Lexical Analysis

Amitabha Sanyal

(www.cse.iitb.ac.in/˜as)

Department of Computer Science and Engineering,

Indian Institute of Technology, Bombay

January 2022

Introduction

The input program – as you see it.

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

Introduction

The same program – as the compiler initially sees it. A continuous sequence of characters without any structure

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

␣	–	The blank space character
↩	–	The return character

How do you make the compiler see what you see?

Discovering the structure of the program

Step 1:

a. Break up this string into the smallest meaningful units.

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

We get a sequence of *lexemes* or *tokens*.

Discovering the structure of the program

Step 1:

b. During this process, remove the ␣ and the ↩ characters.

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

Steps 1a. and 1b. are interleaved.

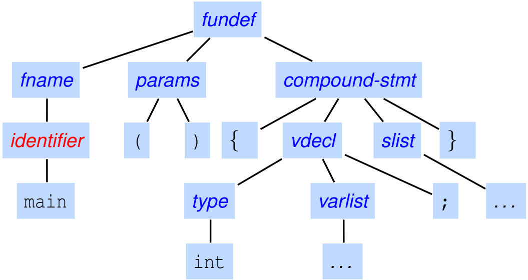
This is *lexical analysis* or *scanning*.

Discovering the structure of the program

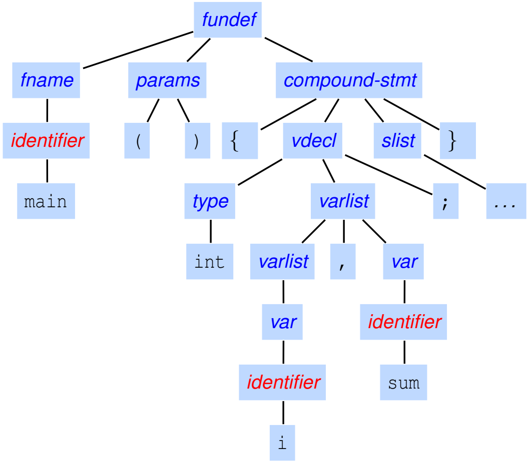
Step 2:

Now group the lexemes to form larger structures.

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



Discovering the structure of the program



This is *syntax analysis* or *parsing*.

Discovering the structure of the program

Why is structure finding done in two steps?

- The process of breaking a program into lexemes (scanning) is easier. Use a separate technique to do this.
- Reduces the work to be done by the parser.

However, there are tools (Antlr for example) that indeed combine scanning with parsing.

Lexemes, Tokens and Patterns

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

Distinguish between

- *lexemes* – smallest logical units (words) of a program.
  Examples – `i, sum, for, 10, ++, "%d\n", <=`.
- *tokens* – sets of similar lexemes, i.e. lexemes which have a common syntactic description.
  Examples –
  identifier = {`i, sum, buffer,` ...}
  int_constant = {`1, 10,` ...}
  addop = {`+, -`}

Lexemes, Tokens and Patterns

What is the basis for grouping lexemes into tokens?

- Why can't `addop` and `mulop` be combined? Why can't + be a token by itself?

Lexemes which play similar roles during syntax analysis are grouped into a common token.

- Operators in `addop` and `mulop` have different roles – `mulop` has an higher precedence than `addop`.
- Each keyword plays a different role – is therefore a token by itself.
- Each punctuation symbol and each delimiter is a token by itself.
- All comments are uniformly ignored. They are all grouped under the same token.
- All identifiers are grouped in a common token.

Lexemes, Tokens and Patterns

Lexemes that are not passed to the later stages of a compiler:

- comments
- white spaces – tab, blanks and newlines
    - White spaces are more like separators between lexemes.

These too have to be detected and then ignored.

Lexemes, Tokens and Patterns

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

EXAMPLE

lexeme	**<token, token value>**
`3`	< const, `3`>
`A`	<identifier, `A`>
`if`	<if, –>
`=`	<assignop, –>
`>`	<relop, `>`>
`;`	<semicolon, –>

Lexemes, Tokens and Patterns

The lexical analyser:

- detects the next lexeme
- categorises it into the right token
- passes to the syntax analyser
  - the token name for further syntax analysis
  - the lexeme itself, in some form, for stages beyond syntax analysis

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:**
   2.1 Integer Constants – 4 byte and 8 byte (ends with a L) representations.
   - Binary – `0b0000011`,
   - Octal – `027` (Note the leading 0),
   - Hex – `0x0f28`,
   - Decimal – `1, -1`

   2.2 Floating point constants
   - Float – `1.0345F, 1.04E-12f, .0345f, 1.04e-13f` – ends with `f` or `F`,
   - Double – `5.6E-120D, 123.4d, 0.1` – ends with d or D, or does not end with any of `f, F, d, D`

   2.3 Boolean constants – `true` and `false`
   2.4 Character constants – `'a'`, `'\u0034'` (Unicode hex), `'\t'`
   2.5 String constants – `""`, `"\""`, `"A string"`.
   2.6 Null constant – `null`.

Example – tokens in Java

3. **Delimiters:** `(`, `)`, $\{$, $\}$, `[`, `]` `,` `;`, `.` and `,`
4. **Operators:** =, >, < ... `>>>=`
5. **Keywords:** `abstract`, `boolean` ... `volatile`, `while`.

Lexemes, Tokens and Patterns

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

Variants in different languages.

- String of alphanumeric characters. The first character is an alphabet.
- a string of alphanumeric characters in which the first character is an alphabet. It has a length of at most 31.
- a string of alphabet or numeric or underline characters in which the first character is an alphabet or an underline. It has a length of at most 31. Any character after the 31st are ignored.

Such descriptions are called *patterns.* The description may be informal or formal. *Regular expressions* are the most commonly used formal patterns.

Lexemes, Tokens and Patterns

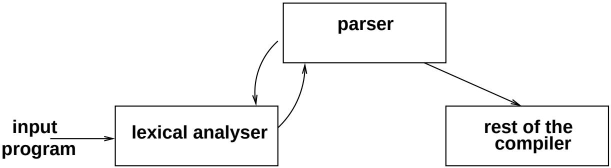
A pattern is used to

- *specify tokens* precisely
- *build a recognizer* from such specifications

Basic concepts and issues

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

- The front end of most compilers is parser driven.
- When the parser needs the next token, it invokes the Lexical Analyser.
- Instead of analysing the entire input string, the lexical analyser sees enough of the input string to return a single token.
- The actions of the lexical analyser and parser are interleaved.



