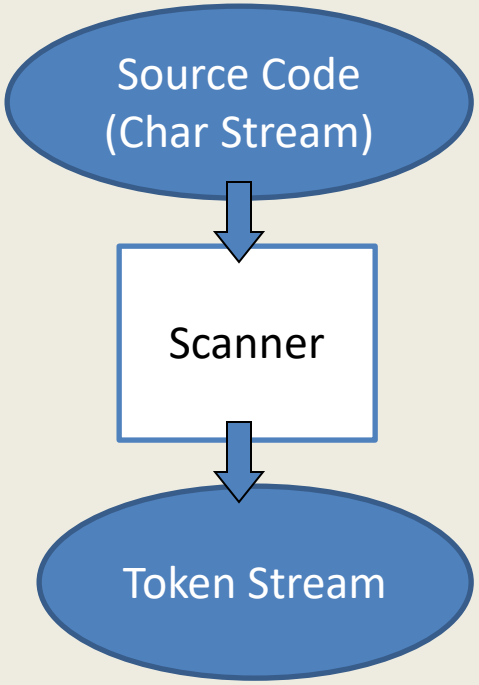
Lexical Analyzer (Scanner)

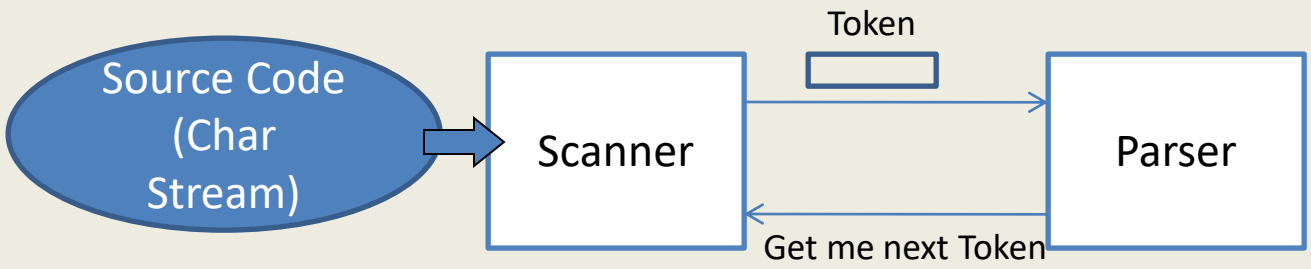
Dinesh Gopalani
dgopalani.cse@mnit.ac.in

Lexical Analyzer (Scanner)

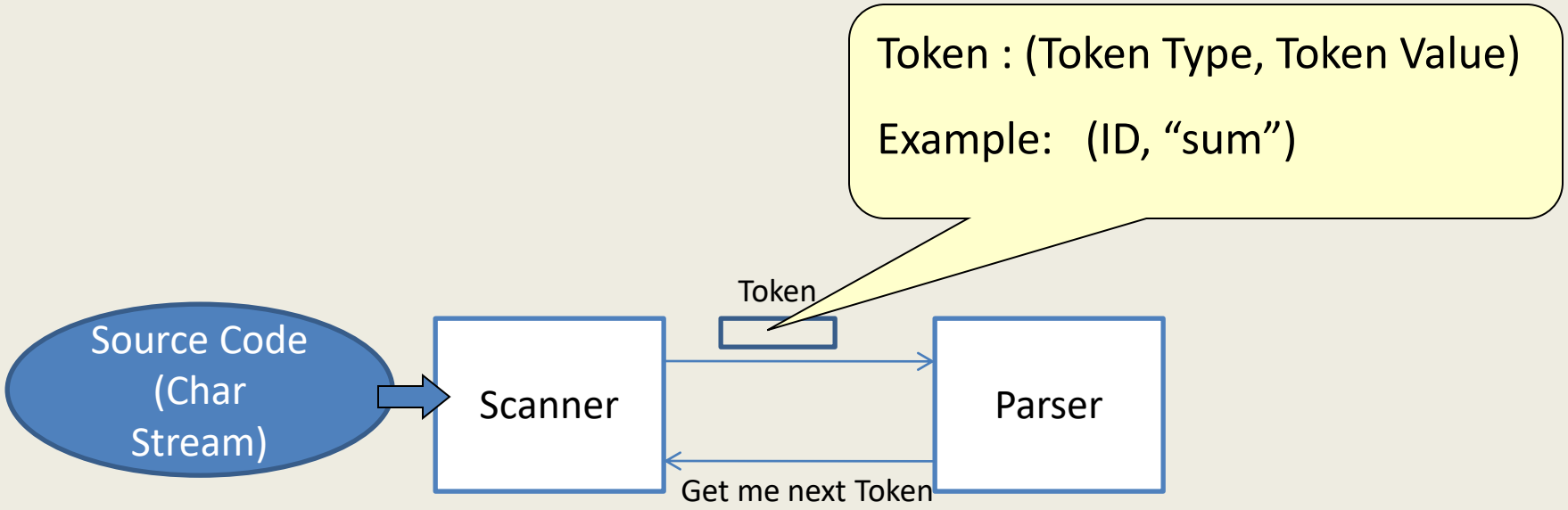
Forms the group of characters that logically belong together - Tokens



