TC3048 Compilers Design

Chapter #2

Lexical Analysis

1

1. Introduction

- The **Lexical Analysis** or **scanning** phase has the task of reading the ***source program*** as a sequence of characters and ***dividing*** it into **tokens**.
  - **Token**: ***pattern of characters*** that represents a unit of information in the source program.
    - The **tokens** are the ***terminal symbols*** of the **language**:

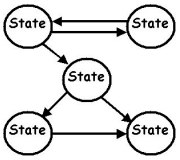
      - ***Reserved words*** or ***keywords***, such as `if` and `while` which are ***fixed*** strings of letters.
      - ***Identifiers***, which are user-defined strings, usually consisting of letters and numbers that begins with a letter.

2

- ***Special symbols***, such as the ***arithmetic*** symbols "**+**" and "*****"; as well as the ***multicharacter*** symbols such as "**>=**" and "**<>**".
- ***Literals*** or ***constants***, which may include ***numeric*** constants, ***string*** literals, and ***single characters***.

☛ Since the job performed by the **Lexical Analyzer** is a ***special case*** of **pattern matching**, it is required to study ***methods*** of ***pattern specification*** and ***recognition***.

- ***Regular Expressions***: standard notation for ***representing*** the **patterns** of strings that form the **lexical structure** of the ***programming language***.
- ***Finite State Machines***: represent ***algorithms*** that ***recognize*** **patterns** of strings given by **regular expressions**.



2. Regular Expressions

- **Regular Expressions** represent **patterns** of strings of characters.
- **Regular Expressions** are a special notation that ***facilitates*** the task of **consistent** and **precisely** ***representing*** the **lexical rules** of a **language**.
- A **Regular Expression** ***r*** is completely defined by the **SET** of strings it matches.
  - This **SET** is called the **language generated by the regular expression** and is expressed as:

$$L(r)$$

- A **language** is composed by a **set of strings**. The strings are formed from a **character set**.
  - The elements of the **character set** are referred as **symbols**.
  - The set of ***legal*** **symbols** is known as the **alphabet** and is written as the Greek symbol ∑ (sigma).

2.1. Definition of Regular Expressions

- **Basic Regular Expressions**:

① Given any character "*a*" from the alphabet ∑:

$$a \in \sum$$

the regular expression ***a*** matches the character "*a*" by the following expression:

$$L(\mathbf{a}) = \{a\}$$

② The **empty string** is the string that **contains *NO*** characters at all. The Greek symbol ε (epsilon) is used to represent the empty string. Therefore:

$$L(\varepsilon) = \{\varepsilon\}$$

③ The **empty set** is the one that **contains *NO*** strings at all. The Greek symbol in bold **Φ** (capital Phi) is used to represent the **regular expression** whose language is the **empty set**, also, Phi in no bold is used to represent the **empty set**:

$$L(\mathbf{\Phi}) = \{\}$$
$$\Phi \leftrightarrow \{\}$$

☝ Note the difference between Φ and {ε}, while the **former** contains ***NO*** string at all, the **latter** contains the single string consisting of no characters.

$$\Phi \neq \{\varepsilon\}$$

❒ **Regular Expressions Operations**:

① The **choice** among alternatives (**OR** operation) which is denoted by the vertical bar "|".

If **r** and **s** are **regular expressions**, then **r|s** is a regular expression that matches any string that is matched either by **r** or by **s**.

In terms of **languages**, the language of **r|s** is the **union** of the languages of **r** and **s**:

$$L(\mathbf{r}|\mathbf{s}) = L(\mathbf{r}) \cup L(\mathbf{s})$$

✎ *Example*:

Consider the regular expression **a|b**: it is matched either by the character "a" or by character "b", i.e.,

$$L(\mathbf{a}|\mathbf{b}) = L(\mathbf{a}) \cup L(\mathbf{b}) = \{a\} \cup \{b\} = \{a,b\}$$

② The **concatenation** which is denoted by **juxtaposition** (without a metacharacter).

If **r** and **s** are **regular expressions**, then **rs** is a regular expression that matches any string that is the concatenation of two strings.

In terms of **languages**, the language of **rs** is:

$$L(\mathbf{rs}) = L(\mathbf{r})L(\mathbf{s})$$

Examples :

1. Consider the regular expression **ab**, it is matched only by the string "ab", i.e.,

$$L(\mathbf{ab}) = L(\mathbf{a})L(\mathbf{b}) = \{a\}\{b\} = \{ab\}$$

2. Consider the regular expression (**a|b**)**c**, it is matched by the strings "ac" and "bc", i.e.,

$$L((\mathbf{a}|\mathbf{b})\mathbf{c}) = L(\mathbf{a}|\mathbf{b})L(\mathbf{c}) = \{a,b\}\{c\} = \{ac,bc\}$$

③ The **Kleen Closure** is the **repetition** operation of a regular expression that represents ***cero or more concatenations*** of the regular expression **with itself**. It is denoted **r***.

If **r** is a **regular expression**, then **r*** is a regular expression that matches any finite concatenation of strings, each of which matches **r**.

In terms of **sets**, given the set **S** of strings, let

$$\mathbf{S}^* = \{\varepsilon\} \cup \mathbf{S} \cup \mathbf{SS} \cup \mathbf{SSS} \cup \ldots\ldots$$

or

$$\mathbf{S}^* = \bigcup_{n=0}^{\infty} S^n$$

In terms of **languages**, the repetition operation for regular expressions is defined as follows:

$$L(\mathbf{r}^*) = L(\mathbf{r})^*$$

Example:

Consider the regular expression **(a|bb)*** which is matched by any of the following strings:

ε, *a*, *bb*, *aa*, *abb*, *bba*, *bbbb*, *aaa*, *aabb*, ……..

In terms of languages:

$$L((\mathbf{a} \mid \mathbf{bb})^*) = \{a, bb\}^*$$
$$= \{\varepsilon, a, bb, aa, abb, bba, bbbb, aaa, aabb, ...\}$$

④ The **Positive Closure** is the **repetition** operation of a regular expression that represents ***one or more concatenations*** of the regular expression **with itself**. It is denoted $\mathbf{r}^+$.

If **r** is a **regular expression**, then $\mathbf{r}^+$ is a regular expression that matches any finite concatenation of strings, each of which matches **r**.

In terms of **sets**, given the set **S** of strings, let

$$\mathbf{S+} = \mathbf{S} \cup \mathbf{SS} \cup \mathbf{SSS} \cup .....$$

or

$$\mathbf{S+} = \bigcup_{n=1}^{\infty} S^n$$

❒ **Regular Expressions Extensions**:

① The **Zero or One** operation of a regular expression that represents ***zero or one occurance*** of the regular expression. It is denoted $\mathbf{r}^?$.

If $\mathbf{r}$ is a **regular expression**, then $\mathbf{r}^?$ is equivalent to $\mathbf{r}|\varepsilon$.

In terms of **languages**,

$$L(\mathbf{r}^?) = L(\mathbf{r}) \cup \{\varepsilon\}$$

② **Character Classes** or **Range of Characters**. Often it is required to write a range of characters, such as lowercase letters or all digits.

For lowercase characters:

a|b|c|...|z = [a-z]

For all digits:

0|1|2|...|9 = [0-9]

Multiple ranges can be included, such as all lowercase and uppercase characters:

[a-zA-Z]

For individual alternatives, e.g.:

a|b|c = [abc]

③ **Names for Regular Expressions**. It is helpful as a notational simplification, to ***assign*** a **name** to a long regular expression, so that it is not required to write the expression itself every time it is used.

For example, a regular expression for a sequence of one or more numeric digits, one option would be:

(0|1|2|...|9) (0|1|2|...|9)*

or it can be used:

digit digit*

where

digit **= 0|1|2|...|9**

④ **Any character**. Sometimes it is required to **match *any* character** in the given alphabet. This can be done by listing every character using the **choice** (|). However, usually the dot (.) metacharacter is used instead for this purpose.

For example, given $\sum = \{a,b,c,\ldots,z\}$, a regular expression for al strings that contain at least one *b* would be:

.*b.*

instead of:

(a|b|c|...|z)* b (a|b|c|...|z)*

Example:

C language identifiers are strings of letters, digits and underscores. The rules for a ***C*** language identifier are:

- Can start with letter or underscore.
- Can follow any amount of letters, underscores, or numbers.
- Can't start with numbers nor any other special symbol.

A regular expression that defines ***C***'s identifiers is the following:

letterUnderscore = **[A-Za-z]|_**
digit = **[0-9]**
id = ***letterUnderscore*** **(*letterUnderscore*|*digit*)***

Example:

Unsigned integer or real nombers are strings such as 1234, 123.456, 123.456E7, 123.456E+2, or 12.34E-5

A regular expression that defines this set of strings:

digit = **[0-9]**
digits = ***digit digit****
optionalfraction = (**.** ***digits***) **|** ε
optionalExponent = (**E**(+|-|ε) ***digits***) **|** ε
numbers = ***digits optionalfraction optionalExponent***

2.2. Algebraic Laws of Regular Expressions

☛ A **language** that can be defined by a **regular expression** is called ***Regular Set***.

❒ **Operators's Precedence**:

① The unary operator * has the highest precedence and is left associative.

② Concatenation has the second precedence and is left associative.

③ Choice | has the lowest precedence and is left associative.

☛ If **r**, **s** and **t** are **regular expressions**, then the algebraic laws that hold are:

Law	Description
r|**s** = **s**|**r**	| is commutative
r|(**s**|**t**) = (**r**|**s**)|**t**	| is associative
r(**st**) = (**rs**)**t**	Concatenation is associative
r(**s**|**t**) = **rs** | **rt**; (**s**|**t**)|**r** = **sr** | **tr**;	Concatenation distributes over |
ε **r** = **r** ε = **r**	ε is the identity of concatenation
Φ|**r** = **r**|Φ = **r**	Φ is the identy of the choice
r* = (**r**|ε)*	ε is guaranteed in a clousure
r** = **r***	* is idempotent

- The ***same language*** can be generated by ***many different*** **regular expressions**.

- In order to describe a set of strings, normally, it is tried to find a **regular expression** ***as simple as possible***.

- However, it ***should never*** be attempted ***to prove*** that the **regular expression** obtained is in fact the simplest or shortest, for two reasons:

  - First, there is ***rarely*** one standard simplest solution. The process to generate a **regular expression** is **cognitive**.

  - Second, the **algorithms** to **recognize** **regular expressions**, ***do not*** require to simplify them.

2.3. Examples of Regular Expressions

Example #1:

Given $\sum = \{a,b,c\}$, write a Regular Expression that represents the set of all strings that contain ***exactly one*** *b*:

Solution:

(a|c)*b(a|c)*

Although **b** appears in the center of the regular expression, it may not necessarily be in the center of a string. The zero or more repetitions of **a** or **c** before and after **b** satisfies the definition.

