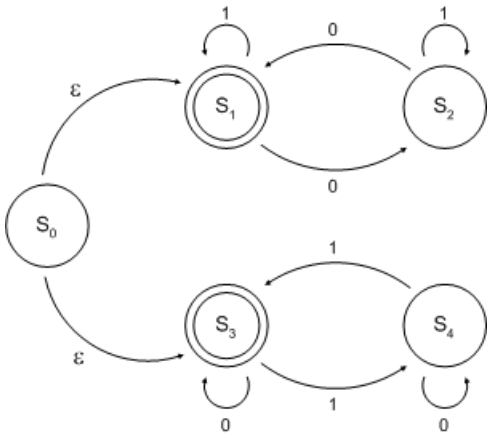
   The grammar belongs to which of the following?
   a) Regular grammar b) Context-free grammar
   c) Context sensitive grammar d) Unrestricted grammar

5. Consider the following grammar rules:
   `S → 0 A 0 S | 1 0`
   `A → 1 | 0 S 0 A`
   Which of the following strings is not a member of this language?
   a) `001001010` b) `10`
   c) `01010` d) `0010110`

6. Which of the following is not described by the regular definition below?

```
D → [1234567890]
N → D+ ( . D+ )? (E( +|- )? D+ )?
a) 1.894                          b) .894E-4
c) 1.894E+4                       d) 1.894E4
```

7. Which of the following cannot be recognized by the following NFA?

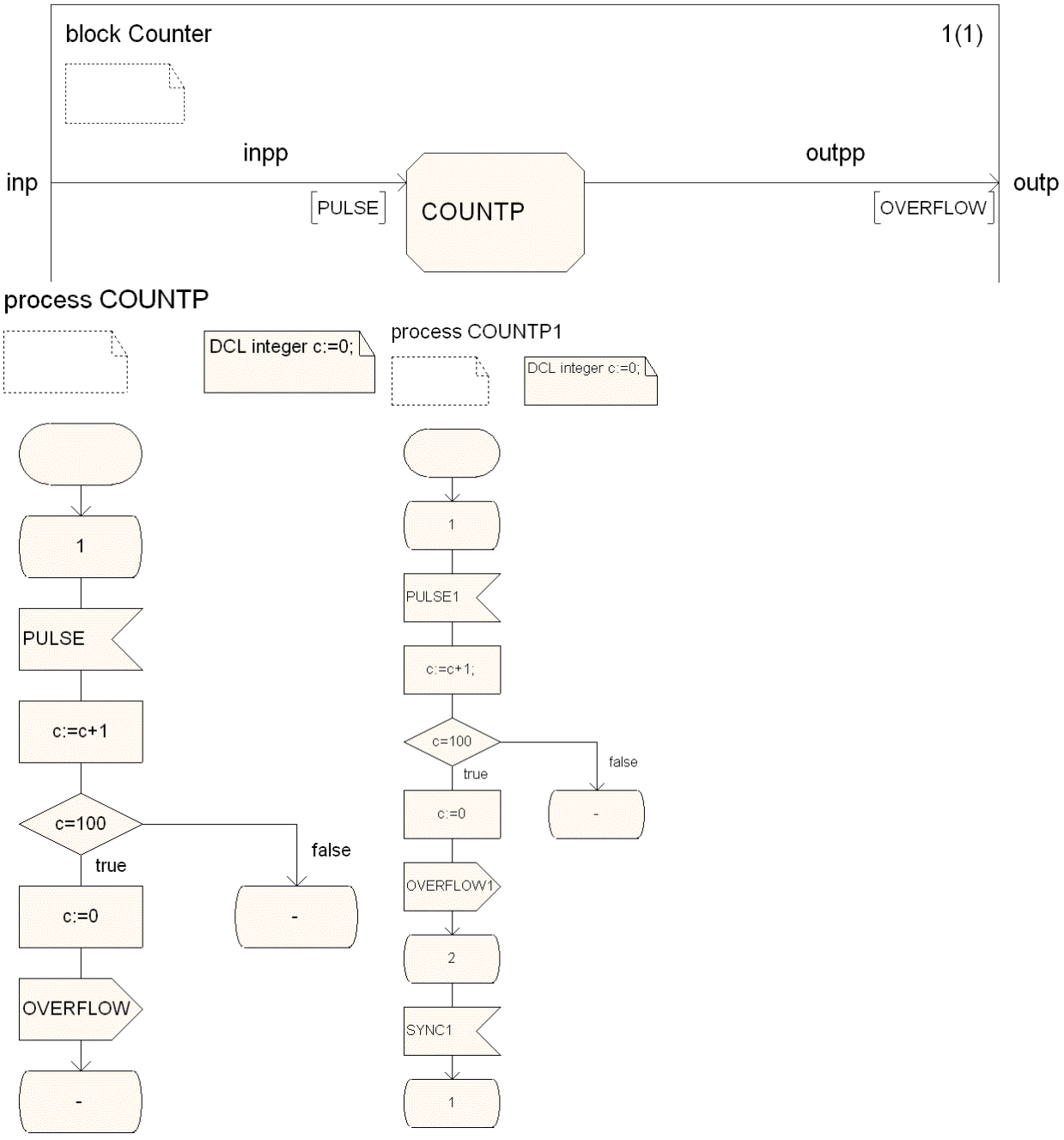


a) An empty string
b) 00001111
c) 10101010
d) 01010111

8. Let Σ={a,b}, which of the following **is not** correct? (where `r`, `s` and `t` are regular expressions)
   a) Regular expression `(a|b)(a|b)` denotes {aa,ab,ba,bb}.
   b) `r**=r*`
   c) `r|(s|t)=(r|s)|t`
   d) `(a|b)*` is not equal to `(a|b)**`

- **Short-answer questions**

9. **[8 marks]** A SDL block named COUNTER consists of one process COUNTP. There is one input signal (PULSE) to the process, and one output signal (OVERFLOW) from the process. Whenever an input signal PULSE is received, the process COUNTP will add 1 to its variable c. The initial value for the variable c is 0. The counting range is from 0 to 99. That is, the value of c goes from 0 to 1 to 2, …, to 99, when it reaches 100 it resets to 0, then the counter will continue to count from 0 to 99 repeatedly. Whenever the value of c is reset from 100 to 0, an output signal OVERFLOW will be generated.