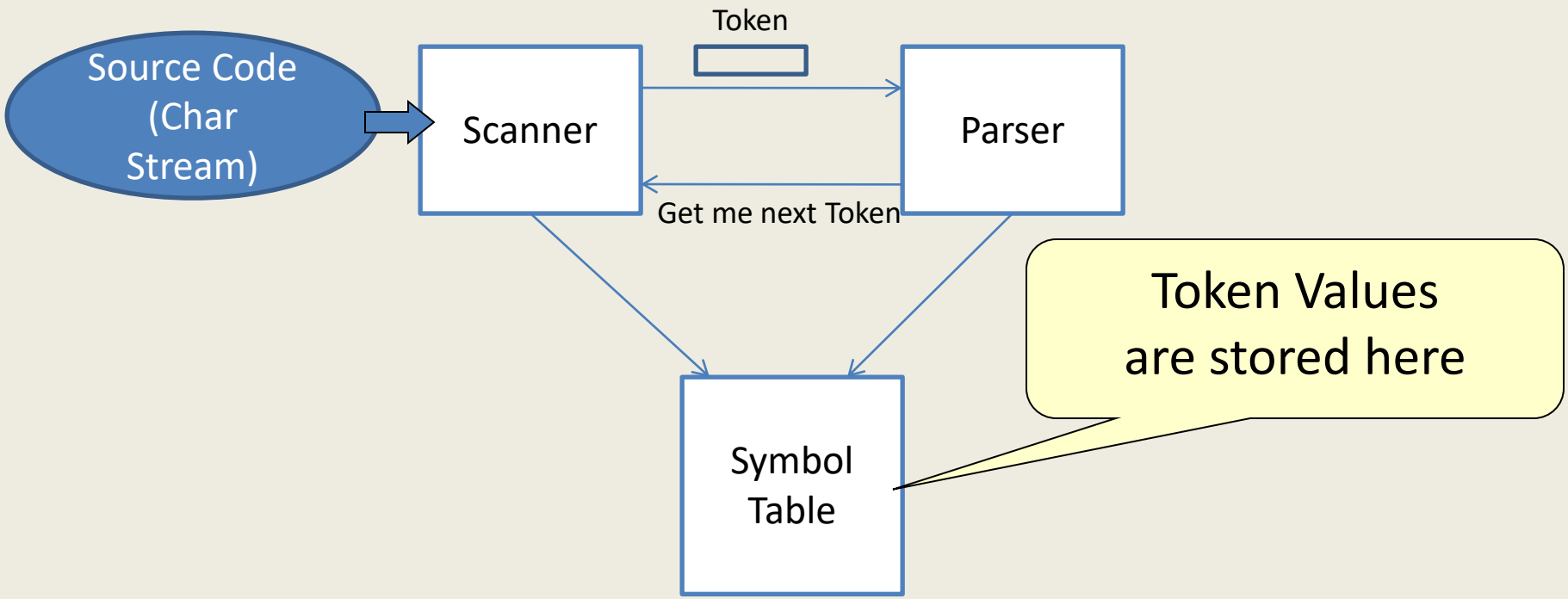
Lexical Analyzer (Scanner)



Lexical Analyzer (Scanner)



Lexical Analyzer (Scanner)



How to develop a Scanner?

