

Lexical Analysis

1. Motivation

Consider an input C program. This is the way an editor will show it.

```
main ()
{
    int i,sum;
    sum = 0;
    for (i=1; i<=10; i++);
    sum = sum + i;
    printf("%d\n",sum);
}
```

The same program as the compiler sees it initially, as a continuous stream of characters.

```
main_()↵{↵_int_i,sum;↵_sum=_0;↵_
for_(i=1;_i<=10;i++);_sum=_sum+_i;↵_
printf("%d\n",sum);↵}
```

Where the symbol _ indicates a blank space and ↵ indicates a newline character.

2. Partitioning the Input

How do we make the compiler see the way that we interpret it, not as a sequence of characters but as a collection of meaningful entities?

Discover the structure in the input.

Step 1:

a. Break up this string into ‘words’–the smallest logical units.

```
main ( ) { int i , sum ; sum = 0 ; for ( i = 1 ; i <= 10 ; i ++ ) ; sum = sum + i ; printf ( "%d\\n" , sum ) ; }
```

Each shaded rectangular box denotes a lexeme of the program. The result is a sequence of lexemes (or tokens).

b. Clean up – remove the blank space `_` and the newline character `↵`.

The resulting sequence after this operation is shown below.

```
main ( ) { int i , sum ; sum = 0 ; for (
i = 1 ; i <= 10 ; i ++ ) ; sum = sum + i
; printf ( "%d\n" , sum ) ; }
```

This is **lexical analysis** or **scanning**.

Important Terms : Lexemes, Tokens and Patterns

Definition: Lexical analysis is the operation of dividing the input program into a sequence of lexemes (tokens).

Let us learn to distinguish between

- **lexemes** – smallest logical units (words) of a program.

Examples – i, sum, for, 10, ++, "%d\n", <=.

- **tokens** – sets of similar lexemes.

Examples –

identifier = {i, sum, buffer, . . . }

int constant = {1, 10, . . . }

addop = {+, -}

Things that are not counted as lexemes

- white spaces – tab, blanks and newlines
- comments

However even these too have to be detected and stripped off the input and ignored.

Q. How does one describe the lexemes that make up the token identifier?

Even for an identifier, there are variants in different languages. We need to know the description of an identifier in the Programming Language (PL) of interest. C is the language for our example program.

Such descriptions are called **patterns**. The description may be informal or formal. An informal description follows.

- a string of alphanumeric characters. The first character is an alphabet.
- a string of alphanumeric characters in which the the first character is an alphabet. There is an additional constraint that the length of an identifier can be at most 31.
- a string of alphanumeric or underline (`_`) characters in which the the first character is an alphabet or an underline. The length of an identifier is restricted cannot exceed 31. However, an identifier with > 31 characters, the first 31st character are retained and the remaining are ignored.

Basic concepts and issues

A pattern in the context of lexical analysis is used to

- specify a token precisely, and
- build a recognizer from such specifications

Example – tokens in Java

1. Identifier: A Javaletter followed by zero or more Javaletterordigits.

A Javaletter includes the characters a-z, A-Z, `_` and `\$`.

2. Constants: The following are all valid constants.

2.1 Integer Constants

- Octal, Hex and Decimal
- 4 byte and 8 byte representation

2.2 Floating point constants

- float - ends with f
- double

2.3 Boolean constants – true and false

2.4 Character constants – 'a', '\u0034', '\t'

2.5 String constants – "", "\", "A string"

2.6 Null constant – null

3. Delimiters: () { } [] ; . and ,

4. Operators: =, >, < . . . >>, >=

5. Keywords: abstract, boolean . . . volatile, while

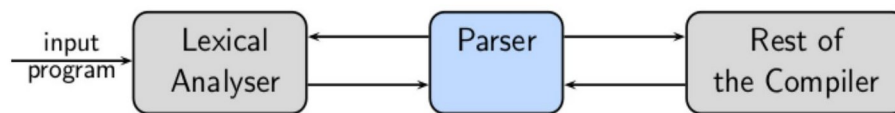
Exercise : Read up the specification of an identifier in language C and write this informally on the same lines as given above.

Context Of A Lexical Analyser

Where does a lexical analyser fit into the rest of the compiler?

- The lexical analyser is used primarily by the parser (part of a compiler that performs syntax analysis). When the parser needs the next token, it calls the lexical analyser.
- Instead of analysing the entire input string, the lexical analyser sees enough of the input string to return a single token to the parser.
- Thus lexical analysis and parsing, which are distinct phases because they perform distinct are not different passes of a compiler, typically belong to the same pass.

EXAMPLE



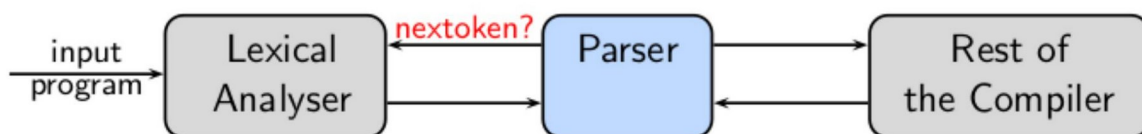
The Lexical Analyzer (or scanner) and the Parser (or Syntax Analyzer) work in tandem as shown below. Consider the initial part of the input program :

```
main ()
```

```
{
```

```
    int i,sum;
```

- Parser asks the lexical analyzer to supply a token.

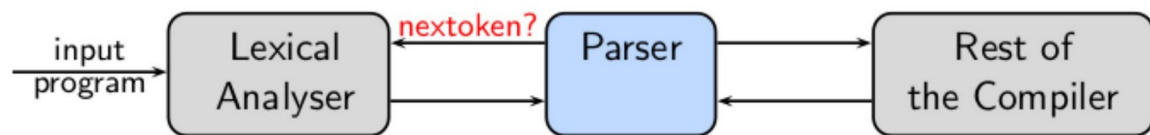
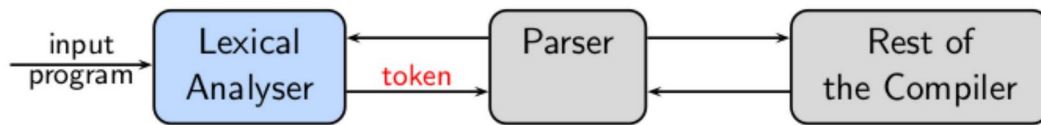


- The scanner identifies a token and returns the same to the parser. In this case, scanner returns : <identifier, "main">. The remaining input is now :

```
)
```

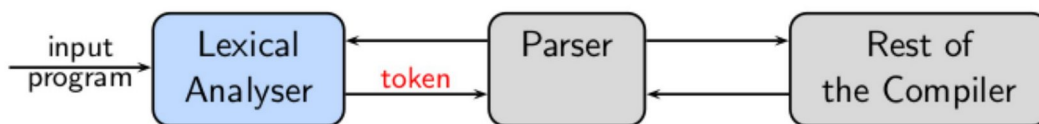
```
{
```

```
    int i,sum;
```

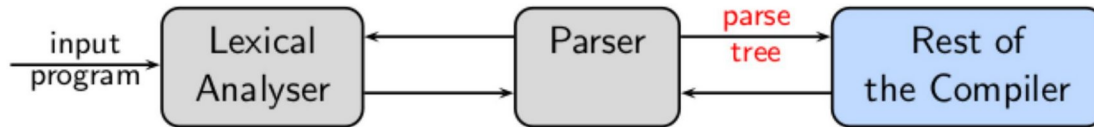


- After consuming “main”, the parser again asks the scanner for the next token, as seen earlier.
- The scanner returns the token “(“ this time with the remaining input as :

)
{
 int i,sum;



Such communication between the parser and continues till the entire program has been processed. At this point the parser generates a parse tree as its output and communicates the tree with the rest of the compiler.