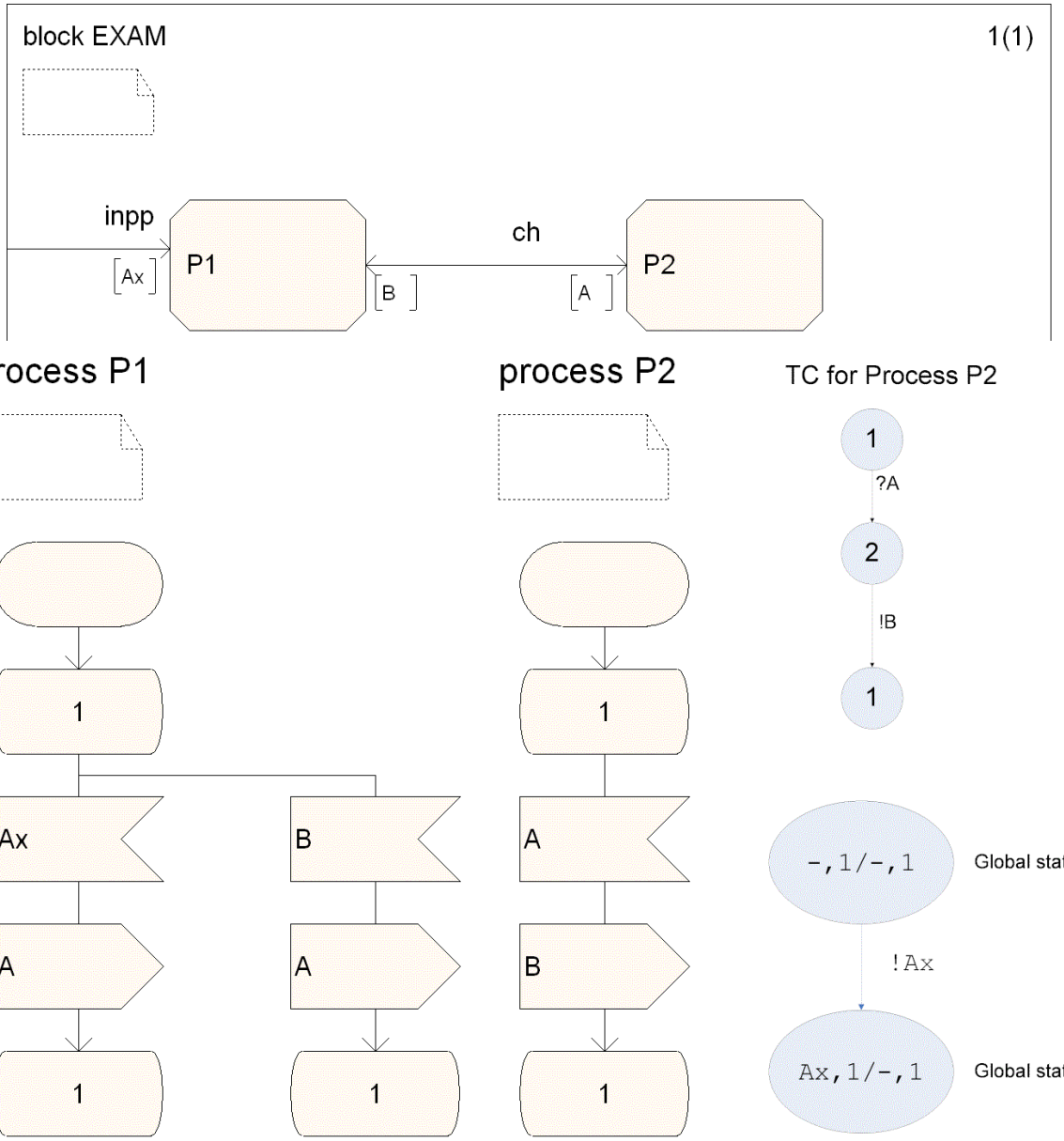


Now you are asked to implement a block similar to COUNTER. That is, you should design a new block, called B-COUNTER, slightly more complex than block COUNTER. The B-COUNTER block should contain four processes (instead of a single one). Three of the four processes (namely, COUNTP1, COUNTP2, and COUNTP3) are the same as

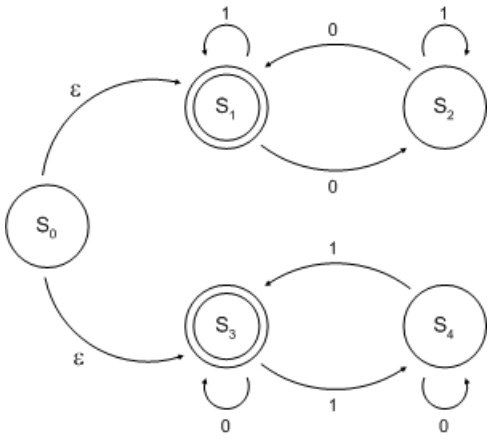
process COUNTP except for the names of the input and output signals (see COUNTP1). That is, each of these three processes counts input signals (namely, PULSE1, PULSE2, and PULSE3 are respectively the inputs for COUNTP1, COUNTP2, and COUNTP3) and generate an output signal (namely, OVERFLOW1, OVERFLOW2, and OVERFLOW3 are respectively the output signals from COUNTP1, COUNTP2, and COUNTP3) when the counter reaches 100. COUNTP1, COUNTP2, and COUNTP3 wait for a signal SYNC1, SYNC2, and SYNC3, respectively, from the fourth process before going back to state 1. The fourth process (namely, BARRIER) takes OVERFLOW1, OVERFLOW2, and OVERFLOW3 as inputs, and sends an OVERFLOW signal to the environment when the three OVERFLOW1, OVERFLOW2, and OVERFLOW3 signals have been received. BARRIER sends a SYNC1, SYNC2, or SYNC3 signal to COUNTP1, COUNTP2, or COUNTP3 when it receives an OVERFLOW1, OVERFLOW2, or OVERFLOW3, respectively. After sending out OVERFLOW, the process BARRIER goes back to the previous state to get ready for the next iteration.

a) Draw the SDL structure diagram of the B-COUNTER block. Draw all processes, channels, signal routes, signals, adding labels, as appropriate.
b) Draw the SDL behavior diagram for the process BARRIER in the BLOCK B-COUNTER. Draw all states, inputs, and outputs, adding labels, as appropriate.

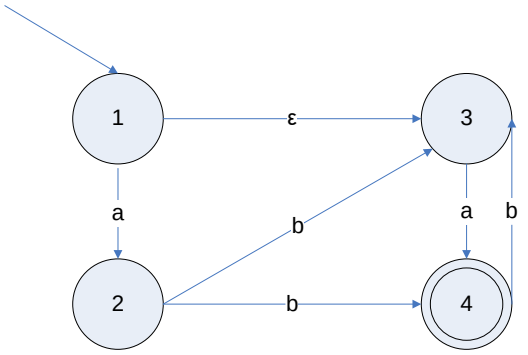
10. **[4 marks]** Refer to the following figure.
    a) Draw the Transition Chart (TC) for process P1.
    b) Draw a global reachability graph showing at least five (5) global states starting from the global state 2.



11. **[4 marks]** Write the corresponding regular expression for the following finite state machine.



12. **[6 marks]** Closely follow Algorithm 3.2 (attached as Appendix A) to convert the following NFA into a DFA by showing the intermediate steps.



13. **[4 marks]** The following grammar has left recursion (refer to Appendix B for possible methods). Find an equivalent grammar (that generates the same language) without left recursion.

    <expression> → <expression> **+** <factor>
    <expression> → <expression> **-** <factor>
    <expression> → <factor>

14. **[4 marks]** Consider the following grammar (0 and 1 are terminals). Left-factor the grammar.

    <expr> -> <term> ADD <expr> | <term> SUB <expr> | <term>
    <term> -> <factor> MUL <term> | <factor> DIV < term > | <factor> | NUM

    Note: ADD, SUB, MUL, DIV, and NUM are terminals.

15. **[4 marks]** Show that the following grammar is ambiguous.

    S → A | S **+** A | S **+** **+** A
    A → a | ε

16. **[10 marks]** Refer to Appendices C, D and F. Given the following grammar:

```
S → id = c ;
S → if (c) {LS} E
LS → S LS
LS → ε
E → else {LS}
E → ε
```

a) Compute the sets FIRST and FOLLOW for all the non-terminals in the above grammar.
b) Build a parsing table for the grammar.
c) Is the grammar LL(1)? – Explain.

17. **[8 marks]** Refer to Appendix E. The table below is the parsing table for the following LL(1) grammar:

S→(L) | a
L→SL′
L′→,SL′ | ε

	(	a	)	,	$	
S	S→(L)	S→a				
L	L→SL′	L→SL′				
L′			L′→ε	L′→,SL′		

Seek the moves made by a nonrecursive predictive parser on the input "(a,(a,a))", by filling the following table.

Stack	**Input**	**Output**
	(a,(a,a))$	

- **Problem Solving**

18. **[10 marks] Define a Java class representing a monitor that controls the access to a resource for reader and writer processes.** We consider the following problem: A resource is used by two different kinds of processes. The Reader processes access the resource without changing its content. The Writer processes access the resource and update the content of the resource. In order to provide a consistent view of the resource to the Readers, no Writer should access the resource while some reader uses the resource. It is also required that at most one Writer process has access to the resource at any given time.

    It is assumed that the Readers and Writers use a monitor Database to synchronize their access to the resource. The monitor offers the following methods: startRead(), endRead(), startWrite(), endWrite(); they are used as follows: Before accessing the resource, a Reader will call startRead(), then it will access the resource and then call the method endRead(). Similarly, a Writer will first call startWrite(), then it will update the resource and then call the method endWrite().

    The following is the incomplete program. Your task is to complete method `startRead()` and `startWrite()`. Please give Writers priority over Readers. That is, if a writer is waiting no new reader may enter the resource (even if other readers are currently using the resource).

```
public class Database
{
   private int readerCount;

   private boolean dbReading;
   private boolean dbWriting;

   public Database()
   {
      readerCount = 0;
      dbReading = false;
      dbWriting = false;
   }

   public synchronized int startRead()
   {









   }

   public synchronized int endRead()
   {
      --readerCount;

      if (readerCount == 0)
         dbReading = false;

      notifyAll();

        System.out.println("Reader Count = " + readerCount);

      return readerCount;
   }
```

```
    public synchronized void startWrite()
    {









    }

    public synchronized void endWrite()
    {
       dbWriting = false;

       notifyAll();
    }

}
```

19. **[8 marks]** The following is a multithreading program. In order to provide fairness to all the five threads, you are asked to modify the `run` method of the classes `PrintChar` and `PrintNum` so that any of the 5 threads must stop before the `for` loop and cannot proceed until all other threads reach this point. This can be implemented in a way similar to a semaphore or a monitor. Please write the class that provides the needed method(s) and make the necessary modification to the following program.

```
public class TestRunnable
{
  public static void main(String[] args)
  {
    Thread printA = new Thread(new PrintChar('a',100));
    Thread printB = new Thread(new PrintChar('b',100));
    Thread printC = new Thread(new PrintChar('c',100));
    Thread print100_1 = new Thread(new PrintNum(100));
    Thread print100_2 = new Thread(new PrintNum(100));

    print100_1.start();
    print100_2.start();
    printA.start();
    printB.start();
    printC.start();
  }
}

class PrintChar implements Runnable
{
  private char charToPrint;
  private int times;

  public PrintChar(char c, int t)
  {
    charToPrint = c;
    times = t;
  }

  public void run()
  {
    for (int i=1; i < times; i++)
      System.out.print(charToPrint);
  }
}

class PrintNum implements Runnable
{
  private int lastNum;

  public PrintNum(int n)
  {
    lastNum = n;
  }

  public void run()
  {
    for (int i=1; i <= lastNum; i++)
      System.out.print(" " + i);
  }
}
```

Algorithm 3.2. (*Subset construction.*) Constructing a DFA from an NFA.