Creating a Lexical Analyzer

Two approaches:

1. *Hand code* – This is only of historical interest now.
   - Possibly more efficient.
2. *Use a generator* – To generate the lexical analyser from a formal description.
   - The generation process is faster.
   - Less prone to errors.

Automatic Generation of Lexical Analysers

- A formal description (specification) of the tokens of the source language, will consist of:
  - a regular expression describing each token, and
  - a code fragment called an action routine describing the action to be performed, on identifying each token.
- Here is a description of whole numbers and identifiers in form accepted by the lexical analyser generator Lex.

```
{DIGIT}+                      { yylval = atoi(yytext);
                                return NUM; }
{LETTER}({LETTER}|{DIGIT})*   { yylval = yytext;
                                return IDENTIFIER; }
```

- The global variable `yylval` holds the token attribute (henceforth to be called token value).

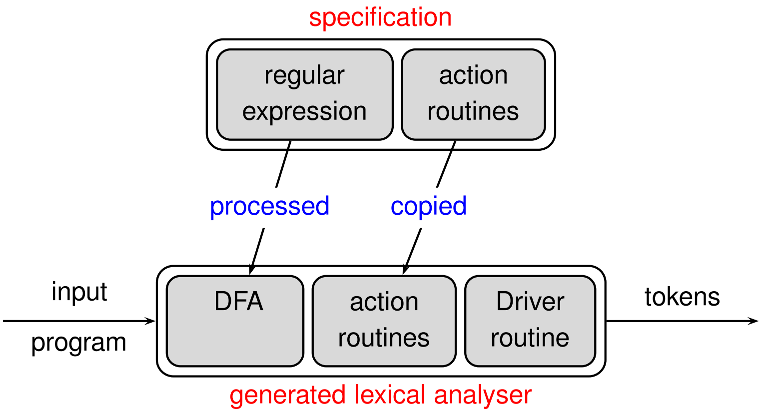
Automatic Generation of Lexical Analysers

Lex can read this description and generate a lexical analyser for whole numbers and identifiers. How?

- The generator puts together:
  - A deterministic finite automaton (DFA) constructed from the token specification.
  - A code fragment called a driver routine which can traverse any DFA.
  - Code for the action routines.
- These three things taken together constitutes the generated lexical analyser.

Automatic Generation of Lexical Analysers

- How is the lexical analyser generated from the description?



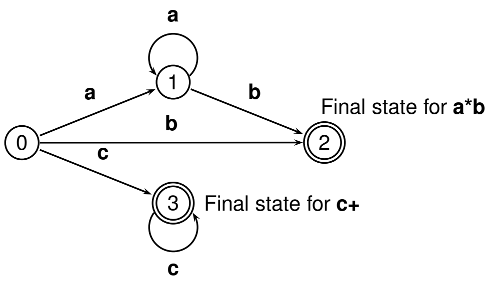
- Note that the driver routine is common for all generated lexical analysers.

Example of Lexical Analyser Generation

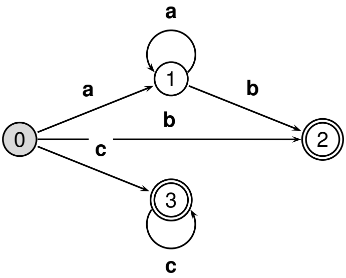
Suppose a language has two tokens

Pattern	Action
a*b	**{ printf("Token 1 found");}**
c+	**{ printf("Token 2 found");}**

From the description, construct a structure called a deterministic finite automaton (DFA).



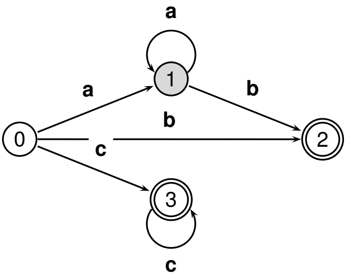
Behaviour of the driver routine



Input: **aabadbcc**↩
Output:

- Track the current state (called **state**) and the last final state **final**. Initially **state** = 0, **final** = -1.

Behaviour of the driver routine

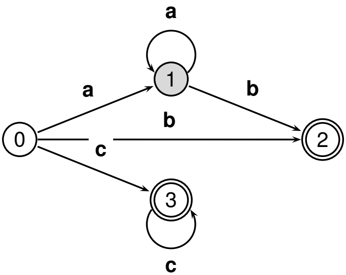


Input: **aabadbcc**$\hookleftarrow$

Output:

- **state** = 1, **final** = -1

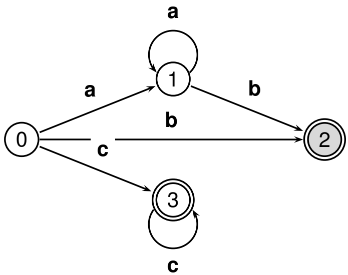
Behaviour of the driver routine



Input: **aabadbcc**$\hookleftarrow$
Output:

- **state** = 1, **final** = -1

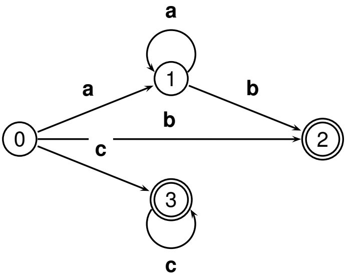
Behaviour of the driver routine



Input: **aabadbcc**↩
Output:

- **state** = 2, **final** = 2. Also mark **a.**

Behaviour of the driver routine

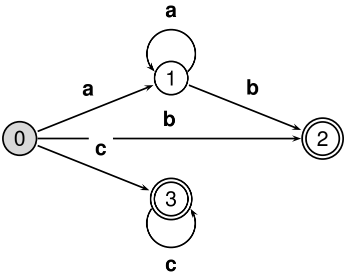


Input: **aabadbcc**↩

Output: **Token 1 found**

- No transition from state 2 on **a**. Perform action corresponding to state 2. The lexeme detected is **aab**.

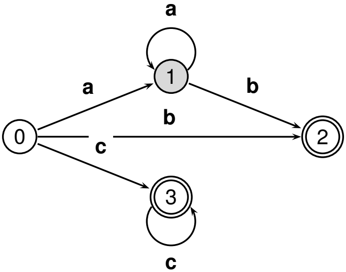
Behaviour of the driver routine



Input:**adbcc**↩
Output: **Token 1 found**

- Start once again in **state** = 0. **final** = -1.