✎ *Example #2*:

Given ∑ = {*a*,*b*,*c*}, write a Regular Expression that represents the set of all strings that contain ***at most one*** *b*:

Solution:

The solution for example #1 can be used. First, we add the part of the regular expression that contains no *b* at all:

(a|c)*

Second, we add the part of the regular expression, which recognize strings with only one *b* with a **choice** operand (**OR**):

(a|c)* | (a|c)*b (a|c)*

An ***alternative*** would allow either *b* or the *empty string* to appear between the two repetitions of *a* or *c*:

(a|c)* (b|ε) (a|c)*

A ***similar*** solution but by using the non-standard ***Zero or One*** operation :

(a|c)* b$^{?}$ (a|c)*

☝ This example shows that the ***same language*** can be generated by ***many different*** **regular expressions**.

✎ *Example #3*:

Consider the set of strings ***S*** over the alphabet ∑ = {*a*,*b*}, consisting of a single *b* surrounded by the same number of *a*'s. Provide its RE:

$$\boldsymbol{S} = \{b, aba, aabaa, aaabaaa, \ldots\} = \{a^n b a^n \mid n \geq 0\}$$

Solution:

Set ***S*** ***can't*** be described by a **regular expression**, due the **Kleen** (*) and the **Positive** (+) closures are the only **repetition** operations. Therefore, the expression **a*ba*** is the ***closest*** regular expression for ***S***. However, due regular expressions ***can't count***, with the previous expression there is no guarantee that the number of *a*'s before and after the *b* will be the same.

✎ *Example #4*:

Consider the alphabet ∑ = {*a*,*b*,*c*} and the following regular expression:

((b|c)*a (b|c)*a (b|c)*)*

Provide a concise English description of the language it generates.

Solution:

It generates the language ***L*** of strings that contains the empty string, or an even number of *a*'s, and any number of *b*'s and *c*'s can appear before, between, or after the two *a*'s:

L = {*ε*, *aa*, *aaaa*, *aaaaaa*, *baab*, *aab*, *baa*, *aba*, *bbaabb*, *baba*, *babab*, *babababa*, *bababababab*, … }

2.4. Regular Expressions For Tokens

① ***Numbers***: they can be ***any*** sequence of ***digits***, or decimal numbers, or numbers with an exponent (using "*e*" or "*E*").

Examples: 23156, 3.53E-2 which is the same as 0.0353

❒ **Regular Expressions**:

***num* = [0-9]+**
***signedNum* = (+|-)? num**
***number* = *signedNum* (“.” *num*)? (E *SignedNum*)?**

② ***Reserved Words***: its regular expressions are the ***simplest*** due they are represented by their fixed sequence of characters.

Examples: **if**, **else**, **while**, **do**, **for**,…….

❒ **Regular Expression**:

***reservedWord* = if | else | while | do | for | …**

③ ***Identifiers***: their regular expression depends on the ***definition*** of the ***programming language***. Normally, and ***identifier*** must begin with a letter and contains only letters and digits.

Examples: *speed*, *frequency*, *var1*, ...

❒ **Regular Expression**:

***letter* = [a-zA-Z]**
***digit* = [0-9]**
identifier* = *letter* (*letter* | *digit*)

④ ***Comments***: They are ***normally ignored*** during the scanning process. They are treated as ***pseudotokens***. That is, they ***must*** be recognized and discarded. The form of ***comments*** depends on the ***definition*** of the ***programming language***.

Examples: The form of ***comments*** in ***C*** language:

/* this is a comment */

❒ **Regular Expression**:

***comment* = "/*".*"*/"**

☝ Since "*" and "/" are metacharacters of regular expressions, writing a regular expression for ***C*** comments is complicated; that it's almost never written in practice.

⑤ ***Ambiguity***, ***White Space***, and ***Lookahead***: When regular expressions are used to describe programming languages, some strings can be matched by ***different*** regular expressions.

Examples: Strings such as **if** and **while** could be either ***identifiers*** or ***keywords***.

The string **<>** might be interpreted as representing two tokens, the "*less-than*" and the "greater-than"; or a single token, the "*not-equal-to*".

☝ A ***language definition must*** give ***disambiguating rules*** that will stablish which meaning is given for each case

☛ The following rules ***shall*** be observed:

① When a string can be either an ***identifier*** or a ***keyword***, the string ***must*** be identified as a ***keyword***. The term ***reserved word*** is also used to denominate a ***keyword***. It means that a ***keyword can not*** be an ***identifier***.

② When a string can be a *single token* or a ***sequence*** of *several tokens*, the ***single-token*** interpretation ***must*** be used. This preference is usually known as the ***principle of longest substring***, i.e., the ***longest*** string of characters that could form a single token will be the next token identified.

Therefore, from the previous example, the string **<>** ***must*** be interpreted as representing a single token, the "*not-equal-to*".

- The usage of the ***principle of longest substring***, arises the necessity to define ***token delimiters***, i.e., those characters that ***unambiguously*** separate one token from another.
  - *Space*, *newline*, and *tab* are ***token delimiters***.
  - ***Token delimiters*** end token strings and they are ***not*** part of the token itself.
  - *Space*, *newline*, and *tab* ***must*** be disregarded.

- However, due tokens such as ***special symbols*** are also *token delimiters* for other tokens, the scanner ***must*** deal with the ***lookahead*** problem:
  - This is presented whenever the scanner encounters a ***special symbol*** token as a ***delimiter***. The special symbol ***must not*** be ***removed*** from the rest of the input either by returning it to the input ("*backing up*") or by looking ahead before removing the character from the input.

Example: in the string ***var1=var2***, the end of the identifier ***var1*** is found when the token "**=**" is found. Therefore, "**=**" ***must*** remain in the input due it represents the next token to be recognized.

3. Finite Automata

- **Finite Automata** or **Finite State Machines** or **Transition Diagrams** are a **mathematical** way of describing **algorithms** (**machines**) for **behavioral** systems.

- **Finite State Machines** can be used to describe the process of recognizing patterns in input strings. Therefore, they can be used to construct **lexical analyzers**.

- There is a ***strong*** relationship between **Finite State Machines** and **Regular Expressions.** A *finite automata* can be ***constructed*** from a *regular expression*.

- *Example*:

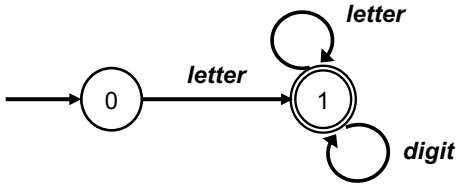
  If the pattern for identifiers, as they are usually defined in programming languages, is provided by the following ***Regular Expression*** :

  letter **= [a-zA-Z]**

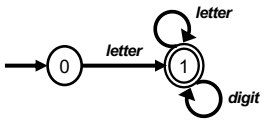
  digit **= [0-9]**

  identifier **=** ***letter*** **(*letter* | *digit*)***

  The ***Finite Automata*** that describes the process of recognizing strings from the previous ***Regular Expression*** is:

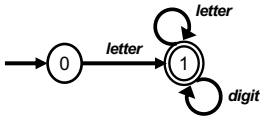


- The ***circles*** represent **states** which are locations in the process of recognition that record how much of the pattern has already been found.



- The ***arrowed lines*** represent **transitions** that record a change from one state (*initial*) to another (*final*) upon a match of the character or characters by which they are ***labeled***.
- In this example, *state 0* is the **start-state** which is where the recognition process begins. The ***standard*** is that the ***start-state*** is indicated by drawing an unlabeled arrowed line to it without initial state (*default transition*).

- In this example, *state 1* is the **accepting-state** that represents where the recognition process ends. ***Accepting-states*** are indicated by a circle with double-line border.



- A **Finite Automata** can have more than one **accepting-states**.

3.1. Definition of Deterministic Finite Automata

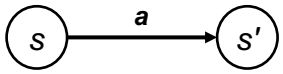
- A **Deterministic Finite Automata** is the one where any **final state** is ***uniquely*** given by the current **initial state** and the **current input** character of its **transition**.

- **Formal Description of a Finite Automaton**:

  Formally speaking, the definition of a **DFA *M*** is a 5-tuple: (***S***, Σ, δ, s_0, ***A***,), where ***S*** is a finite set of states, Σ is a finite input alphabet, a start state $s_0 \in S$, a set of accepting states $A \subseteq S$, and δ is the transition function mapping $S \times \Sigma \longrightarrow S$. That is, $\delta(s,a)$ is a state for each $s \in S$ and each input $a \in \Sigma$.

- ***S*** X Σ refers to the ***Cartesian Product*** or ***Cross Product*** of ***S*** and **Σ**: the set of **pairs** (*s*,*a*), where $s \in S$ and $a \in \Sigma$.

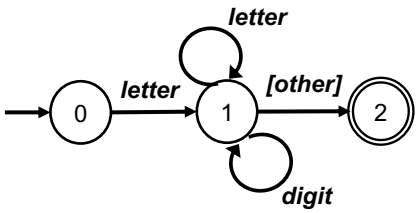
- The function δ records the **transitions**: $\delta(s,a) = s'$, if there is a transition from state *s* to state *s'* labeled by *a*.



3.2. Implementation of a DFA in Code

- There are several ways to translate a **Deterministic Finite Automata** into **code**. The **Direct-Coded** and the **Table-Driven** approaches will be studied.

  The following **DFA** that accepts **identifiers** consisting of a *letter* followed by a ***sequence*** of *letters* and/or *digits*, will be used to illustrate the two implementation approaches.



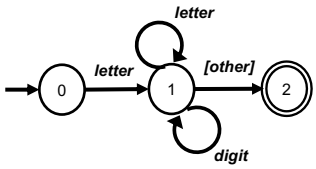
The **DFA** continues to match letters and digits until a ***delimiter*** character (*space*, or any special character defined by the language) is found.

- The **accepting-state** has an input transition labeled with **square brackets** to indicate that a ***delimiter*** character has been found and it should be considered **lookahead**, i.e., it should ***NOT*** be consumed and must be returned to the input string.

- **Direct-Coded Scanners**:

  The ***easiest*** way to simulate a **DFA** is to directly write the code that represents the DFA's behavior.

  For the previous DFA that recognize identifiers, its pseudo code in ***C*** may be the following:




```
state = 0;        /* start */
while (state == 0 || state == 1) {
      switch (state) {
         case 0: switch (input character ) {
                      case letter:
                         advance the input;
                         state = 1;
                         break;
                      default: state = error-state;
                    } /* end switch */
         case 1: switch (input character ) {
                      case letter: case digit:
                         advance the input;
                         state = 1;
                         break;
                      default: state = 2;
                    } /* end switch */
      } /* end switch */
} /* end while */
if (state ==2)
   accept;
else
   error;
```


- This approach is reasonable if there are ***NOT*** too many states and if loops in the **DFA** are small. This type of code is **used** to generate ***small*** scanners.