Input. An NFA N.

Output. A DFA D accepting the same language.

Method. Our algorithm constructs a transition table *Dtran* for D. Each DFA state is a set of NFA states and we construct *Dtran* so that D will simulate "in parallel" all possible moves N can make on a given input string.

We use the operations in Fig. 3.24 to keep track of sets of NFA states (s represents an NFA state and T a set of NFA states).

OPERATION	DESCRIPTION
ϵ-*closure*(s)	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
ϵ-*closure*(T)	Set of NFA states reachable from some NFA state s in T on ϵ-transitions alone.
move(T, a)	Set of NFA states to which there is a transition on input symbol a from some NFA state s in T.

Fig. 3.24. Operations on NFA states.

Before it sees the first input symbol, N can be in any of the states in the set ϵ-*closure*(s_0), where s_0 is the start state of N. Suppose that exactly the states in set T are reachable from s_0 on a given sequence of input symbols, and let a be the next input symbol. On seeing a, N can move to any of the states in the set *move*(T, a). When we allow for ϵ-transitions, N can be in any of the states in ϵ-*closure*(*move*(T, a)), after seeing the a.

```
initially, ε-closure(s0) is the only state in Dstates and it is unmarked;
while there is an unmarked state T in Dstates do begin
    mark T;
    for each input symbol a do begin
        U := ε-closure(move(T, a));
        if U is not in Dstates then
            add U as an unmarked state to Dstates;
        Dtran[T, a] := U
    end
end
```

Fig. 3.25. The subset construction.

We construct *Dstates*, the set of states of D, and *Dtran*, the transition table for D, in the following manner. Each state of D corresponds to a set of NFA

states that N could be in after reading some sequence of input symbols including all possible ϵ-transitions before or after symbols are read. The start state of D is ϵ-*closure*(s_0). States and transitions are added to D using the algorithm of Fig. 3.25. A state of D is an accepting state if it is a set of NFA states containing at least one accepting state of N.

```
push all states in T onto stack;
initialize ε-closure(T) to T;
while stack is not empty do begin
      pop t, the top element, off of stack;
      for each state u with an edge from t to u labeled ε do
            if u is not in ε-closure(T) do begin
                  add u to ε-closure(T);
                  push u onto stack
            end
end
```

Fig. 3.26. Computation of ϵ-*closure*.

The computation of ϵ-*closure*(T) is a typical process of searching a graph for nodes reachable from a given set of nodes. In this case the states of T are the given set of nodes, and the graph consists of just the ϵ-labeled edges of the NFA. A simple algorithm to compute ϵ-*closure*(T) uses a stack to hold states whose edges have not been checked for ϵ-labeled transitions. Such a procedure is shown in Fig. 3.26. □

Example 3.15. Figure 3.27 shows another NFA N accepting the language $(a|b)^*abb$. (It happens to be the one in the next section, which will be mechanically constructed from the regular expression.) Let us apply Algorithm 3.2 to N. The start state of the equivalent DFA is ϵ-*closure*(0), which is $A = \{0, 1, 2, 4, 7\}$, since these are exactly the states reachable from state 0 via a path in which every edge is labeled ϵ. Note that a path can have no edges, so 0 is reached from itself by such a path.

The input symbol alphabet here is $\{a, b\}$. The algorithm of Fig. 3.25 tells us to mark A and then to compute

$$\epsilon\text{-}closure(move(A, a)).$$

We first compute $move(A, a)$, the set of states of N having transitions on a from members of A. Among the states 0, 1, 2, 4 and 7, only 2 and 7 have such transitions, to 3 and 8, so

$$\epsilon\text{-}closure(move(\{0, 1, 2, 4, 7\}, a)) = \epsilon\text{-}closure(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$$

Let us call this set B. Thus, $Dtran[A, a] = B$.

Among the states in A, only 4 has a transition on b to 5, so the DFA has a transition on b from A to

- **Appendix B: Elimination of left recursion**

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$

can be replaced by

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$

- **Appendix C: Computation of FIRST**

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. If X is terminal, then FIRST(X) is $\{X\}$.
2. If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).
3. If X is nonterminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), . . . , FIRST(Y_{i-1}); that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$, then add ϵ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add FIRST($Y_2$) and so on.

- **Appendix D: Computation of FOLLOW**

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol and \$ is the input right endmarker.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except for ϵ is placed in FOLLOW(B).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ (i.e., $\beta \stackrel{*}{\Rightarrow} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B).

- **Appendix E: Nonrecursive predictive parsing**

Input. A string w and a parsing table M for grammar G.

Output. If w is in $L(G)$, a leftmost derivation of w; otherwise, an error indication.

Method. Initially, the parser is in a configuration in which it has \$$S$ on the stack with S, the start symbol of G on top, and w\$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input is shown in Fig. 4.14. □

```
set ip to point to the first symbol of w$;
repeat
    let X be the top stack symbol and a the symbol pointed to by ip;
    if X is a terminal or $ then
        if X = a then
            pop X from the stack and advance ip
        else error()
    else    /* X is a nonterminal */
        if M[X, a] = X → Y1Y2 · · · Yk then begin
            pop X from the stack;
            push Yk, Yk-1, . . . , Y1 onto the stack, with Y1 on top;
            output the production X → Y1Y2 · · · Yk
        end
        else error()
until X = $    /* stack is empty */
```

- **Appendix F: Construction of a predictive parsing table**

Algorithm 4.4. Construction of a predictive parsing table.

Input. Grammar G.

Output. Parsing table M.

Method.

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to $M[A, a]$.
3. If ϵ is in FIRST(α), add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in FOLLOW(A). If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, \$]$.
4. Make each undefined entry of M be **error**.

Initial_____

Université d'Ottawa
Faculté de génie

École d'ingénierie et de technologie de l'information

uOttawa

L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Information Technology and Engineering

COURSE:	SEG2101 Software Construction	**PROF:**	Jiying Zhao
		DATE:	April 28, 2006
SEMESTER:	WINTER 2006	**TIME:**	09:30 – 12:30

Solutions

FINAL EXAMINATION

NAME	
STUDENT NUMBER	

Final Exam

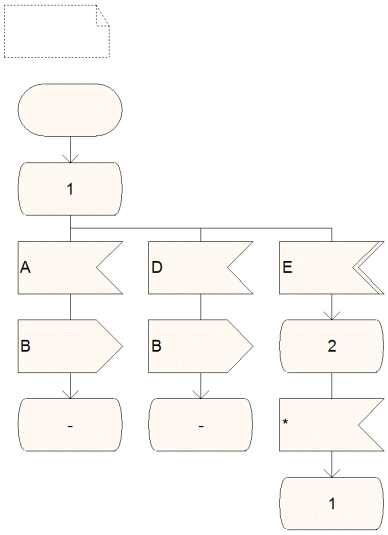
1. There are three (3) types of questions in this examination.

Part 1	**Multiple choice**	**16 marks**	
Part 2	**Short answer**	**52 marks**	
Part 3	**Problem solving**	**18 marks**	
Total		**86 marks**	

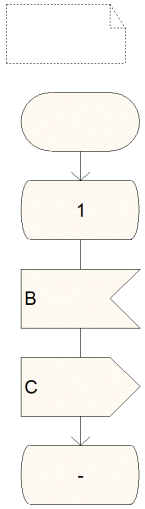
2. The space allocated for each question is limited. In case of necessity you may use the other side of the pages to continue.
3. Initial all the pages.
4. Submit back all the pages.

