

קומפילציה – השלב הראשון:

ניתוח לקסיקלי = וידוא שכל המילים  
שייכות לשפה

# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
	<pre>void f(int a) {     6; }</pre>

# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
Build: 1 succeeded	<pre>void f(int a) {     6; }</pre>

# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
Build: 1 succeeded	<pre>void f(int a) {     6; }</pre>
	<pre>void f(int a) {     6b; }</pre>

# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
Build: 1 succeeded	<pre>void f(int a) {     6; }</pre>
error C2059: 'bad suffix on number' error C2146: missing ';' before identifier 'b' error C2065: 'b' : undeclared identifier	<pre>void f(int a) {     6b; }</pre>

# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
	<pre>void f(int a) {     0r; }</pre>

# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
error C2059: 'bad suffix on number' error C2146: missing ';' before identifier 'r' error C2065: 'r' : undeclared identifier	<pre>void f(int a) {     0r; }</pre>

# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
error C2059: 'bad suffix on number' error C2146: missing ';' before identifier 'r' error C2065: 'r' : undeclared identifier	<pre>void f(int a) {     0r; }</pre>
	<pre>void f(int a) {     0x; }</pre>



# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
error C2059: 'bad suffix on number' error C2146: missing ';' before identifier 'r' error C2065: 'r' : undeclared identifier	<pre>void f(int a) {     0r; }</pre>
error C2153: hex constants must have at least one hex digit	<pre>void f(int a) {     0x; }</pre>

# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
error C2059: 'bad suffix on number' error C2146: missing ';' before identifier 'r' error C2065: 'r' : undeclared identifier	<pre>void f(int a) {     0r; }</pre>
error C2153: hex constants must have at least one hex digit	<pre>void f(int a) {     0x; }</pre>
	<pre>void f(int a) {     0u; }</pre>

# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
error C2059: 'bad suffix on number' error C2146: missing ';' before identifier 'r' error C2065: 'r' : undeclared identifier	<pre>void f(int a) {     0r; }</pre>
error C2153: hex constants must have at least one hex digit	<pre>void f(int a) {     0x; }</pre>
Build: 1 succeeded	<pre>void f(int a) {     0u; }</pre>

# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
	<pre>void f(int a) {     900000000000000000000000; }</pre>

# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
error C2177: constant too big	<pre>void f(int a) {     900000000000000000000000; }</pre>

# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
error C2177: constant too big	<pre>void f(int a) {     9000000000000000000000; }</pre>
	<pre>void f(int a) {     int @gmail=0; }</pre>

# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
error C2177: constant too big	<pre>void f(int a) {     9000000000000000000000; }</pre>
error C2018: unknown character '0x40'	<pre>void f(int a) {     int @gmail=0; }</pre>

# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
	<pre>void f(int a) {     123.45.67; }</pre>



# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
error C2143: missing ';' before 'constant'	<pre>void f(int a) {     123.45.67; }</pre>

# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
error C2143: missing ';' before 'constant'	<pre>void f(int a) {     123.45.67; }</pre>
	<pre>void f(int a) {     123e; }</pre>

# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
error C2143: missing ';' before 'constant'	<pre>void f(int a) {     123.45.67; }</pre>
error C2021: expected exponent value, not ';'	<pre>void f(int a) {     123e; }</pre>

# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
	<pre>void f(void) {     0x123456789; }</pre>

# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
Build: 1 succeeded	<pre>void f(void) {     0x123456789; }</pre>

# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
Build: 1 succeeded	<pre>void f(void) {     0x123456789; }</pre>
	<pre>void f(void) {     int a=0x123456789; }</pre>

# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
Build: 1 succeeded	<pre>void f(void) {     0x123456789; }</pre>
warning C4305: truncation from '__int64' to 'int' warning C4309: truncation of constant value	<pre>void f(void) {     int a=0x123456789; }</pre>

# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
	<pre>void f(void) {     int a=0x00000000000000000000000000000007; }</pre>



# ניתוח לקסיקלי – חימום

שגיאות (?)	קוד
Build: 1 succeeded	<pre>void f(void) {     int a=0x00000000000000000000000000000007; }</pre>

# ניתוח לקסיקלי – תיאור

- קובץ התוכנית הוא למעשה משפט שמורכב ממילים בשפה
- תפקיד המנתח הלקסיקלי, השלב הראשון בקומפילציה, הוא לוודא שאכן כל המילים בשפה. וכך גם בעברית: מי שמצפלב חושב שיש סרפפ – שימו לב שגם תוכנת המצגת עושה את זה, אבל המשימה שלה יותר קלה – למה?
- מה הן מילים בשפת תכנות?

# מילים חוקיות בשפת C

Legal Tokens	Example
Constants	123, 90, 222, ... 19.7, 13e+8, ... 0x80, 0xabcd, ...
Identifiers	numStudentsInMTA, strcpy,
Reserved Keywords	Int, float, char, double, ... If, while, do, goto, ... struct, class, typedef, ...
Parentheses	{, }, (, ), [, ]
Binary Operators	+, -, *, /, =, -, >, ==, ...
Unary operators	-, *, ++,
Comments	/* ... */ , //

# דרוש: מנגנון (יעיל!) המאפשר זיהוי מספר בשפת תכנות

- למה אי אפשר לעשות משהו כמו מילון אם רוצים לבדוק האם לפנינו מספר חוקי?
- מה האורך המקסימלי של שמות משתנים?
- כלומר מה סדר גודל של מספר שמות חוקיים של משתנים?
- מילון הוא בלתי אפשרי בעליל כאן ...

# דרוש: מנגנון (אחיד!) המאפשר זיהוי מספר בשפת תכנות

- ראינו שיש מגוון גדול של סוגי מילים, תזכורת:
- רצף של ספרות הוא תמיד מספר? 000088
- חיובי\שלילי (מה אם אין סימן?)
- נקודה עשרונית (מותרת נקודה אחת ולא יותר)
- ייצוג כבסיס + מעריך (מה לגבי  $34.56E+5.66$ )
- ייצוג אקסהדצימלי (מה לגבי 0xAAA)
- נחפש מנגנון אחיד שמטפל בצורה אחידה בכל המקרים

# ביטויים רגולריים כמנגנון ייצוג של מספר

## אינסופי \* של מילים בשפת C

- למשל, ביטוי רגולרי המאפשר זיהוי של identifiers בשפת C:  
 $[_ a-z A-Z][0-9 a-z _ A-Z]^*$
- או למשל מספר בייצוג הקסה-דצימלי: (מה האורך המותר?)  
 $[0][xX][0-9 a-f A-F]^+$
- מה הביטוי הרגולרי שמגדיר float עם בסיס ואקספוננט?
- המילים בכל שפות התכנות הקיימות ניתנות לייצוג על ידי ביטוי רגולרי מתאים.
- אבל איך הייצוג שלהן כביטויים רגולריים מאפשר מנגנון זיהוי יעיל?