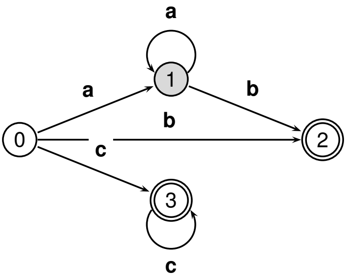
Behaviour of the driver routine



Input:adbcc↩
Output: **Token 1 found**

- **state** = 1, **final** = -1

Behaviour of the driver routine

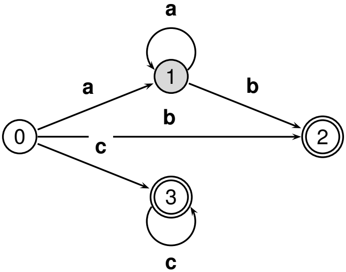


Input:**adbcc**⮐

Output: **Token 1 found**

- No transition from state 1 on **d**. Since **final** is -1 this is a error situation. Perform error action - remove the character **d**. Read the next symbol **b**

Behaviour of the driver routine

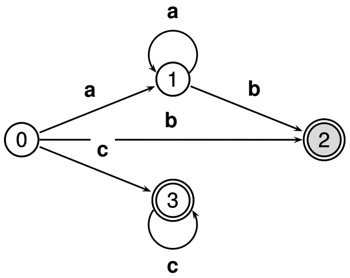


Input:**abcc**⟵

Output: **Token 1 found**

- **state** = 1, **final** = -1

Behaviour of the driver routine

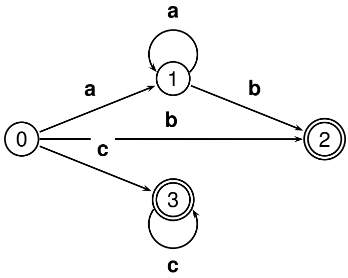


Input:**abcc**⟵

Output: **Token 1 found**

- **state** = 2, **final** = 2. Also mark **c.**

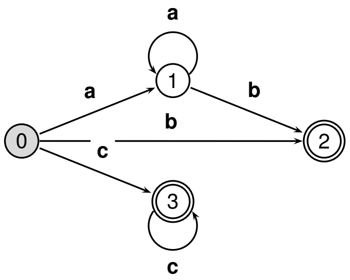
Behaviour of the driver routine



Input:**abcc**⟵
Output: **Token 1 found**
Token 1 found

- No transition from state 2 on **c**. Perform action corresponding to state 2. The lexeme detected is **ab**.

Behaviour of the driver routine



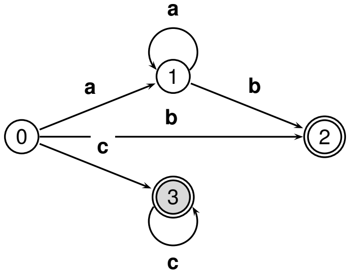
Input:**cc** ↩

Output: **Token 1 found**

Token 1 found

- Start once again in **state** = 0. **final** = -1

Behaviour of the driver routine



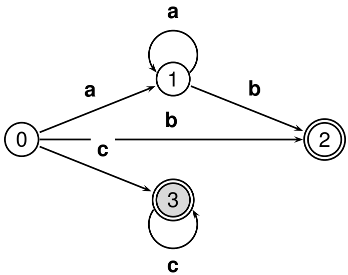
Input:cc ↩

Output: **Token 1 found**

Token 1 found

- **state** = 3, **final** = 3. Mark **c**

Behaviour of the driver routine



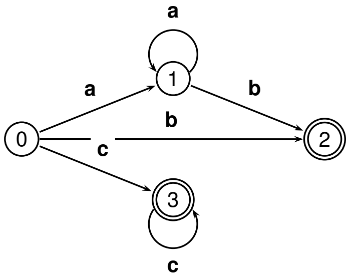
Input:**cc**←↩

Output: **Token 1 found**

Token 1 found

- **state** = 3, **final** = 3. Mark ←↩

Behaviour of the driver routine



Input:**cc** $\hookleftarrow$

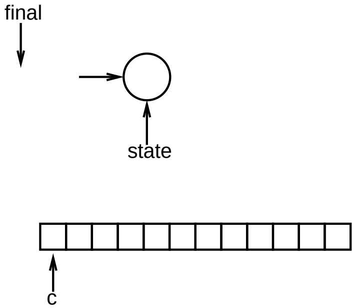
Output: **Token 1 found**

Token 1 found

Token 2 found

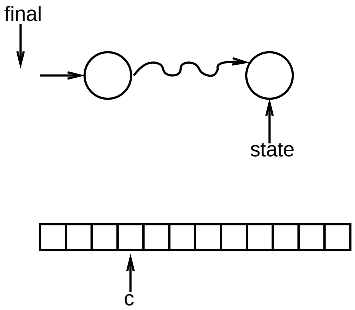
- No transition from state 3 on $\hookleftarrow$. Perform action corresponding to state 3. The lexeme detected is **cc**.

Summary of driver routine behaviour

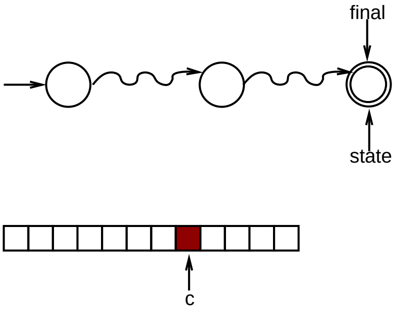


- **state** tracks the current state.
- **final** tracks the last final state traversed.
- **c** tracks the next input character.

Summary of driver routine behaviour

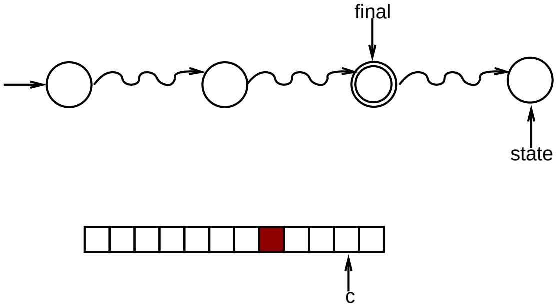


Summary of driver routine behaviour



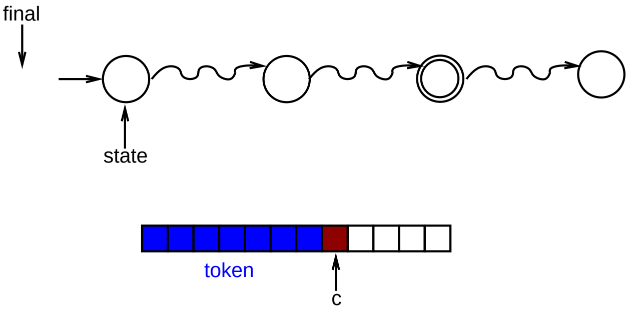
- When a final state is reached **final** is updated and the next input character is marked.

