LEXICAL ANALYSIS

TEACHING MATERIAL

MOTIVATION

The compiler sees an input program as a string of characters *without any structure.* **The program below**

```
var I,J: integer;
begin
     I := I + 1;
     J := J + 1
end.
```

is seen as

var$\boxed{\texttt{b}}$I,J:$\boxed{\texttt{b}}$integer;$\boxed{\texttt{CR}}$ begin$\boxed{\texttt{CR}}$ $\boxed{\texttt{b}}\boxed{\texttt{b}}\boxed{\texttt{b}}\boxed{\texttt{b}}\boxed{\texttt{b}}$I
$\boxed{\texttt{b}}$:=$\boxed{\texttt{b}}$I$\boxed{\texttt{b}}$+$\boxed{\texttt{b}}$1;$\boxed{\texttt{b}}\boxed{\texttt{b}}\boxed{\texttt{b}}\boxed{\texttt{b}}\boxed{\texttt{b}}J\boxed{\texttt{b}}$:=$\boxed{\texttt{b}}J\boxed{\texttt{b}}$+$\boxed{\texttt{b}}1\boxed{\texttt{CR}}$end.

Here $\boxed{\texttt{b}}$ **and** $\boxed{\texttt{CR}}$ **represent the blank and newline characters.**

The first task of the compiler is to identify the structure of the program.

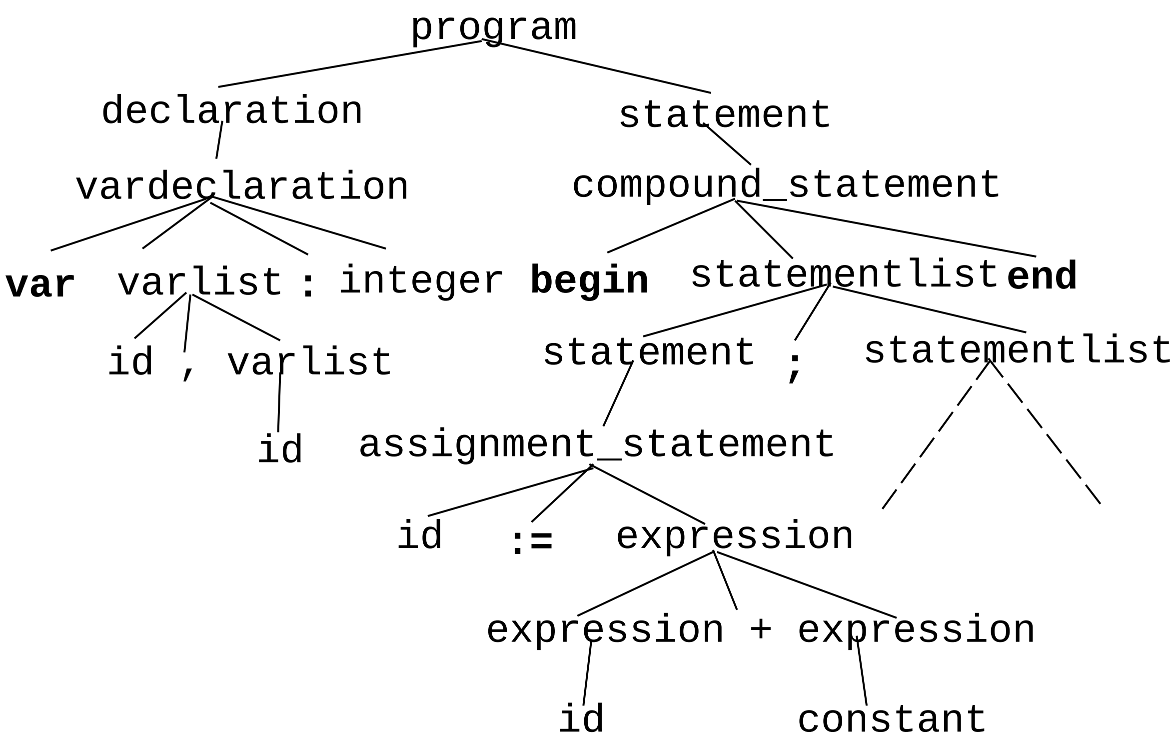
MOTIVATION

To impose structure on the input:

1. **Break up this string into 'words'–the smallest logical units. This is *lexical analysis* or *scanning*.**

`var`	`I`	`,`	`J`	`:`	`integer`	`;`	`begin`	`I`	`:=`	`I`	`+`	`1`	`;`	`J`	`:=`	`J`	`+`	`1`	`end`	`.`

2. **Discover the larger structures in the program. This is *syntax analysis* or *parsing*.**



Note that during parsing we do not consider any entity smaller than a word.

BASIC CONCEPTS AND ISSUES

DEFINITION: ***Lexical analysis*** **is the operation of reading the input program, and breaking it up into a sequence of** *lexemes* **(***tokens***).**

We distinguish between

- *lexemes* – **smallest logical units (words) of a program, such as** `A, B, 1.0, true, +, <=`.
- *tokens* – **classes of similar lexemes, such as** *identifier, constant, operator*
- *pattern* – **an informal or formal description of a token, such as** *an identifier is a string of at most 8 characters, in which the first character is an alphabet, and the successive characters are either digits or alphabets.*

A pattern is used to

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

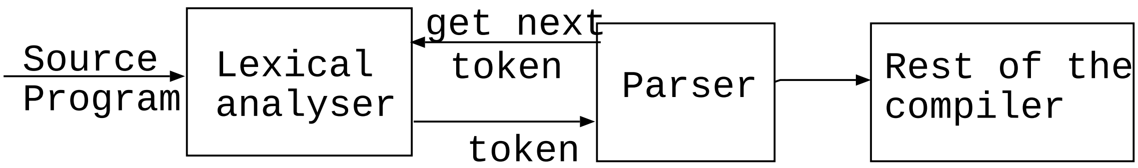
BASIC CONCEPTS AND ISSUES

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. 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.**
- **Thus lexical analysis and parsing are not different passes of a compiler. They typically belong to the same pass.**

EXAMPLE



This is the usual organization for compilers with a single pass front-end.

BASIC CONCEPTS AND ISSUES

TYPES OF TOKENS

What are the different kinds of tokens?

```
function gcd (m, n:  integer):  integer;
begin
    if n = 0 then gcd := m
    else gcd := gcd(n,m mod n)
end; (* of gcd*)
```

tokens	lexemes
keyword	`function, begin, if, then, else, end`
identifier	`gcd, m, n, mod, integer`
constant	`0`
delimiter	`(, :, ), ;`
relop	`=`
assignop	`:=`
comment	`(*of gcd*)`

It is convenient to treat each keyword, each operator and each delimiter as a separate token. We shall, henceforth, assume such a scheme.