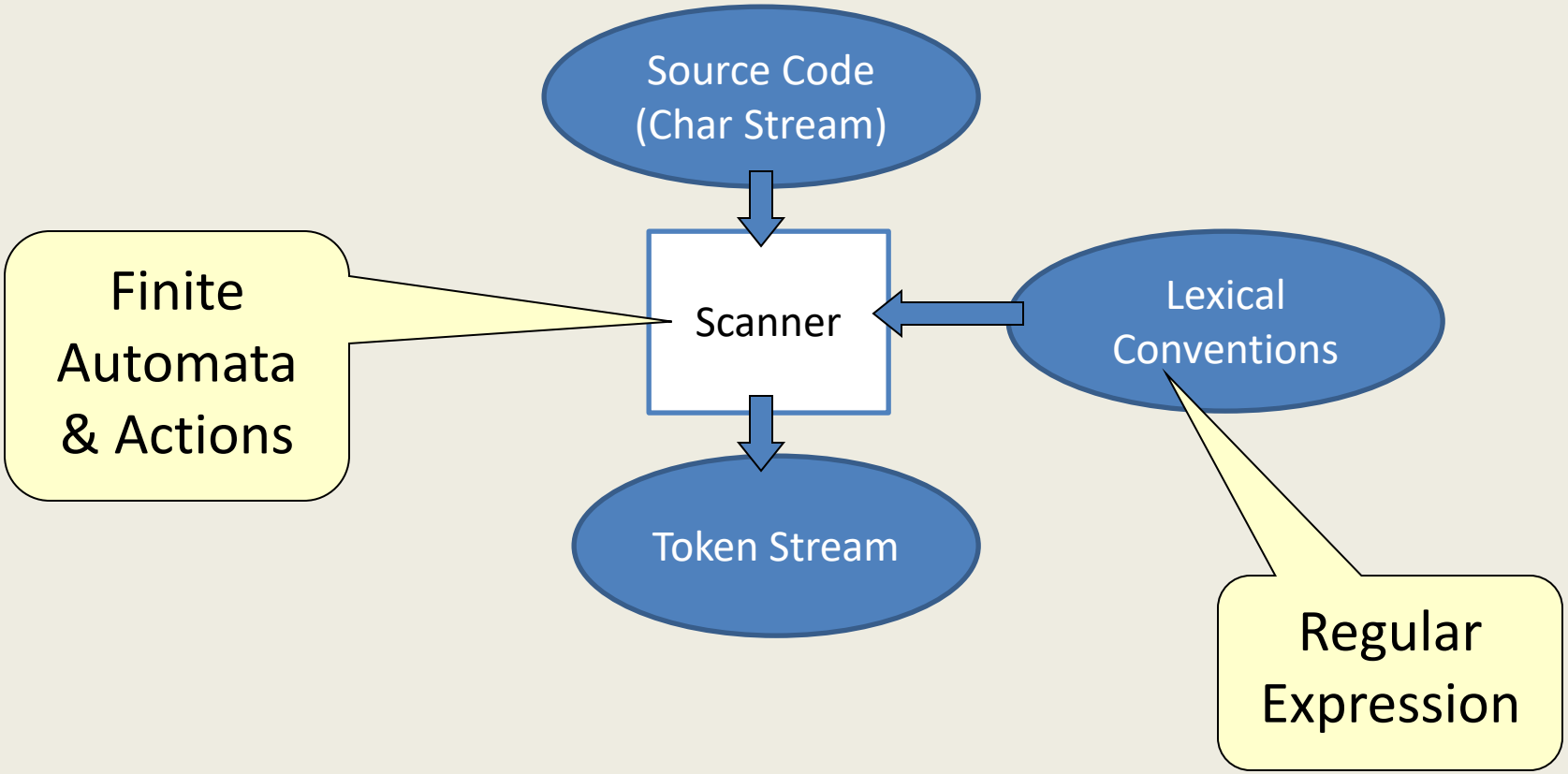
1. Describe the possible tokens that may appear in the source code – Notation used here is Regular Expression
2. Having decided what token are, next some mechanism to identify tokens from the code – Finite Automata may be used
3. Actions to be performed as tokens are recognized

   Examples:

   - To add token value into the symbol table
   - Produce an error message

   …

How to develop a Scanner?



Regular Expression (RE)

Regular expression over the alphabet Σ are constructed with the following rules :

1. ^ is a RE denoting language {^}
2. For each symbol a in Σ, a is a RE denoting language {a}
3. If R and S are REs denoting languages L_R and L_S respectively, then
   a) R | S is a RE denoting $L_R \cup L_S$
   b) R.S or RS is a RE denoting $L_R . L_S$
   c) R* is a RE denoting L_R *

Regular Expression (RE)

Regular expression over the alphabet Σ are constructed with the following rules :

1. ^ is a RE denoting language {^}
2. For each symbol a in Σ, a is a RE denoting language {a}
3. If R and S are REs denoting languages L_R and L_S respectively, then
   a) R | S is a RE denoting $L_R \cup L_S$
   b) R.S or RS is a RE denoting $L_R \cdot L_S$
   c) R* is a RE denoting $L_R{}^*$

$L_R \cup L_S = \{ x \mid x \text{ is in } L_R \text{ or } x \text{ is in } L_S \}$

$L_R \cdot L_S = \{ xy \mid x \text{ is in } L_R \text{ and } y \text{ is in } L_S \}$

$$L_R{}^* = \bigcup_{i=0}^{\infty} L_R^{\,i}$$

Regular Expression (RE)

Regular expression over the alphabet Σ are constructed with the following rules :

1. ^ is a RE denoting language {^}
2. For each symbol a in Σ, a is a RE denoting language {a}
3. If R and S are REs denoting languages L_R and L_S respectively, then
   a) R | S is a RE denoting $L_R \cup L_S$
   b) R.S or RS is a RE denoting $L_R . L_S$
   c) R* is a RE denoting L_R *
   d) (R) is a RE denoting L_R

Regular Expression (RE)

Regular expression over the alphabet Σ are constructed with the following rules :

1. ^ is a RE denoting language {^}
2. For each symbol a in Σ, a is a RE denoting language {a}
3. If R and S are REs denoting languages L_R and L_S respectively, then
   a) R | S is a RE denoting $L_R \cup L_S$
   b) R.S or RS is a RE denoting $L_R \,.\, L_S$
   c) R* is a RE denoting L_R *
   d) (R) is a RE denoting L_R

Precedence Order

*

.

|

Regular Expression (RE)

Examples:

Keyword (while):

while

or

(w|W)(h|H)(i|I)(l|L)(e|E)

Identifier Name (A letter followed by letters and/or digits) :