Summary of driver routine behaviour



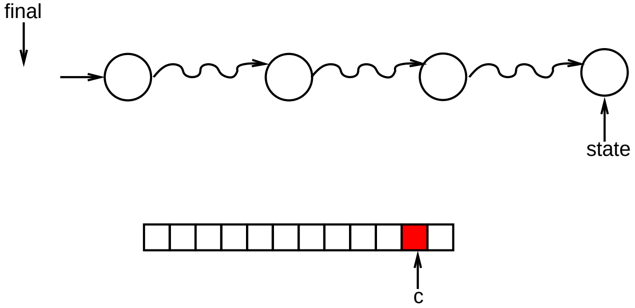
- Reaching a state with no transition on the next symbol implies the discovery of a token only if a final state has been traversed on the way.

Summary of driver routine behaviour



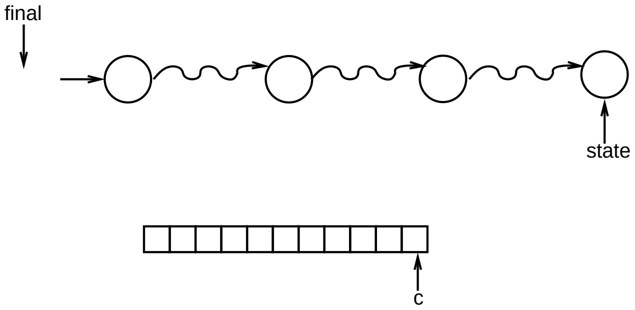
- The sequence of characters till the marked character is returned as the next token.

Summary of driver routine behaviour



- However if no final state has been encountered on the way to a blocked state, it signifies an error condition.

Summary of driver routine behaviour



- Characters are removed from the input till we can make a transition on the current state.

Driver + Action Routine + Error Routine

The Driver Routine

```
void yylex()
{ state = 0;
  if (isfinal(0)) final = 0;
  else final = -1;
  c = nextchar();
  while (true)
    if (valid(nextstate[state,c]))
      { state = nextstate[state,c];
        c = nextchar();
        if (isfinal(state))
          { final = state;
            markinput();
          }
      }
    elseif (isfinal(final))
      { retract (); ACTION
      }
    else error();
}
```

The Action Routine

```
#define ACTION
 switch(final)
  case 2:printf("Token 1 found");
         break;
  case 3:printf("Token 2 found");
         break;
```

The Error Routine

```
void error ()
{
  deletesymbol();
  c = nextchar()
}
```

Recap

In summary:

- The specification of a lexical analyser generator consists of two parts:
    1. Specification of tokens – done through regular expressions.
    2. Specification of actions - done through action routines.
- The lexical analyser generator:
    - Processes the regular expressions and forms a graph called DFA.
    - Copies the action routines without any change.
    - Adds a driver routine whose behaviour we described.

  These three things put together constitutes the lexical analyser.

Issues

- What are regular expressions? How can they be used to describe tokens?
- How can regular expresions be converted to DFA?

Introduction to Regular Expressions

A regular expressions denote a set of strings, also called *a language*. For example, **a*b** denotes the language $\{$**b, ab, aab, aaab, . . .** $\}$. We denote the language of a regular expression r as $L(r)$.

A single character is a regular expression.

- Examples: **a**, **Z**, **n**, **t**.
- Denotes a singleton set containing the character. **a** denotes the set $\{$**a**$\}$.

.

Introduction to Regular Expressions

ε is a regular expression.

- Denotes $\{\varepsilon\}$, the set containing the empty string.

.

Introduction to Regular Expressions

If r and s are regular expressions then $r|s$ is a regular expression.

- Examples: $\mathbf{a}|\mathbf{b}|\ \ldots|\mathbf{z}|\mathbf{A}|\mathbf{B}|\ldots|\mathbf{Z}$ and $\mathbf{0}|\mathbf{1}|\ldots|\mathbf{9}$. Let us call these regular expressions **LETTER** and **DIGIT**.
- $L(r|s)$ is the union of strings in $L(r)$ and $L(s)$.

.

Introduction to Regular Expressions

.

If r and s are regular expressions then rs is a regular expression.

- Examples: `begin` – with an assumed associativity.
- `{LETTER}({LETTER}|{DIGIT})*`.
    - Notice that the braces required around `LETTER` is a lex requirement and denotes that it is a synonym for a regular expression and not the literal `LETTER`.
- $L(rs)$ is the concatenation of strings x and y such that $x \in L(r)$ and $y \in L(s)$.

Introduction to Regular Expressions

.

If r is a regular expressions then r^* is a regular expression.

- Examples: `({LETTER}|{DIGIT})*`
- $L(r^*)$ is the concatenation of zero or more strings from $L(r)$. Concatenation of zero strings is defined to be the null string.

Introduction to Regular Expressions

.

If r is a regular expressions then (r) is a regular expression. Parentheses are used for grouping.

- Examples: `({LETTER}|{DIGIT})*`
- The language denoted by (r) is $L(r)$.

Introduction to Regular Expressions

.

Shorthand: If r is a regular expressions then r^+ is a regular expression.

- Examples: {DIGIT}+
- $L(r^+)$ is the concatenation of one or more strings from $L(r)$.
- $r^+ = rr^*$.

Introduction to Regular Expressions

.

Shorthand: If r is a regular expressions then $r?$ is a regular expression.