BASIC CONCEPTS AND ISSUES

TOKENS HAVE ATTRIBUTES

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

EXAMPLE

lexeme	<token, token attribute>
`3`	< const, `3`>
`A`	<identifier, `A`>
`if`	<if, –>
`:=`	<assignop, –>
`>`	<gt, –>
`;`	<semicolon, –>

BASIC CONCEPTS AND ISSUES

As an example, here is a description of tokens in an actual programming language Java.

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

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

2. Constants:

   (a) Integer Constants

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

   (b) Floating point constants

   - float - ends with `f`
   - double

   (c) Boolean constants – `true` and `false`

   (d) Character constants – `'a'`, `'\u0034'`, `'\t'`

   (e) String constants – `""`, `"\""`, `"A string"`.

   (f) Null constant – `null`.

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

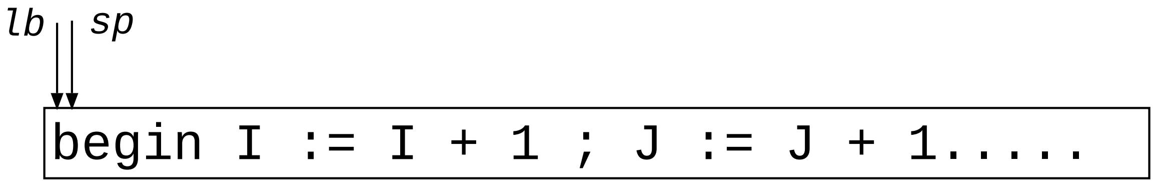
4. Operators: `=`, `>`, `<` ... `>>>=`

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

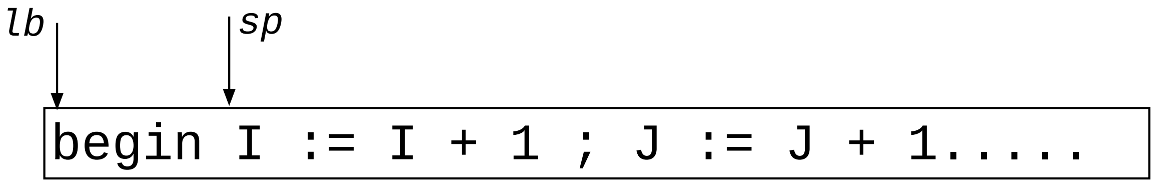
BASIC CONCEPTS AND ISSUES

SCANNING THE INPUT

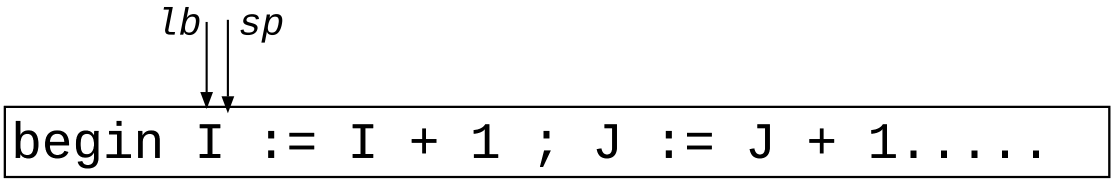
The lexical analyser uses two pointers (*lb:* lexeme_beg ining and *sp:* search_pointer) to keep track of the portion of the input string scanned.



Initially both pointers point to the beginning of a lexeme. *sp* then starts scanning forward to search for the end of the lexeme. The end of the lexeme, in this case, is indicated by the blank space after `begin`.



When the end of the lexeme is identified, the token and the attribute corresponding to this lexeme is returned. *lb* and *sp* are then made to point to the beginning of the next token.

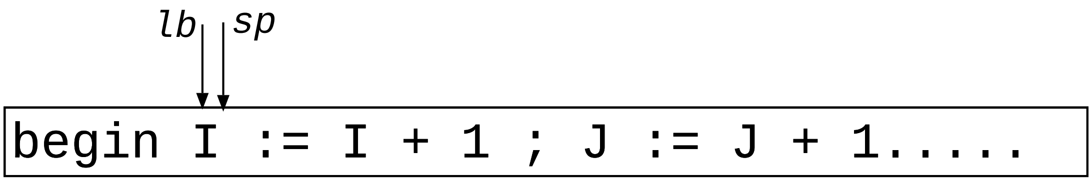


BASIC CONCEPTS AND ISSUES

BUFFERING SCHEMES

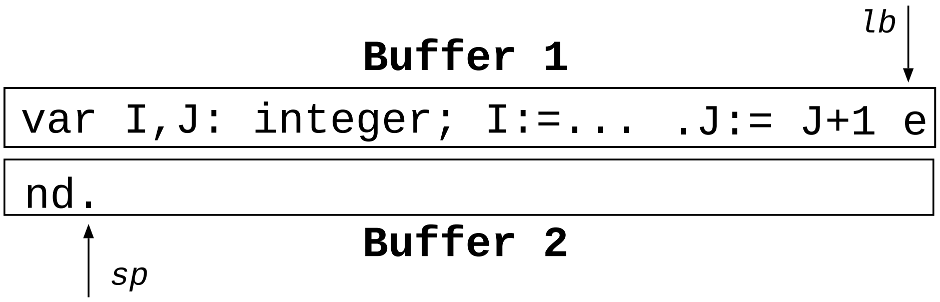
Reading the input character by character from the secondary storage is costly. A block of data is first read into a buffer, and then scanned by the lexical analyser. Buffering methods commonly used are:

1. One buffer scheme.



   There are problems if a lexeme crosses the buffer boundary. To scan the rest of the lexeme, the buffer has to be refilled, thereby overwriting the first part of the lexeme.

2. Two buffer scheme.



   Here buffers 1 and 2 are scanned alternately. When one reaches the end of the current buffer, the other buffer is filled. Clearly the problem in the previous scheme is solved if no lexeme is longer than the length of a buffer.

BASIC CONCEPTS AND ISSUES

DESIGN OF LEXICAL ANALYSERS

There are two main approaches:

1. *Hand code the lexical analyser:* **This involves actually writing a program to do lexical analysis.**

2. *Use a lexical analyser generator:* **This automatically generates the lexical analysis program from a formal description of the tokens of the language and their attributes.**

The lexical analyser generated by the first method is possibly more efficient, but the generation process is faster and is less prone to errors in the second case.

HANDCODING A LEXICAL ANALYSER

Consider a language with the following tokens:

begin – representing the lexeme `begin`

end – representing the lexeme `end`

integer – Examples: `0`, `-5`, `250`

identifier – Examples: `a`, `A1`, `max`

TRANSITION DIAGRAMS

Corresponding to each token, we draw a *transition diagram.* A transition diagram is a directed graph with nodes representing states, and edges representing transitions on input symbols. A state is a representation of the portion of input seen so far. For each transition diagram, there is a *start state* signifying anticipation of the corresponding token, and a *final state* signifying the end of the token.