Note that the above description is a simple model to illustrate the nature of interaction between the parser and the scanner. In a real compiler, the interaction between the parser and the rest of the compiler is more involved as we shall see later in this course.

Token Attributes

Apart from the token itself, the lexical analyser also passes other information regarding the token. These items of information are called token attributes.

lexeme	Token type	token attribute
3	const	3
x1	identifier	x1
if	keyword	(index in keyword list)
>=	relop	>= (or index in list of relops)
;	delimiter	;

Creating a Lexical Analyzer

Two approaches:

1. **Hand code the lexical analyser** – that is write it manually This is of historical interest now. A human may be able to write more efficient code but correctness always remains a concern.

2. : generate the lexical analyser from a formal des **Use a lexical analyser generator** cription of the tokens of the language and their attributes.

- The generation process is faster to yield a working scanner.
- Less prone to errors.

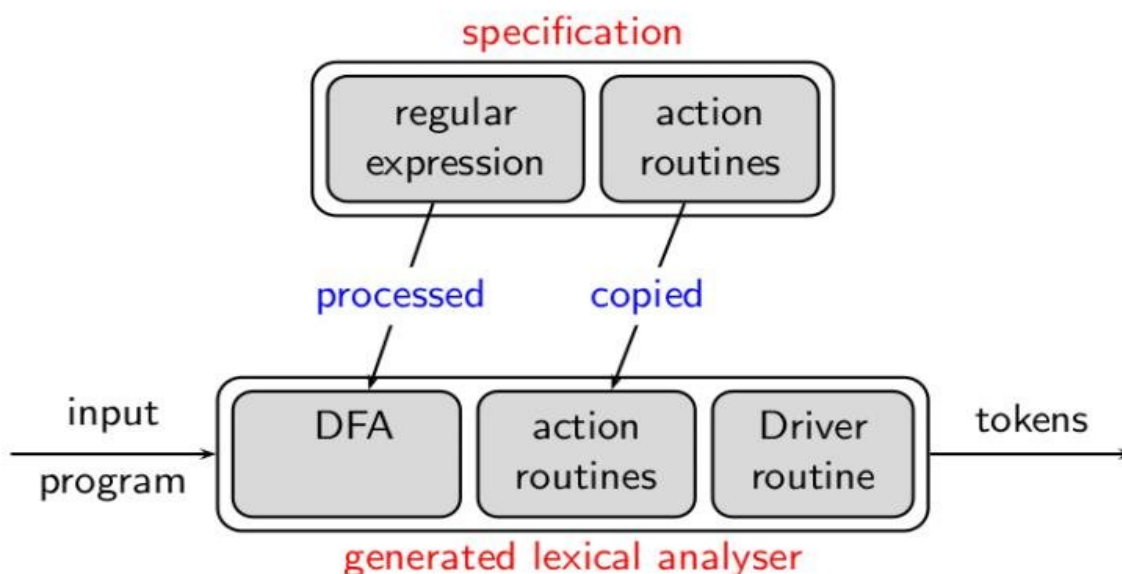
Automatic Generation of Lexical Analysers

Inputs to the lexical analyser generator:

- A specification of the tokens of the source language, which involves :
 - a regular expression describing each token, and
 - a code fragment describing the action to be performed, on identifying each token.

The generated lexical analyser consists of:

- A deterministic finite automaton (DFA) constructed from the token specification.
- A code fragment (a driver routine) which can traverse any DFA.
- Code for the action specifications when a token is recognized.



- The first row indicates the effort at the user level. This involves writing a token in terms of regular expression and state the list of actions to be executed when that token is detected in the input.
- The scanner generation algorithm processes the regular expressions and constructs a DFA first and then minimizes the DFA.
- It copies the user defined actions and converts them into functions that can be called when required.
- A generic driver routine is included, whose purpose is to read a token character by character and use the DFA for recognition. When a final state is encountered, the function corresponding specified action is executed by the driver.
- The components in the bottom row constitutes the lexical analyzer that is automatically generated. It is a one time effort.

The generated lexical analyzer (or scanner) scans a stream of input, recognizes a token and communicates the token to the parser as can be seen in the last row of the above figure.

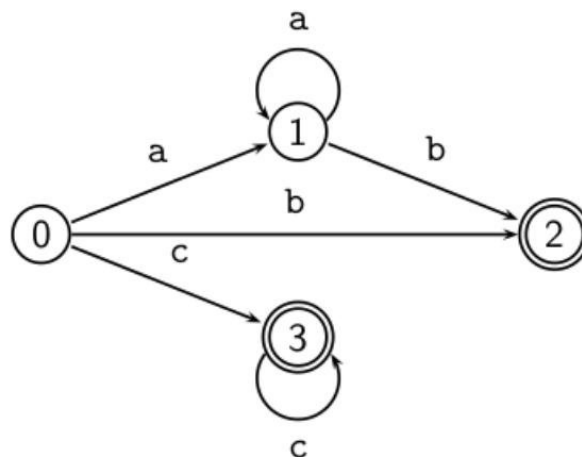
The entire process is illustrated through a working example.

EXAMPLE : Consider a simple language with two tokens along with the associated actions.

Pattern	Action
a^*b	{ printf("Token 1 Recognized \n"); }
c^+	{ printf("Token 2 Recognized \n"); }

The Alphabet is $\Sigma = \{a, b, c\}$ and the $\{*, +\}$ are the usual regex operators.

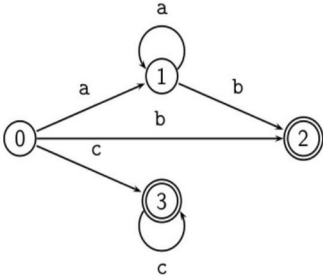
The scanner generation software constructs a dfa from the regex, such as the one shown below. There exists an algorithm for directly constructing a dfa from regular expressions (instead of going round through a NFA). This algorithm is generally not a part of FLAT curriculum and hence we shall discuss it in this course.



The three components that are a part of the generation tool are :

- ▶ a DFA
- ▶ a driver routine
- ▶ action functions

Let us consider these parts together

DFA	Action functions
 <pre> graph LR 0((0)) -- a --> 1((1)) 0 -- b --> 2(((2))) 0 -- c --> 3(((3))) 1 -- a --> 1 1 -- b --> 2 2 -- a --> 2 2 -- b --> 2 3 -- c --> 3 style 0 fill:none,stroke:none style 1 fill:none,stroke:none style 2 fill:none,stroke:none style 3 fill:none,stroke:none </pre>	<p>The actions are converted into a function</p> <pre> function action(state) { case state 2 : { printf("Token 1 Recognized \n"); 3 : { printf("Token 2 Recognized \n"); } } </pre>
Driver routine	Working : input & output
<pre> function nexttoken() { state = 0; c = nextchar(); while (valid(nextstate[state, c])) { state = nextstate[state, c]; c = nextchar(); } if (!final(state)) {error message; return;} else { unput(c); action(state); return;} } </pre>	<p>Input : aabbaccabc</p> <p>The first character in the input is highlighted for clarity.</p>

Data structure and support functions used by the driver routine :

- `nextstate[state s, char c]` is a data structure (a realization of the DFA), which given a state `s` and the character '`c`' in the input, gives the next state.
- `nextchar()` : returns the next character from the input stream
- `final(state s)` : returns true if the argument state `s`, is a final state else returns false
- `unput(c)` : returns the character '`c`' back to input stream. Determine the reason for including this function.
- `action(s)` : executes the action provided by the user on the state `s`; note that only a final state has an associated action corresponding to a token that it has recognized in state `s`.