*(a|b|…|z|A|B|…|Z) (a|b|…|z|A|B|…|Z|0|1|…|9)**

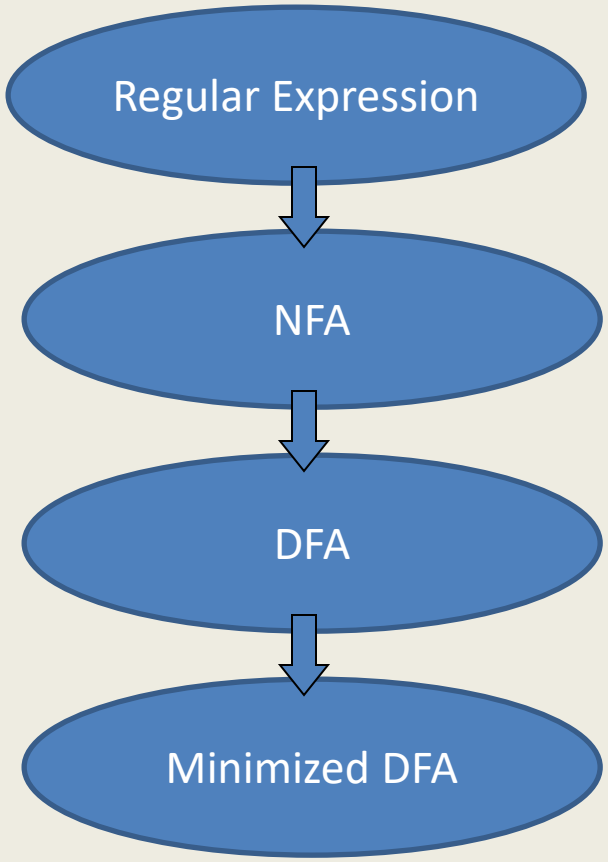
or

letter = a|b|…|z|A|B|…|Z

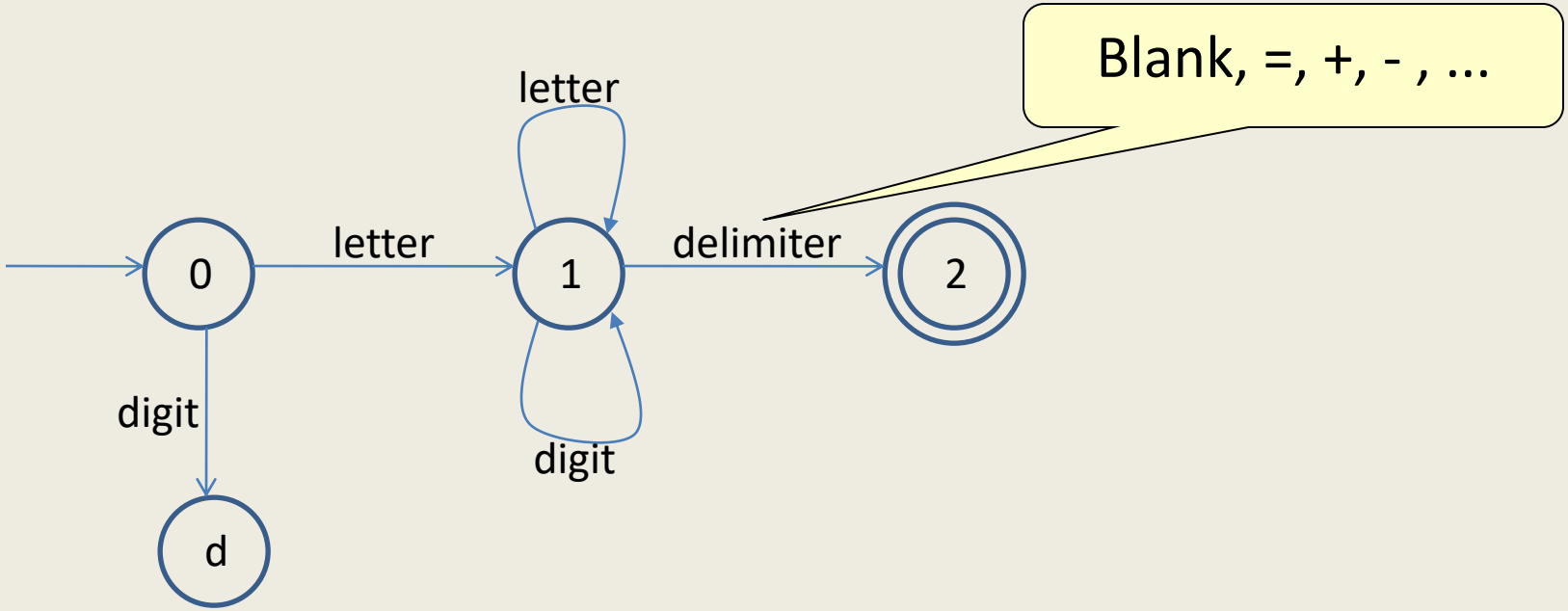
digit = 0|1|…|9

*Identifier = (letter)(letter|digit)**

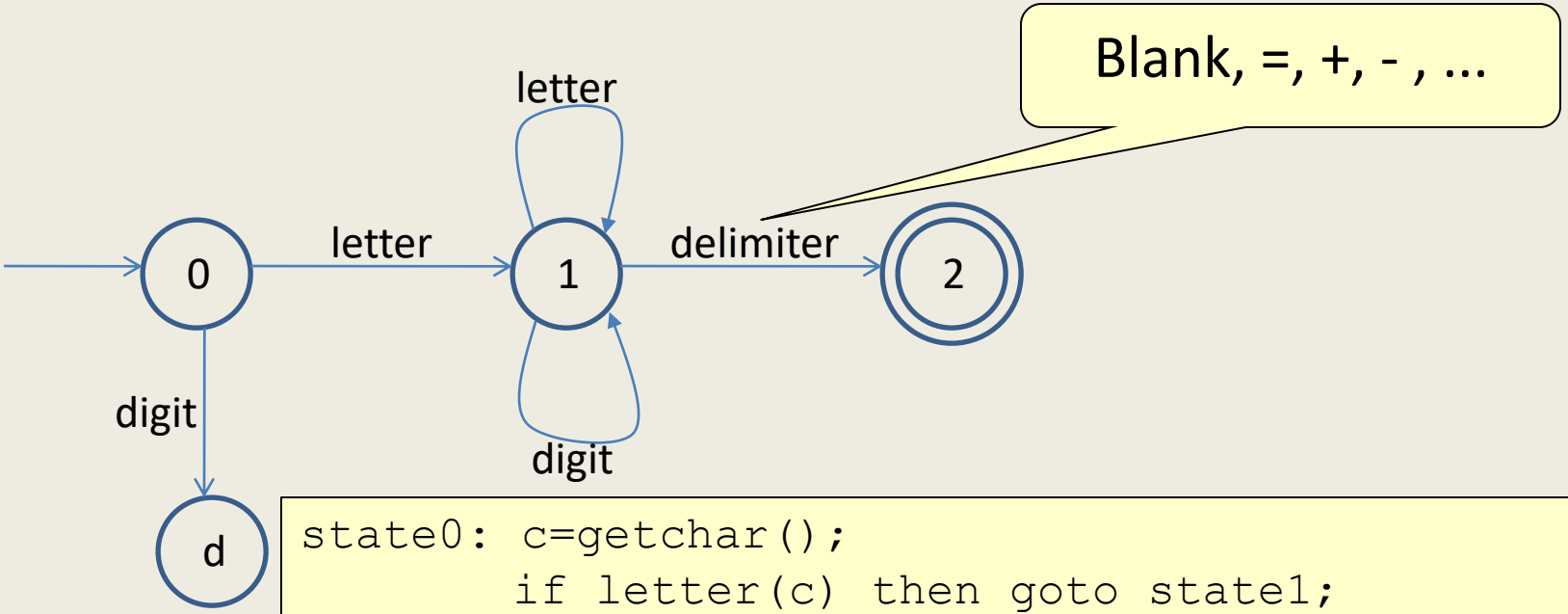
Regular Expression into Finite Automata Conversion



Finite Automata into Code Conversion



Finite Automata into Code Conversion



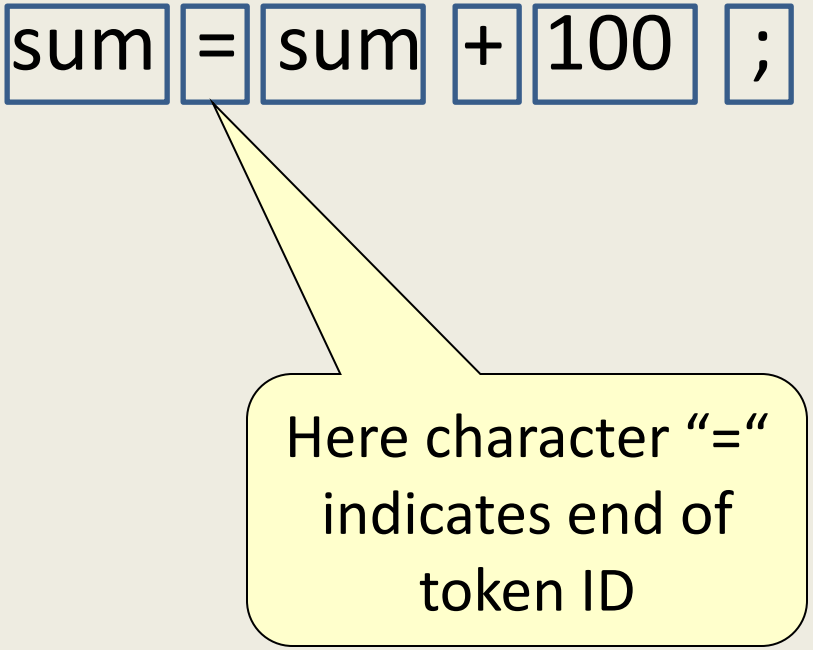

```
state0: c=getchar();
        if letter(c) then goto state1;
          else fail();

state1: c=getchar();
        if letter(c) or digit(c)then goto state1;
          else if delimiter(c) then goto state2;
          else fail();

state2: p=install();
        return (ID, p);
```

Lookahead Technique

In most of the cases token is recognized by reaching at the end of the token :



Lookahead Technique

- Sometimes some more characters need to be scanned in order to recognize the token
- This may happen due to certain language conventions
- Example is “Blanks are not significant”
- DO Loop of Fortran:

```
   ...
   DO 55 I = 1 , 100
      ...
      ...
      ...
55 CONTINUE
   ...
```

Lookahead Technique

- Sometimes some more characters need to be scanned in order to recognize the token
- This may happen due to certain lan s
- Example is “Blanks are not significa
- DO Loop of Fortran:

After removing blank spaces

```
   ...
   DO 55 I = 1 , 100
      ...
      ...
      ...
55 CONTINUE
   ...
```

```
   ...
   DO55I=1,100
      ...
      ...
      ...
55 CONTINUE
   ...
```

Lookahead Technique

Compare two Code fragments of Fortran

```
   ...
   DO55I=1,100
      ...
      ...
      ...
55 CONTINUE
   ...
```

```
   ...
   DO55I=1.100
   ...
   ...
   ...
```

Lookahead Technique

Compare two Code fragments of Fortran

```
   ...
   DO55I=1,100
      ...
      ...
      ...
55 CONTINUE
   ...
```

```
   ...
   DO55I=1.100
   ...
   ...
   ...
```

DO is a token keyword but confirmed only when some more characters are scanned

Lookahead Technique

Compare two Code fragments of Fortran

```
   ...
   DO55I=1,100
      ...
      ...
      ...
55 CONTINUE
   ...
```

```
   ...
   DO55I=1.100
   ...
   ...
   ...
```

After seeing this DO keyword is confirmed

Lookahead Technique

Compare two Code fragments of Fortran

```
   ...
   DO55I=1,100
      ...
      ...
      ...
55 CONTINUE
   ...
```

```
   ..
   DO55I=1.100
   ...
   .
   ..
```

Here "DO" is only part of a bigger token ID "DO55I"

Lookahead Technique

Another Example:

"Keywords are not reserved"

```
...
IF(I .LT. 100)
   ...
   ...
   ...
   ...
```

```
...
IF(I,J)=100
...
...
...
```

Lookahead Technique

Another Example:

"Keywords are not reserved"

```
...
IF(I .LT. 100)
  ...
  ...
  ...
  ..
```

Here "IF" is a Keyword

```
...
IF(I,J)=100
...
.
..
```

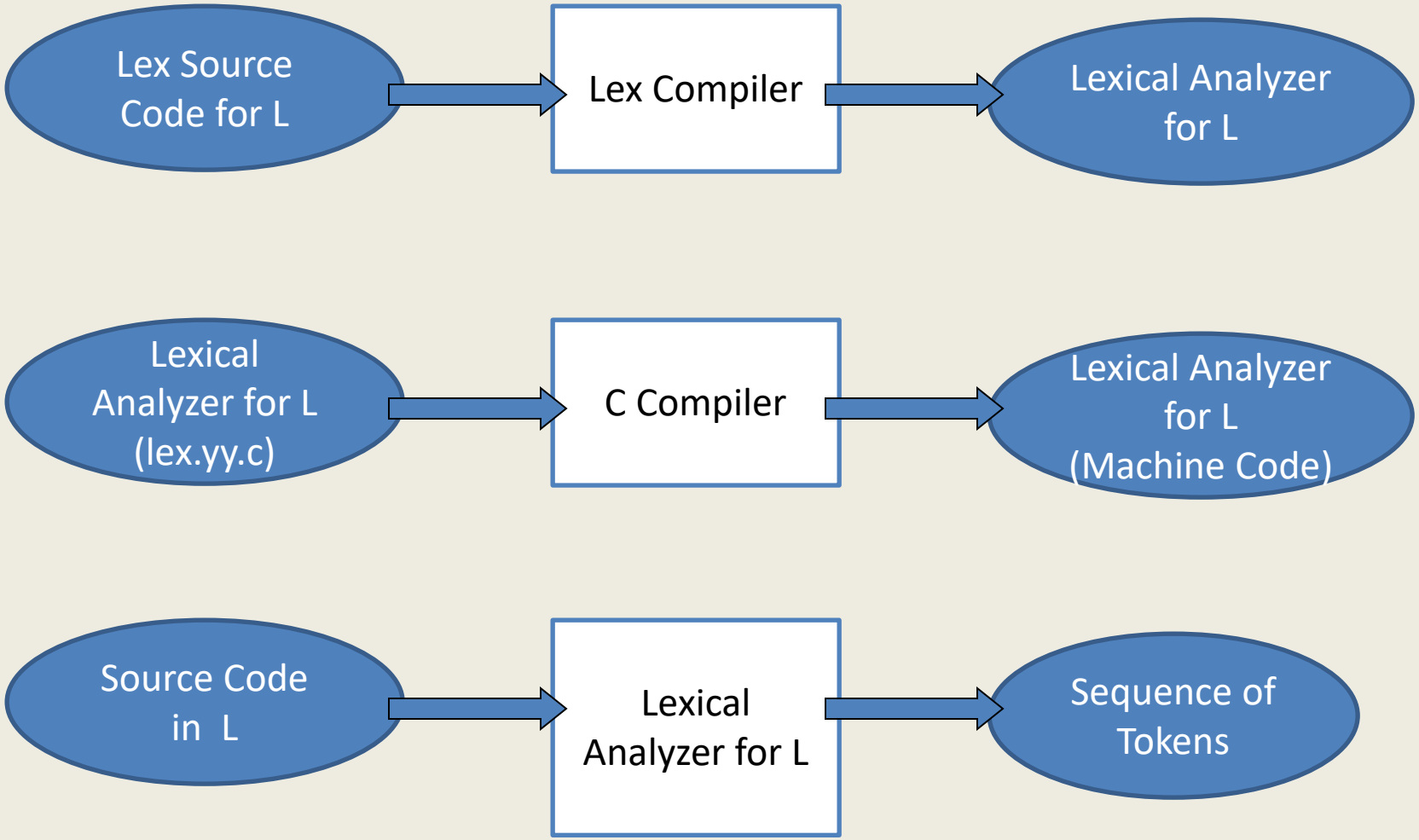
Here "IF" is the Array name

The LEX Tool

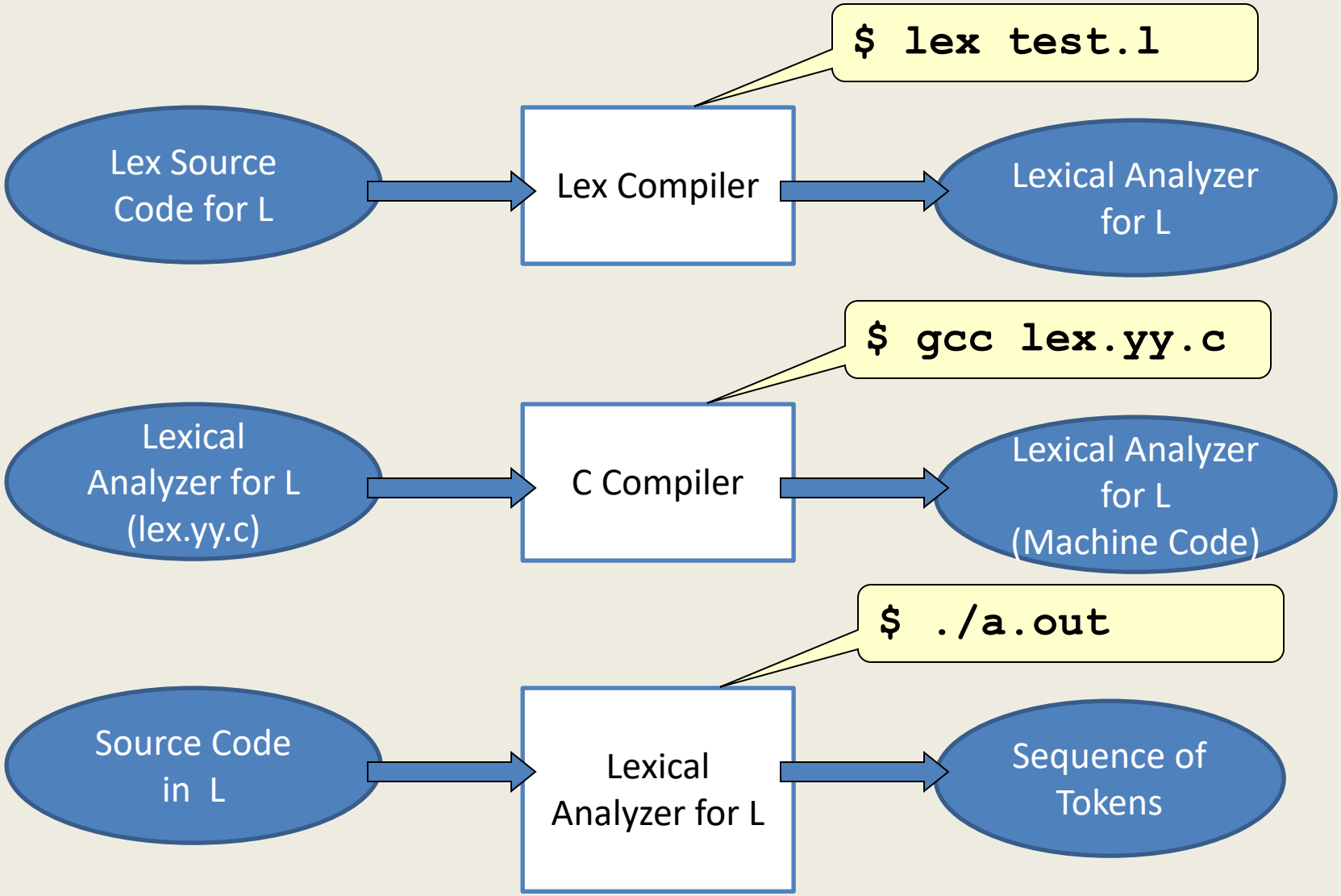
- The Lex is a software tool that automatically constructs a Lexical Analyzer
- Need to write a program in the Lex language –

  Lex Source Code
- A Lex source code is a specification of a Lexical Analyzer consists of
  - Set of Regular Expressions
  - Set of Actions for each RE

The LEX Tool - Working



The LEX Tool - Working



LEX Source Code

A Lex source code consists of three parts

Auxiliary Definitions

%%

Translation Rules

%%

Auxiliary Functions

LEX Source Code

The Auxiliary Definitions are statements of the form

D_1 R_1

D_2 R_2

…

…

D_n R_n

LEX Source Code

The Auxiliary Definitions are statements of the form

$D_1 \quad R_1$

$D_2 \quad R_2$

...

...