# ביטויים רגולריים – תזכורת

בהינתן אלפבית כלשהו  $\Sigma$ , שפה מעליו, היא אוסף כלשהו של מחרוזות (מחרוזת היא תמיד סופית).

- בהינתן אלפבית כלשהו  $\Sigma$ , ביטוי רגולרי  $R$  מייצג שפה  $L(R)$  מעל האלפבית באופן הבא:

- הביטוי הרגולרי  $a$  מייצג את השפה  $\{a\}$

- הביטוי הרגולרי  $\epsilon$  (המחרוזת הריקה) מייצג את השפה  $\{\epsilon\}$

- הביטוי הרגולרי  $\emptyset$  מייצג את השפה הריקה

- בהינתן שני ביטויים רגולריים  $R_1, R_2$  הפעולות הבאות מוגדרות ליצירת ביטוי רגולרי חדש:

- שרשור:  $R_1 R_2$  מייצג את השפה  $\{w_1 w_2 \mid w_i \in L(R_i)\}$

- איחוד:  $R_1 \mid R_2$  מייצג את השפה  $L(R_1) \cup L(R_2)$

- הסגור של קליין:  $R_1^*$  מייצג שפה המורכבת משרשורים (סופיים!) כלשהם של מילים מתוך  $L(R_1)$

# ביטוי רגולרי ← לאוטומט דטרמיניסטי סופי

- לכל ביטוי רגולרי קיים אוטומט דטרמיניסטי סופי שמקבל בדיוק את אותה שפה שהביטוי הרגולרי מייצג.
- ההוכחה לטענה מעלה היא קונסטרוקטיבית, כלומר, בהינתן ביטוי רגולרי, קיים אלגוריתם אשר בונה את האוטומט הנ"ל.
- בהינתן אוטומט דטרמיניסטי סופי שמקבל את השפה שהביטוי הרגולרי המקורי ייצג, אפשר לממש אותו כטבלה דו מימדית



ביטוי רגולרי ←

אוטומט לא דטרמיניסטי, עם מעברי אפסילון ←  
אוטומט דטרמיניסטי סופי

- המעבר מביטוי רגולרי לאוטומט דטרמיניסטי סופי, עובר דרך אוטומט לא דטרמיניסטי, עם מעברי אפסילון.

- ניזכר שביטוי רגולרי יכול להיות ביטוי אטומי:

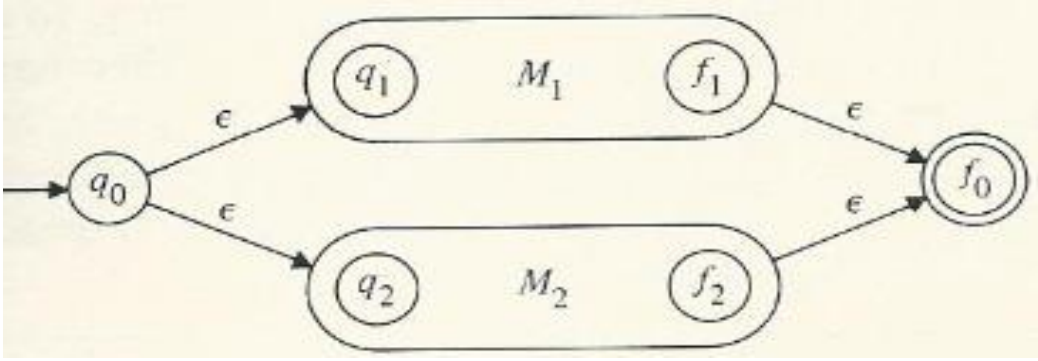
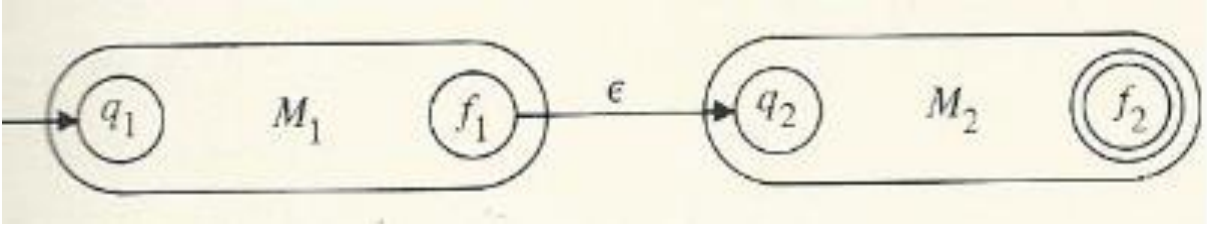
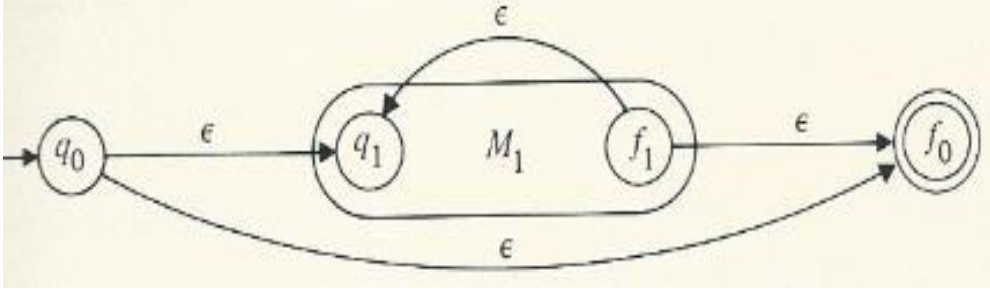
$a, \epsilon, \emptyset$  –

- או ביטוי מורכב:

- $rfsd$

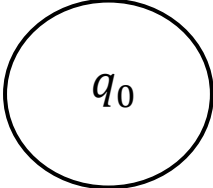
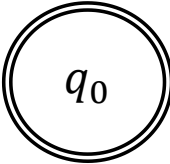
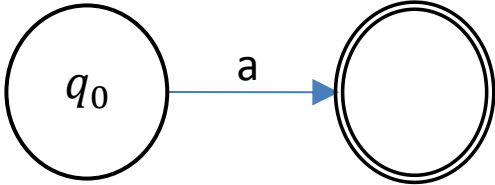
# ביטוי רגולרי מורכב ←