The driver routine starts with start state 0.

The input string is : `aabbaccabc`

The configuration of the system at the end of each iteration of the while loop is tabulated below.

Observe that each iteration of the while loop starts with a given state `s` and a character `c`.

On entry to the loop body, a change of state is executed and the next character in the input is read.

The 1st column gives the iteration number. The 2nd and 3rd columns denote the current state and the next character in the input before entering the loop body. The 4th and 5th columns give the changes to state and the `nextchar` caused by the statements in the body of the loop. The output of the generated scanner is highlighted in green as also the final states.

Examine the table given below that shows how the generated scanner processes the input : `aabbaccabc`

For a better understanding, show the effect of each row of the table by drawing the dfa and the state reached, the input symbols consumed and the input that remains to be scanned yet.

Iter	state s	nextchar c	nextstate [s,c]	nextchar c	Rest of input	Move / Output
1	0	a	1	a	bbaccabc	Move to state 1
2	1	a	1	b	baccabc	Remain in state 1
3	1	b	2	b	accabc	Move to state 2
4	2	b	–	b	baccabc	unput(b); return Token 1 Recognized
5	0	b	2	a	ccabc	Restart; move to state 2
6	2	a	–	a	accabc	unput(a); return Token 1 Recognized
7	0	a	1	c	cabc	Restart; move to state 1
8	1	c	–	c	ccabc	unput(c); Error; skip a; return
9	0	c	3	c	abc	Restart; move to state 3
10	3	c	3	b	c	Remain in state 3
11	3	b	–	b	bc	unput(b); return Token 2 Recognized
12	0	b	2	c	NIL	Restart; move to state 2
13	2	c	–	c	c	unput(c); return Token 1 Recognized
14	0	c	3	–	NIL	Restart; move to state 3
15	3	NIL	–	–	NIL	Return; Token 2 Recognized scanner terminates

In conclusion we have shown that the DFA, the driver routine and the action routines, all taken together constitute the lexical analyser. Further, given specifications of patterns and actions by the user

- patterns are used to automatically construct a final DFA
- actions that are supplied as part of specification are converted into callable functions
- a generic driver routine is written and included as a one time effort that is common to all generated lexical analyzers

LEXICAL ERRORS

Lexical errors are essentially of two kinds.

1. Lexemes whose length exceed the bound specified by the language.

For example, In Fortran, an identifier more than 7 characters long is a lexical error.

Most languages have a bound on the precision of numeric constants.

A constant whose length exceeds this bound is a lexical error.

2. Illegal characters in the program

The characters ~, & and @ occurring in a Pascal program (but not within a string or a comment) are lexical errors.

3. Unterminated strings or comments

In C / C++, a string that does not have the delimiter “ at the end of the string is an unterminated string

multi-line comment that does not end with “*/”

Handling of Lexical Errors

The action taken on detection of a lexical error are:

Issue an appropriate error message indicating the error to the program writer.

- Error of the first type – the entire lexeme is read and then truncated to the specified length
- Error of the second type – skip illegal character.

Alternative is to pass the illegal character to the parser which has better knowledge of the context in which error has occurred. There exist more possibilities of recovery replacement instead of deletion.

Error of the third type – process till end of file and then issue error message if the terminating string is not found.

An Efficient Algorithm for Direct Construction of DFA from Regular Expressions

Recapitulate the Basics of Regular Expressions from FLAT.

- An Alphabet Σ is any finite set of symbols.
- A string over an alphabet is a finite (possibly zero) sequence of symbols from Σ .
- A Language L is any set of strings over an Alphabet Σ , formally expressed as $L \subseteq \Sigma^*$; where $L = \bigcup_{i=0}^{\infty} L^i$
- The empty set Φ is the smallest language over any Σ while the largest is Σ^*
- Operations on languages : union, intersection, concatenation
- The basic regular expression operators and the language denoted by them is given below.

Expression	Describes	Language	Example
ϵ	Empty string	$\{\epsilon\}$	
c	Any non-metalinguistic character c	$\{c\}$	a
r^*	Zero or more r 's	$L(r)^*$	a^*
r^+	One or more r 's	$L(r)^+$	a^+
$r1 \cdot r2$	$r1$ followed by $r2$	$L(r1) \cdot L(r2)$	$a \cdot b$ or ab
$r1 \mid r2$	$r1$ or $r2$	$L(r1) \cup L(r2)$	$a \mid b$
(r)	r	$L(r)$	$(a \mid b)$

Table 1 : Regular Expression Operators and Underlying Languages

Like the operators in programming languages, the regular expression operators also have precedence and associativity operators as given below.

	Regular expression operators with precedence values			
Operator	Parentheses ()	Closure * Reflexive closure +	Concatenation •	Alternate
Precedence	4	3	2	1
Associativity	Left	Right	Left	Left

Table 2 : Properties of Regular Expression Operators

Consider the following regular expression notation for the two tokens used earlier.

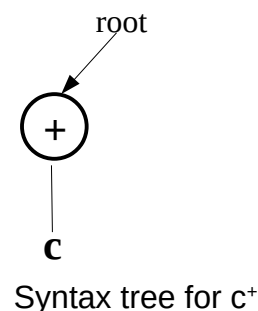
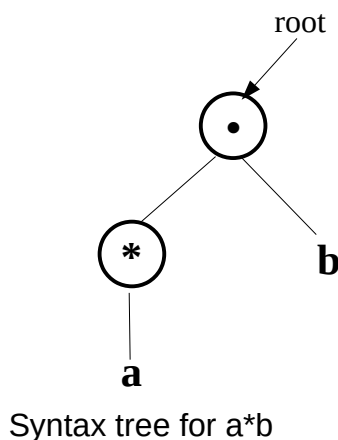
a^*b Token 1
 c^+ Token 2

First we combine all the regular expressions into a single regex as shown below.

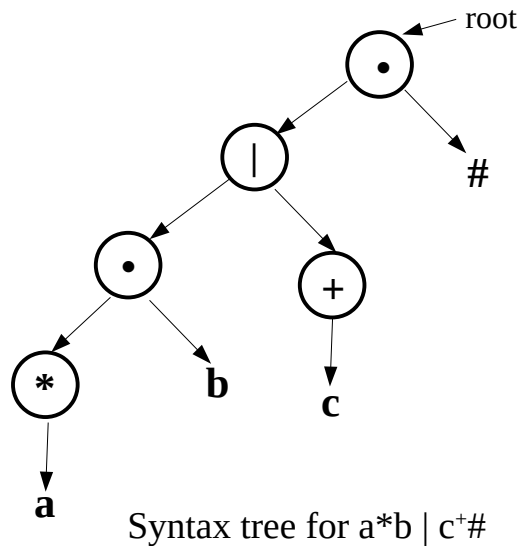
$a^*b \mid c^+$

We construct a syntax tree for the above regex that honours all the operator properties.

The syntax tree for a^*b is same as $a^* \cdot b$; since $*$ has higher precedence than \cdot , the tree is as drawn below. Similarly the syntax tree for c^+ is also drawn.



Combining the two trees with '|' and appending an endmarker symbol (#) yields the tree for the complete regex, rooted at •



It should be clear from the construction process outlined above that given a set of token specifications, one can combine all the regexes to construct a single syntax tree.

We know that a regex denotes a set of strings from the underlying alphabet.