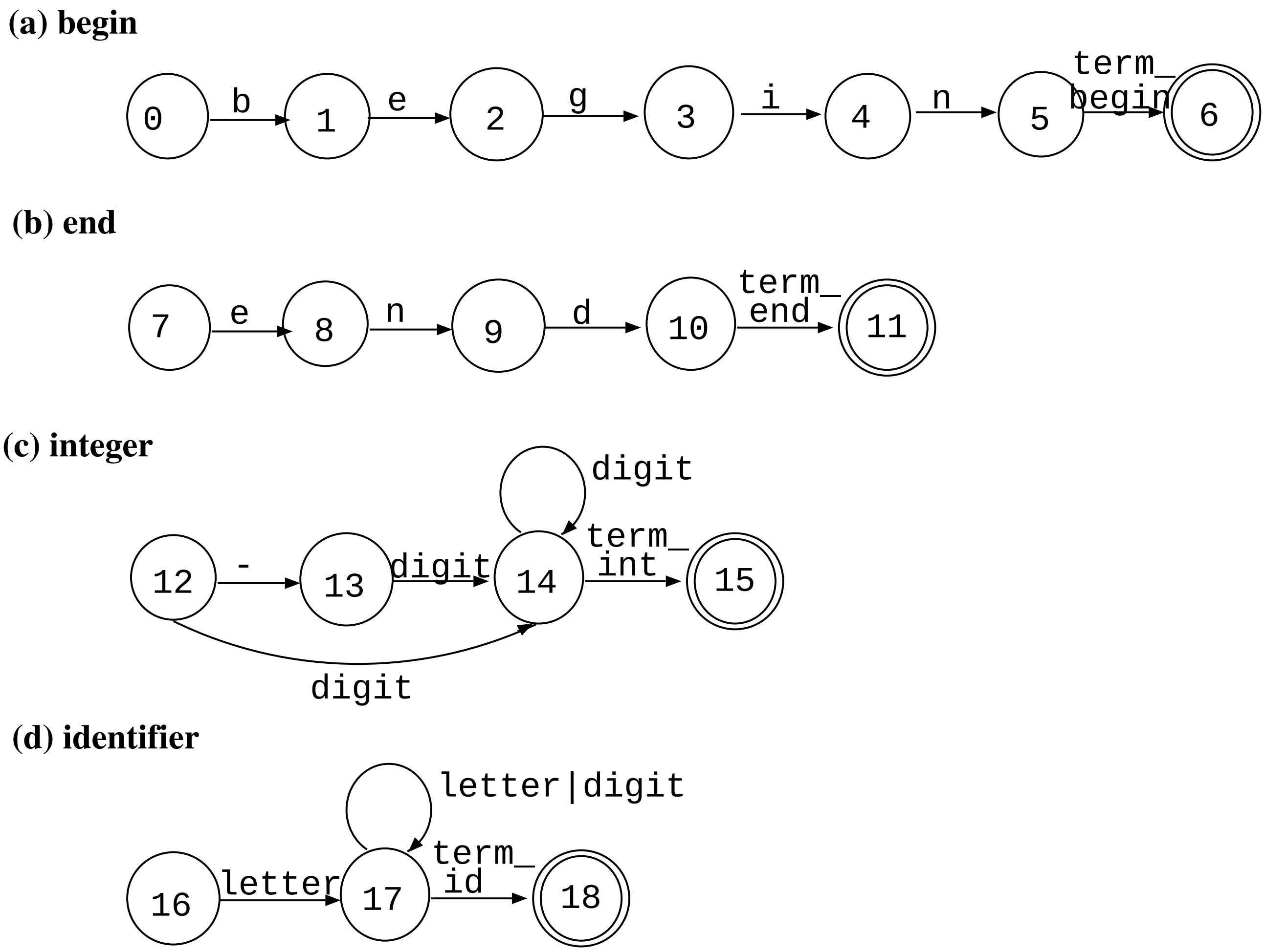
Transition diagrams are useful in

- **precisely specifying tokens, and**
- **structuring the lexical analyser program.**

HANDCODING A LEXICAL ANALYSER

TRANSITION DIAGRAMS

Transition diagram for



We shall explain the transitions on term_begin, term_end etc. later.

HANDCODING A LEXICAL ANALYSER

CONVERTING TRANSITION DIAGRAMS INTO CODE

The function *nexttoken* (next slide) returns the next token and its attribute. It tries the transition diagrams successively starting with the diagram beginning in state 0. On failing to traverse the current transition diagram, it backtracks to the beginning of the lexeme and tries the next diagram specified by the function *fail*. The following functions are used in the code fragment.

1. *error*: an unspecified error routine.
2. *getnonblank*: returns the next non-blank character.
3. *nextchar*: returns the next character.
4. *letter*(c)/*digit*(c): checks whether c is a letter/digit.

```
function fail;
  begin
    sp := lb;
    case start of
      0: start := 7; return := 7;
      7: start := 12; return := 12;
      12: start := 16; return := 16;
      other: error
    end
  end;
```

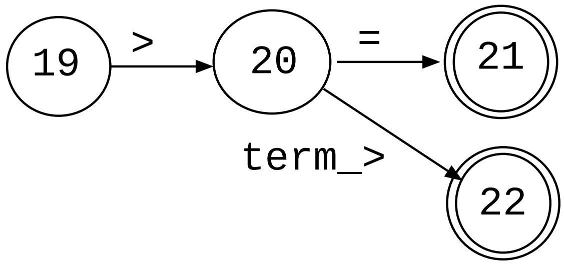
HANDCODING A LEXICAL ANALYSER

```
function nexttoken;
  begin
    state := 0; start := 0;
    while true do
      case state of
        0: getnonblank; lb := sp;
           if c = 'b' then state := 1 else state = fail;
        ...
        5: c:= nextchar;
           if letter(c) or digit(c) then state := fail else state :=6;
        6: sp := sp -1; return(<begin, ->)
        ...
        16: c := nextchar;
           if letter(c) then state := 17 else state := fail;
        17: c := nextchar;
           if letter(c) or digit(c) then state := 17 else state := 18;
        18: sp := sp - 1; return (<identifier, input[lb,sp]>)
      end /*case*/
  end; /*nexttoken*/
```

HANDCODING A LEXICAL ANALYSER

EXERCISE

Add the token relop consisting of the lexemes > and >=



and try to scan the example input string shown below:

begin$\boxed{\texttt{b}}\boxed{\texttt{b}}$amax$\boxed{\texttt{CR}}\boxed{\texttt{b}}\boxed{\texttt{b}}\boxed{\texttt{b}}I\boxed{\texttt{b}}$>=$\boxed{\texttt{b}}I\boxed{\texttt{b}}$>$\boxed{\texttt{b}}1\boxed{\texttt{CR}}\boxed{\texttt{b}}\boxed{\texttt{b}}\boxed{\texttt{b}}\boxed{\texttt{b}}J\boxed{\texttt{b}}$>=$\boxed{\texttt{b}}$
J$\boxed{\texttt{b}}$>$\boxed{\texttt{b}}$1$\boxed{\texttt{CR}}$end