- **Multiple choice questions [2 marks each, please circle the best answer]:**

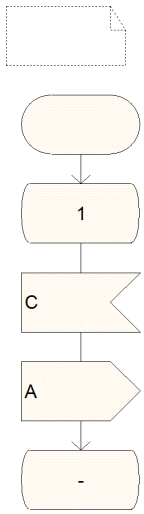
process final1_1



process final1_2



process final1_3

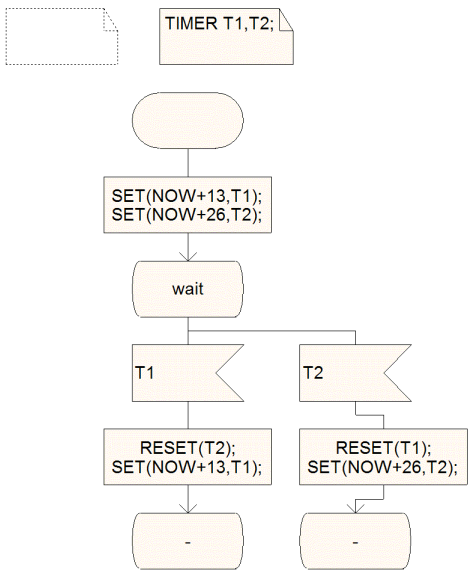


1. Refer to the diagram above. Signal D and E are from the environment. Which of the following **is not** correct?
   a) Suppose final1_1, final1_2, and final1_3 have been just initialized. When there is no signal sent from the environment, final1_1, final1_2, and final1_3 will not be able to output any signals.
   b) Suppose final1_1, final1_2, and final1_3 stay at state 1 and are not generating any signal. When signal D is sent from the environment, final1_1, final1_2, and final1_3 will start to output signals.
   c) When final1_1, final1_2, and final1_3 start generating signals and there is no further signal sent from the environment, the three processes will not stop and will generate signals endlessly.
   d) Whenever signal E is sent from the environment, the three processes will always stop generating signals.
   e) At least one of the above (a,b,c,) is wrong.

ASSUMPTION : after initialization and after E is sent, no other signal is sent from environment => state 2 is waiting for any signal, but none arrive. All processes shown will halt.

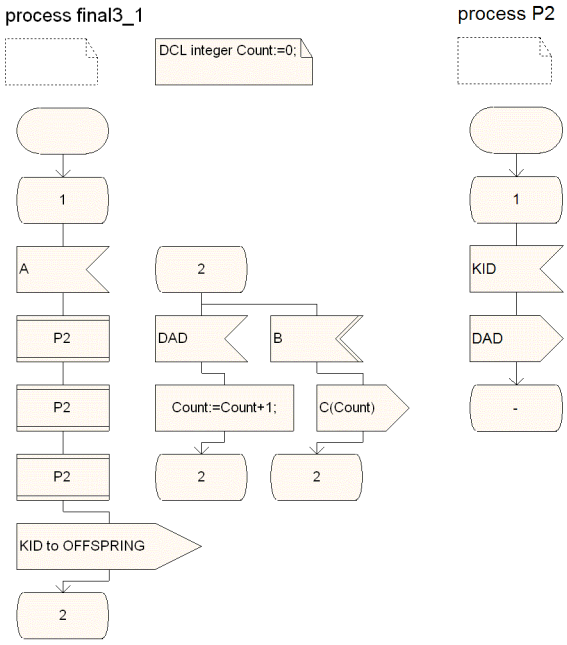
2. Refer to process final 2. In 100 time unites, how many T2 signal(s) will be generated?
   a) 0
   b) 1
   c) 2
   d) 3

process final2



T2 will be reset (becoming inactive) when T1 expires the first time.

3. Refer to the diagram below (final3_1 and P2). Signals A and B are from the environment. Sinal C is to the environment. Long time after inputting signal A, the user inputs signal B. What value will be returned by the signal C (the integer value carried by C)?
   a) 0 b) 1
   c) 2 d)



The message KID is sent to the last offspring (the last process instance created at this point by the process final2_1. The process has created 3 instances at this point, only the last one will receive this signal. Therefore only this last one will return a DAD message and the counter will be incremented only once

4. Refer to the grammar below.

```
S --> NP VP
NP --> N
NP --> Det N
Det --> the
N --> man
N --> dog
N --> cat
VP --> V NP
V --> bites
V --> catches
```

   The grammar belongs to which of the following?
   a) Regular grammar b) Context-free grammar
   c) Context sensitive grammar d) Unrestricted grammar

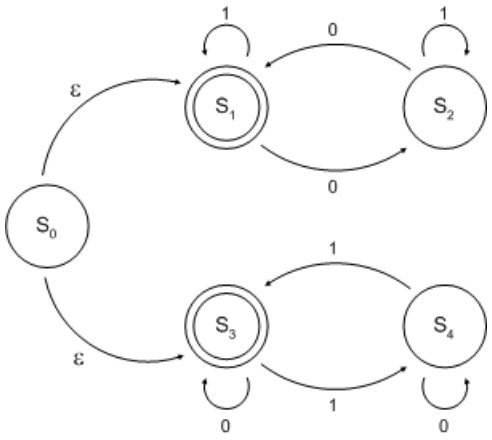
5. Consider the following grammar rules:
   `S → 0 A 0 S | 1 0`
   `A → 1 | 0 S 0 A`
   Which of the following strings is not a member of this language?
   a) 001001010 b) 10
   c) 01010 d) 0010110 --- 0010 requires 0 after

6. Which of the following is not described by the regular definition below?

```
D → [1234567890]
N → D+ ( . D+ )? (E( +|- )? D+ )?
```

a) `1.894`
b) `.894E-4`
c) `1.894E+4`
d) `1.894E4`

7. Which of the following cannot be recognized by the following NFA?



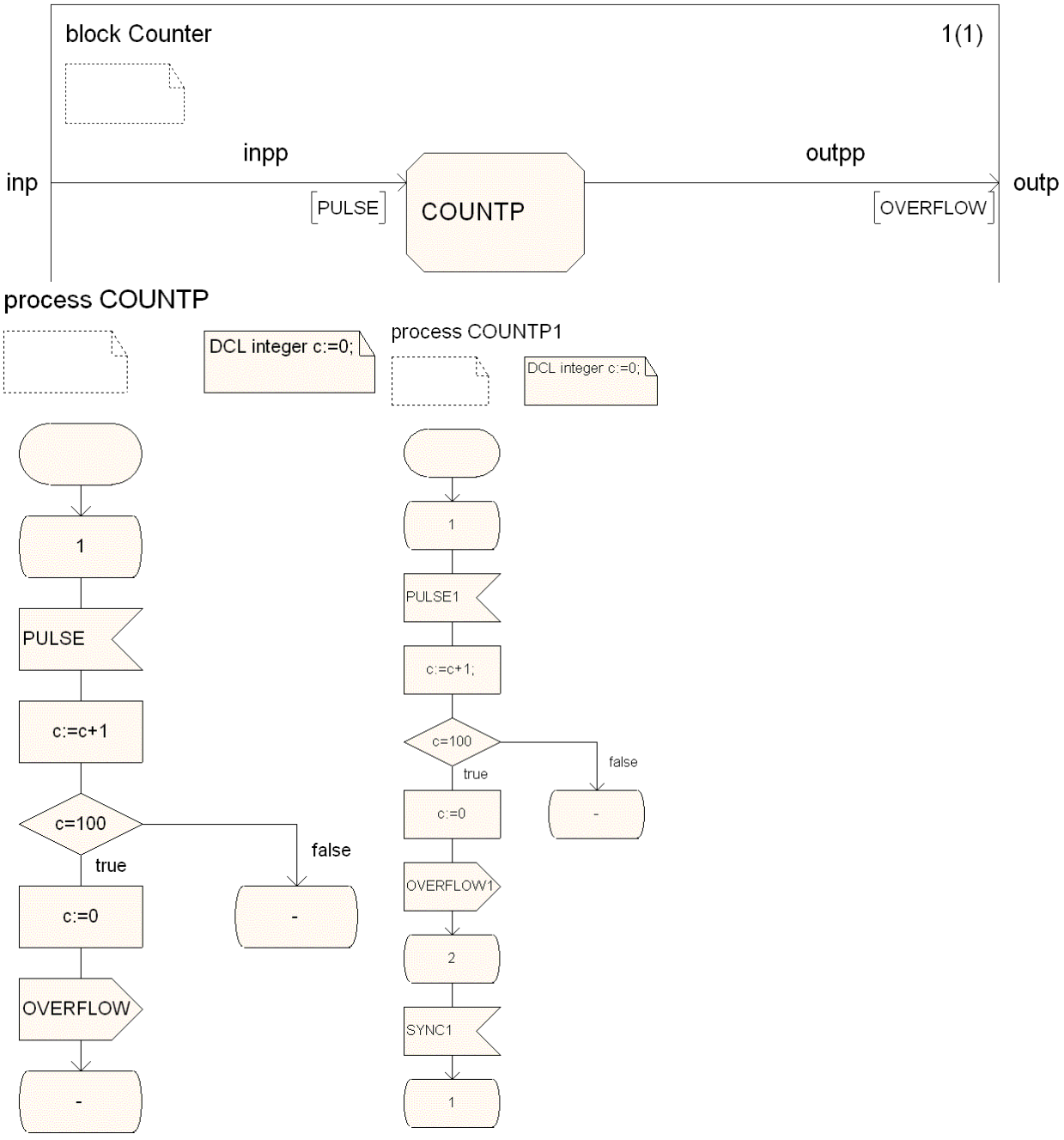
a) An empty string
b) 00001111
c) 10101010
d) 01010111