For example, the regex **a** denotes the singleton string {a}.

The regex **a*** denotes the infinite set of strings { ϵ , **a**, aa, aaa, aaaa, }.

The regex **a*b** denotes the infinite set of strings {**b**, ab, aab, aaab, aaaab, ... }.

The regex **c+** denotes the infinite set of strings {**c**, cc, ccc, }.

Examining the regex, $a^*b \mid c^+$, and its associated strings reveal that the first character in any string belonging to the language, $L(a^*b \mid c^+)$, is one of 'a', 'b' or 'c' (highlighted). Since the same character may appear more than once in a regular expression, such as, $(ab)^+ \mid (a \mid b)^*b$, indicating a character symbol by name is ambiguous.

The set of strings of $(ab)^+$ is {ab, abab,}.

The set of strings of $(a \mid b)^*b$ is {b, ab, bb, aab, abb, bab,}.

For the regex, $(ab)^+ \mid (a \mid b)^*b$, the first character in $L((ab)^+ \mid (a \mid b)^*b)$ can be written as either a or b, which is misleading and wrong.

Instead of the character symbol, if we can replace its position index, a precise information is obtained. All the alphabet symbols have been given a unique index value.

$$(a\ b)^+ \mid (a\ \mid\ b)^* b$$

1 2 3 4 5

The set of strings of $(ab)^+$ is {ab, abab,}.

12 1212

The set of strings of $(a \mid b)^*b$ is {b, ab, bb, aab, abb, bab,}.

5 35 45 335 345 435

For the regex, $(ab)^+ \mid (a \mid b)^*b$, the first character in $L((ab)^+ \mid (a \mid b)^*b)$ can now be written as {1, 3, 4, 5} because of the highlighted strings above.

Using the position of alphabet symbols, an algorithm was designed that uses the syntax tree as input and directly constructs the dfa using the position of symbols in the input and the semantics of the regex operators.

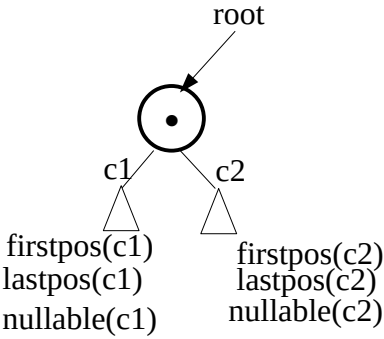
A few terms and definitions are needed. It is assumed that a syntax tree for the regex has been created. The internal nodes of the tree are the regex operators and the the alphabet symbols are place at the leaf nodes.

Four functions are used in the construction process. Consider a syntax tree rooted at a node, say x.

- $nullable(x)$ indicates whether the tree rooted at x can generate the null string ϵ in which case the function returns True otherwise it returns False.
- $firstpos(x)$ denotes the set of characters, by position, that are the first characters in any string generated from x
- $lastpos(x)$ denotes the set of characters, by position, that are the last characters in any string generated from x
- $followpos(i)$ denotes the set of characters, by position, that can follow the character at position i in any string generated from th regex.

Let Σ be the underlying alphabet. The rules for construction of the sets, $\text{firstpos}()$, $\text{lastpos}()$ and $\text{followpos}()$ are summarized in the following table. It is assumed that $\text{followpos}(i)$ is initialized to \emptyset , \forall positions labeled i . The table entry for $c_1 \bullet c_2$ is explained below.

Concatenation Operator \bullet : Consider a tree rooted at \bullet with two subtrees rooted at c_1 and c_2 respectively. It is assumed that $\text{nullable}()$, $\text{firstpos}()$ and $\text{lastpos}()$ are already known for both c_1 and c_2 .

	<p>The symbols that happen to be at the first positions of $\text{firstpos}(\bullet)$ are definitely those positions that are present in $\text{firstpos}(c_1)$.</p> <p>However in case ϵ belongs to $\text{firstpos}(c_1)$, then all the positions in $\text{firstpos}(c_2)$ are also included in $\text{firstpos}(\bullet)$. The calculations for $\text{nullable}(\bullet)$, $\text{firstpos}(\bullet)$ and $\text{lastpos}(\bullet)$, given in the table, in terms of their children are based on these relations.</p>
--	--

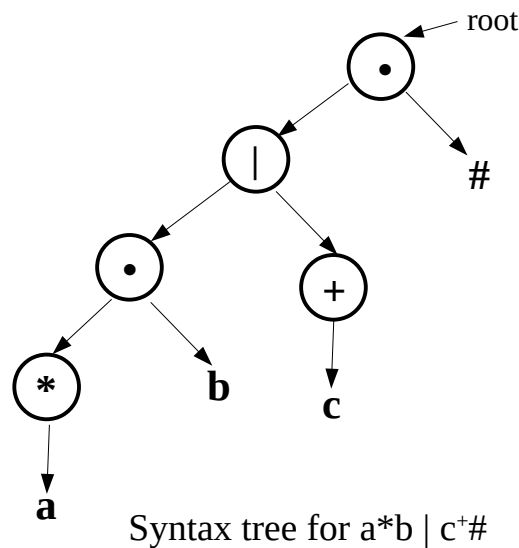
Rules for computation of the four functions

<i>node n</i>	<i>nullable(n)</i>	<i>firstpos(n)</i>	<i>lastpos(n)</i>	<i>followpos(i)</i>
Leaf labeled ϵ	true	\emptyset	\emptyset	
Leaf labeled with $a \in \Sigma$ at position i	false	$\{i\}$	$\{i\}$	
$c_1 \mid c_2$	$\text{nullable}(c_1)$ or $\text{nullable}(c_2)$	$\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$	$\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$	
$c_1 \bullet c_2$	$\text{nullable}(c_1)$ and $\text{nullable}(c_2)$	if $\text{nullable}(c_1)$ then $\text{firstpos}(c_1) \cup \text{firstpos}(c_2)$	if $\text{nullable}(c_2)$ then $\text{lastpos}(c_1) \cup \text{lastpos}(c_2)$ else $\text{lastpos}(c_2)$	if $i \in \text{lastpos}(c_1)$ then $\text{followpos}(i) \neq \text{firstpos}(c_2)$

<i>node n</i>	<i>nullable(n)</i>	<i>firstpos(n)</i>	<i>lastpos(n)</i>	<i>followpos(i)</i>
		else firstpos(c_1)		
c^*	true	firstpos(c)	lastpos(c)	if $i \in \text{lastpos}(c)$ then followpos(i) += firstpos(c)
c^+	nullable(c)	firstpos(c)	lastpos(c)	if $i \in \text{lastpos}(c)$ then followpos(i) += firstpos(c)

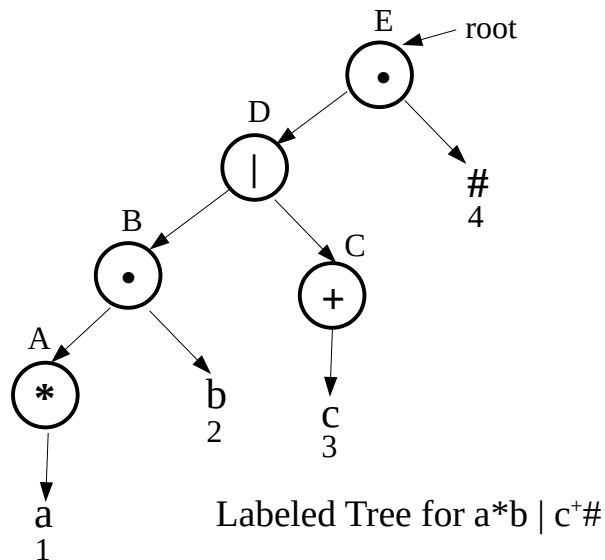
From the table above, it can be seen that only 3 operators, $\{*, +, \bullet\}$ contribute to the followpos() information, while the other operators '|' and the leaf nodes do not affect followpos().