NOTE:

1. **Some tokens like begin and end have fixed lengths, whereas others, like those corresponding to numbers, have variable lengths.**

2. **Associated with each token, is a set of characters called its terminator. For example:**

term_num	**any character other than a digit.**
term_id	**any character other than a letter or a digit.**
term_begin	**any character other than a letter or a digit.**
term_>	**any character other than =**

3. **A lookahead is necessary for a token, only if it is a prefix of a larger token.**

HANDCODING A LEXICAL ANALYSER

4. **Between identifiers and keywords, the transition diagrams for keywords must be tried first, failing which the transition diagram for the identifier must be tried.**

5. **Keywords cause a proliferation in the number of states (and hence increases the code size). To prevent this:**

   (i) **Enter all keywords in a keyword table.**

   (ii) **Detect a keyword as an identifier. If the identifier is found in the keyword table, declare it as a keyword, Otherwise it is actually an identifier.**

EXERCISE

Suppose you wanted to extend the function *nexttoken* **by adding a token for reals. How would you do it?**

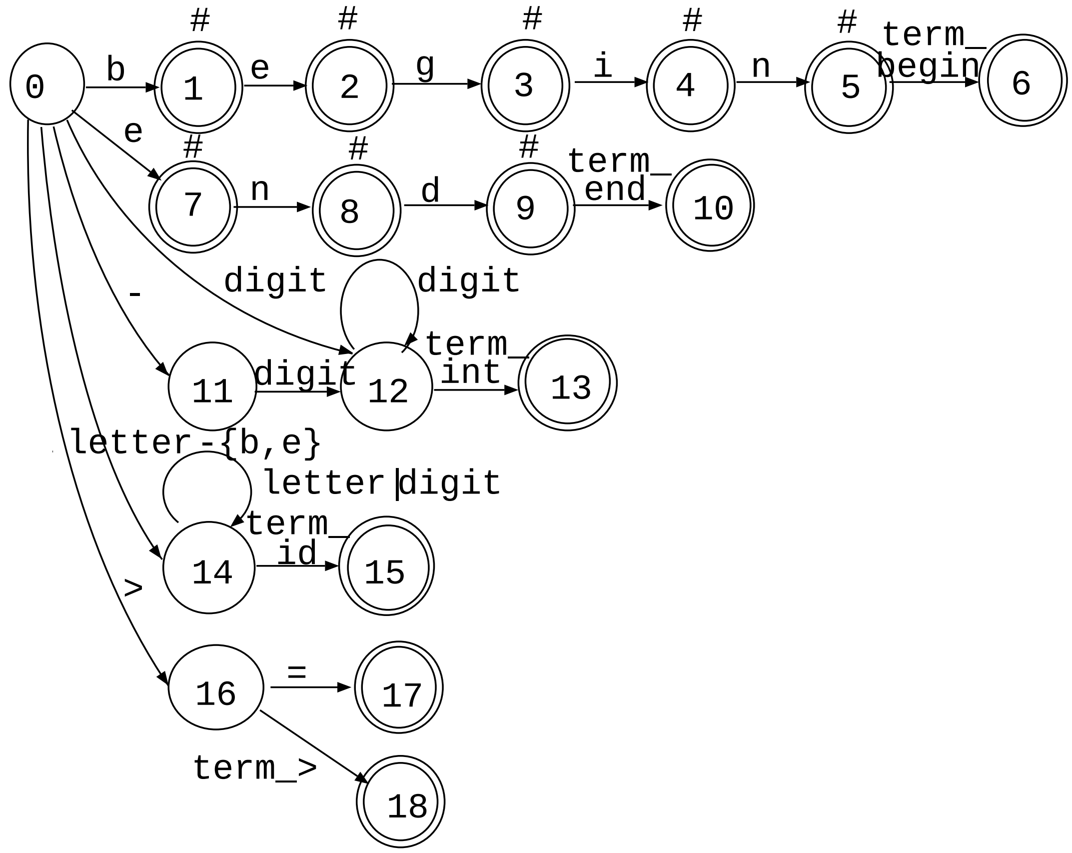
HANDCODING A LEXICAL ANALYSER

COMBINING TRANSITION DIAGRAMS

Transition diagrams can be combined to yield two benefits:

(i) The resultant transition diagram is of smaller size.

(ii) No backtracking.



NOTE: There is a transition from each state marked # to state 14.

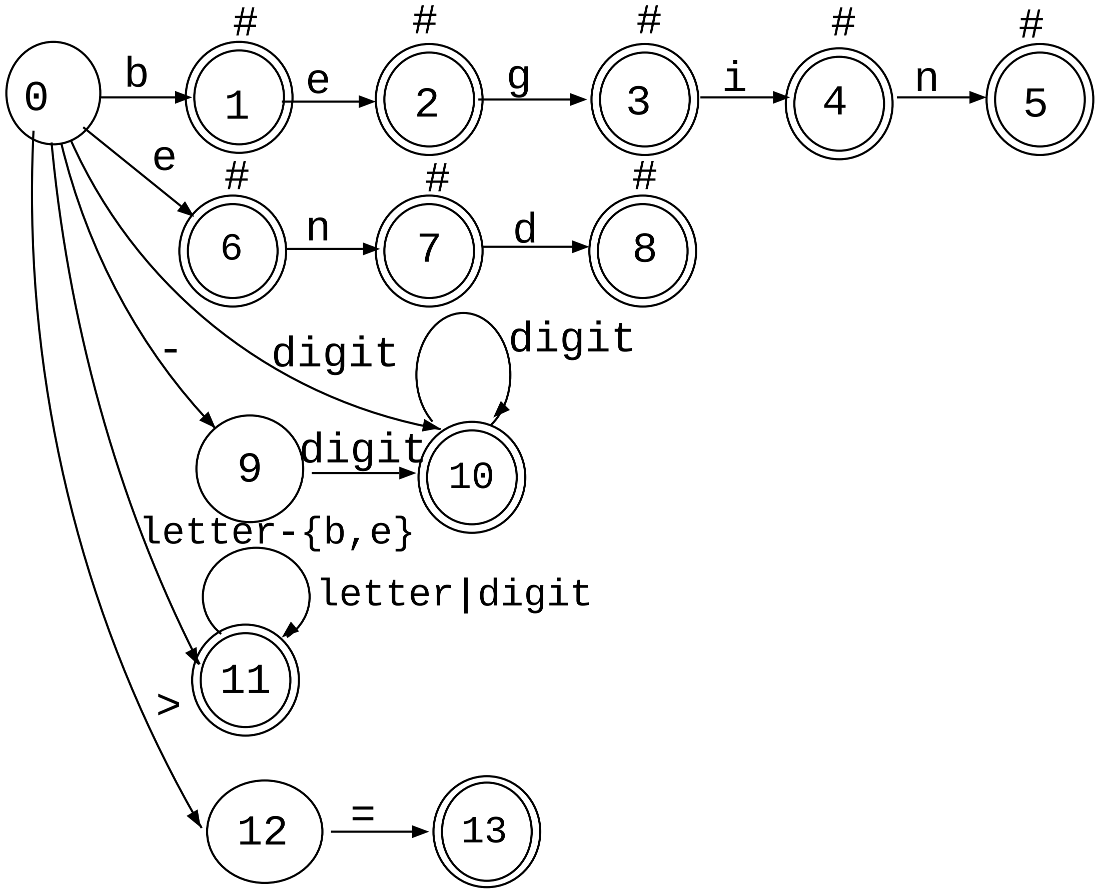
A deterministic transition diagram with a single starting state as shown above is also called a DFA.