אוטומט לא דטרמיניסטי סופי, עם מעברי אפסילון

	<p>איחוד של שני ביטויים רגולריים</p>
	<p>שרשור של שני ביטויים רגולריים</p>
	<p>הסגור של קלין של ביטוי רגולרי</p>

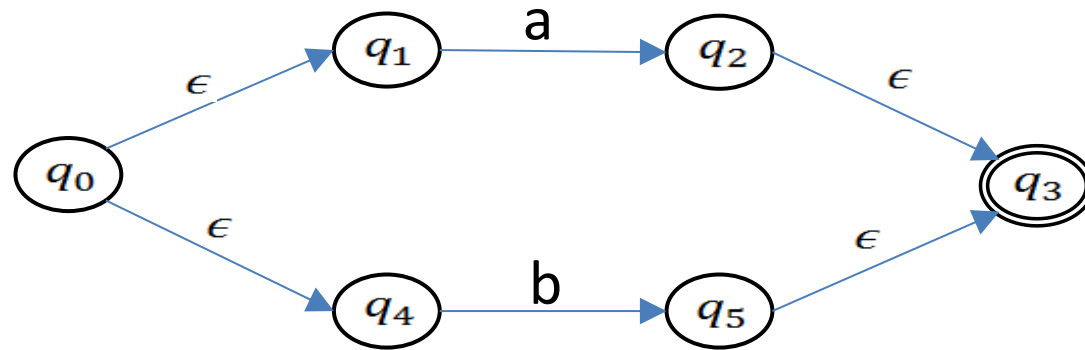
# ומה לגבי ביטויים רגולריים אטומיים?

## מה האוטומט שלהם?

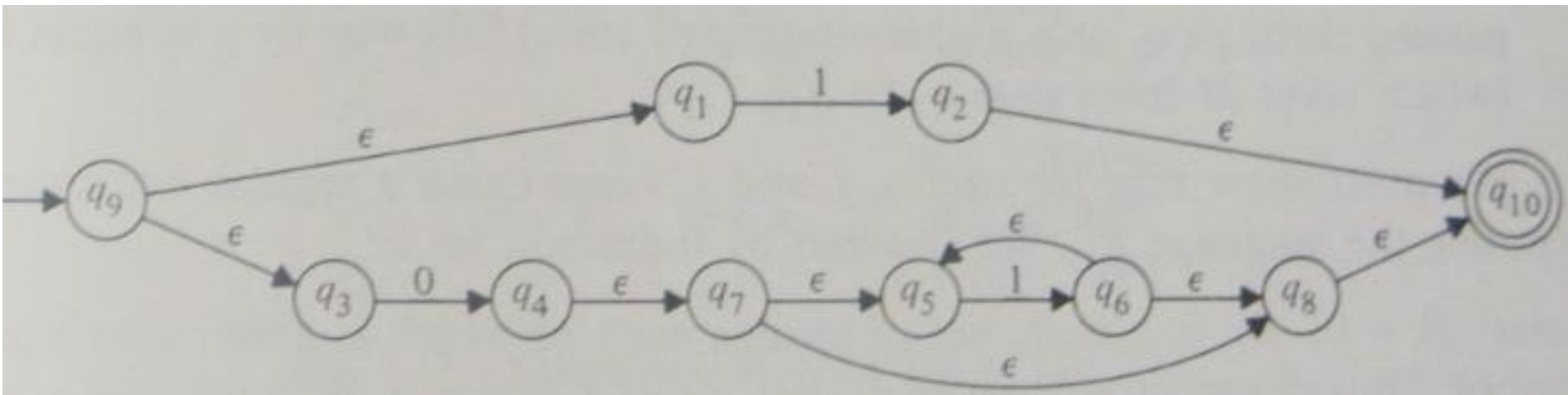
	הקבוצה הריקה
	המחרוזת הריקה
	מחרוזת של תו אחד

תרגיל כיתה של שלב 1: ביטוי רגולרי  
 אוטומט לא דטרמיניסטי סופי, עם מעברי אפסילון

$a \mid b$  •



$R_3 = 01^*|1$  •



# אוטומט דטרמיניסטי סופי – הגדרה פורמלית

- אוטומט דטרמיניסטי סופי הוא חמישיה סדורה  $(Q, \Sigma, \delta, q_0, F)$  כאשר  $Q$  היא קבוצת המצבים,  $\Sigma$  הוא האלפבית,  $\delta: V \times \Sigma \rightarrow V$  היא פונקציית המעבר,  $q_0 \in Q$  הוא המצב ההתחלתי, ו  $F \subseteq Q$  הוא קבוצת המצבים המקבילים
- נניח שקראנו  $i$  אותיות מהקלט והגענו לקודקוד  $q'$ , ונניח שהוא הבאה בקלט היא  $\sigma$  אז מסתכלים בפונקציית המעבר  $\delta: V \times \Sigma \rightarrow V$  כדי לדעת לאיזה קודקוד לעבור. נניח שכאן  $\delta(q', \sigma) = q''$  אז נעבור למצב  $q''$  ונמשיך לקרוא את האות הבאה בקלט

# אוטומטים לא דטרמיניסטיים סופיים – הצגה פורמלית

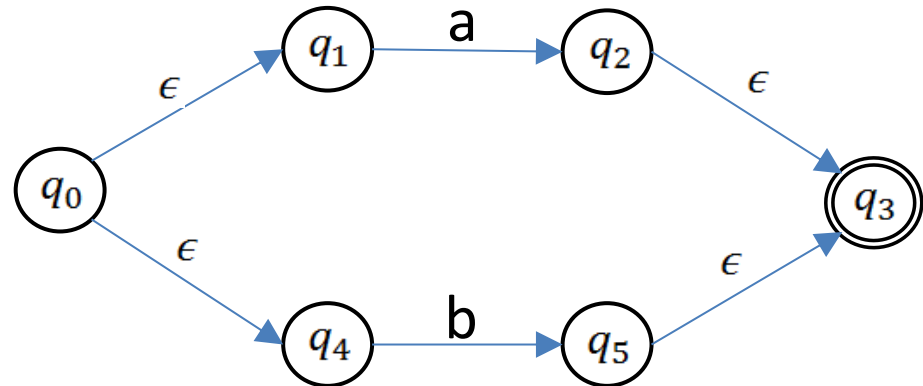
- אוטומט לא דטרמיניסטי סופי הוא חמשיה סדורה:  
 $\{Q, \Sigma, \Delta, q_0, F\}$  כאשר  $Q$  היא קבוצת המצבים,  $\Sigma$  הוא האלפבית,  $\Delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow P(Q)$  היא פונקציית המעברים,  $q_0 \in Q$  הוא המצב ההתחלתי, ו  $F \subseteq Q$  היא קבוצת המצבים המקבילים.  $\epsilon$  הוא הקלט הריק.
- כל אוטומט לא דטרמיניסטי סופי מגדיר שפה מעל האלפבית  $\Sigma$  – שפת המילים שהוא מקבל. כלומר, מילים שאם נריץ אותן על האוטומט נסיים בקבוצת מצבים שמכילה מצב מקבל.