- ✗ There are two ***disadvantages*** of this method:
  - It is **ad hoc**. Each **DFA** has its own singular piece of code.
  - The **complexity** of the code ***increases dramatically*** as the number of states grows.

❒ **Table-Driven Scanners**:

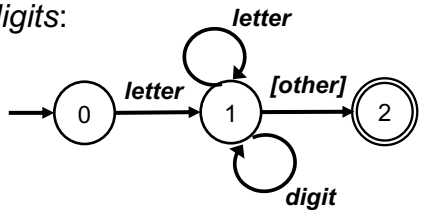
A **DFA** is expressed as a ***data structure*** together with a ***generic*** code that will use the information stored in the data structure in order to take an action.

- A simple ***data structure*** that is is used for this method is a **Transition Table** implemented as a **two-dimensional array**. The transition table is **indexed** by a **state** and by an **input character** that provides the values for the transition function δ.

States s	Character in the Alphabet c
	$\delta(s,c) = s'$

✎ *Example*:

Let's use the same previously described **DFA** that accepts **identifiers** consisting of a *letter* followed by a ***sequence*** of *letters* and/or *digits*:



The **Transition Table** for this **DFA** is the following:

state \ Input char	letter	digit	other	accepting
0	**1**			**no**
1	**1**	**1**	**[2]**	**no**
2				**yes**

- States between square **brackets** represent ***non-consuming* transitions**.
- **Blank entries** in the ***transition table shall*** represent ***error*** states.

☛ The **generic code** for the previous ***transition table*** considers that:

- Transitions are kept in the transition array ***T*** indexed by *states* and *input characters*.
- Transitions that ***advance the input*** (those not marked with square brackets) are given by the Boolean array ***Advance***, indexed by *states* and *input characters*.

- **Accepting states** are given by the Boolean array ***Accept***, indexed by *states*.
- **Error states** are given by the Boolean array ***Error***, indexed by *states*.

Generic Code to Process a Transition Table

```
state = 0;        /* start */
ch = next input character;
while (!Accept[state] && !Error[state] ) {
    newState = T[state, ch];
    if (Advance[state, ch])
        ch = next input character ;
    state = newState;
} /* end while */
if (Accept[state] )
   RecordToken;
else
   error;
```

✓ **Table-Driven** has certain ***advantages*** :

- The **size** of the **code** is significantly ***reduced***.
- The **same code** will work for many different DFAs.
- The **code** is ***easier*** to **change**.

✗ The ***disadvantage*** is that **transition tables** can become ***very large***, thus generating a significant increase in the space used by the scanner.

❍ **Additonal Considerations for Implementation**

① In order to make the previous **generic code** able to process the set of **DFA's** that described a **programming language**, the code should keep reading the input file until ***EOF*** is found.

② The **generic code** ***must*** handle the **Symbol Table**. An approach could be to use ***separate*** **tables** for each type of token: *variables*, *integers*, *floats*, *strings*, *chars*, etc. `

③ The **generic code** ***must*** provide the corresponding behavior for each **accepting state**, i.e., generate the list of tokens, fill the corresponding symbol table, etc.

④ The **generic code** ***shall*** provide a suitable ***error message*** whenever an **error state** is reached.

Example:

The programming language ***Mini*** has a very simple structure:

1. It consists of a sequence of statements separated by semicolons ("**;**").
2. There are no procedures and no declarations.
3. All ***variables*** are integer variables, and they are declared by assigning values to them. The assignation operator is "**:=**".
4. There are only two control statements: an **if-then** statement and a **do-while** statement. Both control statements may contain inside themselves statement sequences.

5. The **if-then** statement has an optional **else** part, and must terminate with the keyword **end**.
6. It has **read** and **write** statements to perform input/output operations.
7. Comments are **C** language style, i.e., **/*** … ***/**
8. Expressions are limited to Boolean and integer arithmetic expressions.
9. An arithmetic expression may involve **integer constants**, **variables**, **parenthesis**, and any of the four integer operators **+**, **-**, ***** and **/**.

10. A Boolean expression is a comparison between two arithmetic expressions.
    - The comparison operators are: **<**, **>**, and **=**.
    - Boolean expressions may appear only as a test in control statements.
    - There are no Boolean variables, assignment, or I/O.

An example program:

```
/*Sample program in Mini language
  that computes the factorial function*/
read x; /*inputs an integer*/
If  x > 0 then /*do not compute if x<=0*/
    fact := 1;
    do
        fact := fact * x;
        x := x - 1;
    while x = 0;
    write fact; /*output factorial of  x*/
end /* End of IF Statement */
```


11. The regular expression for identifiers is:

letter = [a-zA-Z]
digit = [0-9]
identifier = *letter* (*letter* | *digit*)*

12. The regular expression for integer constants is:

digit = [0-9]
integers = *digit*+

13. From the previous information, programming language ***Mini*** has the following list of valid tokens:

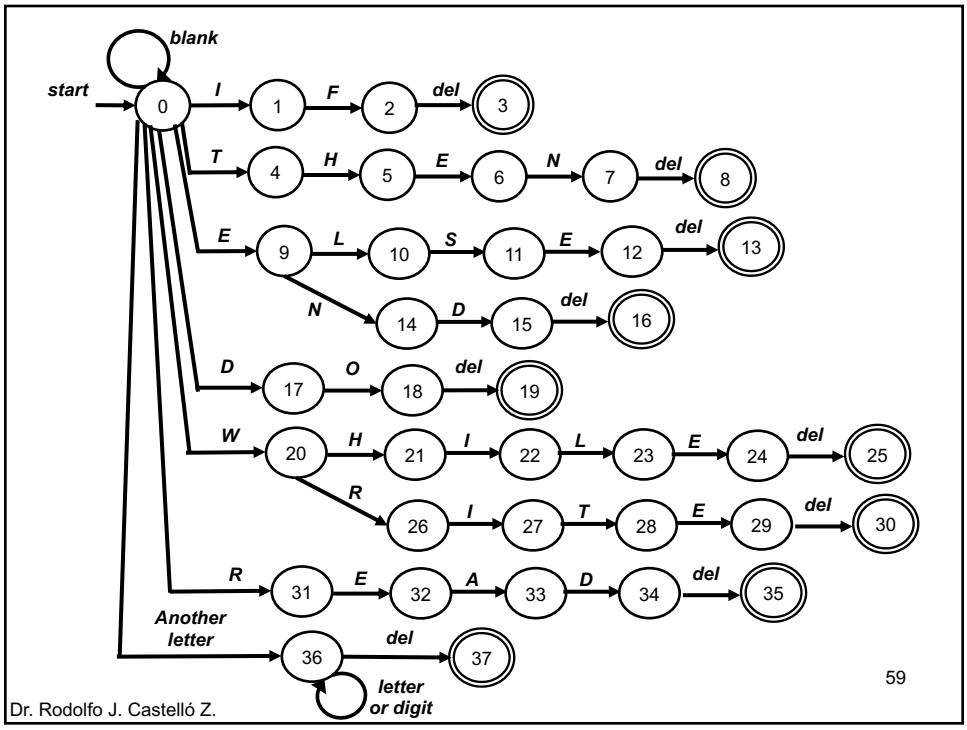
1. IF	9. ;	17. =
2. THEN	10. :=	18. <
3. ELSE	11. +	19. >
4. END	12. -	20. Identifier
5. DO	13. *	21. Integer constant
6. WHILE	14. /	22. Comments
7. READ	15. (	
8. WRITE	16.)	

14. Set of ***delimiters***:

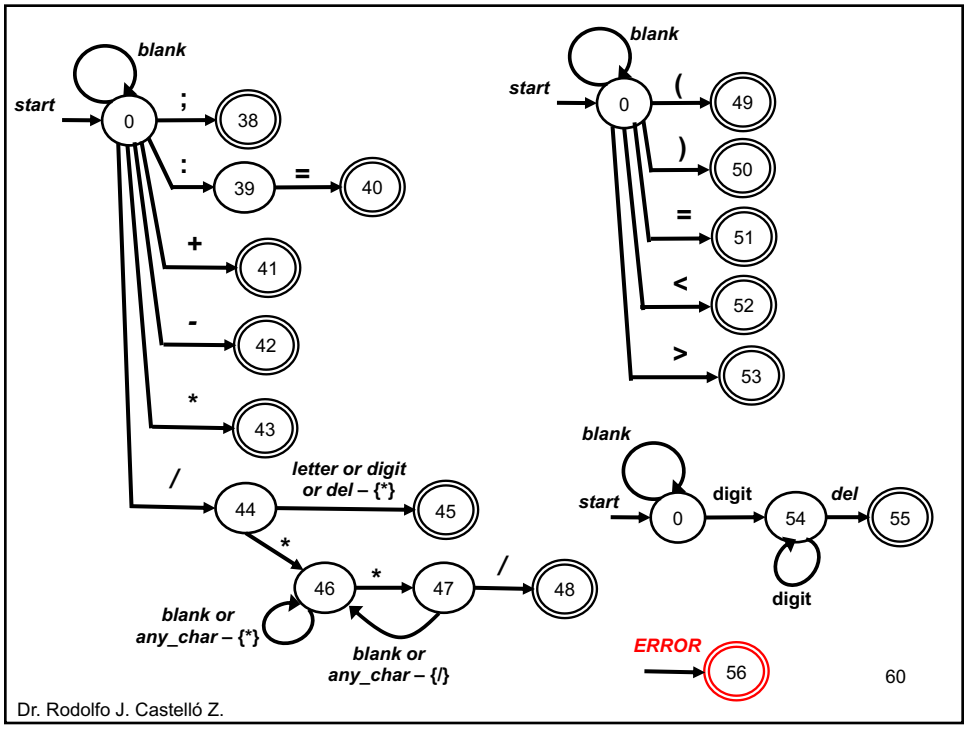
del = {*blank*, **;**, **:**, **=**, *****, **+**, **-**, **/**, **(**, **)**, **<**, **>**}

✎ *Exercise #1*:

Provide the Transition Diagrams for the tokens:


blank
start
0
I
1
F
2
del
3
T
4
H
5
E
6
N
7
del
8
E
9
L
10
S
11
E
12
del
13
N
14
D
15
del
16
D
17
O
18
del
19
W
20
H
21
I
22
L
23
E
24
del
25
R
26
I
27
T
28
E
29
del
30
R
31
E
32
A
33
D
34
del
35
Another letter
36
del
37
letter or digit
Dr. Rodolfo J. Castelló Z.
59