The syntax tree for the regex is used to illustrate the computation of the 4 functions by making a bottom up traversal of the tree.



All the leaf nodes are numbered from 1 to 4 while the internal nodes are labeled by upper case alphabets.

The tree with all its nodes with labels are shown below.



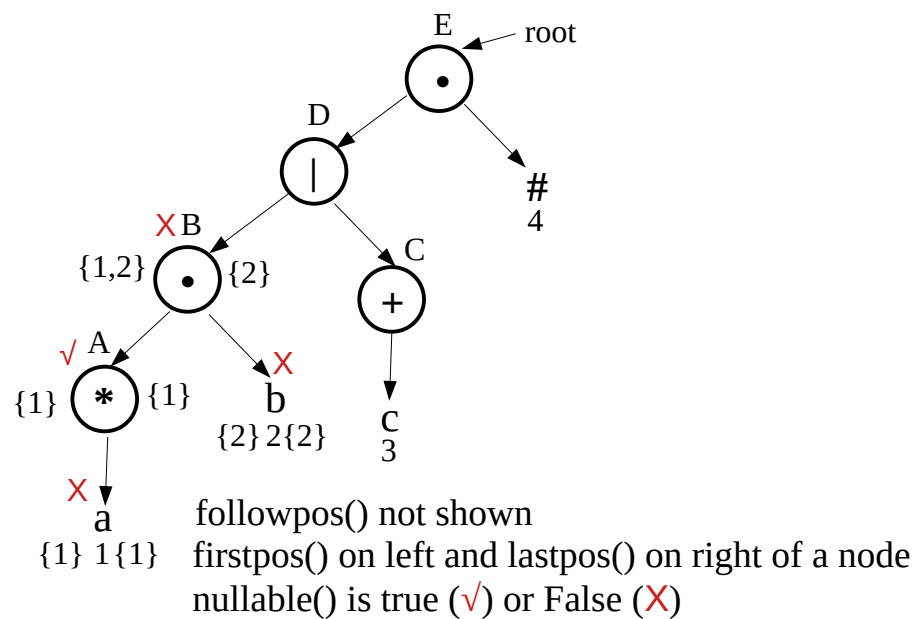
The symbol X is used to denote that the node is NOT nullable while the symbol ✓ is used to denote that the node is nullable. The firstpos() and lastpos() are placed at the left and right of a given node.

1. Start with the leaf node 1. This is not nullable, so nullable(1) = False, and firstpos(1) = lastpos(1) = {1}
2. For the internal node labeled A, nullable(A) is True, firstpos(A) = firstpos(1) = {1} and lastpos(A) = lastpos(1) = {1}. Since A has operator *, it also contributes to followpos(). Since 1 belongs to lastpos(1), we create followpos(1) = {1}, where the {1} in the RHS is because of firstpos(1).
3. The leaf node 2 is NOT nullable and firstpos(2) = lastpos(2) = {2}
4. The internal node B has • operation whose left and right children have all the 3 sets computed. The nullable(B) is False because the right child is not nullable. firstpos(B) = firstpos(A) \cup firstpos(2) = {1, 2}
lastpos(B) = lastpos(2) = {2}.

Since B has • , it contributes to followpos(). For all elements in lastpos(c1), which turns out to be lastpos(1) = {1}, we have to add firstpos(c2), that is 2, to the existing followpos(1), which now becomes {1, 2}.

The annotated tree and a table that records all the calculations performed thus far are given below.

Node / leaf	Symbol	nullable (node)	firstpos()	lastpos()	followpos()
1	a	False (X)	{1}	{1}	{1} ∪ {2}
A	*	True (✓)	{1}	{1}	
2	b	False (X)	{2}	{2}	
B	•	False (X)	{1,2}	{2}	



Labeled Tree for a*b | c⁺# : values at subtree rooted at B

Note that $\text{followpos}(3) = \{3\}$ due to $+$ in node C. Further the \bullet operation at root E, results in assigning $\text{followpos}(\text{lastpos}(D))$ to $\text{firstpos}(4)$, that is updating the $\text{followpos}()$ at 2 and 3 as indicated by, $\text{followpos}(2) \cup = \{4\}$, and $\text{followpos}(3) \cup = \{4\}$. Continuing along the same lines, when the entire syntax has been processed, the final information of all the 4 sets are as shown in the following table.

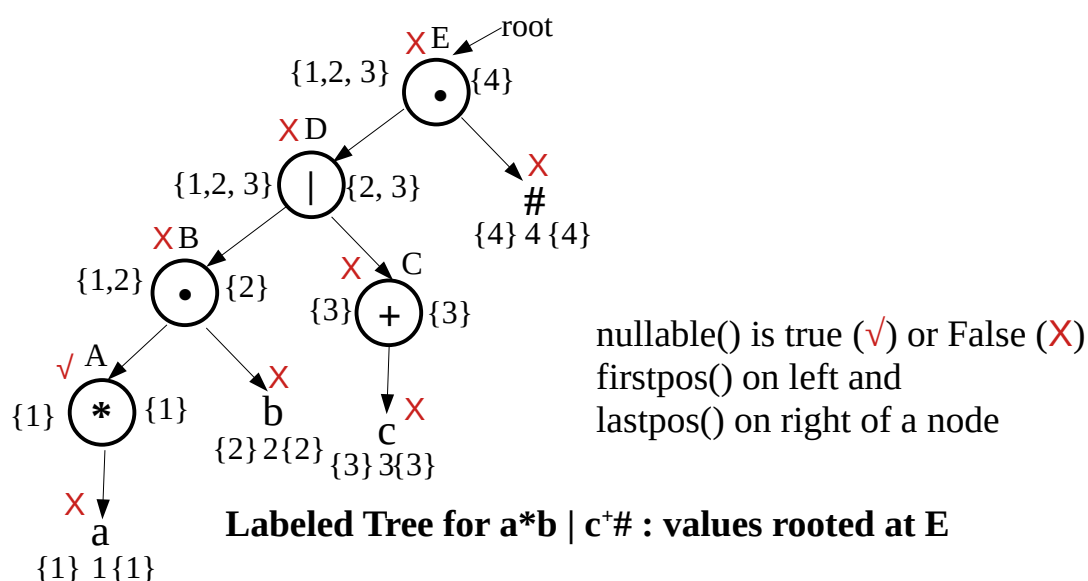
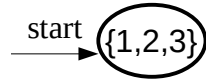


Table 3 : Values of 4 functions at each node of the Tree

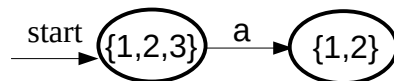
Node / leaf	Symbol	nullable (node)	firstpos()	lastpos()	followpos()
1	a	False	{1}	{1}	$\{1\} \cup \{2\}$
A	*	True	{1}	{1}	
2	b	False	{2}	{2}	{4}
B	•	False	{1,2}	{2}	
3	c	False	{3}	{3}	$\{3\} \cup \{4\}$
C	+	False	{3}	{3}	
D		False	{1,2,3}	{2,3}	
4	#	False	{4}	{4}	
E	•	False	{1,2,3}	{4}	

The final step is to construct the dfa using the firstpos(), lastpos() and followpos() information of Table 3.