- Examples: $\{$`DIGIT`$\}$`?` denotes zero or one occurrence of a digit.
- $r?$ stands for zero or one occurrence of strings in r.
- $r? = \varepsilon | r$

Regular expressions provided by Lex

<table>
<tr><th>Expression</th><th>Describes</th><th>Example</th></tr>
<tr><td>c</td><td>any character c</td><td><code>a</code></td></tr>
<tr><td><code>\c</code></td><td>character c literally</td><td><code>*</code></td></tr>
<tr><td><code>"s"</code></td><td>string s literally</td><td><code>"**"</code></td></tr>
<tr><td><code>.</code></td><td>any character except newline</td><td><code>a.*b</code></td></tr>
<tr><td><code>^</code></td><td>beginning of a line</td><td><code>^abc</code></td></tr>
<tr><td><code>$</code></td><td>end of line</td><td><code>abc$</code></td></tr>
<tr><td><code>[s]</code></td><td>any character in s</td><td><code>[abc]</code></td></tr>
<tr><td><code>[^s]</code></td><td>any character not in s</td><td><code>[^abc]</code></td></tr>
<tr><td>$r*$</td><td>zero or more r’s</td><td><code>a*</code></td></tr>
<tr><td>$r+$</td><td>one or more r’s</td><td><code>a+</code></td></tr>
<tr><td>$r?$</td><td>zero or one r</td><td><code>a?</code></td></tr>
<tr><td>$r_1 r_2$</td><td>r_1 then r_2</td><td><code>ab</code></td></tr>
<tr><td>$r_1 \mid r_2$</td><td>r_1 or r_2</td><td><code>a|b</code></td></tr>
<tr><td>(r)</td><td>r</td><td><code>(a|b)</code></td></tr>
<tr><td>r_1 / r_2</td><td>r_1 when followed by r_2</td><td><code>abc/123</code></td></tr>
</table>

Example of token specification in Lex

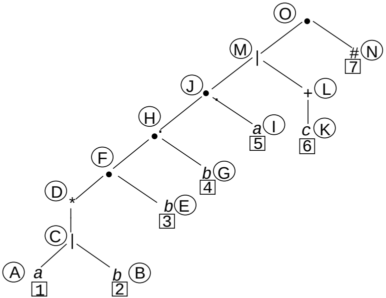
```
[ \t\n]+                        {/*no action, no return*/}
if                              {return(IF);}
then                            {return(THEN);}
else                            {return(ELSE);}
{letter}({letter}|{digit})*     {yylval=install_id(); return(ID);}

-?{digit}+(\.{digit}+)?(E[+\-]?{digit}+)?
                                {yylval=atof(yytext); return(NUM);}

"<"                             {yylval=LT; return(RELOP);}
"<="                            {yylval=LE; return(RELOP);}
"+"                             {yylval=PLUS; return(ADDOP);}
"*"                             {yylval=MULT; return(MULOP);}
```

Conversion of Regular Expressions to DFA

$((a|b)^* bba)|c^+$ expressed as a tree:



- Leaves labeled both with positions (1,2 etc.) and alphabets (A,B etc.).
- Interior nodes are labeled with alphabetic labels only.
- • represents concatenation.
- # signifies end of a token.

Conversion of Regular Expressions to DFA

Key idea: Identify a state with a set of positions in the regular expression.

- The starting state of the DFA for $(a|b)^* bba|c^+$ could expect a a (from position 1 of the corresponding tree), a b (from position 2 or 3) or a c (from position 6).
- Initial state consists of the set of positions 1,2,3 and 6.
- These are the positions from where the first symbol a, b or c could be generated.

1236

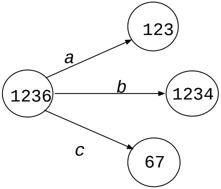
Conversion of Regular Expressions to DFA

- If the first symbol is a *b* coming from position 1,
  - then the next symbol could be a *a* coming from position 1 or a *b* coming from position 2 or a *b* coming from 3.
- If the first symbol is a *b* coming from position 3,
  - the next symbol can only be a *b* coming from position 4.

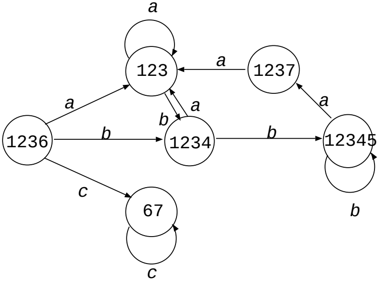
if we are in	then on symbol	we could go to
position 1	a	1,2,3
position 2	b	1,2,3
position 3	b	4
position 6	c	6,7

Conversion of Regular Expressions to DFA

Identifying groups of positions with states, we get:



We can ask the same question for each of the new states. Completing the diagram we get:



Conversion of Regular Expressions to DFA

Formalization:

- Define a function *followpos* which takes a position as an argument and returns a set of positions as result.
- *followpos*(i) answers the following question:
  - Suppose a string takes us to the position i of the syntax tree. Then, on seeing the symbol associated with this position, what are the positions from which the next symbol could come?

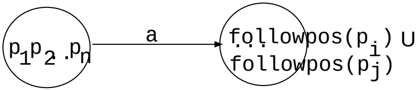
We can write the earlier table in terms of *followpos* as,

if we are in	then on symbol	we could go to
position 1	a	followpos(1)
position 2	b	followpos(2)
position 3	b	followpos(3)
position 6	c	followpos(6)

Conversion of Regular Expressions to DFA

- Assume that the current state consists of positions $p_1, \ldots, p_n$.
- Of these $p_i, \ldots, p_j$ are the only positions associated with a certain symbol a.

Then the next state on symbol a can be described in terms of *followpos* as:



Conversion of Regular Expressions to DFA

To find *followpos*, we have to first define:

1. *nullable*(n): *nullable*(n) is true if the subexpression represented by the node n can generate a null string. As examples:
   - *nullable*(D) = true
   - *nullable*(F) = false
2. *firstpos*(n): *firstpos* of a node n is the set of positions from which the first symbol of some string derivable from the subexpression represented by n could come.
   - *firstpos*(D) = $\{1, 2\}$
   - *firstpos*(F) = $\{1, 2, 3\}$

   This is because the first symbol a string derivable from D could either be a *a* coming from 1, or a *b* coming from 2. Similarly, the first symbol of a string derivable from node F could either come from 1 (*a*), 2 (*b*), or from 3 (*b*).