59


blank
start
0
;
38
:
39
=
40
+
41
-
42
*
43
/
44
letter or digit or del – {*}
45
*
46
*
47
/
48
blank or any_char – {*}
blank or any_char – {/}
blank
start
0
(
49
)
50
=
51
<
52
>
53
blank
start
0
digit
54
del
55
digit
ERROR
56
Dr. Rodolfo J. Castelló Z.
60

60

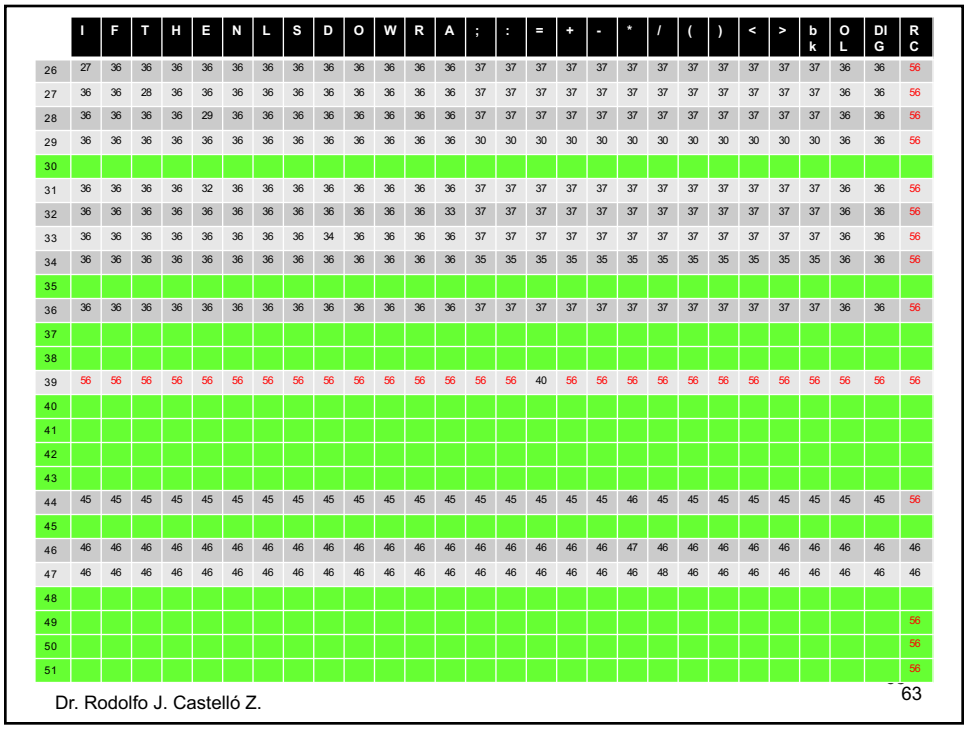
- *Exercise #2*:

  Provide the Transition Table for the previous transition diagrams:

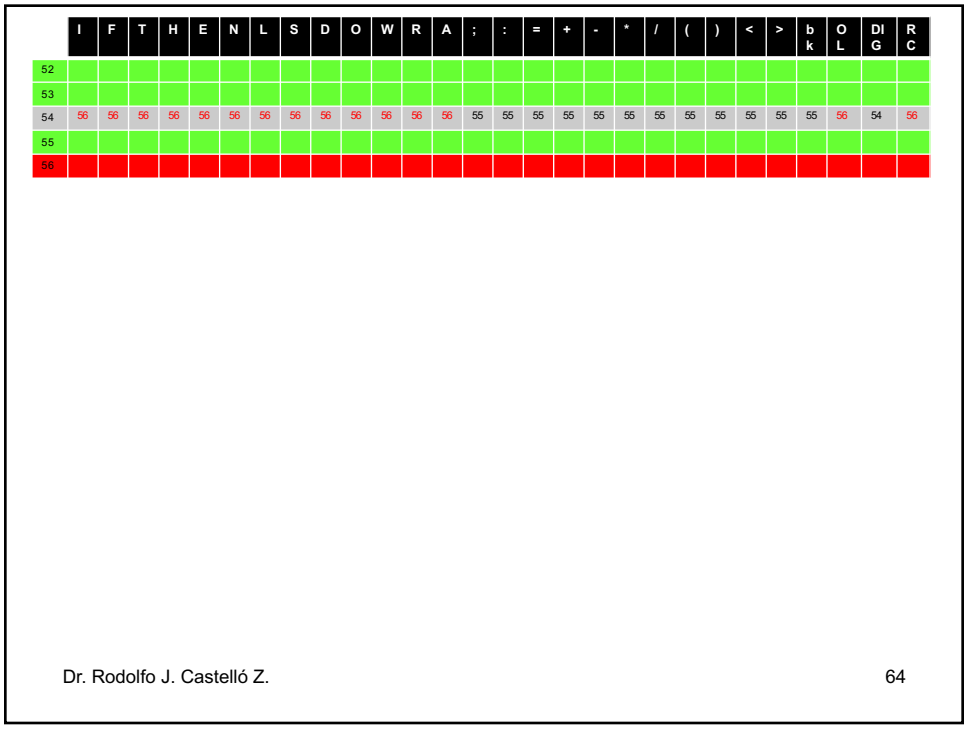
61

	I	F	T	H	E	N	L	S	D	O	W	R	A	;	:	=	+	-	*	/	(	)	<	>	bk	OL	DIGI	RC
0	1	36	4	36	9	36	36	36	17	36	20	31	36	38	39	51	41	42	43	44	49	50	52	53	0	36	54	56
1	36	2	36	36	36	36	36	36	36	36	36	36	36	37	37	37	37	37	37	37	37	37	37	37	37	36	36	56
2	36	36	36	36	36	36	36	36	36	36	36	36	36	3	3	3	3	3	3	3	3	3	3	3	3	36	36	56
3																												
4	36	36	36	5	36	36	36	36	36	36	36	36	36	37	37	37	37	37	37	37	37	37	37	37	37	36	36	56
5	36	36	36	36	6	36	36	36	36	36	36	36	36	37	37	37	37	37	37	37	37	37	37	37	37	36	36	56
6	36	36	36	36	36	7	36	36	36	36	36	36	36	37	37	37	37	37	37	37	37	37	37	37	37	36	36	56
7	36	36	36	36	36	36	36	36	36	36	36	36	36	8	8	8	8	8	8	8	8	8	8	8	8	36	36	56
8																												
9	36	36	36	36	36	14	10	36	36	36	36	36	36	37	37	37	37	37	37	37	37	37	37	37	37	36	36	56
10	36	36	36	36	36	36	36	11	36	36	36	36	36	37	37	37	37	37	37	37	37	37	37	37	37	36	36	56
11	36	36	36	36	12	36	36	36	36	36	36	36	36	37	37	37	37	37	37	37	37	37	37	37	37	36	36	56
12	36	36	36	36	36	36	36	36	36	36	36	36	36	13	13	13	13	13	13	13	13	13	13	13	13	36	36	56
13																												
14	36	36	36	36	36	36	36	36	15	36	36	36	36	37	37	37	37	37	37	37	37	37	37	37	37	36	36	56
15	36	36	36	36	36	36	36	36	36	36	36	36	36	16	16	16	16	16	16	16	16	16	16	16	16	36	36	56
16																												
17	36	36	36	36	36	36	36	36	36	18	36	36	36	37	37	37	37	37	37	37	37	37	37	37	37	36	36	56
18	36	36	36	36	36	36	36	36	36	36	36	36	36	19	19	19	19	19	19	19	19	19	19	19	19	36	36	56
19																												
20	36	36	36	21	36	36	36	36	36	36	36	26	36	37	37	37	37	37	37	37	37	37	37	37	37	36	36	56
21	22	36	36	36	36	36	36	36	36	36	36	36	36	37	37	37	37	37	37	37	37	37	37	37	37	36	36	56
22	36	36	36	36	36	36	23	36	36	36	36	36	36	37	37	37	37	37	37	37	37	37	37	37	37	36	36	56
23	36	36	36	36	24	36	36	36	36	36	36	36	36	37	37	37	37	37	37	37	37	37	37	37	37	36	36	56
24	36	36	36	36	36	36	36	36	36	36	36	36	36	25	25	25	25	25	25	25	25	25	25	25	25	36	36	56
25																												

62



	I	F	T	H	E	N	L	S	D	O	W	R	A	;	:	=	+	-	*	/	(	)	<	>	bk	OL	DlG	RC
26	27	36	36	36	36	36	36	36	36	36	36	36	36	37	37	37	37	37	37	37	37	37	37	37	37	36	36	56
27	36	36	28	36	36	36	36	36	36	36	36	36	36	37	37	37	37	37	37	37	37	37	37	37	37	37	36	36	56
28	36	36	36	36	29	36	36	36	36	36	36	36	36	37	37	37	37	37	37	37	37	37	37	37	37	37	36	36	56
29	36	36	36	36	36	36	36	36	36	36	36	36	36	30	30	30	30	30	30	30	30	30	30	30	30	36	36	56
30																												
31	36	36	36	36	32	36	36	36	36	36	36	36	36	37	37	37	37	37	37	37	37	37	37	37	37	36	36	56
32	36	36	36	36	36	36	36	36	36	36	36	36	33	37	37	37	37	37	37	37	37	37	37	37	37	36	36	56
33	36	36	36	36	36	36	36	36	34	36	36	36	36	37	37	37	37	37	37	37	37	37	37	37	37	36	36	56
34	36	36	36	36	36	36	36	36	36	36	36	36	36	35	35	35	35	35	35	35	35	35	35	35	35	36	36	56
35																												
36	36	36	36	36	36	36	36	36	36	36	36	36	36	37	37	37	37	37	37	37	37	37	37	37	37	36	36	56
37																												
38																												
39	56	56	56	56	56	56	56	56	56	56	56	56	56	56	56	40	56	56	56	56	56	56	56	56	56	56	56	56
40																												
41																												
42																												
43																												
44	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	46	45	45	45	45	45	45	45	45	56
45																												
46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	47	46	46	46	46	46	46	46	46	46
47	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	48	46	46	46	46	46	46	46	46
48																												
49																												56
50																												56
51																												56



	I	F	T	H	E	N	L	S	D	O	W	R	A	;	:	=	+	-	*	/	(	)	<	>	bk	OL	DlG	RC
52																												
53																												
54	56	56	56	56	56	56	56	56	56	56	56	56	56	55	55	55	55	55	55	55	55	55	55	55	55	56	54	56
55																												
56																												

✎ *Exercise #3*:

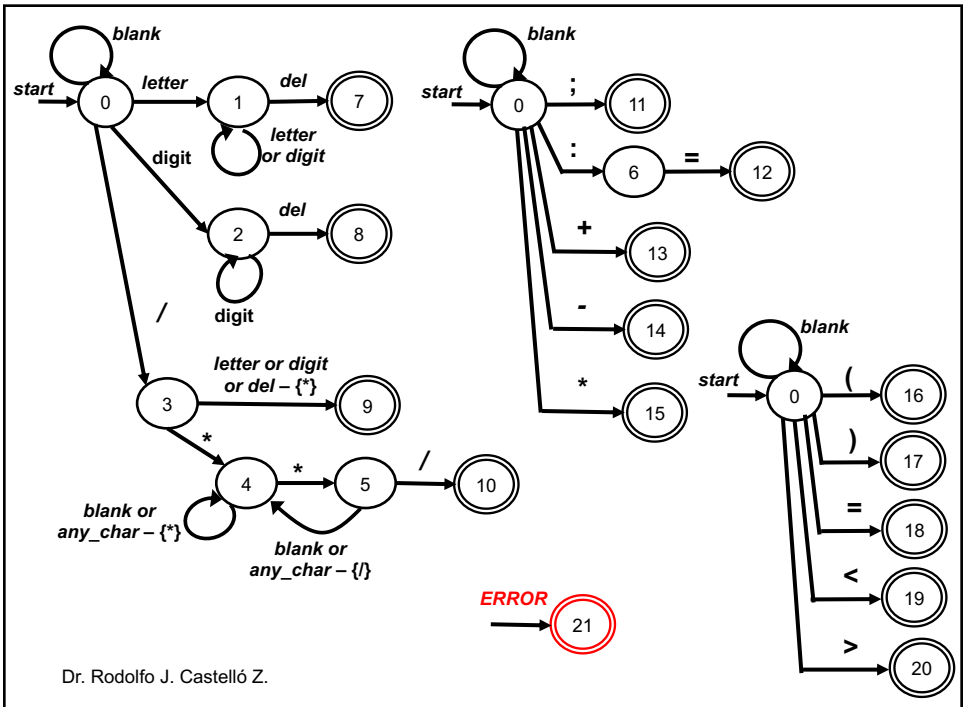
Provide the code in ***C*** language for the ***Mini*** language scanner:

☝ Before developing the scanner program two ***improvements*** can be done to simplify the transition table :

① In order to eliminate the **Transition Diagram** to identify each **Reserved Word**, every time any **identifier** is found, the **scanner *must*** ask if it is reserved word.

② In order to reduce the size of the **Transition table**, all **accepting** and **error states *shall*** be assigned the last number of states.

65



66

	letra	digito	;	:	=	+	-	*	/	(	)	<	>	blank	Rare Char	
0	1	2	11	6	18	13	14	15	3	16	17	19	20	0	21	
1	1	1	7	7	7	7	7	7	7	7	7	7	7	7	21	
2	21	2	8	8	8	8	8	8	8	8	8	8	8	8	21	
3	9	9	9	9	9	9	9	4	9	9	9	9	9	9	21	
4	4	4	4	4	4	4	4	5	4	4	4	4	4	4	4	
5	4	4	4	4	4	4	4	4	10	4	4	4	4	4	4	
6	21	21	21	21	12	21	21	21	21	21	21	21	21	21	21	
7																→ Identifier
8																→ Integer constant
9																→ /
10																→ comments
11																→ ;
12																→ :=
13																→ +
14																→ -
15																→ *
16																→ (
17																→)
18																→ =
19																→ <
20																→ >
21																→ *Error*

67

67

4. How to Manage Symbol Tables

- **Symbol Tables** are data structures that are used by the compiler to hold ***information*** about source-programs **constructs**, i.e., *variables*, *constants*, *procedures*, *functions*, *labels*, *structures*, etc.
- Entries in the **Symbol Tables** contain information about an **identifier** such as its **string** or **lexeme**, its type, and any other relevant information.
- Information is collected ***incrementally*** by the **Analysis phase** (**Lexical**) and used by the **Synthesis phases.**
- **Symbol Tables *must*** support multiple declarations of the same identifier. Therefore, the **scope** of a declaration ***shall*** be considered. This aspect is delayed until the **Synthesis phases.**

Dr. Rodolfo J. Castelló Z. 68

68

4.1. Symbol Tables for the Lexical Analysis Phase

- **Symbol Table for Identifiers**:

  ① During the Analysis phase, this table ***may*** have only one field, the **string** or **lexeme** found.

  ② Information such as if it's a *variable*, *array*, *data type*, *function*, *structure*, or *class*, will be introduced during the **Syntax phase**.

Entry #	Lexeme
0	ABC
1	Var1
2	X
3	Fun1

- **Symbol Table for Numbers**:

  ① During the Analysis phase, this table ***may*** have only two fields, the **string** or **lexeme** found and the type which is found by the corresponding transition diagram.

  ② Information such as it's *scope* will be introduced during the **Syntax phase**.

Entry #	Lexeme	Type
0	1234	int
1	56.78	float
2		
3		

❒ Symbol Table for Strings:

① During the Analysis phase, this table ***may*** have only one field, the **string** or **lexeme** found.

② Information such as it's *scope* will be introduced during the **Syntax phase**.

Entry #	Lexeme
0	"ABC"
1	"xyz"
2	
3	

✎ *Example of Scanner Output*:

Given the Following Source Program File written in ***Mini*** Language and their corresponding Token IDs:

Token IDs

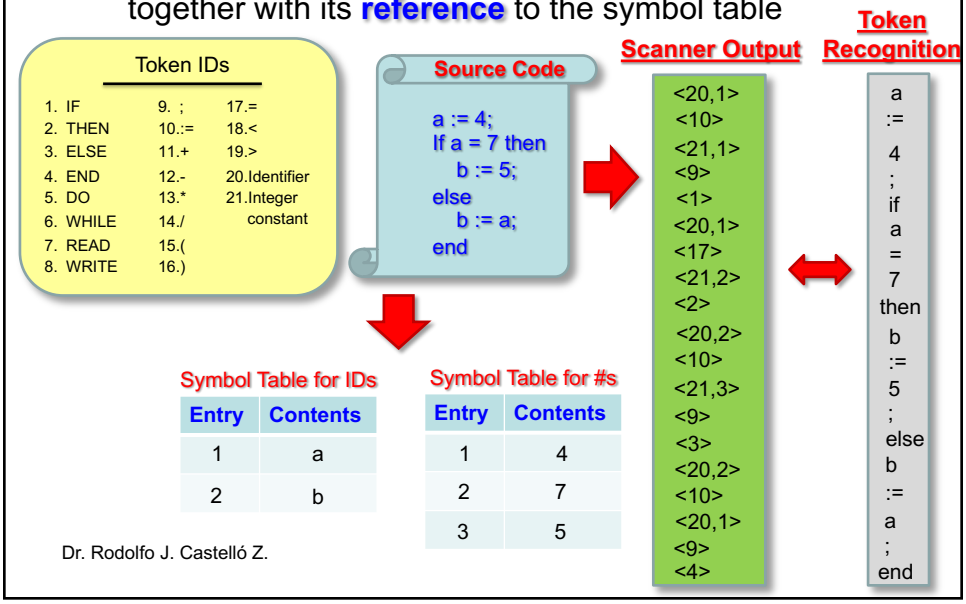
1. IF	9. ;	17.=
2. THEN	10.:=	18.<
3. ELSE	11.+	19.>
4. END	12.-	20.Identifier
5. DO	13.*	21.Integer constant
6. WHILE	14./	
7. READ	15.(	
8. WRITE	16.)	

Source Code

```
a := 4;
If a = 7 then
   b := 5;
else
   b := a;
end
```

Provide the **Scanner's** output and the corresponding **symbol table**.

Remember that the structure of a **token** is composed by the ***token-id***, and in case of an ***identifier*** or ***number***, together with its **reference** to the symbol table



Symbol Table for IDs

Entry	Contents
1	a
2	b

Symbol Table for #s

Entry	Contents
1	4
2	7
3	5

Dr. Rodolfo J. Castelló Z.

73

5. From Regular Expressions to DFAs

The **Regular Expression** is the preferred notation for describing **Lexical Analyzers** and other **pattern-processing software**. An example is the **Lex**/**Flex** tool, that automatically generates a Scanner from a Regular Expression.

The process to convert a **Regular Expression** (**RE**) into a **Deterministic Finite Automata** (**DFA**) involves two steps: convert the **RE** to a **Non-Deterministic Finite Automata** (**NFA**), then convert the **NFA** to a **DFA**.

Dr. Rodolfo J. Castelló Z. 74

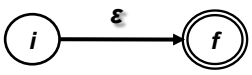
74

5.1. From RE to NFA

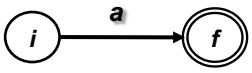
- The **Thompson's Construction** approach will be used.
- It uses **ε-transitions** to join all the machines that correspond to each part of a **RE** in order to form the final machine that represents the whole **RE**.

① **Machines for Basic REs**.

- **NFA** that accepts the expression **ε** (empty string):

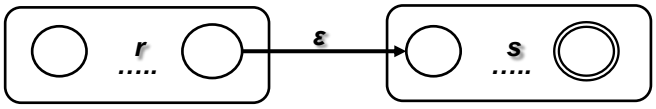


- **NFA** for any subexpression **a** ∈ ∑ :



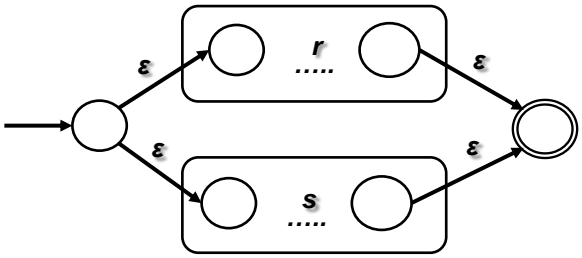
② **Machine for Concatenation of two REs**.

- **NFA** equivalent to the RE ***rs***, where ***r*** and ***s*** are regular expressions:



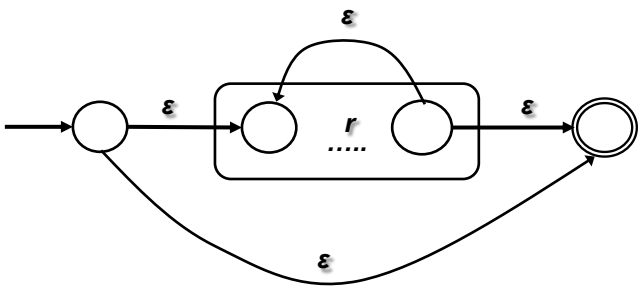
③ Machine for Choice Among two REs.