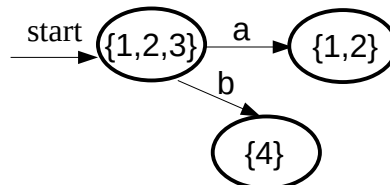
- The start state of the dfa is given by firstpos(root) which is $\{1,2,3\}$ for our regex.



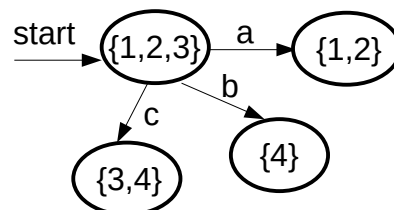
- The state transitions on symbols a, b and c are now calculated.
- For transition on 'a' from the start state, we need to find all positions that correspond to 'a' in $\{1,2,3\}$. The position 1 is the only one that has label 'a'. The next state of $\{1,2,3\}$ on symbol 'a' is given by followpos(1) = $\{1,2\}$.



- For transition on 'b' from the start state, we need to find all positions that correspond to 'b' in $\{1,2,3\}$. The position 2 is the only one that has label 'b'. The next state of $\{1,2,3\}$ on symbol 'b' is given by followpos(2) = $\{4\}$. The dfa at this stage is

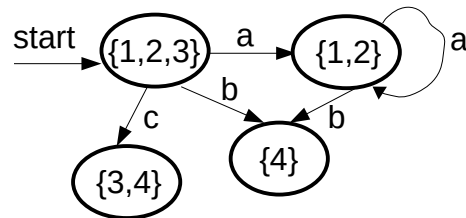


- For transition on 'c' from the start state, we need to find all positions that correspond to 'c' in $\{1,2,3\}$. The position 3 is the only one that has label 'c'. The next state of $\{1,2,3\}$ on symbol 'c' is given by followpos(3) = $\{3, 4\}$. The dfa at this stage is

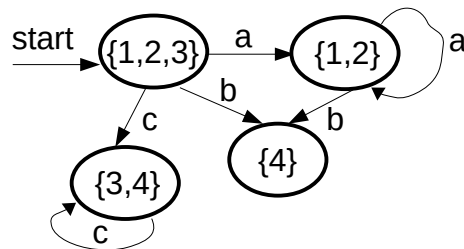


- The state with $\{1,2\}$ is now picked up for possible transitions. On symbol 'a', the next state is given by followpos(1) = $\{1,2\}$, that is the same state. On 'b', the next state is followpos(2) = $\{4\}$. There is no transition from $\{1,2\}$ on symbol 'c'.

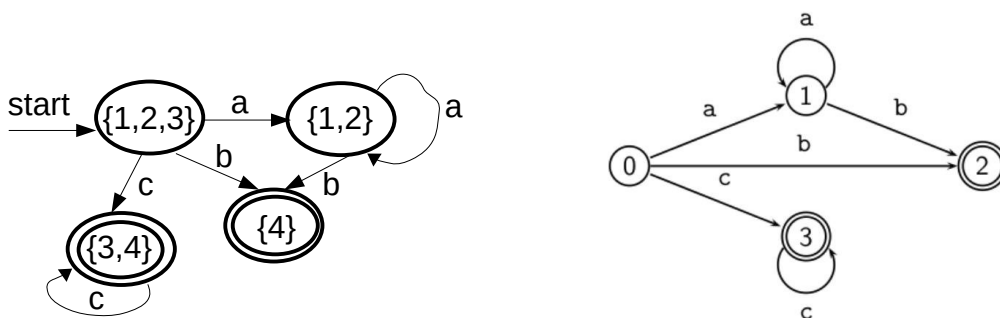
- Adding these transitions produces the following dfa.



- State corresponding to $\{4\}$ has no transitions as 4 does not correspond to any of the alphabet symbols $\{a, b, c\}$.
- The state with $\{3,4\}$ is now picked up for exploration. The only symbol is 'c' that corresponds to position 3. The next state of $\{3, 4\}$ on 'c' is given by $\text{followpos}(3) = \{3,4\}$, that is the same state. The resulting dfa at this stage is



- The dfa is now complete. The final states of the dfa are the states that contain the endmarker symbol #, which has label 4. Hence there are 2 final states, namely $\{4\}$ and $\{3,4\}$. Marking the final states we get the dfa shown below.



Comparing with the dfa given earlier, we can see that the two dfas are identical, except for the labelling of the states.

The direct dfa construction from a regex is summarized in the following algorithm.

Algorithm

1. Construct the tree for $r\#$ for the given regular expression r
2. Construct functions *nullable()*, *firstpos()*, *lastpos()* and *followpos()*
3. Let *firstpos(root)* be the start state. Push it on top of a stack.
While (stack not empty)
do begin
 pop the top state U off the stack; mark it;
 for each input symbol a do
 begin
 let $p_1, p_2, p_3, \dots, p_k$ be the positions in U corresponding to symbol a ;
 Let $V = \text{followpos}(p_1) \cup \text{followpos}(p_2) \cup \dots \cup \text{followpos}(p_k)$;
 place V on stack if not marked and not already in stack;
 make a transition from U to V labeled a ;
 end
end
4. Final states are the states containing the positions corresponding to $\#$.

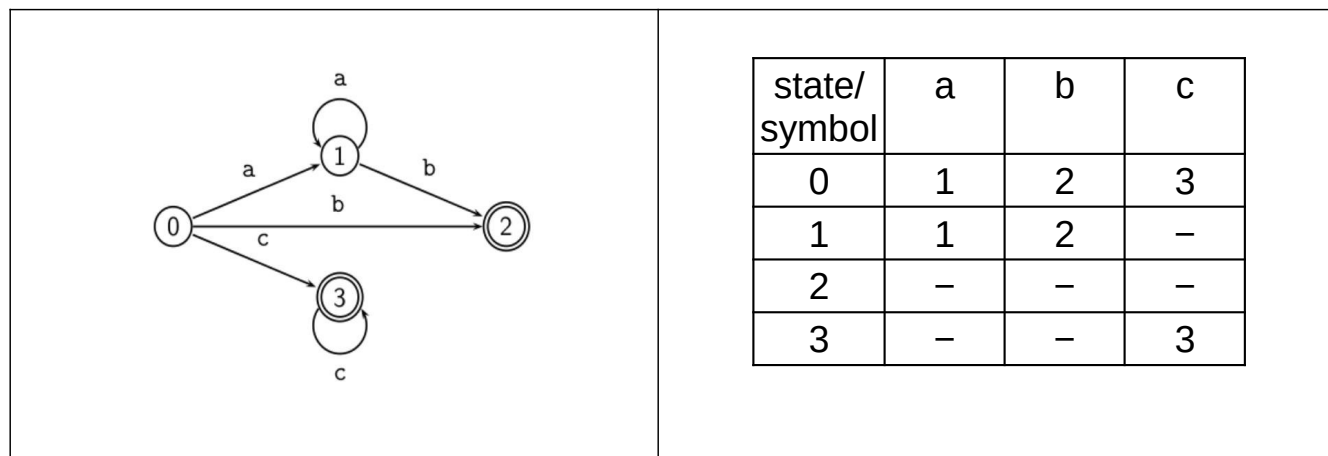
The DFA that is constructed directly from regex is not guaranteed to be the minimal dfa. DFA minimization algorithm is applied to minimize the dfa constructed by the above algorithm.