Conversion of Regular Expressions to DFA

3. *lastpos*(n): *lastpos* is a function from a node to a set of positions. *lastpos* of a node n is the set of positions from which the last symbol of some string derivable from the subexpression represented by n could come.
   lastpos(D) = $\{1,2\}$
   lastpos(F) = $\{3\}$

Conversion of Regular Expressions to DFA

Rules for constructing the functions *nullable*, *firstpos* and *lastpos*.

node n	*nullable*(n)
n is a leaf labeled ε	true
n is a leaf	false
n is $c_1 \mid c_2$	*nullable*(c_1) or *nullable*(c_2)
n is $c_1 \bullet c_2$	*nullable*(c_1) and *nullable*(c_2)
n is c^*	true
n is c^+	nullable(c)

node n	*firstpos*(n)
n is a leaf labeled ε	ϕ
n is a leaf at position i	$\{i\}$
n is $c_1 \mid c_2$	*firstpos*(c_1) $\bigcup$ *firstpos*(c_2)
n is $c_1 \bullet c_2$	if *nullable*(c_1) then *firstpos*(c_1) $\bigcup$ *firstpos*(c_2) else *firstpos*(c_1)
n is c^*	*firstpos*(c)
n is c^+	*firstpos*(c)

Conversion of Regular Expressions to DFA

node n	$lastpos(n)$
n is a leaf labeled ε	ϕ
n is a leaf at position i	$\{i\}$
n is $c_1 \mid c_2$	$lastpos(c_1) \bigcup lastpos(c_2)$
n is $c_1 \bullet c_2$	if $nullable(c_2)$ then $lastpos(c_1) \bigcup lastpos(c_2)$ else $lastpos(c_2)$
n is c^*	$lastpos(c)$
n is c^+	$lastpos(c)$

Conversion of Regular Expressions to DFA

Given *nullable*,*firstpos* and *lastpos*, *followpos* can be found out by repeated application of the three rules given below.

1. $c_1 \bullet c_2$: If i is a position in *lastpos*(c_1), then everything in *firstpos*(c_2) is in followpos(i).
2. c^*: If i is a position in *lastpos*(c), then every position in *firstpos*(c) is in *followpos*(i).
3. c^+: If i is a position in *lastpos*(c), then every position in *firstpos*(c) is in *followpos*(i).

Conversion of Regular Expressions to DFA

For the example, the functions *nullable*,*firstpos*, *lastpos* and *followpos*. Notice that *followpos* is defined only for positions (leaf nodes).

node	*firstpos*	*lastpos*	*followpos*
A	{1}	{1}	{1,2,3}
B	{2}	{2}	{1,2,3}
C	{1,2}	{1,2}	
D	{1,2}	{1,2}	
E	{3}	{3}	{4}
F	{1,2,3}	{3}	
G	{4}	{4}	{5}
H	{1,2,3}	{4}	
I	{5}	{5}	{7}
J	{1,2,3}	{5}	
K	{6}	{6}	{6,7}
L	{6}	{6}	
M	{1,2,3,6}	{5,6}	
N	{7}	{7}	
O	{1,2,3,6}	{7}	

Conversion of Regular Expressions to DFA – Algorithm

1. Construct the tree for *r#*.
2. Construct functions *nullable*, *firstpos*, *lastpos* and *followpos*.
3. Let *firstpos*(*root*) be the first state. Push it on top of stack;

   while stack not empty
   do begin
       pop the top state U off the stack; mark U;
       for each input symbol a do
         begin
           let $p_1, \ldots, p_k$ be the positions in U
           corresponding to the symbol a;
           let $V = \mathit{followpos}(p_1) \bigcup \ldots \bigcup \mathit{followpos}(p_k)$;
           put V in 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 position corresponding to #.

Features of Lex/Flex

1. Starting from a input position, detect the longest lexeme that could match a pattern (maximal munch).

   Example: Return `begin` and not `b`, `be`, `beg` . . . .

   Where has this decision been incorporated in our description of the generated lexical analyser?
2. If a lexeme matches more than one patterns, declare the lexeme to have matched the earliest pattern.

   Example: The state numbered 5 corresponds to the pattern `begin` and not `identifier`.

Lexical Errors

Primarily of two kinds:

1. Lexemes whose length exceed the bound specified by the language.
   - In Fortran, an identifier more than 31 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 `@` occuring in a Pascal program (but not within a string or a comment) are lexical errors.
3. Unterminated strings or comments.

Handling Lexical Errors

The action taken on detection of an error are:

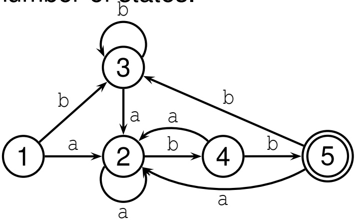
1. Issue an appropriate error message.
2. Error of the first type—
   - Truncate identifiers to the specified length.
   - Read the numeric constant and pass to the parser. The parser has the type information and can genrate a warning/error.
3. Error of the second type—
   - Skip illegal character—this is what was discussed earlier. What does `flex++` do?
   - Pass the character to the parser which has better knowledge of the context in which error has occurred.
4. Error of the third type—wait till end of file and issue error message.

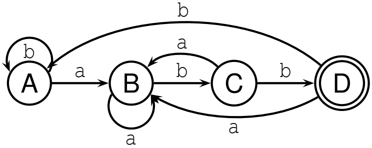
See tokens generated by a real compiler (clang):

`clang -fsyntax-only -Xclang -dump-tokens filename.c`

Minimizing the Number of States

- The DFA constructed for $(b|\varepsilon)(a|b)^* abb$.
- There is another DFA for the same regular expression with lesser number of states.





Minimizing the Number of States

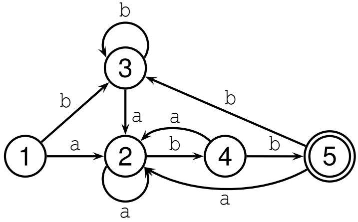
- For a typical language, the number of states of the DFA is in order of hundreds.
- Therefore we should try to minimize the number of states.

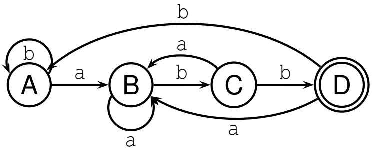
However, lex/flex does not! See discussion in:

`https://stackoverflow.com/questions/33393831/flex-lex-analyzer-generator-dfa-minimization`

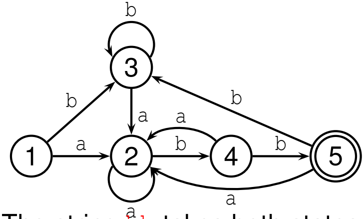
Minimizing the Number of States

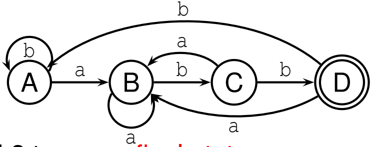
- The second DFA has been obtained by merging states 1 and 3 of the first DFA.
- Under what conditions can this merging take place?