- **NFA** equivalent to the RE ***r|s***, where ***r*** and ***s*** are regular expressions:



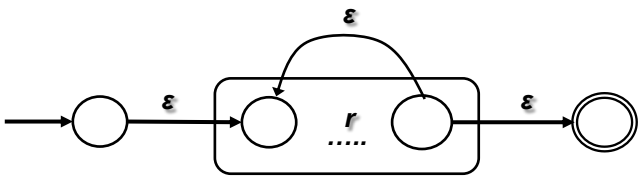
④ Machine for Kleen Closure (repetition of cero or more) of a RE.

- **NFA** equivalent to the RE ***r****, where ***r*** is a regular expression:



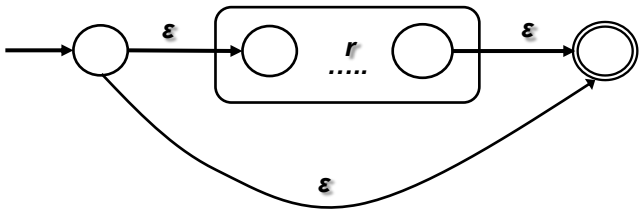
⑤ Machine for Positive Closure (repetition of one or more) of a RE.

- **NFA** equivalent to the RE ***r+***, where ***r*** is a regular expression:



⑥ Machine for cero or one of a RE.

- **NFA** equivalent to the RE $r^?$, where ***r*** is a regular expression:

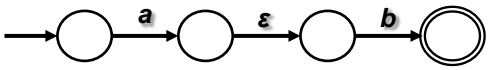


✎ *Example #1*:

Given the RE ***ab|a***, generate its corresponding NFA:

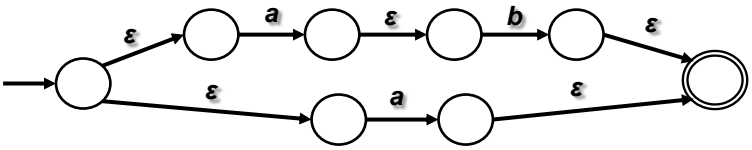
① First, the NFA for each basic RE ***a*** and ***b*** are generated.

② Second, the NFA for the RE ***ab*** is generated.

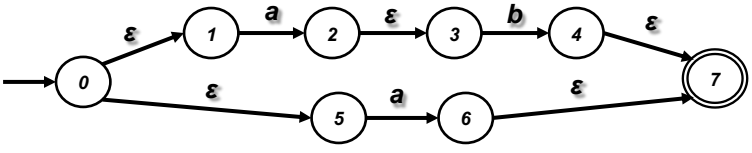


③ Third, the NFA for the RE ***ab|a*** is generated.

ab|a



④ Finally, all states are enumerated.

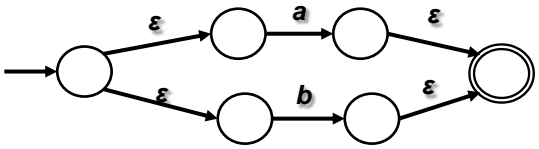


Example #2:

Given the RE (***a***|***b***)****abb***, generate its corresponding NFA:

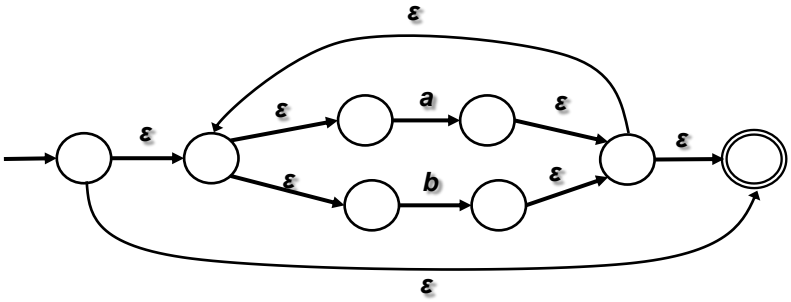
① The NFA for each basic RE ***a*** and ***b*** are generated.

② The NFA for the RE ***a***|***b*** is generated.

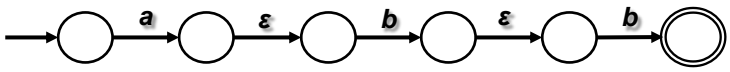


③ The NFA for the RE (***a***|***b***)* is generated.

(***a***|***b***)****abb***

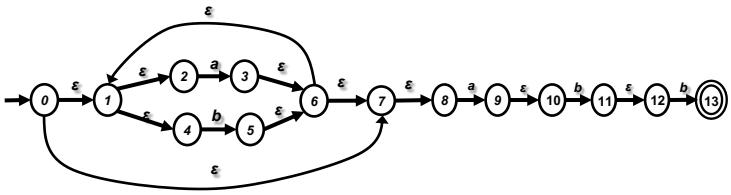


④ The NFA for the RE ***abb*** is generated.



⑤ The NFA for the RE (***a***|***b***)* ***abb*** is generated.

(***a***|***b***)****abb***

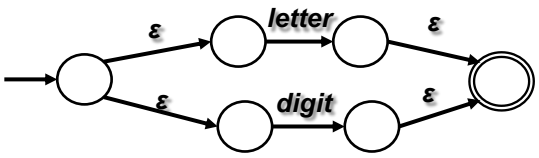


Example #3:

Given the RE ***letter***(***letter***|***digit***)* used to recognize **identifiers**, generate its corresponding NFA:

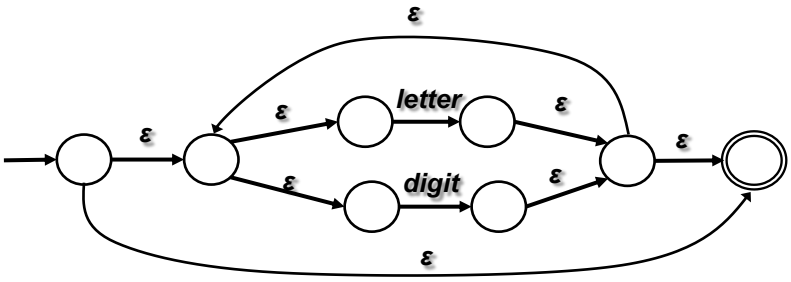
① The NFA for each basic RE ***letter*** and ***digit*** are generated.

② The NFA for the RE ***letter***|***digit*** is generated.

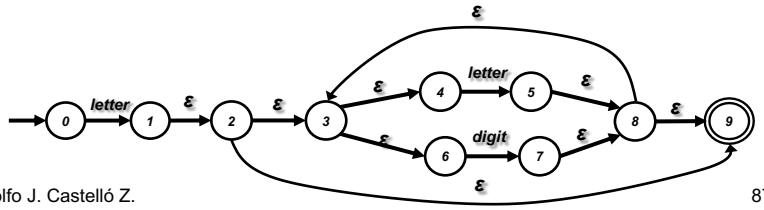


*letter(letter|digit)**

③ The NFA for the RE (***letter|digit***)* is generated.



④ The NFA for the RE ***letter***(***letter|digit***)* is generated.



5.2. From NFA to DFA

☛ The **Subset Construction** algorithm takes an **NFA** and constructs its equivalent **DFA**, i.e., one that accepts precisely the same strings but ***without*** **ε-transitions**.

☝ In order to eliminate **ε-transitions** it is needed the construction and usage of **ε-closures**.

- ➪ The **ε-closure** of a single state **s**, is the set of states that are reachable by a series of zero or more **ε-transitions**.
- ➪ Therefore, the **ε-closure** of a state **s**, always contains **s** itself.

❒ **Subset Construction Algorithm** :

In order to generate a **DFA** ***D*** from a given **NFA** ***N***, where s_i is a *state* of ***N*** and A_i is a subset of *states* of ***N*** which at the same time, becomes a *state* of ***D***:

① First, compute the ***ε-closure*** of the *start state* of ***N*** (s_0) which becomes the *start state* of ***D*** (A_0):

$$A_0 = \varepsilon\text{-closure}\ (s_0)$$

② For the set *start state* of ***D*** (A_0) and for each subsequent set A_i, compute transitions on each character $a \in \sum$ as follows:

$$A_i = \varepsilon\text{-closure}\ (move\ (A_{i-1}, a))$$

where *move* (A_{i-1}, *a*) is a set of states *t* of ***N***, such that there is a transition from a state $s \in A_{i-1}$ to state *t* on input symbol *a*.

$$move\ (A_{i-1}, a) = \{\ \forall t \mid (t \in N)\ \wedge (s \in A_{i-1})\ \wedge (a \in \sum) \wedge (s \xrightarrow{a} t)\ \}$$

A_i is a new state in ***D***, and also there is a new transition:

$$A_{i-1} \xrightarrow{a} A_i$$

③ The ***accepting states*** of ***D*** are all those sets A_i that include at least one *accepting state* of ***N***.

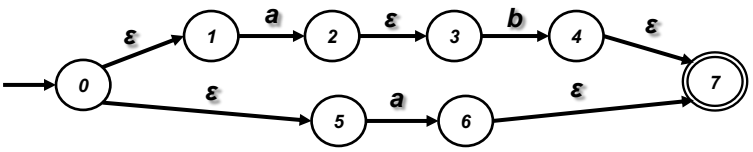
④ Continue with this process until no new set of states or transitions are created. This becomes the **DFA *D***.

⑤ For the purpose of clarity, a table with the following structure will be built during the **Subset Construction** algorithm.

NFA States	DFA States	a_i	a_{i+1}	a_{i+2}	.....