Representation of a DFA

The dfa that results from the tokens of most programming languages are rather large in size. Therefore a 2-dimensional representation of the dfa is memory intensive, though this representation permits the determination of $\text{nextstate}[s, c]$, where s is a state of the dfa and c is a symbol of the alphabet in constant time, $O(1)$. The dfa for lexical analysers is also fairly sparse leading to wasteful memory space.

The challenge before designers of scanner generation was to find a representation that was memory efficient while retaining the constant access time of $\text{nextstate}[s, c]$.

This issue is explained with the dfa constructed directly from the regex $a^*b \mid c^+$. An obvious 2-dimensional representation of the dfa is shown.



The 2D representation needs 12 entries as outlined above.

Incomplete : Example for 4-array scheme to be inserted

Illustration of 4 Array Scheme for representing a DFA

4 array layout of a DFA is the internal representation used by the lexical analyzer generation software, Lex or Flex. The efficacy of this interesting data structure is explained with the help of an example. Consider a lexical analyser for tokens encountered in programming languages. The following 3 tokens, {Keyword, Identifier and Number} are specified and the alphabet is restricted to 15 characters, {a, b, c, d, e, f, g, h, i, j, 0, 1, 2, 3, 4} to contain the size of the dfa that recognizes all the tokens. The symbols from the alphabet are shown in bold. Note that the token Number may admit numbers like {00, 000, ...} but are irrelevant for the present discussion.

```

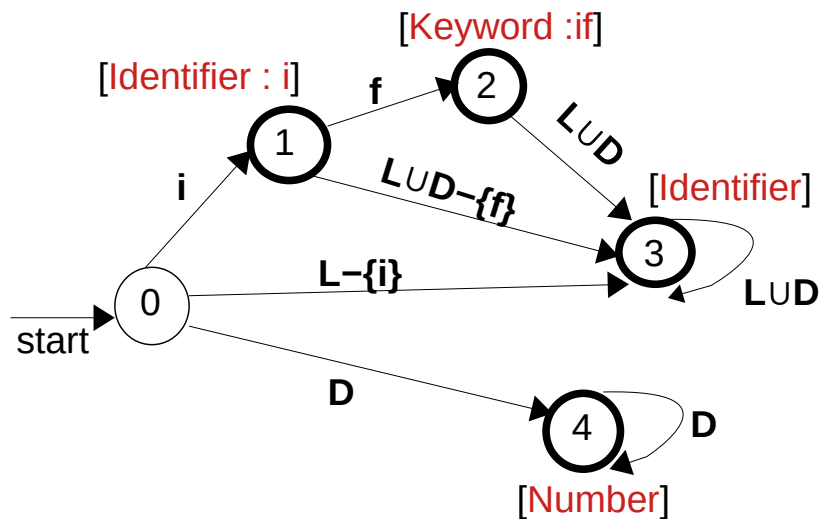
Letter    a | b | c | d | e | f | g | h | i | j
Digit     0 | 1 | 2 | 3 | 4
Keyword   if
Identifier letter (letter | digit)*
Number    digit+

```

Manually constructed DFA is given below. It is left to the reader to determine the difference of this dfa with respect to the one constructed by the direct regular

expression to DFA algorithm. We use L and D be the abbreviations instead of the expressions Letter and Digit respectively. Note that L has 10 symbols from alphabet while D has 5 numeric symbols. The final states are nodes with bold boundary. The edge labels denote the set of symbols for which transition along the edge happens. The edge labels use sets to indicate the collection of symbols. For example,

L- {i} is a shorthand for the collection, {a, b, c, d, e, f, g, h, j}. The other edge labels are to be elaborated analogously.



The tokens that are recognized at the 4 final states are attached with them in red font.

The symbols of the alphabet are assigned the following offset values :

Table 4 : Symbols and offset values

offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Symbol	a	b	c	d	e	f	g	h	i	j	0	1	2	3	4

1. The 2D representation of this DFA is shown in Table 5.

Table 5 : DFA : 2-Dimensional Representation

State	15 Alphabet Symbols by their offset value														
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	3	3	3	3	3	3	3	3	1	3	4	4	4	4	4
1	3	3	3	3	3	2	3	3	3	3	3	3	3	3	3
2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
4	–	–	–	–	–	–	–	–	–	–	4	4	4	4	4

Observations from the DFA of Table 5

- The '–' indicates there is no transition from state 4 for all the 10 alphabet characters, 'a' through 'j'
- The space required is for $5 \times 15 = 75$ entries
- All the 15 entries for states 2 and 3 are identical
- 14 entries of state 1 is identical to that of state 2 (and 3); only one entry, [1,5] is different than that of [2, 5] (or [3,5])
- State 0 has a considerable overlap with that of state 2 (and 3); 9 entries are identical while 6 are different.

The C Alphabet, that is the full character set of C, has at least 96 symbols, Alphabet(52 characters), digits (10), and 34 symbols ! " # % & ' () * + , - . / : ; < = > ? [\] ^ _ { | } ~ space horizontal-tab vertical-tab form-feed, newline

Assuming 100 states in the complete DFA, the space requirement is for 10000 entries. However the full table is usually sparse, so there is inefficient space utilization, if the 2D data structure is directly used. The advantage is that the `nextstate[s, offset]`, for any state `s` and any offset (in the range[0, 95]), is obtained in $O(1)$ time.

Because of the size and cost of RAM, during the period when Lex was under development, researchers were looking for a data structure that had the following characteristics :

- Determination of a state transition, `nextstate[s, offset]`, should be efficient, possibly $O(1)$ in practice. This rules out a linked list data structure as a possible candidate, even though it is well known that the sparsity in the dfa can be exploited by a list data structure.

- The overlap between 2 and more states (found to occur very frequently in such dfa), should be exploited to eliminate redundant entries.
- The sparsity present in the DFA be possibly reduced by plugging the holes with useful data.

The data structure employed by the designers of lex to represent the DFA is a novel one and squarely meets the requirements.

Proposed Data Structure using 4 one-dimensional arrays

Four arrays named as Default[], Base[], Next[] and Check[] are used.

The 2 arrays, Default and Base are of the same size as the number of states, $|S|$. The size of the remaining 2 arrays, say m , where $m > |S|$. The challenge is to keep m as low as possible. The function of the arrays, in brief, are described in the following.

- The value of Base[k], for the state k , $0 \leq k \leq |S|-1$, is used as an index in the Next[] and Check[] arrays. Let n be the number of alphabet symbols, $|\Sigma| = n$, with assigned offset indices from 0 to $n-1$.
- Then the n state transitions of state k are laid out in the array positions, Next[Base[k] + 0] to Next[Base[k] + $n-1$].
- The value of Check[j], $0 \leq j \leq n-1$, denote the state for which Next[j] is the destination state.
- Default[k], $0 \leq k \leq |S|-1$, denotes a state that is to be used as a default for state k . A default state, Default[k] shares many transitions with the original state k .

The construction process is better appreciated by working it out for the dfa represented by Table 5.

There are several choices for choosing the layout and in general finding an optimal layout is a NP-hard problem. Heuristics are used to find a good layout, that is an appropriate contiguous segment in the Next[] and Check[] arrays.

1. States 1, 2 and 3 have a large overlap. We choose state 2 to start the layout. WLOG we choose Base[2] to be 0.
 - $\text{nextstate}[2, i] = 3$ for $0 \leq i \leq 14$; therefore $\text{Next}[\text{Base}[2] + i] = 3$; $0 \leq i \leq 14$. Also since all these transitions are for state 2, we assign $\text{Check}[\text{Base}[2] + i] = 2$; $0 \leq i \leq 14$; Since there is no state that 2 shares its entry with (being the first chosen state), we set $\text{Default}[2]$ to be Null (denoted by $-$)
 - The entries of state 2 is now complete and shown in Table 6.
 - You should verify that all the transitions of state 2 are obtained by if $(\text{CHECK}[\text{BASE}[2] + i] == 2)$ then $\text{return}(\text{NEXT}[\text{BASE}[2] + i])$ for all i in $0 \leq i \leq 14$; essentially 3 array references.