Minimizing the Number of States



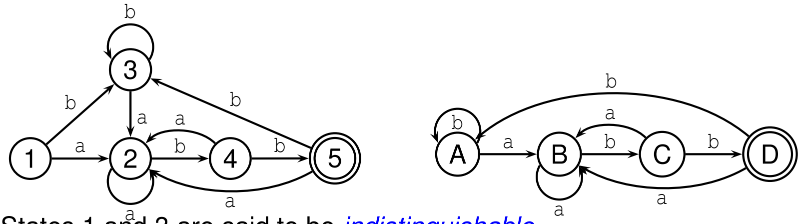


- The string bb takes both states 1 and 3 to a non-final state.
- The string aba takes both states 1 and 3 to a non-final state.
- The string ε takes both states 1 and 3 to a non-final state.
- The string bbabb takes both states 1 and 3 to a final state.

Observation:

Any string that takes state 1 to a final state also takes 3 to the same final state. Conversely, any string that takes state 1 to a non-final state also takes 3 to the same non-final state.

Minimizing the Number of States



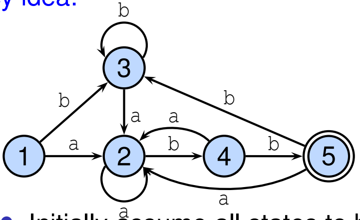
- States 1 and 3 are said to be *indistinguishable*.
- Minimimization strategy:
  - Find indistinguishable states.
  - Merge them.
- Question: How does one find indistingushable states?

Minimizing the Number of States

Key idea:

Minimizing the Number of States

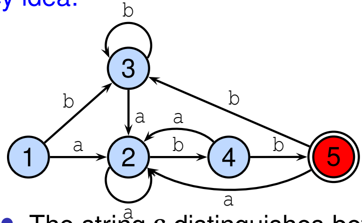
Key idea:



- Initially assume all states to be indistinguishable. Put them in a single set.

Minimizing the Number of States

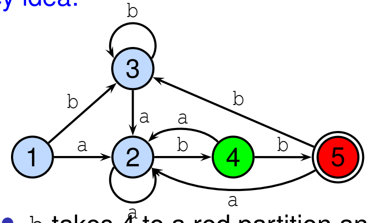
Key idea:



- The string ε distinguishes between final states and non-final states. Create two partitions.

Minimizing the Number of States

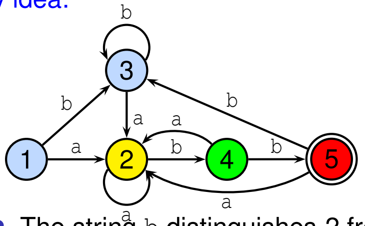
Key idea:



- b takes 4 to a red partition and retains other blue states in blue partition. Put 4 in a separate partition.

Minimizing the Number of States

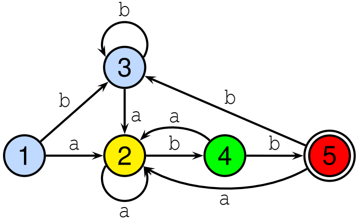
Key idea:

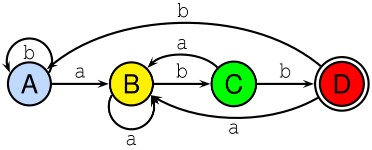


- The string `b` distinguishes 2 from other states in the blue partition.

Minimizing the Number of States

Key idea:





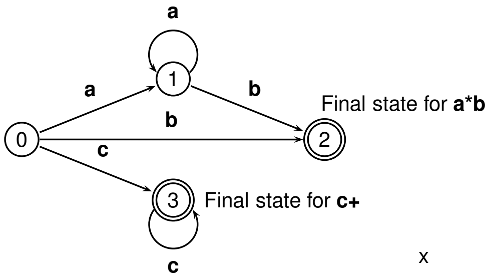
- No other partition possible. Merge all states in the same partition.

Summary of the Method

1. Construct an initial partition $\pi = \{S - F, F_1, \ldots, F_n,\}$, where $F = F_1 \cup F_2 \cup \ldots F_n s$, and each F_i is the set of final states for some token i.
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, states s and t have transitions
   onto states in the same set of π;
   replace G in π_{new} by the set of all subsets formed
3. If $\pi_{new} = \pi$, let $\pi_{final} := \pi$ and continue with step 4. Otherwise repeat step 2 with $\pi := \pi_{new}$.
4. Merge states in the same set of the partition.
5. Remove any dead states.

Efficient Representation of DFA

A naive method to represent a DFA uses a two dimensional array.



x

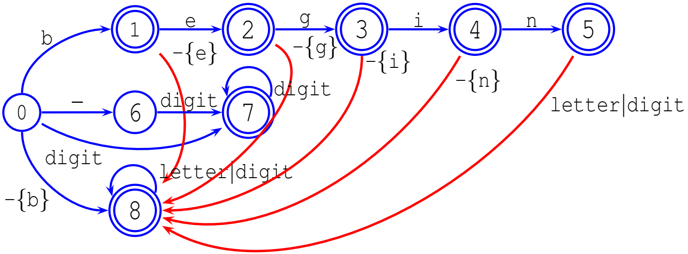
	a	b	c
0	1	2	3
2	1	2	_
2	_	_	_
3	_	_	3

- For a typical language:
    - the number of DFA states is in the order of hundreds (sometimes 1000),
    - the number of input symbols is greater than 100.
- It is desirable to find a space-efficient representation of the DFA.

An Example Language

Consider a language with the following tokens:

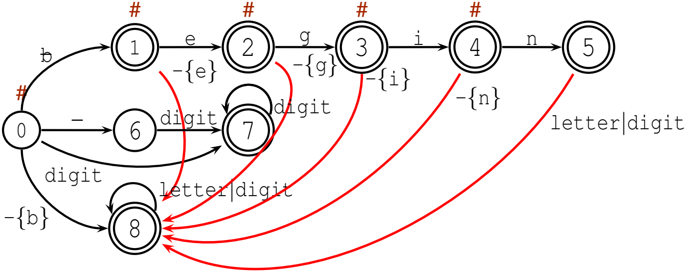
- *begin* – representing the lexeme `begin`
- *integer* – Examples: `0`, `-5`, `250`
- *identifier* – Examples: `a`, `a1`, `max`



$-\{x\}$ is a shorthand for $\texttt{letter}-\{x\}$.