HANDCODING A LEXICAL ANALYSER

The transition on terminators can also be eliminated as follows:

(i) Starting at state 0, traverse the transition diagram on the input string as long as possible.

(ii) If the last state on which there is no transition on the next input character is a final state, then return the token corresponding to the final state, else announce an error.



There is a transition from each state marked # to state 11.

NOTE: This works if the end of a token can be detected by examining *a single extra character (a lookahead of one).* **Are there examples of tokens which require more than one lookaheads?**

AUTOMATIC GENERATION OF LEXICAL ANALYSERS

To generate a lexical analyser, we need the following:

- **a** *linguistic* **specification of tokens.**
- **a specification of the action to be performed, on identifying each token.**

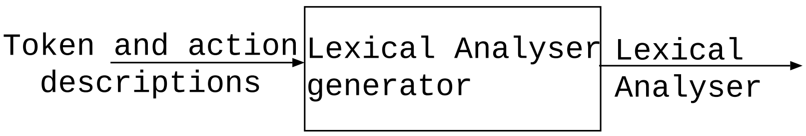
The generator program consists of:

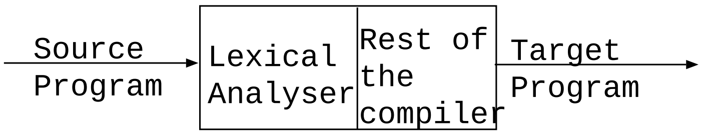
(a) A transition diagram constructed from the token specification.

(b) A code fragment which can traverse ***any transition diagram*** **(a driver routine).**

(c) Code for the action specifications.

The resulting program is the lexical analyser. This can now be incorporated into a compiler.





The UNIX tool lex is such a generator.

AUTOMATIC GENERATION OF LEXICAL ANALYSERS

MOTIVATION

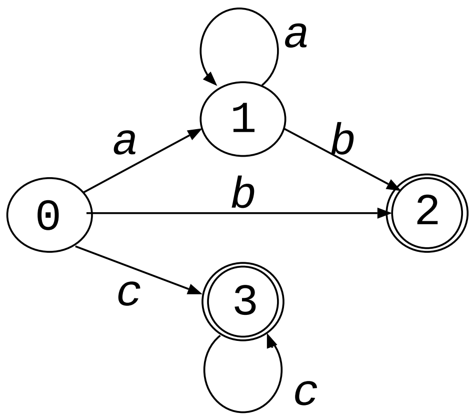
Suppose a language has two tokens

1. **Token 1 is the set of strings consisting of zero or more** a**'s and ending with a single** b**.**
2. **Token 2 is the set of strings consisting of one or more** c**'s.**

Formally describe the two tokens as

$a^* b$	**{print('token_1 found')}**
c^+	**{print('token_2 found')}**

This formalism, called a regular expression, is used to *describe* **the set of strings constituting a token.**
From the description, construct a structure called a deterministic finite automaton (DFA).



AUTOMATIC GENERATION OF LEXICAL ANALYSERS

Now consider the following three together:

1. The same DFA stored in a tabular form in an array called *nextstate*.

state ↓ \ symbol →	*a*	*b*	*c*
0	1	2	3
1	1	2	–
2	–	–	–
3	–	–	3

2. A driver routine which can traverse *any* **given DFA**

```
function nexttoken;
begin
    state := 0; c := nextchar;
    while nextstate[state,c] <> _ do
          begin
              state := nextstate[state,c]; c := nextchar
          end
    if not final(state) then
          begin error; return end
    else begin
              unput(c); action; return
        end
    end;
```

AUTOMATIC GENERATION OF LEXICAL ANALYSERS

3. The action part of the token specification.

```
function action;
begin
    case state of
        2: print('token_1 found');
        3: print('token_2 found');
    end
end;
```

The DFA, the driver routine and the action routines taken together, constitute the lexical analyser.

AUTOMATIC GENERATION OF LEXICAL ANALYSERS

SPECIFYING TOKENS

To specify tokens, we use regular expressions.
Roughly, a regular expression denotes a set of strings, as does a token. A set of strings is also called a *language*. **We talk about a token in the context of programming languages, whereas the concept of regular expression is used in much more general contexts.**

We shall

1. **Formally define the notion of a language.**
2. **Define the syntax of regular expressions.**
3. **Specify what is the language that each regular expression denotes.**
4. **See how to specify tokens through regular expressions.**

AUTOMATIC GENERATION OF LEXICAL ANALYSERS

LANGUAGE

Definition An *alphabet* is any finite set of symbols.

Examples $L = \{\mathbf{A},\ldots,\mathbf{Z},\mathbf{a},\ldots,\mathbf{z}\}$, $D = \{\mathbf{0},\mathbf{1},\ldots,\mathbf{9}\}$, $L \cup D$.

Definition A *string* over an alphabet is a finite sequence of symbols drawn from the alphabet.

Examples begin, max1, 123, ϵ (the empty string), are strings over $L \cup D$.

Definition If s and r are strings, then the *concatenation of* s *and* r, written sr is the string formed by appending r to s. Further $s\epsilon = \epsilon s = s$.

Definition A string, exponentiated i times, is defined inductively as:

$$\mathrm{s}^0 = \epsilon,$$
$$\mathrm{s}^i = \mathrm{ss}^{i-1}$$

Definition A *language* is any set of strings over some fixed alphabet.

Examples {max, max1, A123}, {1, 123, -87}, {begin}, {end}, $\{\epsilon\}$, ϕ (the empty set).