← אוטומט לא דטרמיניסטי סופי, עם מעברי אפסילון  
אוטומט דטרמיניסטי סופי

- כדי לממש בתוכנה אוטומט, הוא חייב להיות דטרמיניסטי.
- השלב השני אם כן, הוא דטרמיניזציה של האוטומטים הלא דטרמינסטיים לאוטומטים כן דטרמינסטיים.
- בשקפים הבאים כמה דוגמאות

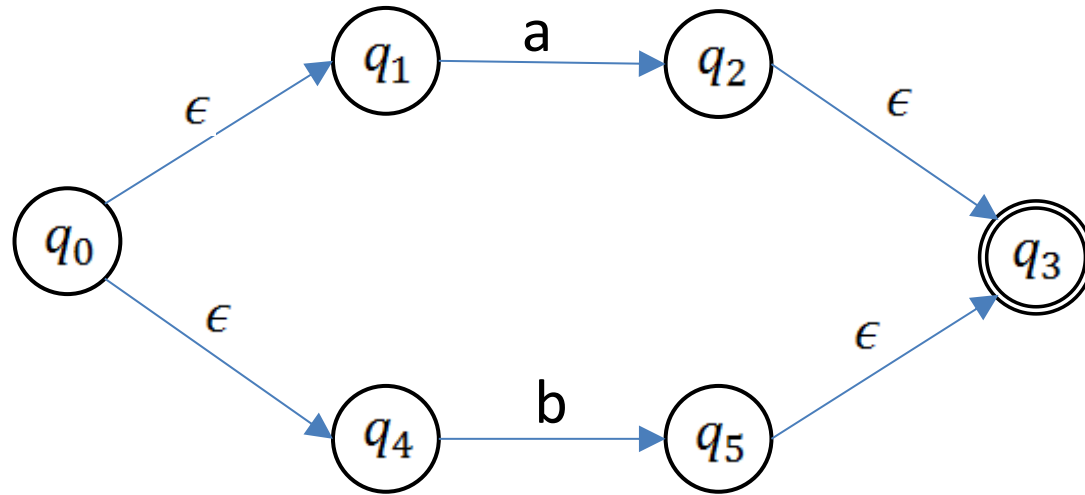
## תרגיל כיתה של שלב II:

אוטומט לא דטרמיניסטי סופי, עם מעברי אפסילון ←  
אוטומט דטרמיניסטי סופי

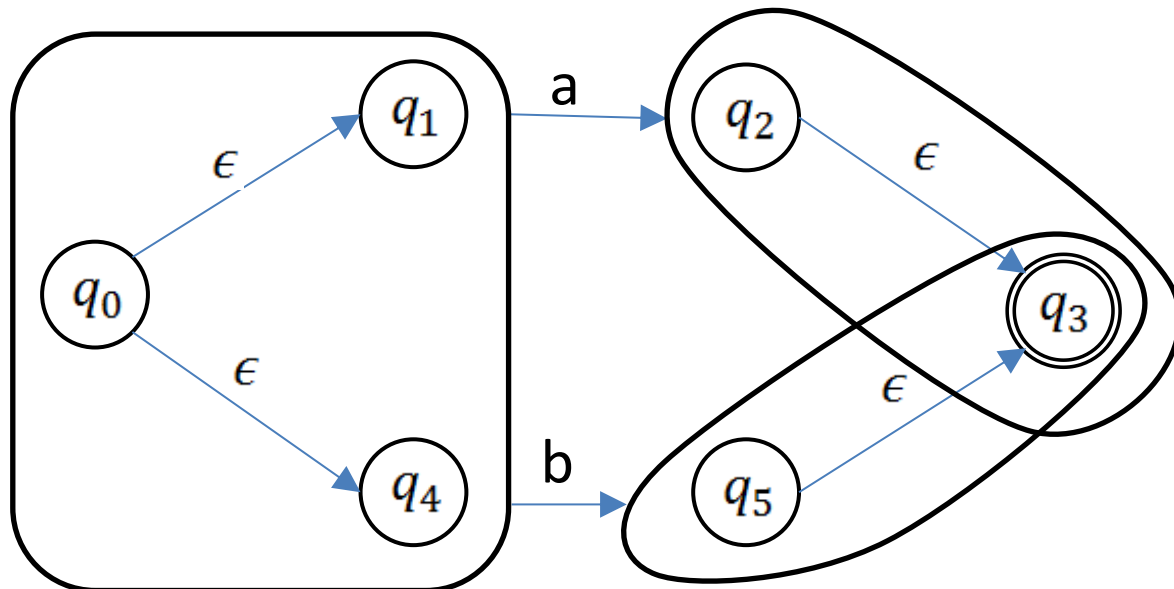


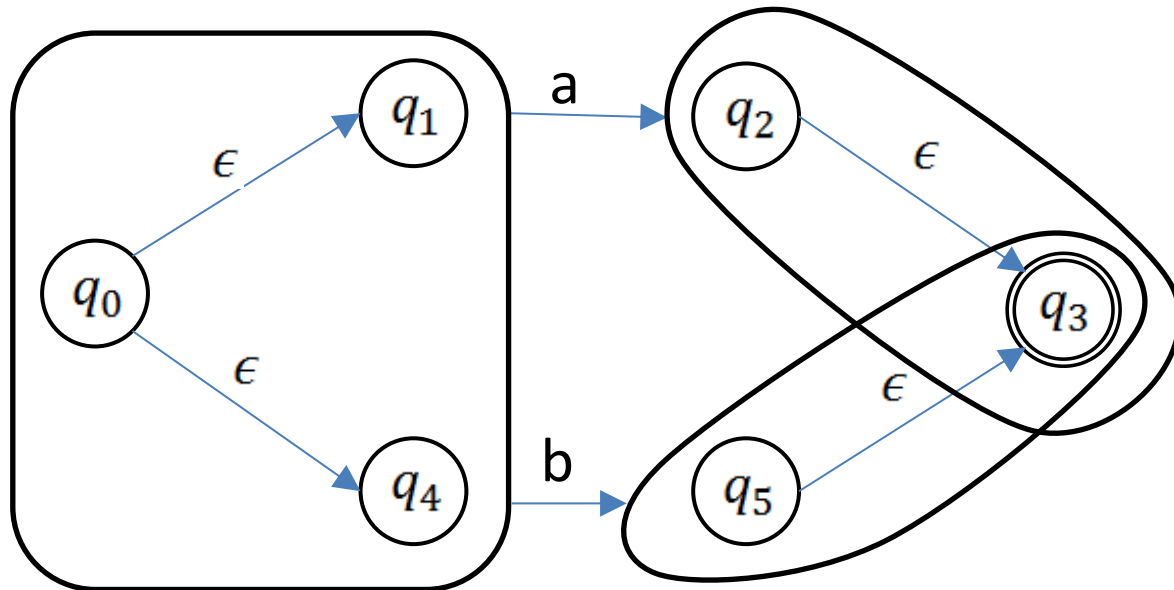
- בהתחלת הריצה, אנחנו יכולים להיות במצבים 0,1,4.
- אם ראינו את האות a אז אנחנו במצב 2 או 3
- אם ראינו את האות b אז אנחנו במצב 5 או 3
- אם הסתיים הקלט, ואנחנו בקבוצת מצבים שיש בה מצב מקבל, אז נקבל את המילה.
- בכל מצב אחר לדחות.



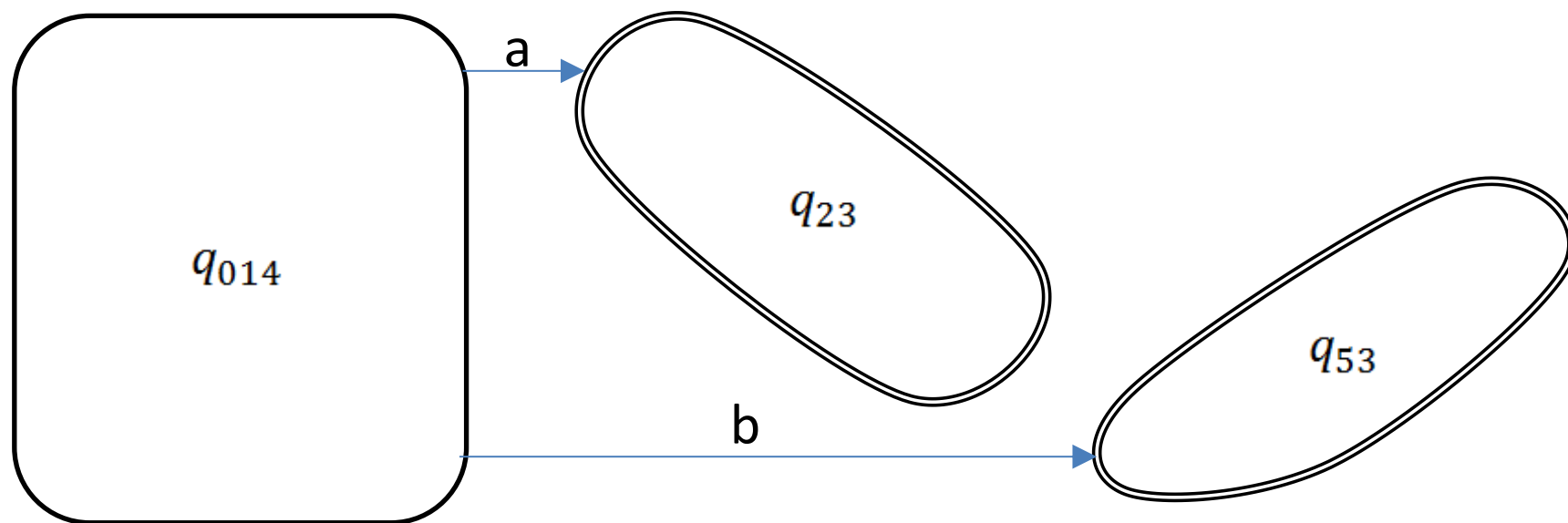


- כלומר עברנו מהאוטומט הלא דטרמיניסטי מעלה, לאוטומט הדטרמיניסטי למטה



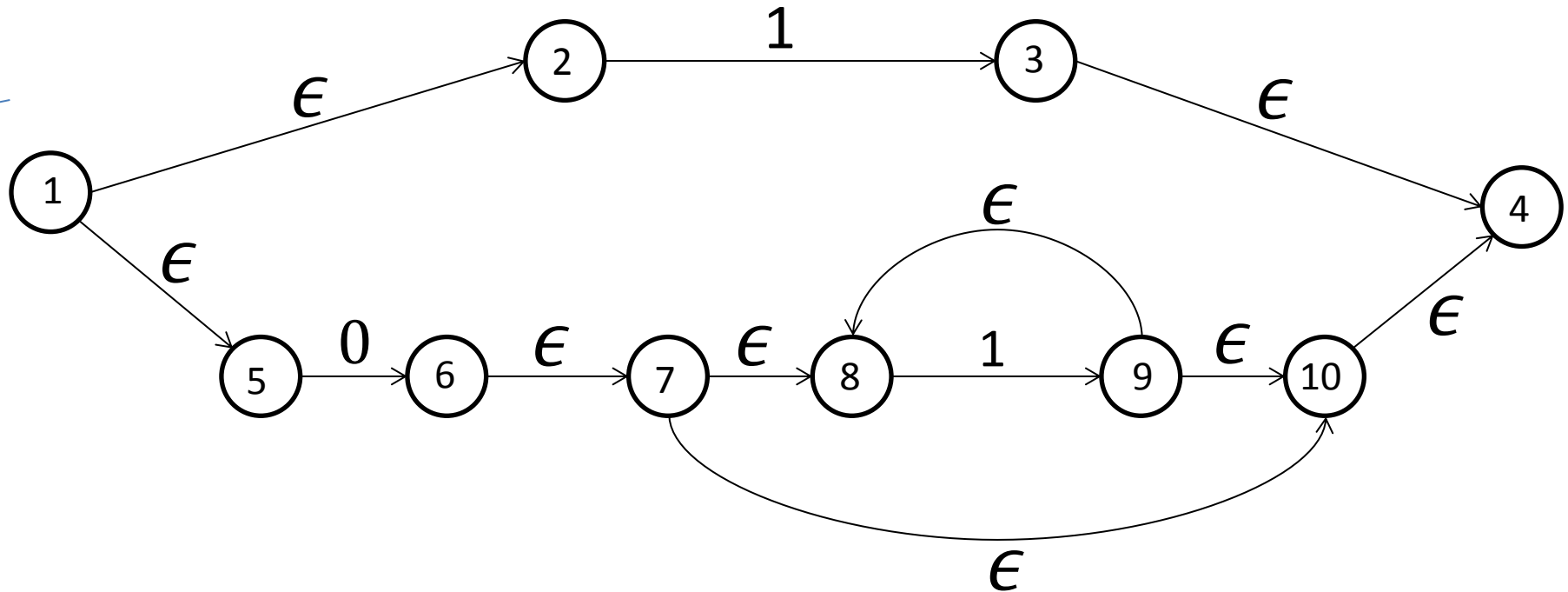


• ואותו אפשר לרשום בנוחות כך:



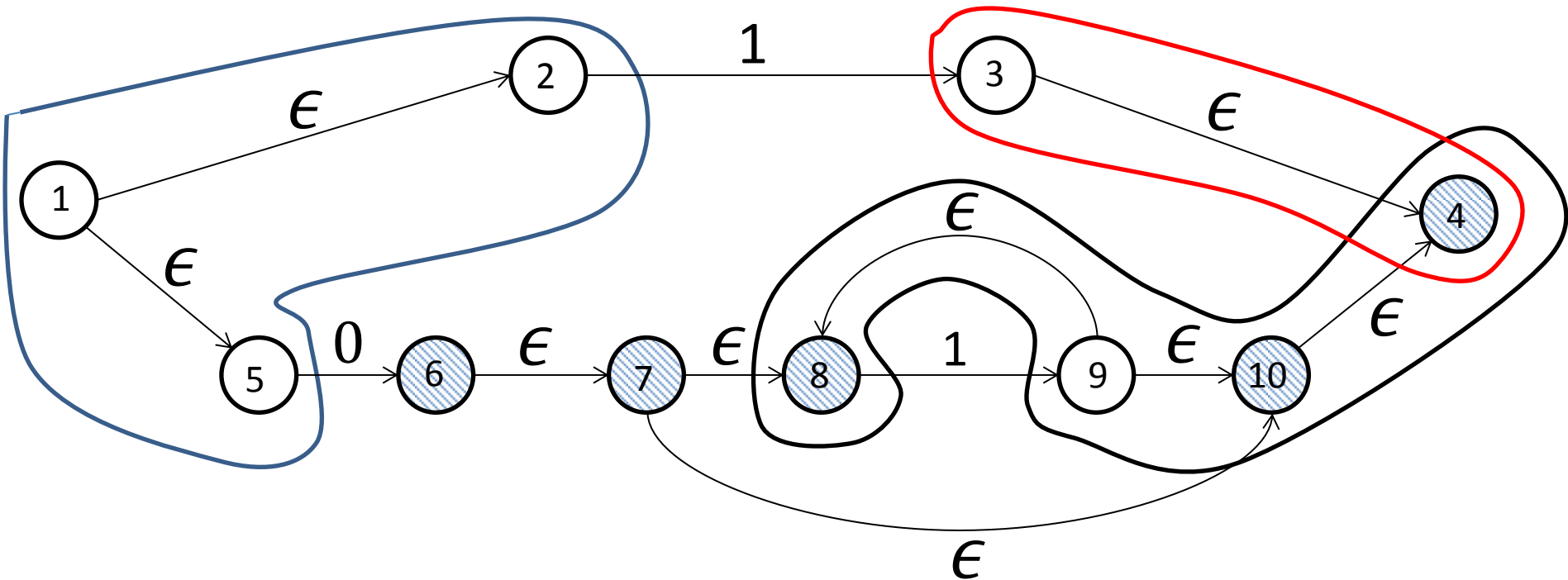
תרגיל כיתה:

אוטומט לא דטרמיניסטי סופי, עם מעברי אפסילון  
אוטומט דטרמיניסטי סופי



עוד תרגיל כיתה של שלב II:

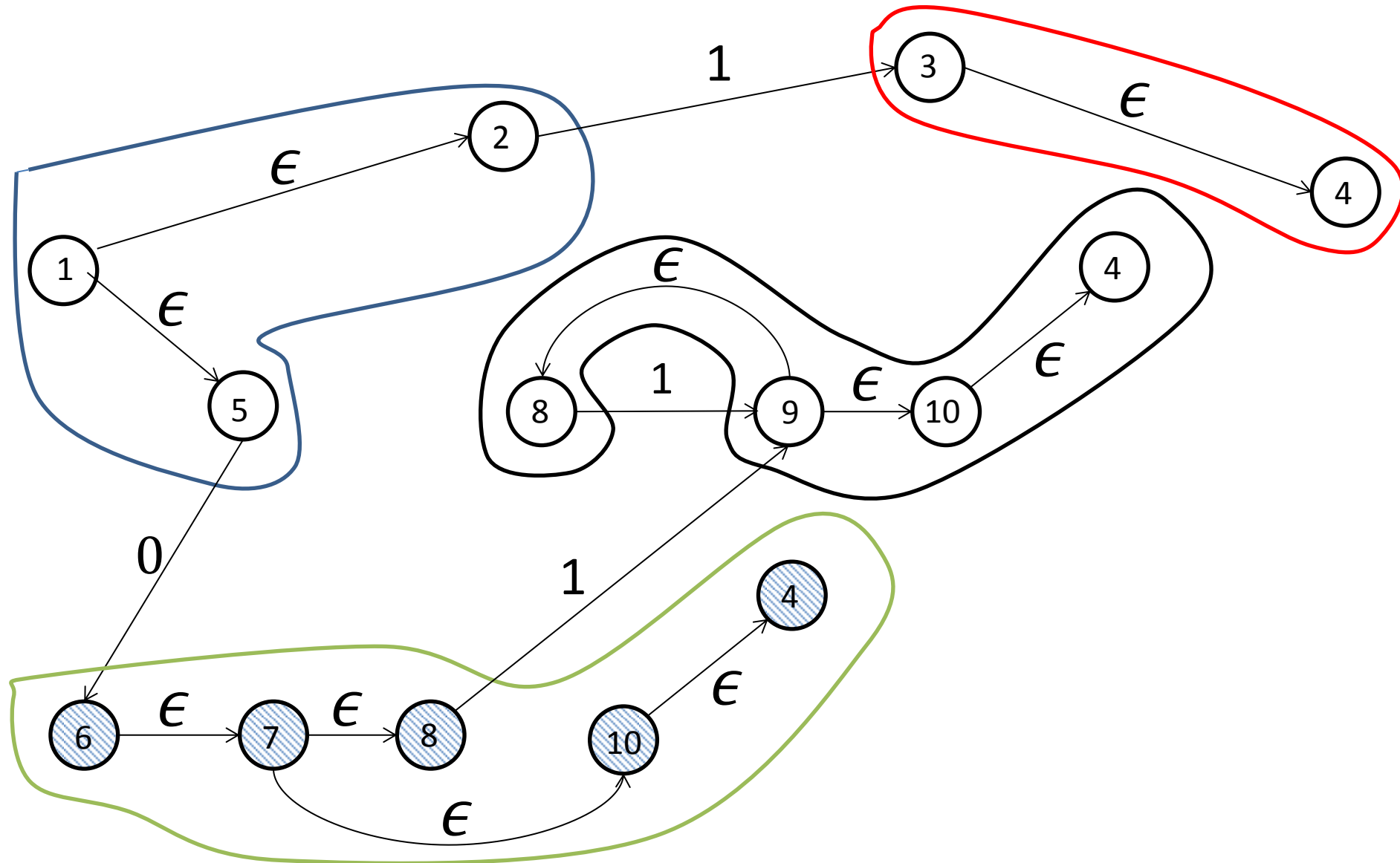
אוטומט לא דטרמיניסטי סופי, עם מעברי אפסילון  
אוטומט דטרמיניסטי סופי