The Four Arrays Scheme

Key Observation For a DFA that we have seen earlier, the states marked with # behave like state 8 on all symbols *except for a few symbols*.

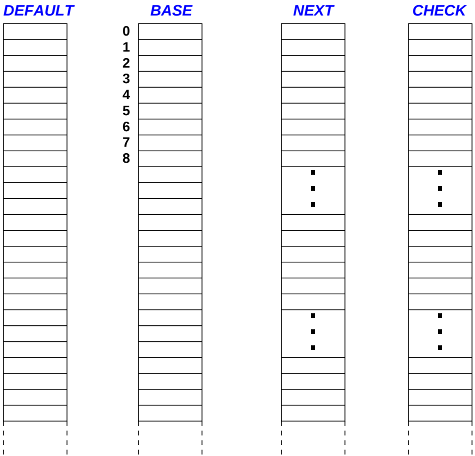


Therefore information about state 8 can also be used for these states.

Four Arrays Representation of DFA

Symbols and their numbering

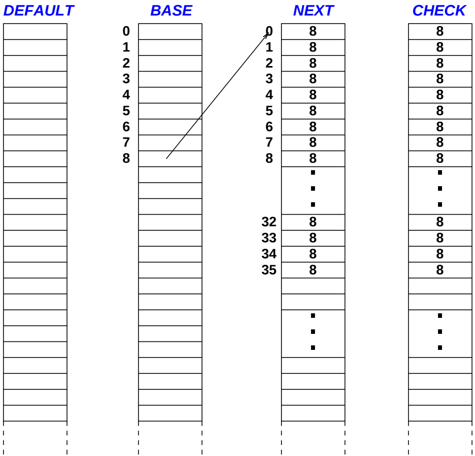
`a–z`	0–25
`0–9`	26–35
`-`	36



Four Arrays Representation of DFA

Symbols and their numbering

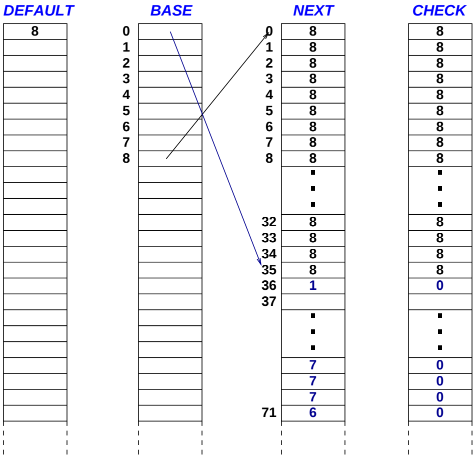
`a–z`	0–25
`0–9`	26–35
`-`	36



Four Arrays Representation of DFA

Symbols and their numbering

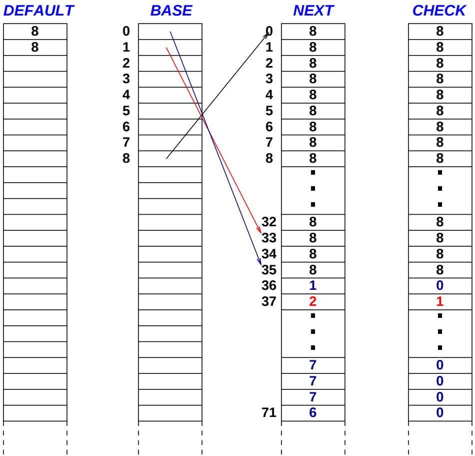
`a–z`	0–25
`0–9`	26–35
`_`	36



Four Arrays Representation of DFA

Symbols and their numbering

`a–z`	0–25
`0–9`	26–35
`-`	36



Four Arrays Representation of DFA

If s is a state and a is the numeric representation of a symbol, then

1. $BASE[s]$ gives the base location for the information stored about state s.
2. $NEXT[BASE[s]+a]$ gives the next state for s and symbol a, only if $CHECK[BASE[s]+a] = s$.
3. If $CHECK[BASE[s]+a] \neq s$, then the next state information is associated with $DEFAULT[s]$.

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

Four Arrays Representation of DFA

- All the entries for state 8 have been stored in the array *NEXT*. The *CHECK* array shows that the entries are valid for state 8.
- State 1 has a transition on e(4) which must be stored explicitly. This differing entry is stored in *NEXT*[37]. Therefore *BASE*[1] is set to $37 - 4 = 33$.
- To find *nextstate*$[1, 0]$, we first refer to *NEXT*$[33 + 0]$, But since *CHECK*$[33 + 0]$ is not 1 we have to refer to *DEFAULT*[1] which is 8. So the correct next state is found from $NEXT[BASE[8] + 0] = 8$.
- To fill up the four arrays, we have to use a heuristic method. One possibility, which works well in practice, is to find for a given state, the lowest *BASE*, so that the special entries can be filled without conflicting with existing entries.

A Heuristic for Filling The Arrays

1. Order states by the number of transitions requiring explicit filling. 8 (36), 0(12), 7(10), 1(1), 2(1), 3(1), 4(1), 6(0), 5(0).
2. For each state s in this order do:
   2.1 Let i and j be the indices of the lowest and highest numbered symbols for which information has to be stored explicitly.
   2.2 Find the earliest contiguous gap $[g, h]$ of length $j - i + 1$.
   2.3 Set $BASE[s] = g - i$. Fill $NEXT[k]$ for all symbols indices k for which explicit information has to be stored. Also set $CHECK[k]$ to s.
   There may be gaps in $[g, h]$.
   2.4 If s has no transitions requiring explicit filling, then it is exactly mimicked by some default state t. Set $BASE[s] = BASE[t]$.
   2.5 If the default state of s is t, set $DEFAULT[s] = t$.

Flex Employs the 4-Array Scheme

```
while ( yy_chk[yy_base[yy_current_state] + yy_c] !=
        yy_current_state )
{
  yy_current_state = (int) yy_def[yy_current_state];
  ...
}
yy_current_state = yy_nxt[yy_base[yy_current_state] +
                          yy_c];
```