8. Let Σ=`{a,b}`, which of the following **is not** correct? (where `r`, `s` and `t` are regular expressions)

a) Regular expression `(a|b)(a|b)` denotes `{aa,ab,ba,bb}`.
b) `r**=r*`
c) `r|(s|t)=(r|s)|t`
d) `(a|b)*` is not equal to `(a|b)**`

- **Short-answer questions**

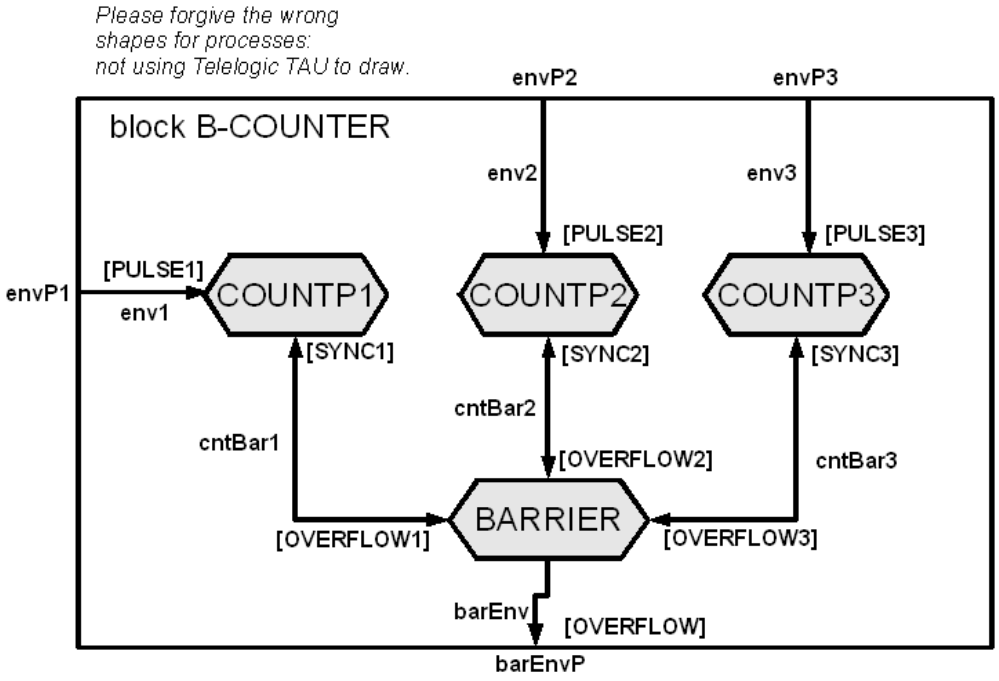
9. **[8 marks]** A SDL block named COUNTER consists of one process COUNTP. There is one input signal (PULSE) to the process, and one output signal (OVERFLOW) from the process. Whenever an input signal PULSE is received, the process COUNTP will add 1 to its variable c. The initial value for the variable c is 0. The counting range is from 0 to 99. That is, the value of c goes from 0 to 1 to 2, …, to 99, when it reaches 100 it resets to 0, then the counter will continue to count from 0 to 99 repeatedly. Whenever the value of c is reset from 100 to 0, an output signal OVERFLOW will be generated.

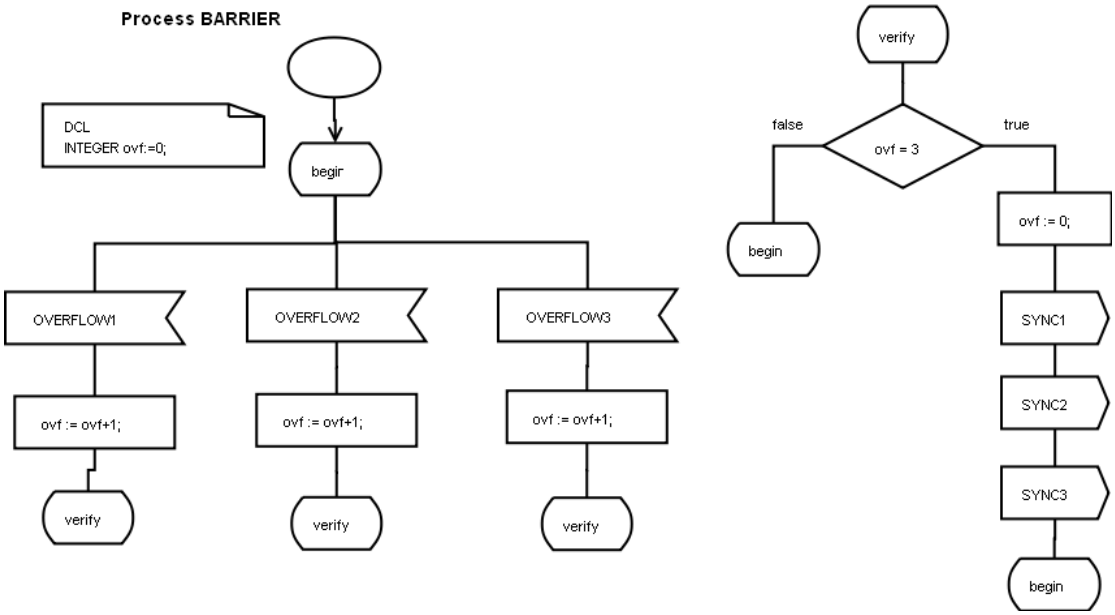


Now you are asked to implement a block similar to COUNTER. That is, you should design a new block, called B-COUNTER, slightly more complex than block COUNTER. The B-COUNTER block should contain four processes (instead of a single one). Three of the four processes (namely, COUNTP1, COUNTP2, and COUNTP3) are the same as

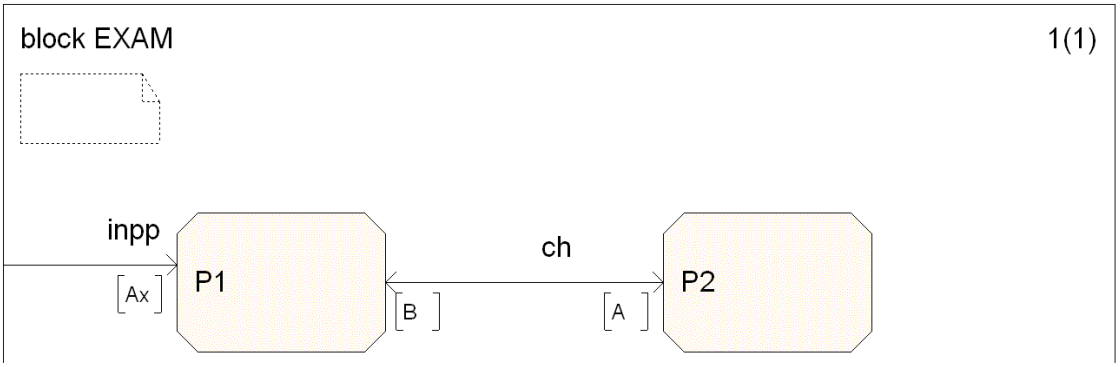
process COUNTP except for the names of the input and output signals (see COUNTP1). That is, each of these three processes counts input signals (namely, PULSE1, PULSE2, and PULSE3 are respectively the inputs for COUNTP1, COUNTP2, and COUNTP3) and generate an output signal (namely, OVERFLOW1, OVERFLOW2, and OVERFLOW3 are respectively the output signals from COUNTP1, COUNTP2, and COUNTP3) when the counter reaches 100. COUNTP1, COUNTP2, and COUNTP3 wait for a signal SYNC1, SYNC2, and SYNC3, respectively, from the fourth process before going back to state 1. The fourth process (namely, BARRIER) takes OVERFLOW1, OVERFLOW2, and OVERFLOW3 as inputs, and sends an OVERFLOW signal to the environment when the three OVERFLOW1, OVERFLOW2, and OVERFLOW3 signals have been received. BARRIER sends a SYNC1, SYNC2, or SYNC3 signal to COUNTP1, COUNTP2, or COUNTP3 when it receives an OVERFLOW1, OVERFLOW2, or OVERFLOW3, respectively. After sending out OVERFLOW, the process BARRIER goes back to the previous state to get ready for the next iteration.

a) Draw the SDL structure diagram of the B-COUNTER block. Draw all processes, channels, signal routes, signals, adding labels, as appropriate.
b) Draw the SDL behavior diagram for the process BARRIER in the BLOCK B-COUNTER. Draw all states, inputs, and outputs, adding labels, as appropriate.