AUTOMATIC GENERATION OF LEXICAL ANALYSERS

OPERATIONS ON LANGUAGES

Languages can be combined to give bigger languages using the operations shown below. If L and M are languages, then:

1. **Union**

$$L \cup M = \{s | s \in L \textbf{ or } s \in M\}$$

2. **Intersection**

$$L \cap M = \{s | s \in L \textbf{ and } s \in M\}$$

3. **Concatenation**

$$LM = \{sr | s \in L \textbf{ and } r \in M\}$$

4. **Exponentiation**

$$L^i = LL^{i-1}$$

5. **Kleene Closure (zero or more concatenations)**

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

6. **Positive Closure (one or more concatenations)**

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

AUTOMATIC GENERATION OF LEXICAL ANALYSERS

EXAMPLES

If $L = \{A, \ldots, Z, a, \ldots, z\}, D = \{0, 1, \ldots, 9\}$, then

1. **$L \cup D$ is the set of letters and digits.**
2. **LD is the set of strings consisting of a letter followed by a digit.**
3. **L^4 is the set of four letter words.**
4. **L^* is the set of all strings of letters, including ϵ.**
5. **$L(L \cup D)^*$ is the set of all strings of letters and digits, beginning with a letter. This is the language represented by the token identifier.**
6. **D^+ is the set of all strings of one or more digits**

AUTOMATIC GENERATION OF LEXICAL ANALYSERS

REGULAR EXPRESSIONS

The table below shows possible regular expressions over an alphabet Σ and the languages they denote. Here r and s stand for regular expressions, and $L(r)$ and $L(s)$ stand for the languages denoted by them.

regular expression	language
ϵ	$\{\epsilon\}$
$a \in \Sigma$	$\{a\}$
$r\|s$	$L(r) \cup L(s)$
rs	$L(r)L(s)$
r^*	$L(r)^*$
(r)	$L(r)$

A language defined by a regular expression is called a *regular language.*

AUTOMATIC GENERATION OF LEXICAL ANALYSERS

EXTENDED REGULAR EXPRESSIONS

The language provided by Lex to describe patterns is an extension of regular expressions that we have seen earlier. The extension serves two purposes:

1. It is more concise than the usual regular expressions. For example, instead of writing `A | B | C | D | E`, we can write `[A-E]`.
2. It provides for machine readability. The symbols of the language whose tokens are being specified and the metalanguage in which the regular expressions are being expressed are clearly distinguished. Thus `|` is the metalinguistic character denoting 'alternate', Whereas `\|` is the character `|` itself.

AUTOMATIC GENERATION OF LEXICAL ANALYSERS

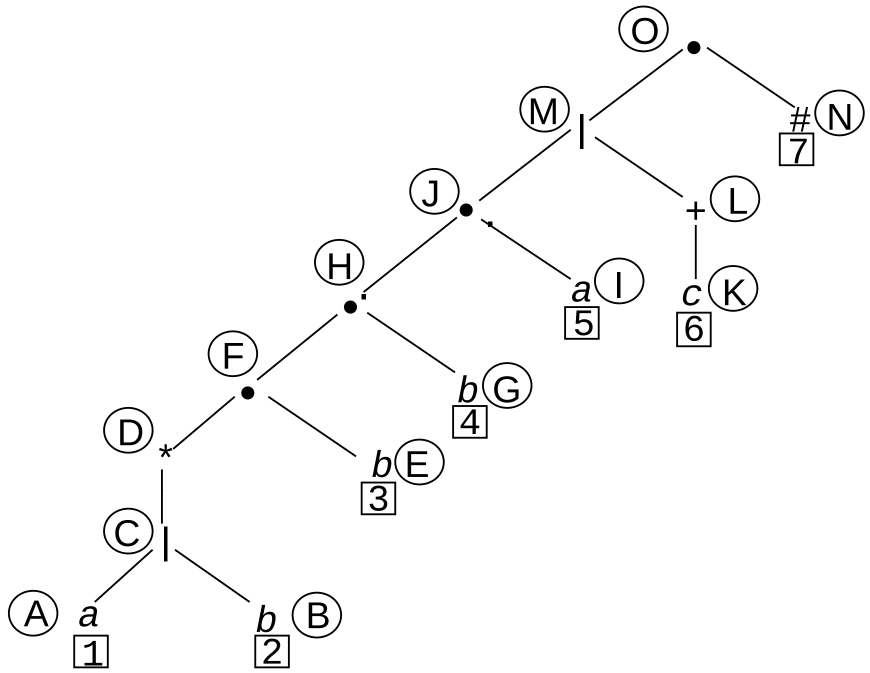
<table>
<tr><th>Expression</th><th>Describes</th><th>Example</th></tr>
<tr><td>c</td><td>any non-metalinguistic character c</td><td>a</td></tr>
<tr><td>\c</td><td>character c literally</td><td>*</td></tr>
<tr><td>''s''</td><td>string s literally</td><td>''**''</td></tr>
<tr><td>.</td><td>any character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

AUTOMATIC GENERATION OF LEXICAL ANALYSERS

CONVERTING REGULAR EXPRESSION TO DFA

Consider the regular expression $(a|b)^*bba|c^+$

We can represent this regular expression in the form of a tree:



NOTE:

1. **Leaves have been labeled with positions (1,2 etc.) and alphabetic labels (A,B etc.), whereas interior nodes are labeled with alphabetic labels only.**

2. **Concatenation has been represented by a •.**

3. **To signify the end of a token, we use # as a end marker.**

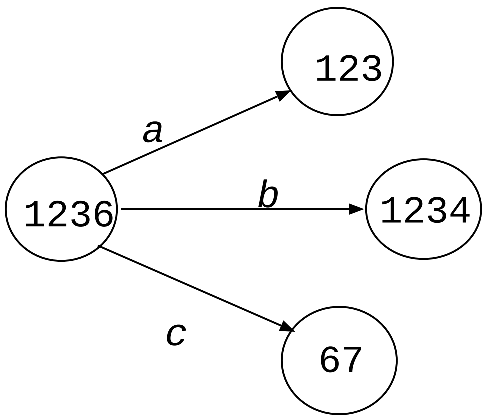
AUTOMATIC GENERATION OF LEXICAL ANALYSERS

The key idea is to identify a state with a set of positions in the regular expression. For example, the starting state of the DFA for $(a|b)^*bba|c^+$ could expect a a (in position 1 of the corresponding tree), a b (in position 2 or 3) or a c (in position 6). Therefore we shall identify this state with positions 1,2,3 and 6.