✎ *Example #1*:

Given the following NFA for the RE ***ab|a***, generate its corresponding DFA:

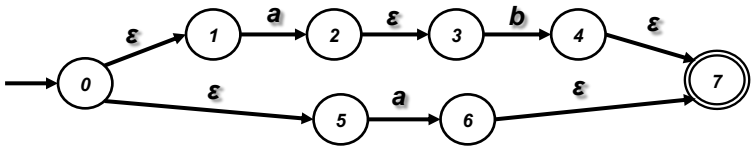


① First, the start state for the DFA is:

A = *ε-closure* (***0***) = {0,1,5}

NFA states	DFA states	a	b
{0,1,5}	A		

② Second, compute the following states of the DFA using $A_i = \varepsilon\text{-closure}(move(A_{i-1}, a))$:



$A = \{0,1,5\}$

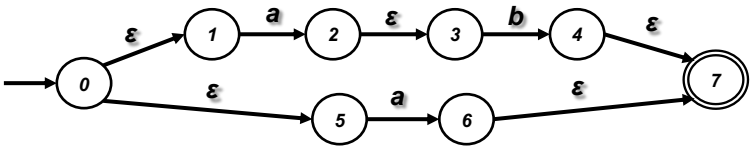
$\varepsilon\text{-closure}(move(A, a)) = \{2,3,6,7\} = B$

$\varepsilon\text{-closure}(move(A, b)) = \{\ \}$

NFA states	DFA states	a	b
{0,1,5}	A	B	-
{2,3,6,7}	B		

93

93



$B = \{2,3,6,7\}$

$\varepsilon\text{-closure}(move(B, a)) = \{\ \}$

$\varepsilon\text{-closure}(move(B, b)) = \{4,7\} = C$

NFA states	DFA states	a	b
{0,1,5}	A	B	-
{2,3,6,7}	B	-	C
{4,7}	C		

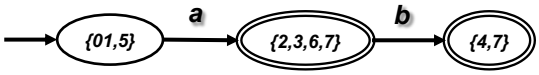
94

94

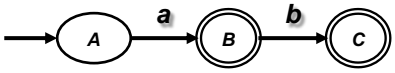
③ Due there are no more transitions on *a* or *b* from *C*, then the resulting ***Transition Table*** for the DFA, where *B* and *C* are **final states** for the DFA due they have state #7 as member, is the following:

NFA states	DFA states	a	b
{0,1,5}	A	B	-
{2,3,6,7}	B	-	C
{4,7}	C	-	-

Therefore, the corresponding DFA for the previous NFA is:

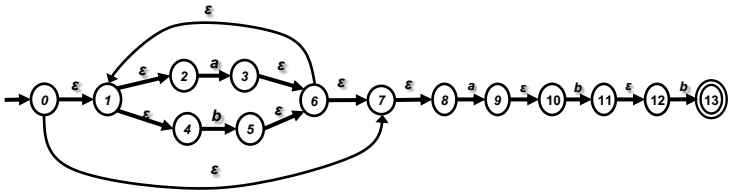


Which is the same as:



✎ *Example #2*:

Given the following NFA for the RE (***a|b***)****abb***, generate its corresponding DFA:

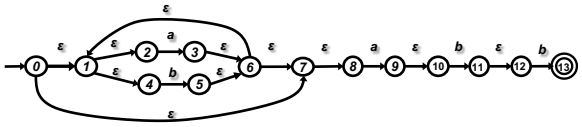


① First, the start state for the DFA is:

A = *ε-closure* (***0***) = {0,1,2,4,7,8}

NFA states	DFA states	a	b
{0,1,2,4,7,8}	A		

② Second, compute the following states of the DFA using $A_i = \varepsilon\text{-closure}(move(A_{i-1}, a))$:



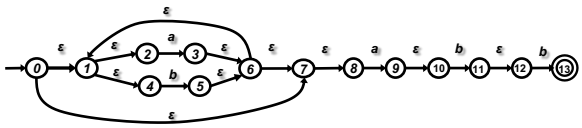
A = {0,1,2,4,7,8}
ε-closure (move (**A**, a)) = {1,2,3,4,6,7,8,9,10} = **B**
ε-closure (move (**A**, b)) = {1,2,4,5,6,7,8} = **C**

B = {1,2,3,4,6,7,8,9,10}
ε-closure (move (**B**, a)) = {1,2,3,4,6,7,8,9,10} = **B**
ε-closure (move (**B**, b)) = {1,2,4,5,6,7,8,11,12} = **D**

NFA states	DFA states	a	b
{0,1,2,4,7,8}	A	B	C
{1,2,3,4,6,7,8,9,10}	B	B	D
{1,2,4,5,6,7,8}	C		
{1,2,4,5,6,7,8,11,12}	D		

97

97



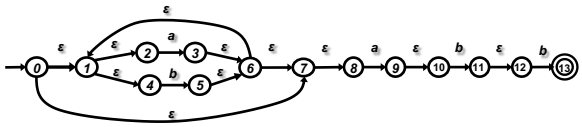
C = {1,2,4,5,6,7,8}
ε-closure (move (**C**, a)) = {1,2,3,4,6,7,8,9,10} = **B**
ε-closure (move (**C**, b)) = {1,2,4,5,6,7,8} = **C**

D = {1,2,4,5,6,7,8,11,12}
ε-closure (move (**D**, a)) = {1,2,3,4,6,7,8,9,10} = **B**
ε-closure (move (**D**, b)) = {1,2,4,5,6,7,8,13} = **E**

NFA states	DFA states	a	b
{0,1,2,4,7,8}	*A*	*B*	*C*
{1,2,3,4,6,7,8,9,10}	*B*	*B*	*D*
{1,2,4,5,6,7,8}	*C*	*B*	*C*
{1,2,4,5,6,7,8,11,12}	*D*	*B*	*E*
{1,2,4,5,6,7,8,13}	*E*		

98

98



E = {1,2,4,5,6,7,8,13}

ε-closure (*move* (***E***, *a*)) = {1,2,3,4,6,7,8,9,10} = ***B***

ε-closure (*move* (***E***, *b*)) = {1,2,4,5,6,7,8} = ***C***

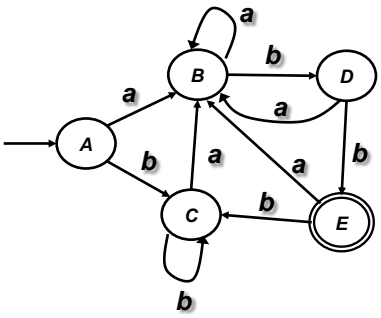
NFA states	DFA states	a	b
{0,1,2,4,7,8}	*A*	*B*	*C*
{1,2,3,4,6,7,8,9,10}	*B*	*B*	*D*
{1,2,4,5,6,7,8}	*C*	*B*	*C*
{1,2,4,5,6,7,8,11,12}	*D*	*B*	*E*
{1,2,4,5,6,7,8,13}	*E*	*B*	*C*

99

99

③ Due there are no new states, then the previous table is the resulting ***Transition Table*** for the DFA, where *E* is the **final state** for the DFA due it has state #13 as member.

Therefore, the corresponding DFA for the previous NFA with RE (***a***|***b***)****abb*** is:



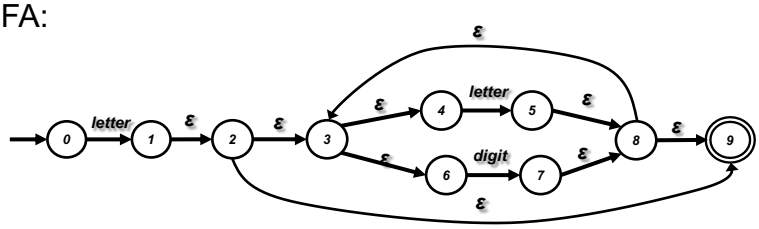
Dr. Rodolfo J. Castelló Z.

100

100

✎ *Example #3*:

Given the following NFA for the RE ***letter***(***letter***|***digit***)* used to recognize **identifiers**, generate its corresponding DFA:

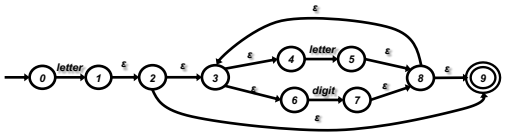


① First, the start state for the DFA is:

A = *ε-closure* (***0***) = {0}

NFA states	DFA states	letter	digit
{0}	A		

② Second, compute the following states of the DFA using $A_i = \varepsilon\text{-closure}(move(A_{i-1}, a))$:

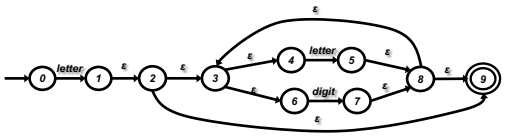


A = {0}

ε-closure (*move* (***A***, *letter*)) = {1,2,3,4,6,9} = ***B***

ε-closure (*move* (***A***, *digit*)) = { }

NFA states	DFA states	letter	digit
{0}	A	B	-



B = {1,2,3,4,6,9}

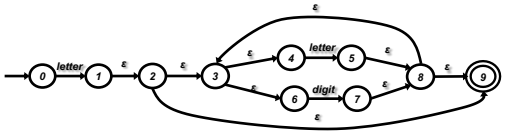
ε-closure (*move* (***B***, *letter*)) = {3,4,5,6,8,9} = ***C***

ε-closure (*move* (***B***, *digit*)) = {3,4,6,7,8,9} = ***D***

NFA states	DFA states	letter	digit
{0}	A	B	-
{1,2,3,4,6,9}	B	C	D
{3,4,5,6,8,9}	C		
{3,4,6,7,8,9}	D		

103

103



C = {3,4,5,6,8,9}

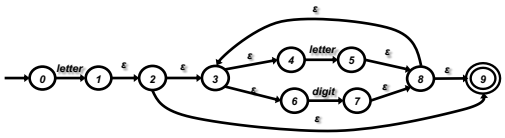
ε-closure (*move* (***C***, *letter*)) = {3,4,5,6,8,9} = ***C***

ε-closure (*move* (***C***, *digit*)) = {3,4,6,7,8,9 } = ***D***

NFA states	DFA states	letter	digit
{0}	A	B	-
{1,2,3,4,6,9}	B	C	D
{3,4,5,6,8,9}	C	C	D
{3,4,6,7,8,9}	D		

104

104



D = {3,4,6,7,8,9}
ε-closure (*move* (***D***, *letter*)) = {3,4,5,6,8,9} = ***C***
ε-closure (*move* (***D***, *digit*)) = {3,4,6,7,8,9} = ***D***

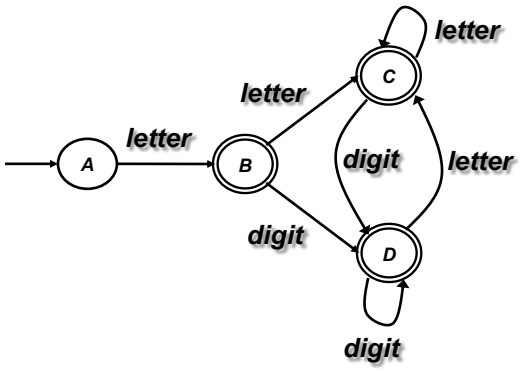
NFA states	DFA states	letter	digit
{0}	A	B	-
{1,2,3,4,6,9}	B	C	D
{3,4,5,6,8,9}	C	C	D
{3,4,6,7,8,9}	D	C	D

③ Due there are no new states, then the previous table is the resulting ***Transition Table*** for the DFA, where *B*, *C*, and *D*, are **final states** for the DFA due they have state #9 as member.