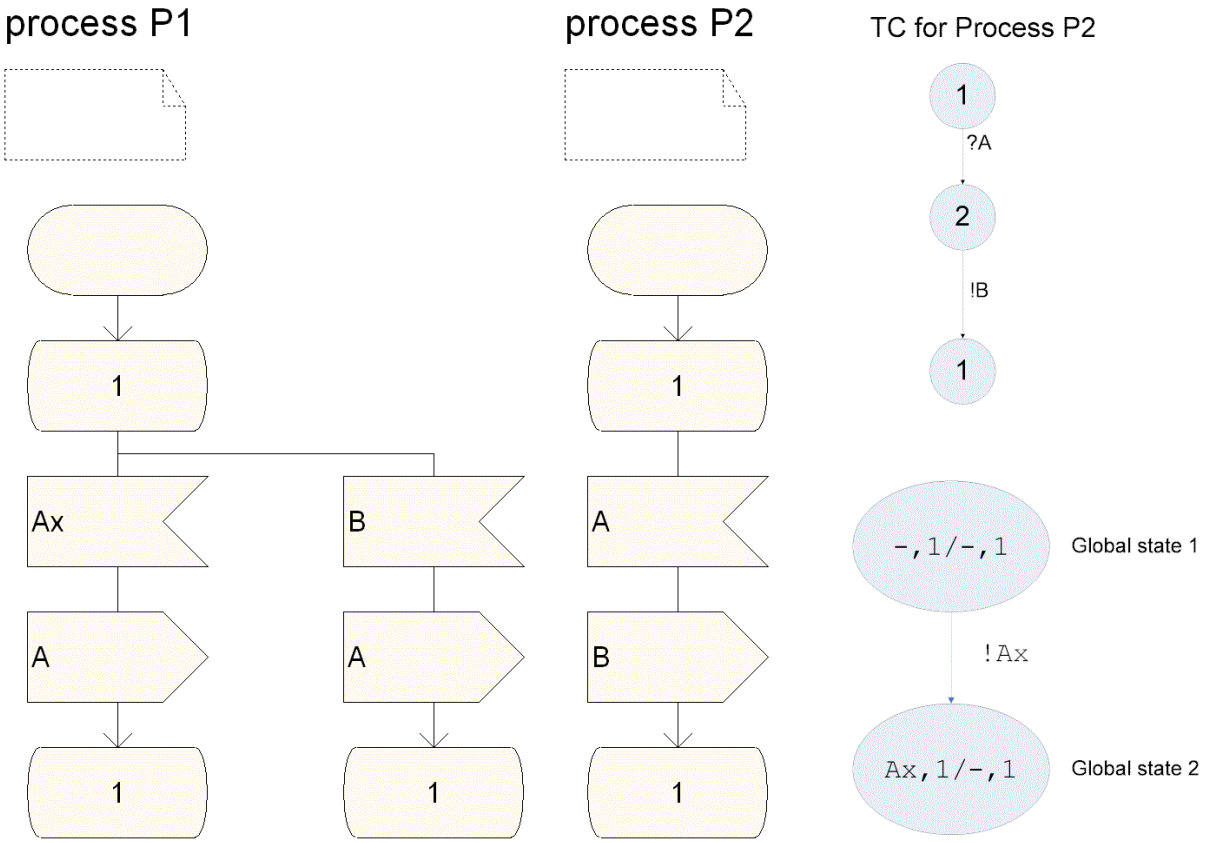


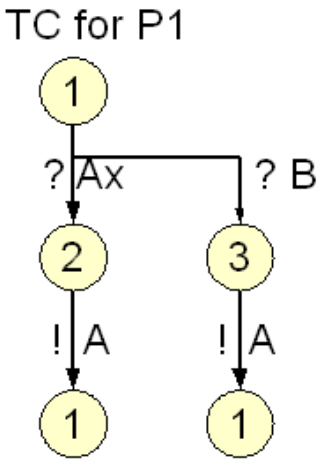

Process BARRIER
DCL
INTEGER ovf:=0;
begin
OVERFLOW1
OVERFLOW2
OVERFLOW3
ovf := ovf+1;
ovf := ovf+1;
ovf := ovf+1;
verify
verify
verify
verify
false
ovf = 3
true
begin
ovf := 0;
SYNC1
SYNC2
SYNC3
begin

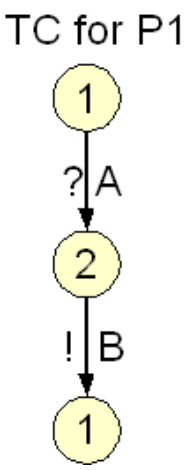
10. **[4 marks]** Refer to the following figure.
    a) Draw the Transition Chart (TC) for process P1.
    b) Draw a global reachability graph showing at least five (5) global states starting from the global state 2

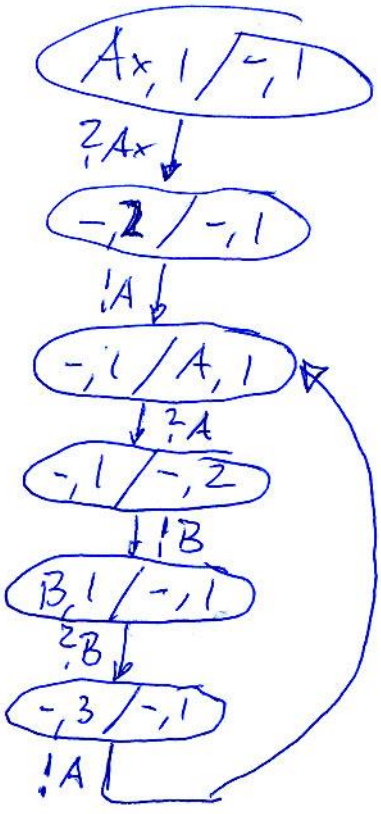


c)

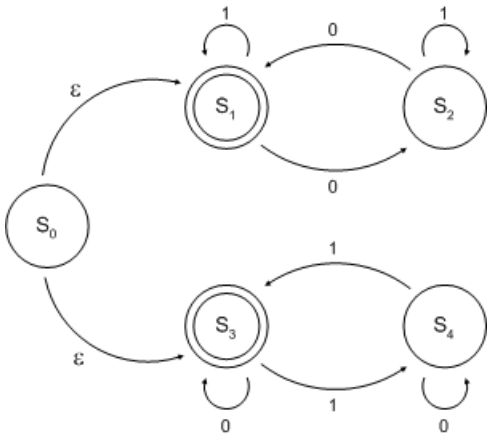



TC for P1
1
? Ax
? B
2
3
! A
! A
1
1


TC for P1
1
? A
2
! B
1

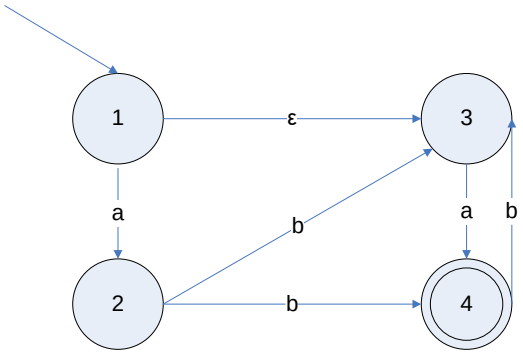

Ax, 1 / -, 1
?, Ax
-, 2 / -, 1
!A
-, 1 / A, 1
?, A
-, 1 / -, 2
!B
B, 1 / -, 1
?B
-, 3 / -, 1
!A

11. **[4 marks]** Write the corresponding regular expression for the following finite state machine.



[1* (0 1* 0)* 1*] | [0* (1 0* 1)* 0*]

12. **[6 marks]** Closely follow Algorithm 3.2 (attached as Appendix A) to convert the following NFA into a DFA by showing the intermediate steps.