Starting from this state, we could make the following transitions:

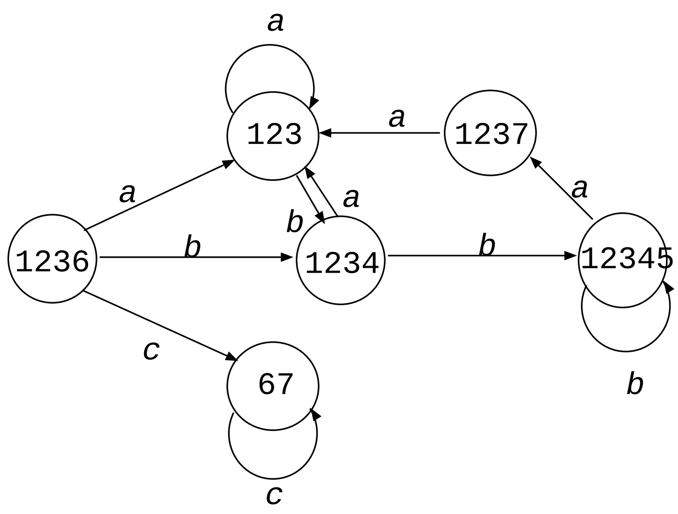
if we are in	then on symbol	we could go to
position 1	a	1,2,3
position 2	b	1,2,3
position 3	b	4
position 6	c	6,7

This gives us three new states as shown below:



AUTOMATIC GENERATION OF LEXICAL ANALYSERS

Continuing in this manner, the final DFA will be:



The final states are those which contain position 7, i.e. the position expecting #.

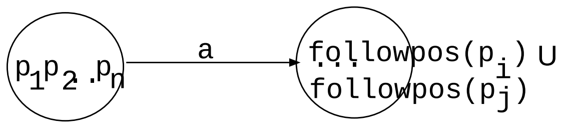
AUTOMATIC GENERATION OF LEXICAL ANALYSERS

FORMALISING THE METHOD

Define a function $followpos$ where $followpos(i)$ answers the following question: If we are at position i of the syntax tree, then which positions does the (unique) symbol corresponding to i take us to?
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)

Then our transition diagram becomes:



where position p_i to p_j are the only positions associated with symbol a.

AUTOMATIC GENERATION OF LEXICAL ANALYSERS

FINDING $followpos$

To find $followpos$, we have to first define some additional functions:

1. $nullable(n)$: **is true if the string represented by the node can generate a null string.**

   $nullable(\mathsf{D})$ = **true;**
   $nullable(\mathsf{F})$ = false;

2. $firstpos(n)$: $firstpos$ **of a node** n **is the set of positions that can match the first symbol of some string derivable from the subexpression rooted at** n.

   $firstpos(\mathsf{D}) = \{1, 2\}$
   $firstpos(\mathsf{F}) = \{1, 2, 3\}$

3. $lastpos(n)$: $lastpos$ **of a node** n **is the set of positions that can match the last symbol of some string derivable from the subexpression rooted at** n.

   $lastpos(\mathsf{D}) = \{1,2\}$
   $lastpos(\mathsf{F}) = \{3\}$

AUTOMATIC GENERATION OF LEXICAL ANALYSERS

RULES FOR $nullable$, $firstpos$ AND $lastpos$

node n	$nullable(n)$
n **is a leaf labeled** ϵ	**true**
n **is a leaf labeled with symbol** a	**false**
n **is** $c_1 \| c_2$	$nullable(c_1)$ **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 labeled** a **at position** i	$\{i\}$
n **is** $c_1 \| c_2$	$firstpos(c_1) \cup firstpos(c_2)$
n **is** $c_1 \bullet c_2$	**if** $nullable(c_1)$ **then**
	$firstpos(c_1) \cup firstpos(c_2)$
	else $firstpos(c_1)$
n **is** c^*	$firstpos(c)$
n **is** c^+	$firstpos(c)$

AUTOMATIC GENERATION OF LEXICAL ANALYSERS

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

RULES DEFINING $followpos$

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** $followpos(i)$**.**

AUTOMATIC GENERATION OF LEXICAL ANALYSERS

EXAMPLE

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}**	

AUTOMATIC GENERATION OF LEXICAL ANALYSERS

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 $V = followpos(p_1) \cup \ldots \cup followpos(p_k)$;
           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 $\#$.**

LEXICAL ERRORS

CLASSIFICATION OF ERRORS

There are primarily two kinds of lexical errors:

1. Lexemes whose length exceed the bound specified by the language. As examples:
   (a) In Fortran, an identifier more than 7 characters long is a lexical error.
   (b) 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 appearing in the input stream. For example the characters `~`, `&` and `@` occuring in a Pascal program (but not within a string or a comment) are lexical errors.

LEXICAL ERRORS

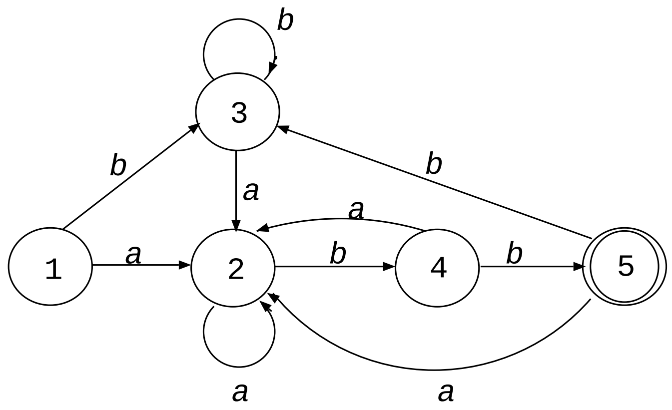
ERROR HANDLING

Lexical errors can easily be detected by lexical analyser. Apart from issuing an error message, the action taken on detection of an error are:

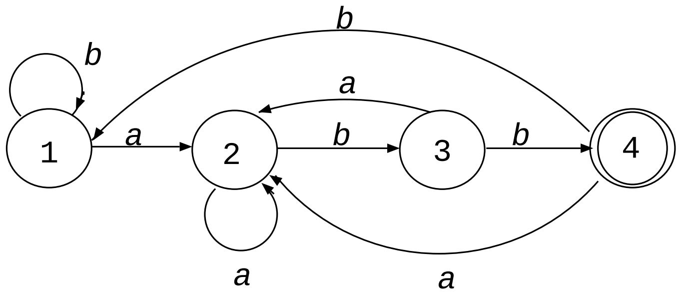
1. For an error of the first type, the entire lexeme is read and then truncated to the specified length.
2. In case of an error of the second kind, a simple strategy would be to skip illegal characters till one detects a character which can begin a token. Another alternative would be to pass the character to the parser and depend on the error handling capability of the parser. (see module on Error Analysis). This might better in certain situations because the parser has better knowledge of the context in which error has occurred. This helps the parser to recover from the error by not only deletion (like the strategy mentioned above) but also by insertion.

MINIMIZING THE NUMBER OF STATES

For the regular expression $(b|\epsilon)(a|b)^*abb$, the DFA constructed by the earlier method is:



It can be verified, that there is another DFA, with lesser number of states, which accepts the same language.



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, if possible.