105

105

Therefore, the corresponding DFA for the previous NFA with RE ***letter***(***letter***|***digit***)* is:

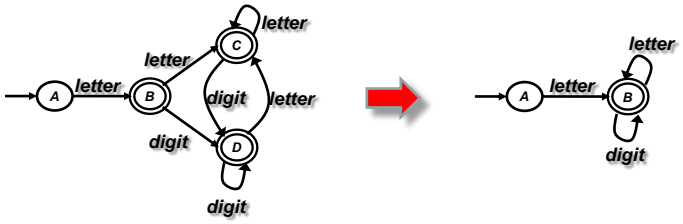


106

106

5.3. Minimizing the Number of States of a DFA

- The previous algorithm used to derive a DFA from a RE may result in a DFA that is ***more complex*** than necessary. An example of this remark is the result from the RE ***letter***(***letter***|***digit***)* of the last exercise :



- An important result from **automata theory** establishes that, given any DFA, there is an equivalent DFA containing a ***minimum number of states***, and that this minimum-state DFA is unique.

- The **state-minimization** algorithm works by ***partitioning*** the states of a DFA into ***group of states*** that ***can NOT*** be distinguished. Each group of states is then merged into a single state of the ***minimum-state DFA***:

  ① First, generate ***two*** sets, one consisting of all the ***accepting*** states, and the other consisting of all ***non-accepting*** states.

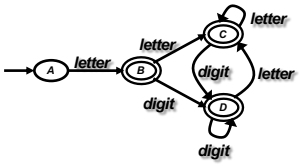
  ② Second, take any group of states, say group I=$\{s_1, s_2, \ldots s_n\}$ and some input $a \in \sum$, and verify if a can be used to separate any states in group I, i.e., if there is a transition form s_i to s_k on a that is unique or different from all other states s_m of group I, then split the group:

IF $(\exists r, \exists s, \forall t : I \mid (a \in \sum) \wedge (s \neq t) \wedge (r \xrightarrow{a} s) \neq (r \xrightarrow{a} t))$
THEN split I

③ **Continue** the process of ***refining the partition of states into sets*** **until** ***all sets contain only one element*** (which is the same as the original DFA) or **until** ***no further splitting of sets occur***.

✎ *Example #1*:

Let's take the DFA obtained from the transformation process of RE ***letter***(***letter***|***digit***)* used to recognize **identifiers**, and obtain the the equivalent minimum-state DFA:

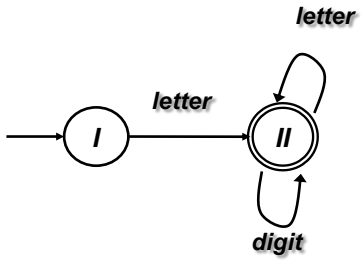


① First, split the original transition table into two sets, one for accepting states and the other for non-accepting states:

NFA states	DFA states	letter	digit	MS-DFA	letter	digit
{0}	A	B	-	I	II	-
{1,2,3,4,6,9}	B	C	D	II	II	II
{3,4,5,6,8,9}	C	C	D	II		
{3,4,6,7,8,9}	D	C	D	II		

② Form the first step, we can clearly observe that no further splitting can be performed. Therefore, the minimum-state DFA is obtained.

NFA states	DFA states	letter	digit	MS-DFA	letter	digit
{0}	A	B	-	I	II	-
{1,2,3,4,6,9}	B	C	D	II	II	II
{3,4,5,6,8,9}	C	C	D	II		
{3,4,6,7,8,9}	D	C	D	II		



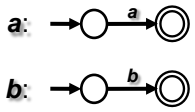
111

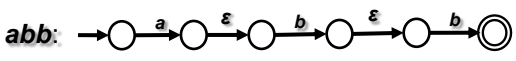
✎ *Example #2*:

Let's have a complete exercise. Obtain the minimum-state DFA for the RE ***a|abb |a*b+***

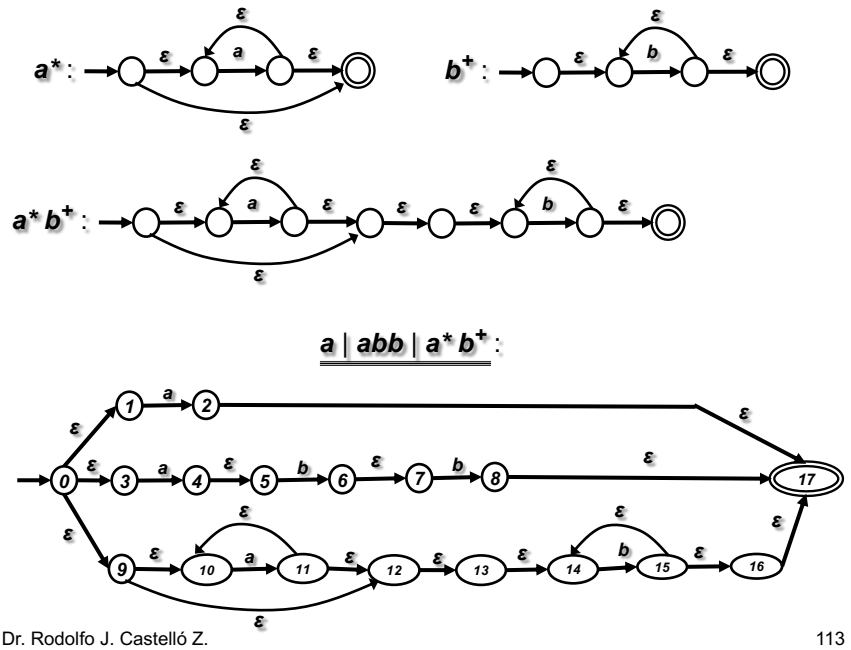
✎ **Solution**:

① First, we obtained the NFA for the given RE:





112



② Second, we transform the NFA to a DFA:

A = *ε-closure* (***0***) = {0,1,3,9,10,12,13,14}

ε-closure (*move* (***A***, *a*)) = {2,4,5,10,11,12,13,14,17} = ***B***
ε-closure (*move* (***A***, *b*)) = {14,15,16,17} = ***C***
ε-closure (*move* (***B***, *a*)) = {10,11,12,13,14} = ***D***
ε-closure (*move* (***B***, *b*)) = {6,7,14,15,16,17} = ***E***
ε-closure (*move* (***C***, *a*)) = {}
ε-closure (*move* (***C***, *b*)) = {14,15,16,17} = ***C***

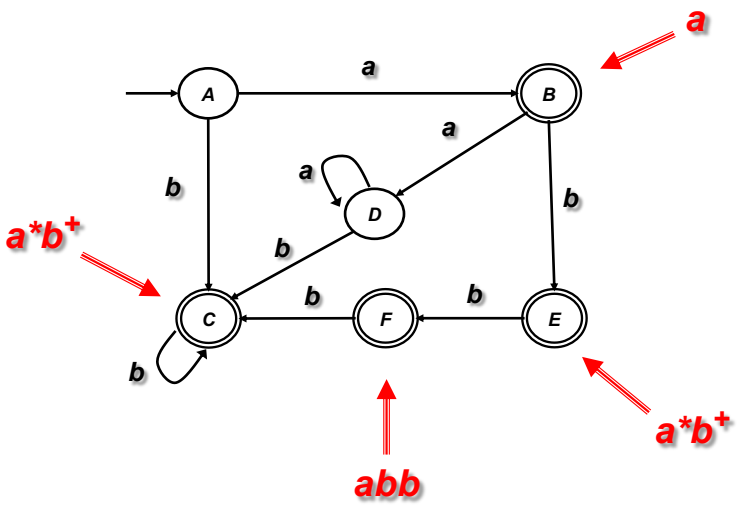
NFA states	DFA states	a	b
{0,1,3,9,10,12,13,14}	A	B	C
{2,4,5,10,11,12,13,14,17}	B	D	E
{14,15,16,17}	C	-	C
{10,11,12,13,14}	D		
{6,7,14,15,16,17}	E		

ε-closure (*move* (***D***, *a*)) = {10,11,12,13,14} = ***D***
ε-closure (*move* (***D***, *b*)) = {14,15,16,17} = ***C***
ε-closure (*move* (***E***, *a*)) = {}
ε-closure (*move* (***E***, *b*)) = {8,14,15,16,17} = ***F***
ε-closure (*move* (***F***, *a*)) = {}
ε-closure (*move* (***F***, *b*)) = {14,15,16,17} = ***C***

NFA states	DFA states	a	b
{0,1,3,9,10,12,13,14}	A	B	C
{2,4,5,10,11,12,13,14,17}	B	D	E
{14,15,16,17}	C	-	C
{10,11,12,13,14}	D	D	C
{6,7,14,15,16,17}	E	-	F
{8,14,15,16,17]	F	-	C

115

Therefore, the corresponding DFA for RE ***a|abb |a*b+*** is:



116

③ Finally, we obtain the minimum-state DFA:

NFA states	DFA states	a	b	MS-DFA
{0,1,3,9,10,12,13,14}	A	B	C	
{10,11,12,13,14}	D	D	C	
{2,4,5,10,11,12,13,14,17}	B	D	E	
{14,15,16,17}	C	-	C	
{6,7,14,15,16,17}	E	-	F	
{8,14,15,16,17]	F	-	C	

Non-Accepting States

Accepting States

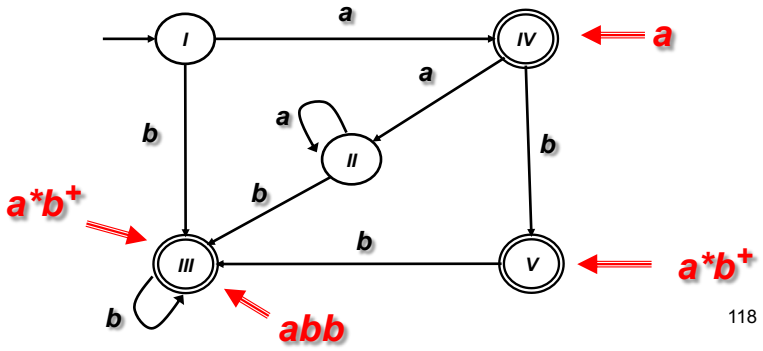
NFA states	DFA states	a	b	MS-DFA
{0,1,3,9,10,12,13,14}	A	B	C	I
{10,11,12,13,14}	D	D	C	II
{2,4,5,10,11,12,13,14,17}	B	D	E	
{14,15,16,17}	C	-	C	
{6,7,14,15,16,17}	E	-	F	
{8,14,15,16,17]	F	-	C	

Common Transitions

Dr. Rodolfo J. Castelló Z. 117

117

NFA states	DFA states	a	b	MS-DFA	a	b
{0,1,3,9,10,12,13,14}	A	B	C	I	IV	III
{10,11,12,13,14}	D	D	C	II	II	III
{14,15,16,17}	C	-	C	III	-	III
{8,14,15,16,17]	F	-	C	III		
{2,4,5,10,11,12,13,14,17}	B	D	E	IV	II	V
{6,7,14,15,16,17}	E	-	F	V	-	III



118

118