Table 6 : 4-arrays after placing transitions of state 2

Index	Default	Base		Index	Next	Check
0				0	3	2
1				1	3	2
2	-	0		2	3	2
3				3	3	2
4				4	3	2
				5	3	2
				6	3	2
				7	3	2
				8	3	2
				9	3	2
				10	3	2
				11	3	2
				12	3	2
				13	3	2
				14	3	2

2. We now choose state 3 for laying out its transitions. Since state 3 is identical to state 2 (already placed), we choose Base[3] as 0 and assign Default[3] to 2.

The transitions of state 3 are now determined by the following logic.

```
if (CHECK[BASE[3] + i] != 3)
    then return (nextstate( DEFAULT[3], i))
```

Interesting to observe that only Base[3] and Default[3] changed while Next[] and Check[] remains unchanged.

Q. How many array references are required to find nextstate[3, i];
 $0 \leq i \leq 14$?

Table 7 : 4-arrays after placing transitions of states 2 and 3

Index	Default	Base		Index	Next	Check
0				0	3	2
1				1	3	2
2	–	0		2	3	2
3	2	0		3	3	2
4				4	3	2
				5	3	2
				6	3	2
				7	3	2
				8	3	2
				9	3	2
				10	3	2
				11	3	2
				12	3	2
				13	3	2
				14	3	2

- The state to be placed now is 0. nextstate[0, 8] is 1 which is different from nextstate[2, 8], but nextstate[0, i] = nextstate[2,i]; $0 \leq i \leq 7$. The setting of Default[0] = 2 will take care of the first 8 transitions of 0. We need a free slot to place nextstate[0, 8] and the first free slot is available at index 15.

To do this, Base[0] must be 7. The nextstate[0, 9] can be served by the default state 2, provided index 16, which base value 7 plus 9 is kept vacant (set to -1). The indices 17 to 21 can be used to save nextstate[0, k]; $10 \leq k \leq 14$. The 4-arrays after placing state 0 is shown in Table 8.

Table 8 : 4-arrays after placing states 2, 3 and 0

Index	Default	Base		Index	Next	Check
0	2	7		0	3	2
1				1	3	2
2	–	0		2	3	2
3	2	0		3	3	2
4				4	3	2
				5	3	2
				6	3	2
				7	3	2
				8	3	2
				9	3	2
				10	3	2
				11	3	2
				12	3	2
				13	3	2
				14	3	2
				15	1	0
				16		-1
				17	4	0
				18	4	0
				19	4	0
				20	4	0
				21	4	0

- State 1 is now chosen for placement. Except for nextstate[1, 5] = 2, all other entries are same as state 2. State 2 should therefore be used as the default state for 1. To place [1, 5], we use the free slot at index 16 by

setting Base[1] to 11. The indices that state 1 will use are from 11 to 25. By setting Check[22] = Check[23] = Check[24] = Check[25] = -1, we ensure that default state 2 will be used for next state information.

Table 9 : 4-arrays after placing states 2, 3, 0 and 1

Index	Default	Base	Index	Next	Check
0	2	7	0	3	2
1	2	11	1	3	2
2	-	0	2	3	2
3	2	0	3	3	2
4			4	3	2
			5	3	2
			6	3	2
			7	3	2
			8	3	2
			9	3	2
			10	3	2
			11	3	2
			12	3	2
			13	3	2
			14	3	2
			15	1	0
			16	2	1
			17	4	0
			18	4	0
			19	4	0
			20	4	0
			21	4	0
			22		-1
			23		-1
			24		-1
			25		-1

5. Finally we have to place state 4. State 4 has 5 common transitions with state 0 but that is not easy to reuse (Why?). So we use the indices 22 to 26 to place the transitions for this state by Base[4] set to 12 with no default state.

Table 10 : 4-arrays after placing all 5 states

Index	Default	Base		Index	Next	Check
0	2	7		0	3	2
1	2	11		1	3	2
2	–	0		2	3	2
3	2	0		3	3	2
4	–	12		4	3	2
				5	3	2
				6	3	2
				7	3	2
				8	3	2
				9	3	2
				10	3	2
				11	3	2
				12	3	2
				13	3	2
				14	3	2
				15	1	0
				16	2	1
				17	4	0
				18	4	0
				19	4	0
				20	4	0
				21	4	0
				22	4	4
				23	4	4
				24	4	4
				25	4	4

Index	Default	Base	Index	Next	Check
			26	4	4

The construction of the 4-array representation for the given DFA is complete. Verify that all the state transition information is identical in both the representations.

The benefits of the scheme for this example may be summarized as

- 4-array scheme uses $26*2 + 5*2 = 62$ units as compared to 75 units for a dense matrix representation
- There is no hole in the Next[] and Check[] arrays so we can claim that dense packing has been achieved. However it is not obvious whether the size of these 2 arrays could be reduced further by better exploitation of common entries.
- While the cost of nextstate[s, i] for any i is O(1), the 4-array scheme has a higher time complexity. What is the largest number of array references required for nextstate[s, i] for any i?

This completes the layout of a DFA in the 4-array scheme. The calculation of the next state information in a 4array scheme is outlined in the following function.

```
function nextstate (s, a)
{ if CHECK[BASE[s] + a] = s
  then NEXT[BASE[s] + a]
  else
    return (nextstate( DEFAULT[s], a))
}
```

A heuristic, which works well in practice to fill up the four arrays, is to find for a given state, the lowest BASE, so that the special entries of the state can be filled without conflicting with the existing entries.

Take-away from Lexical Analyser Generators

- A different algorithm, developed by compiler designers, for constructing a DFA directly from a regular expression. The algorithm is efficient and provably correct. The lexical analyzer generator tool Lex uses this algorithm.
- A novel data structure named as 4 array scheme has been designed to store a large DFA efficiently (reduced space requirement) with marginal increase in the cost for finding the nextstate. Lex uses the 4 array scheme as its internal representation for a DFA.
- The direct DFA construction algorithm does not claim that the generated dfa is minimal. Therefore the generated dfa requires a minimization algorithm to produce the minimal dfa.
- For the sake of completeness, an algorithm for state minimization (covered in FLAT) is reproduced below, without explanation, for your perusal.

Minimizing states of a dfa

Outline of an algorithm that minimizes the states S of a given dfa. This has been covered in the course on FLAT and hence details are omitted.

1. Construct an initial partition $\Pi = \{ S - F, F_1, F_2, \dots, F_n \}$, where $F = F_1 \cup F_2 \cup \dots \cup F_n$ and each F_i is the a final state.
2. For each set G in Π do
 - Partition G into subsets such that two states s and t of G are in the same subset if and only if
For all input symbols a , and for states s and t such that
states s and t have transitions onto states in the same set of Π ;
construct a new partition Π_{new} by replacing G in Π by the set of its subsets.
3. If $\Pi_{\text{new}} == \Pi$, then $\Pi_{\text{final}} := \Pi$ and continue with step 4.
Otherwise repeat step 2 with $\Pi := \Pi_{\text{new}}$.
4. Merge states in the same set of the partition.

5. Remove any dead (unreachable) states.

Reporting of typos / errors in this document will be appreciated.

End of Notes on Lexical Analysis