עוד תרגיל כיתה של שלב II:

אוטומט לא דטרמיניסטי סופי, עם מעברי אפסילון ←

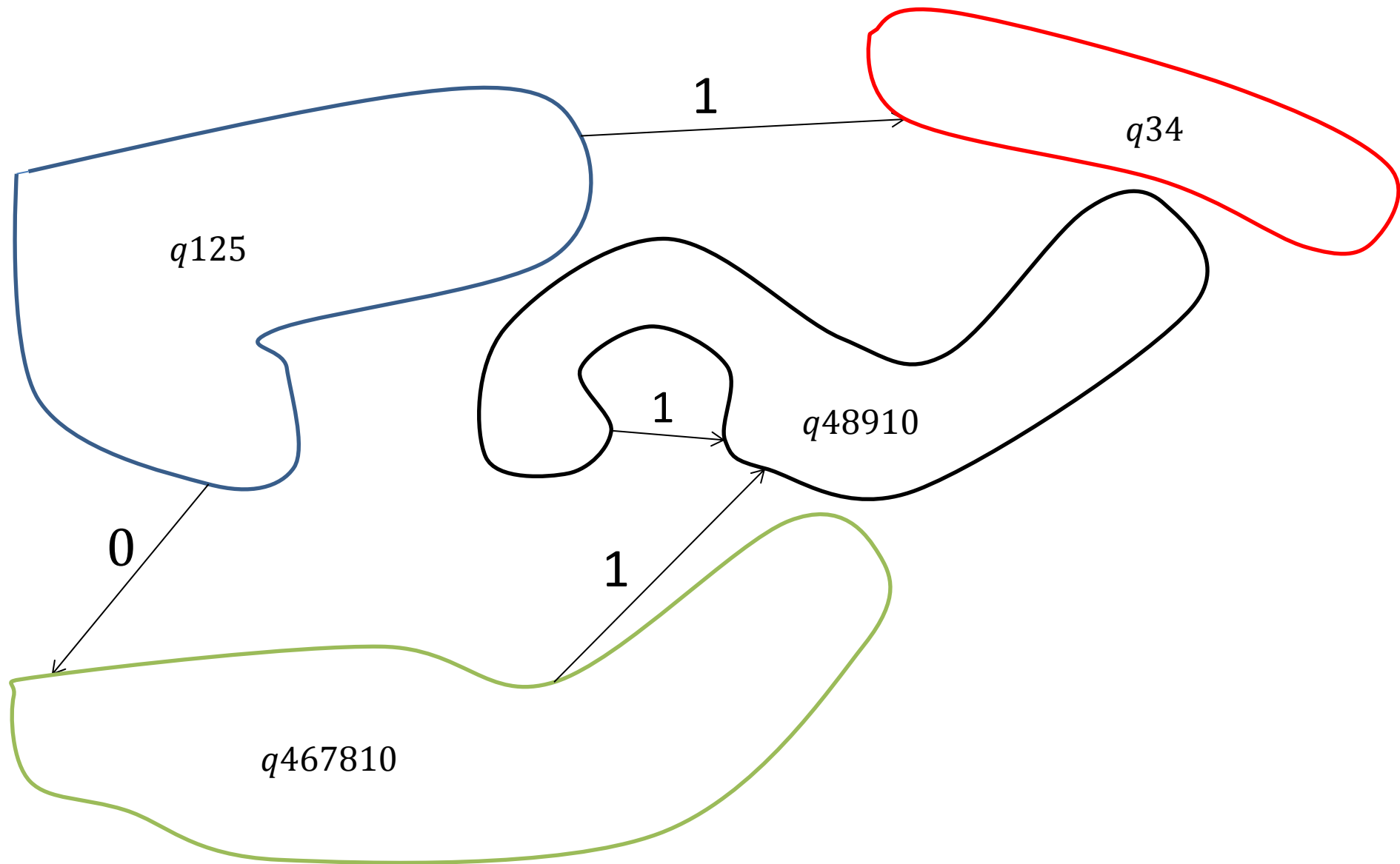
אוטומט דטרמיניסטי סופי



עוד תרגיל כיתה של שלב II:

← אוטומט לא דטרמיניסטי סופי, עם מעברי אפסילון

אוטומט דטרמיניסטי סופי

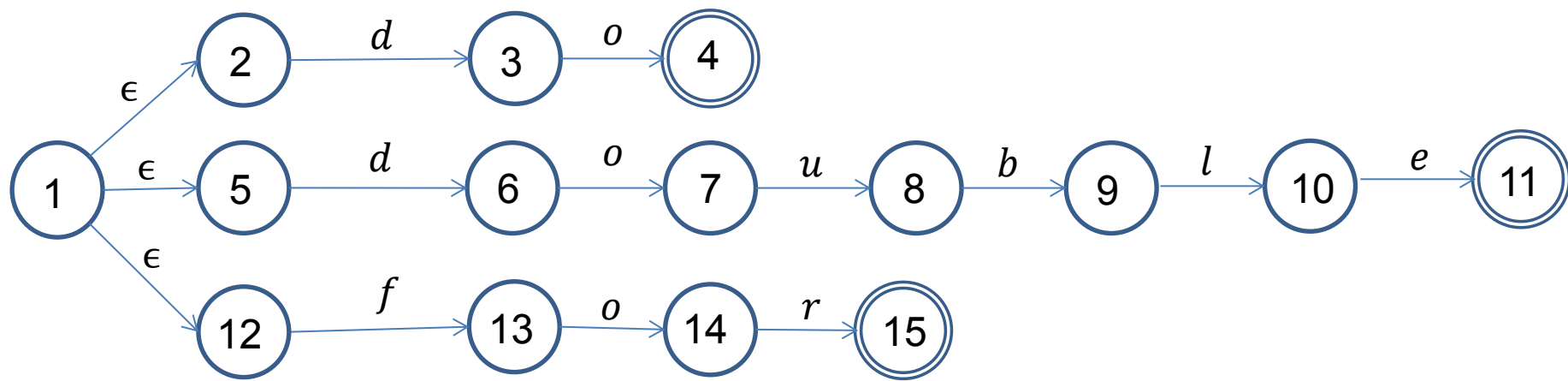
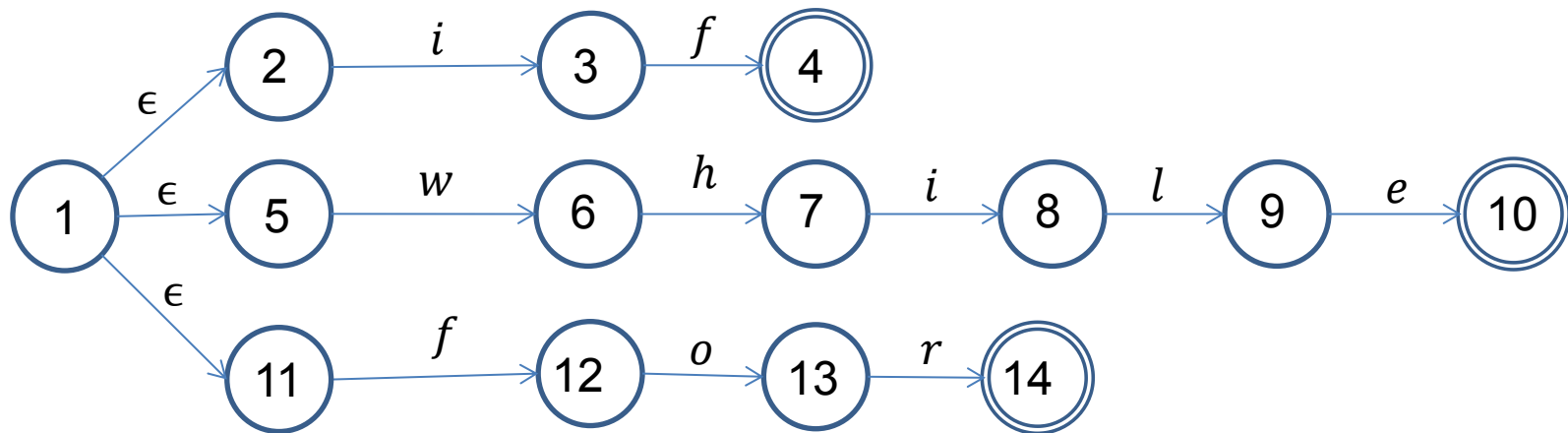


# מאוטומט לא דטרמיניסטי סופי

## לאוטומט דטרמיניסטי סופי – תיאור פורמלי

1. קבוצת המצבים היא קבוצת החזקה של המצבים באוטומט הלא דטרמיניסטי
2. המצב ההתחלתי הוא הסגור-אפסילון של המצב ההתחלתי
3. פונקציית המעבר תעבוד כך: לכל מצב בקבוצה, נחשב את קבוצת המצבים שהוא עובר אליה וניקח את הסגור אפסילון שלה. אחר כך ניקח את האיחוד של כל הקבוצות האלה.
4. מצב מקבל הוא קבוצה של מצבים באוטומט הלא דטרמיניסטי, שהכילה מצב מקבל

# ואיך משלבים כמה ביטויים רגולריים?





# בניית מנתח לקסיקלי מהחיים האמתיים – סיכום

- בנו ביטויים רגולריים המתארים את סוגי המילים השונות הקיימות בשפת התכנות (identifiers, hexadecimal numbers, reserved keywords, float numbers, strings ועוד). שימו לב: הסיווג נעשה כדי לבנות ביטויים פשוטים ככל הניתן.
- במידה ויש התנגשויות, כלומר אותה מילה יכולה להתקבל על ידי יותר מאוטומט אחד (המילה for), יש להגדיר עדיפות!

# מילים בשפת C הן ביטויים רגולריים!

- הנה הקובץ שמתאר את הביטויים הרגולריים של שפת C:
- <http://www.lysator.liu.se/c/ANSI-C-grammar-l.html>
- שימו לב כמה פשוט הוא כל ביטוי. שימו לב לגמישות התיאור: למשל, אם רוצים לאסור על מספר שלם להתחיל באפסים, או שבמקום הסימן = נשתמש בחץ להשמה ( $a \leftarrow 80$ ) השינויים האלה יכולים להיעשות מהר מאוד.

ביטויים רגולריים ←

אוטומט לא דטרמיניסטי סופי ←

אוטומט דטרמיניסטי סופי

- בכל שפת תכנות שימושית כיום, המילים החוקיות הן ביטויים רגולריים.
- הצעדים הנ"ל הם פשוטים מספיק להיעשות באופן אוטומטי: המשתמש (ממציא השפה) יתאר את המילים החוקיות כאוסף של ביטויים רגולריים. התוכנה (JFlex) תהפוך את האוסף לאוטומט לא דטרמיניסטי סופי. אחר כך, היא תהפוך אותו לאוטומט דטרמיניסטי סופי ותוציא כפלט את פונקציית המעברים כטבלה

# JFlex – Java Fast Lexical Analyzer

- JFlex הוא מנתח לקסיקלי שמקבל קובץ של ביטויים רגולריים המתארים מילים חוקיות בשפה, ומחזיר קובץ בשפת Java שמייצא פונקציה אחת בלבד `yylex()`. פונקציה זאת קוראת מהקלט מילה ומחזירה האם היא בשפה.
- ליתר דיוק, כל ביטוי רגולרי מתאר מילה חוקית בשפה מסוג מסוים (`identifier, float, int, string` etc.) והפונקציה `yylex()` הממומשת על ידי JFlex תחזיר את סוג המילה, במידה והיא בשפה, או שגיאה, אם המילה לא בשפה.