$D_n \quad R_n$

Each D_i is a distinct name

Each R_i is a RE over $\Sigma \cup \{D_1, D_2, \ldots, D_{i-1}\}$

LEX Source Code

The Translation Rules are statements of the form

$P_1 \quad \{A_1\}$

$P_2 \quad \{A_2\}$

…

…

$P_m \quad \{A_m\}$

LEX Source Code

The Translation Rules are statements of the form

P_1 $\{A_1\}$

P_2 $\{A_2\}$

…

…

P_m $\{A_m\}$

Each P_i is a RE over $\Sigma \cup \{D_1, D_2, \ldots, D_n\}$

Each A_i is a code fragment – Action when token is found matching pattern P_i

LEX Source Code

The Auxiliary Functions are regular C Functions

```
int f1(...)
{
}

void f2(...)
{
}
...
...
```

Regular Expression for LEX

```
R|S        Union
RS         Concatenation
R*         Closure
R+         One or more R's
R?         Zero or one R
R+         One or more R's
R{n}       n number of R's
R{m,n}     Min m and Max n R's
{name}     Definition name
R/S        Matches with R but only
           when followed by S
.          Matches any one char except "\n"
[...]      Matches any one char within [...]
"..."      Matches everything literally
```

Regular Expression for LEX

R|S	Union
RS	Concatenation
R*	Closure
R+	One or more R's
R?	Zero or one R
R+	One or more R's
R{n}	n number of R's
R{m,n}	Min m
{name}	Definition
R/S	Matches R when followed by S
.	Matches any one char except "\n"
[...]	Matches any one char within [...]
"..."	Matches everything literally

For range - is used
Ex. [A-Z] or [a-Z0-9]

Metacharacters lose their meaning
Ex. "+", "*", "{ }"

Lookahead Operator in LEX

R / S

Matches a string in **R** but only when followed by a string in **S**.

The RE **S** after the lookahead operator (**/**) indicates the right context for a match, it is used only to restrict a match and not part of the current token.

Lookahead Operator in LEX

R / S

Matches a string in **R** but only when followed by a string in **S.**

Example(DO Keyword of Fortran):

DO / {digit}+ {identifier} = {digit}+ “,”

```
   ...
   DO55I=1,100
      ...
      ...
      ...
55 CONTINUE
   ...
```

Sample LEX Source Code

Token	Type	Value
if	101	-
else	102	-
while	103	-
for	104	-
Identifier	201	Pointer to symbol Table
Integer constant	202	Pointer to symbol Table
+	301	10
-	301	20
*	301	30
/	301	40

Sample LEX Source Code

```
%{
   …
   …
%}

letter      [A-Za-z]
digit       [0-9]
%%
if | IF                          { return 101; }
else | ELSE                      { return 102; }
while | WHILE                    { return 103; }
for | FOR                        { return 104; }
{letter}({letter}|{digit})*      { yylval=install_id();   return 201; }
{digit}+                         { yylval=install_int();  return 202; }
```

All C declarations written here (Global variables, function prototypes, ...)

Sample LEX Source Code

```
"+"                    { yylval=10;   return 301; }
"-"                    { yylval=20;   return 301; }
"*"                    { yylval=30;   return 301; }
"/"                    { yylval=40;   return 301; }
%%
int install_id()
{           ...
            ...
            return p;
}

int install_int()
{           ...
            ...
            return p;
}
```

Sample LEX Source Code

```
"+"                    { yylval=10;   return 301; }
 "-"                   { yylval=20;   return 301; }
 "*"                   { yylval=30;   return 301; }
 "/"                   { yylval=40;   return 301; }
 %%
int install_id()
{           ...
            ...
            return p;
}

int install_int()
{           ...
            ...
            return p;
}
void main()
{
            yylex();
}
```

Sample LEX Source Code

```
"+"                    { yylval=10;   return 301; }
"-"                    { yylval=20;   return 301; }
"*"                    { yylval=30;   return 301; }
"/"                    { yylval=40;   return 301; }
%%
int install_id()
{           ...
            ...
            return p;
}

int install_int()
{           ...
            ...
            return p;
}
void main()
{
            yylex();
}
```

Required when used independently (without yacc)

Conflict Resolution in LEX

The following two rules are used:

Rule 1 :

"Always prefer a longer prefix to a shorter prefix"

If it finds a string matching with more than one patterns, it takes the one matching the longer text.

Example:

```
...
if (a <= b)
...
...
ifa=10;
...
```

Conflict Resolution in LEX

The following two rules are used:

Rule 1 :

"Always prefer a longer prefix to a shorter prefix"

If it finds a string matching with more than one patterns, it takes the one matching the longer text.

Example:

```
...
if (a <= b)
...
...
ifa=10;
...
```

Here <= treated as one token rather than < and = as two separate tokens

Conflict Resolution in LEX

The following two rules are used:

Rule 1 :

"Always prefer a longer prefix to a shorter prefix"

If it finds a string matching with more than one patterns, it takes the one matching the longer text.

Example:

```
...
if (a <= b)
...
...
ifa=10;
...
```

Here ifa is an identifier name

Conflict Resolution in LEX

The following two rules are used:

Rule 2 :

"Always prefer a match whose pattern comes earlier of the same length"

If the longest possible prefix matches two more patterns, the pattern listed earlier in the Lex code is preferred.

Example:

Keyword vs. Identifier