Non-deterministic: State 2 has 2 'b' transitions.

For verification later - this machine realises the regular expression:
a b? (ba)*

Start with e-closures from beginning:
A = { 1, 3 }

From there, e-closure on any state arrived at with an 'a'
B = { 2, 4 } - *stop*

From A, e-closure on any state arrived at with a 'b'
C = {} : *not used*

From B, e-closure on any state arrived at with an 'a'
D = {} : *not used*

From B, e-closure on any state arrived with a 'b'
E = { 3, 4 } - *stop*

---------- *all states visited : finish-up* ----------

From E, e-closure on any state arrived at with 'a'

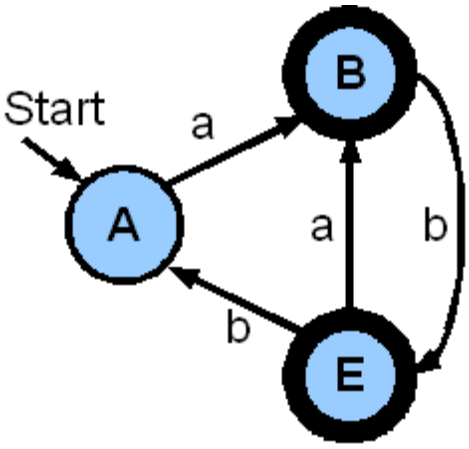
F = { 4 } - stop & redundant : ***use B***

From E, e-closure on any state arrived at wit a 'b'
G = { 3 } - redundant : ***use A***

Note, the other way with {x,y,z} in the resulting nodes is the exact same thing, just a different notation.

	Transitions	
States	'a'	'b'
A	B	
B	-	E
E	B	A

Final result:



Verification :
a b? (ba)*
yes, realized.

13. **[4 marks]** The following grammar has left recursion (refer to Appendix B for possible methods). Find an equivalent grammar (that generates the same language) without left recursion.

```
<expression> → <expression> + <factor>
<expression> → <expression> - <factor>
<expression> → <factor>
```

```
<expression> → <factor> <Exp'>
<Exp'> → + <factor> <Exp'> | ε
<Exp'> → - <factor> <Exp'> | ε
```

14. **[4 marks]** Consider the following grammar (0 and 1 are terminals). Left-factor the grammar.

```
<expr> → <term> ADD <expr> | <term> SUB <expr> | <term>
<term> → <factor> MUL <term> | <factor> DIV < term > | <factor> | NUM
```

Note: `ADD`, `SUB`, `MUL`, `DIV`, and `NUM` are terminals.

```
<expr> → <term> ( a | b | ε )
a → ADD <expr>
b → SUB <expr>
<term> → <factor> MUL <term> | <factor> DIV < term > | <factor> | NUM
```

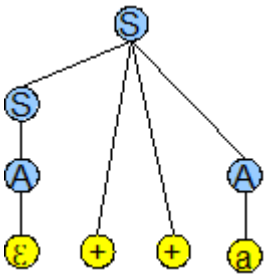
NOTE : cannot factor more directly, BUT:

```
<term> → c | d
c → <factor> ( e|f )
e → MUL <term>
f → DIV < term >
d → NUM
```

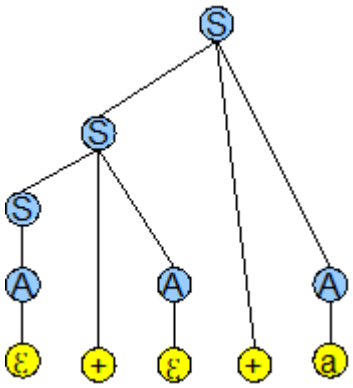
15. **[4 marks]** Show that the following grammar is ambiguous.

```
S → A | S + A | S + + A
A → a | ε
```

"++a" is given by :



or by :



16. **[10 marks]** Refer to Appendices C, D and F. Given the following grammar:

```
S → id = c ;
S → if (c) {LS} E
LS → S LS
LS → ε
E → else {LS}
E → ε
```

a) Compute the sets FIRST and FOLLOW for all the non-terminals in the above grammar.
b) Build a parsing table for the grammar.
c) Is the grammar LL(1)? – Explain.

Important note:
The following solution is wrong – please ignore it.
Gregor v. Bochmann, April 15, 2009

a)

RECALL : productions are of the form
X → aSb where a and b can be strings of characters
Therefore, there are 4 terminals :

1. "id = c ;"
2. "if (c) {"
3. "else {"
4. "}"

First(S) = { "id = c ;", "if (c) {" }
First(LS) = First(S) = { "id = c ;", "if (c) {" }
First(E) = { "else {", ε }

//add 'before root' production to satisfy follow rules :

```
PRE → S$
```

Follow(S) = First(LS) //because S on right side is only followed by non-terminal LS
= { $, "id = c ;", "if (c) {" }
Follow(LS) = {"}"} //straight-forward
Follow(E) = Follow(S) //because E on right is only followed by void
= { $, "id = c ;", "if (c) {" }

Refer to T.A.'s notes for "plain-language" explanation.
Another walk-through example can be found at:
www.jambe.co.nz/UN/FirstAndFollowSets.html

b)

	"id = c ;"	"if (c) {"	"else {"	"}"
S	S → id = c ;	S → if (c) {LS} E		
LS			E → else {LS}	
E				S → if (c) {LS} E

c)

Yes, the two conditions for LL(1) are satisfied (please check course notes).

17. **[8 marks]** Refer to Appendix E. The table below is the parsing table for the following LL(1) grammar:

S→(L) | a
L→SL′
L′→,SL′ | ε

	(	a	)	,	$	
S	S→(L)	S→a				
L	L→SL′	L→SL′				
L′			L′→ε	L′→,SL′		

Seek the moves made by a nonrecursive predictive parser on the input "(a,(a,a))", by filling the following table.

Stack	**Input**	**Output**
S$	(a,(a,a))$	S → (L)
(L)$	(a,(a,a))$	
L)$	a,(a,a))$	L → SL′
SL')$	a,(a,a))$	
SL')$	a,(a,a))$	S → a
aL')$	a,(a,a))$	
L')$	,(a,a))$	L' → ,SL'
,SL')$	,(a,a))$	
SL')$	(a,a))$	S → (L)
(L)L')$	(a,a))$	
L)L')$	a,a))$	L → SL'
SL')L')$	a,a))$	S → a
aL')L')$	a,a))$	
L')L')$	,a))$	L' → ,SL'
,SL')L')$	,a))$	
SL')L')$	a))$	S → a
aL')L')$	a))$	
L')L')$	))$	L → ε
ε)L')$	))$	
)L')$	))$	
L')$	)$	L → ε
ε)$	)$	
)$	)$	
$	$	ACCEPT!

For an explanation, please refer to the link:

http://web.cs.wpi.edu/~kal/PLT/PLTex4.4.html

Note that they "flip" the characters in the Stack column in order to eliminate from the right. Here, I am eliminating terminals from the left, so no need to flip.

- **Problem Solving**

18. **[10 marks] Define a Java class representing a monitor that controls the access to a resource for reader and writer processes.** We consider the following problem: A resource is used by two different kinds of processes. The Reader processes access the resource without changing its content. The Writer processes access the resource and update the content of the resource. In order to provide a consistent view of the resource to the Readers, no Writer should access the resource while some reader uses the resource. It is also required that at most one Writer process has access to the resource at any given time.

    It is assumed that the Readers and Writers use a monitor Database to synchronize their access to the resource. The monitor offers the following methods: startRead(), endRead(), startWrite(), endWrite(); they are used as follows: Before accessing the resource, a Reader will call startRead(), then it will access the resource and then call the method endRead(). Similarly, a Writer will first call startWrite(), then it will update the resource and then call the method endWrite().

    The following is the incomplete program. Your task is to complete method `startRead()` and `startWrite()`. ***Please give Writers priority over Readers.*** That is, if a writer is waiting no new reader may enter the resource (even if other readers are currently using the resource).

```
public class Database
{
   private void readerCount;
        //missing : ensure no writer is waiting
        private volatile int writerCount;

   private boolean dbReading;
   private boolean dbWriting;

   public Database()
   {
        readerCount = 0;
        writerCount=0;
        dbReading = false;
        dbWriting = false;
   }

   public synchronized void startRead()
   {
       while( dbWriting || dbReading ){ // ERROR: the text "|| dbReading" should
be deleted.
              wait();
        }
        //notify does not mean no writer is waiting
        // => ensure no writer is in queue:
        while(  writerCount > 0 ){
              wait();
        }
        //at this point, readerCount should necessarily be == 0
        ++readerCount;
        dbReading = true;
   }

   public synchronized int endRead()
   {
        --readerCount;
        if (readerCount == 0){
              dbReading = false;
        }
        notifyAll();
        System.out.println("Reader Count = " + readerCount);
        return readerCount;
   }
```

```
    public synchronized void startWrite()
    {
        ++writerCount;      //>0 => writers in queue :no reading permitted

        //if ressource is used by ANYONE, wait
        while( dbWriting || dbReading ){
               wait();
        }

        dbWriting = true;

        //the notify all in end Write might have set dBWriting to false
        //either way, set it to true to prevent other
        //readers/writers to access the resource
    }

    public synchronized void endWrite()
    {
       dbWriting = false;
          --writerCount;
       notifyAll();
    }

}
```

19. **[8 marks]** The following is a multithreading program. In order to provide fairness to all the five threads, you are asked to modify the `run` method of the classes `PrintChar` and `PrintNum` so that any of the 5 threads must stop before the `for` loop and cannot proceed until all other threads reach this point. This can be implemented in a way similar to a semaphore or a monitor. Please write the class that provides the needed method(s) and make the necessary modification to the following program.

join

```
public class TestRunnable
{
  public static void main(String[] args)
  {
    Thread printA = new Thread(new PrintChar('a',100));
    Thread printB = new Thread(new PrintChar('b',100));
    Thread printC = new Thread(new PrintChar('c',100));
    Thread print100_1 = new Thread(new PrintNum(100));
    Thread print100_2 = new Thread(new PrintNum(100));

    print100_1.start();
    print100_2.start();
    printA.start();
    printB.start();
    printC.start();
  }
}

class PrintChar implements Runnable
{
  private char charToPrint;
  private int times;

  public PrintChar(char c, int t)
  {
    charToPrint = c;
    times = t;
  }

  public void run()
  {
    JoinSem.join();
    for (int i=1; i < times; i++)
      System.out.print(charToPrint);
  }
}

class PrintNum implements Runnable
{
  private int lastNum;

  public PrintNum(int n)
  {
    lastNum = n;
  }

  public void run()
  {
    JoinSem.join();
    for (int i=1; i <= lastNum; i++)
      System.out.print(" " + i);
  }
}
```

Many solutions possible.

Proposed : a semaphore-like class that notifies all 5 only when all 5 have requested to 'join'.

Here, the 'protected ressource' (by definition of semaphore) is simply the ability to continue.

```
class JoinSem{

    private static final int numberJoiners = 5;
    private static int haveJoined = 0;

    public static synchronized void join(){
        haveJoined++;
        if( haveJoined < numberJoiners ){
            wait();
        }else{
            haveJoined = 0;
            notifyAll();
        }
    }//end constructor

}//end class joinSem
```

- **Appendix A: Algorithm 3.2**

Algorithm 3.2. (*Subset construction.*) Constructing a DFA from an NFA.

Input. An NFA N.

Output. A DFA D accepting the same language.

Method. Our algorithm constructs a transition table *Dtran* for D. Each DFA state is a set of NFA states and we construct *Dtran* so that D will simulate "in parallel" all possible moves N can make on a given input string.

We use the operations in Fig. 3.24 to keep track of sets of NFA states (s represents an NFA state and T a set of NFA states).

OPERATION	DESCRIPTION
ϵ-*closure*(s)	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
ϵ-*closure*(T)	Set of NFA states reachable from some NFA state s in T on ϵ-transitions alone.
move(T, a)	Set of NFA states to which there is a transition on input symbol a from some NFA state s in T.

Fig. 3.24. Operations on NFA states.

Before it sees the first input symbol, N can be in any of the states in the set ϵ-*closure*(s_0), where s_0 is the start state of N. Suppose that exactly the states in set T are reachable from s_0 on a given sequence of input symbols, and let a be the next input symbol. On seeing a, N can move to any of the states in the set *move*(T, a). When we allow for ϵ-transitions, N can be in any of the states in ϵ-*closure*(*move*(T, a)), after seeing the a.

```
initially, ε-closure(s0) is the only state in Dstates and it is unmarked;
while there is an unmarked state T in Dstates do begin
    mark T;
    for each input symbol a do begin
        U := ε-closure(move(T, a));
        if U is not in Dstates then
            add U as an unmarked state to Dstates;
        Dtran[T, a] := U
    end
end
```

Fig. 3.25. The subset construction.

We construct *Dstates*, the set of states of D, and *Dtran*, the transition table for D, in the following manner. Each state of D corresponds to a set of NFA

states that N could be in after reading some sequence of input symbols including all possible ϵ-transitions before or after symbols are read. The start state of D is ϵ-*closure*(s_0). States and transitions are added to D using the algorithm of Fig. 3.25. A state of D is an accepting state if it is a set of NFA states containing at least one accepting state of N.

```
push all states in T onto stack;
initialize ε-closure(T) to T;
while stack is not empty do begin
        pop t, the top element, off of stack;
        for each state u with an edge from t to u labeled ε do
            if u is not in ε-closure(T) do begin
                add u to ε-closure(T);
                push u onto stack
            end
end
```

Fig. 3.26. Computation of ϵ-*closure*.

The computation of ϵ-*closure*(T) is a typical process of searching a graph for nodes reachable from a given set of nodes. In this case the states of T are the given set of nodes, and the graph consists of just the ϵ-labeled edges of the NFA. A simple algorithm to compute ϵ-*closure*(T) uses a stack to hold states whose edges have not been checked for ϵ-labeled transitions. Such a procedure is shown in Fig. 3.26. □

Example 3.15. Figure 3.27 shows another NFA N accepting the language $(a|b)^*abb$. (It happens to be the one in the next section, which will be mechanically constructed from the regular expression.) Let us apply Algorithm 3.2 to N. The start state of the equivalent DFA is ϵ-*closure*(0), which is $A = \{0, 1, 2, 4, 7\}$, since these are exactly the states reachable from state 0 via a path in which every edge is labeled ϵ. Note that a path can have no edges, so 0 is reached from itself by such a path.

The input symbol alphabet here is $\{a, b\}$. The algorithm of Fig. 3.25 tells us to mark A and then to compute

$$\epsilon\text{-}closure(move(A, a)).$$

We first compute $move(A, a)$, the set of states of N having transitions on a from members of A. Among the states 0, 1, 2, 4 and 7, only 2 and 7 have such transitions, to 3 and 8, so

$$\epsilon\text{-}closure(move(\{0, 1, 2, 4, 7\}, a)) = \epsilon\text{-}closure(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$$

Let us call this set B. Thus, $Dtran[A, a] = B$.

Among the states in A, only 4 has a transition on b to 5, so the DFA has a transition on b from A to

- **Appendix B: Elimination of left recursion**

  $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$

  can be replaced by

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

- **Appendix C: Computation of FIRST**

  To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

  1. If X is terminal, then FIRST(X) is $\{X\}$.
  2. If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).
  3. If X is nonterminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), . . . , FIRST(Y_{i-1}); that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$, then add ϵ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add FIRST($Y_2$) and so on.

- **Appendix D: Computation of FOLLOW**

  To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

  1. Place \$ in FOLLOW(S), where S is the start symbol and \$ is the input right endmarker.
  2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except for ϵ is placed in FOLLOW(B).
  3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ (i.e., $\beta \stackrel{*}{\Rightarrow} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B).

- **Appendix E: Nonrecursive predictive parsing**

Input. A string w and a parsing table M for grammar G.

Output. If w is in $L(G)$, a leftmost derivation of w; otherwise, an error indication.

Method. Initially, the parser is in a configuration in which it has \$$S$ on the stack with S, the start symbol of G on top, and w\$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input is shown in Fig. 4.14. □

```
set ip to point to the first symbol of w$;
repeat
    let X be the top stack symbol and a the symbol pointed to by ip;
    if X is a terminal or $ then
        if X = a then
            pop X from the stack and advance ip
        else error()
    else    /* X is a nonterminal */
        if M[X, a] = X → Y1Y2 · · · Yk then begin
            pop X from the stack;
            push Yk, Yk-1, . . . , Y1 onto the stack, with Y1 on top;
            output the production X → Y1Y2 · · · Yk
        end
        else error()
until X = $    /* stack is empty */
```

- **Appendix F: Construction of a predictive parsing table**

Algorithm 4.4. Construction of a predictive parsing table.

Input. Grammar G.

Output. Parsing table M.

Method.

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to $M[A, a]$.
3. If ϵ is in FIRST(α), add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in FOLLOW(A). If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, \$]$.
4. Make each undefined entry of M be **error**.