MINIMIZING THE NUMBER OF STATES

CENTRAL IDEA BEHIND MINIMIZATION

The key idea is that certain states can possibly be merged in a DFA without affecting acceptability. As example, note that *any string takes 1 to a final state, iff it takes 3 to a final state.* **The states 1 and 3 are therefore, as far acceptance is concerned,** *indistinguishable.*

States 2 and 4, on the other hand, are *distinguishable*. **The string** b **takes 4 to the final state 5, but takes 2 to 4.** b **is called a** ***distinguishing string*** **for states 2 and 4.**

So 1 and 3 can be merged, but 2 and 4 have to be kept separate.

DEFINITION

Two states are *distinguishable* **if there exists a string** s **which takes one of the states to a final state, and the other to a non-final state.** s **is then called a** *distinguishing string* **for the two states.**

MINIMIZING THE NUMBER OF STATES

HOW TO FIND DISTINGUISHABLE STATES?

Firstly, note that the set of final states are distinguishable from the set of non-final states. This is true because the null-string ϵ distinguishes them.
Assume initially that the set of states have been partitioned into two groups, one consisting of the set of final states, and the other the set of non-final states. The states within a group are deemed to be indistinguishable. Therefore,

$\Pi_1 = \{\{1,2,3,4\},\{5\}\}$

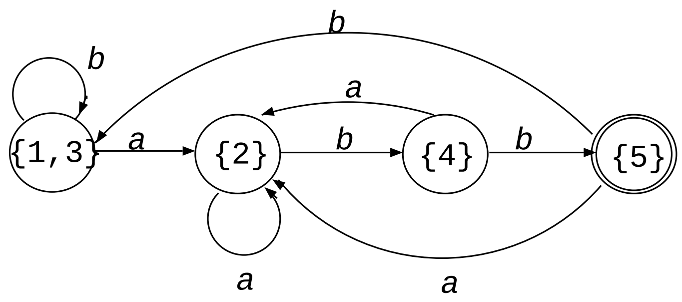
Now consider states 3 and 4. The string b takes 3 to itself and 4 to 5. Since 3 and 5 are already distinguishable on the basis of ϵ, states 3 and 4 become distinguishable on the basis of the string $b\epsilon$. The new partition Π_2 and the next partition Π_3 are shown in the table below.

Partition n	Distinguishing string
$\Pi_1 = \{\{1,2,3,4\},\{5\}\}$	ϵ
$\Pi_2 = \{\{1,2,3\},\{4\},\{5\}\}$	$b\epsilon$
$\Pi_3 = \{\{1,3\},\{2\},\{4\},\{5\}\}$	$bb\epsilon$

MINIMIZING THE NUMBER OF STATES

THE MINIMIZED DFA

In the minimized DFA, there will be a state corresponding to each group of the partition. If a group has more than one state of the original DFA, then, on a particular symbol, they will all transit to states of the same group. The resulting DFA is



NOTE: In the context of lexical analyser generators, all the final states should not merged in the initial partition. This is because two different final states may recognise two different tokens, and therefore should be distinguished. So, in the initial partition, only those final states which recognise the same token are merged in a group.

MINIMIZING THE NUMBER OF STATES

ALGORITHM FOR MINIMIZING DFA

1. **Construct an initial partition $\Pi = \{S - F, F\}$, where F is the set of final states and $S - F$ is the state of non-final states.**

2. **for each group G of Π do**
   partition G into subgroups such that two states
   s and t of G are in the same subgroup if and only if
   for all input symbols a, states s and t have transitions
   onto states in the same group of Π;
   replace G in Π_{new} by the set of all subgroups formed

3. **If $\Pi_{new} = \Pi$, let $\Pi_{final} := \Pi$ and continue with step 2. Otherwise repeat step 4. with $\Pi := \Pi_{new}$.**

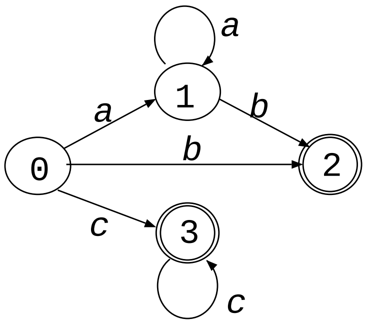
4. **Construct one state in each group of the partition Π_{final} as the representative for that group. Let s be a representative state. If, in the old DFA, there is a transition from s to t on a, then in the new DFA, there is a transition from s to the representative of the group containing t in Π_{final}.**

5. **Remove any dead states.**

EFFICIENT REPRESENTATION OF DFA

MOTIVATION

A naive method to represent a DFA is to use a two dimensional array indexed by state and input symbol, whose entries are, in turn, states.



symbol →

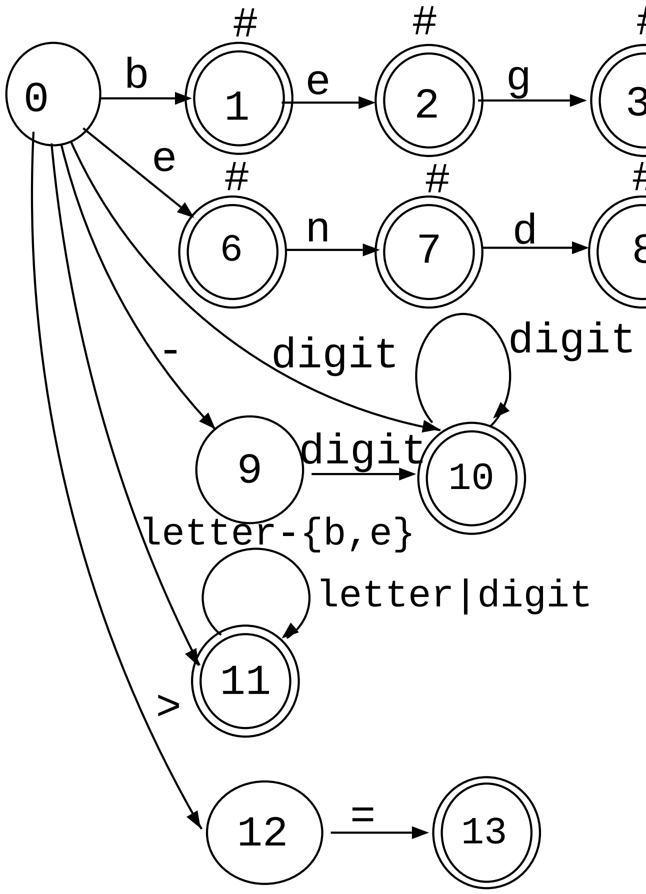
state ↓	*a*	*b*	*c*
0	1	2	3
1	1	2	–
2	–	–	–
3	–	–	3

For a typical language, the number of DFA states is in the order of hundreds (1111 in the case of a particular Fortran-77 compiler), and the number of input symbols is roughly 100. The size of the resulting DFA array is extremely large.
Therefore it is necessary to find a representation which is smaller in size without compromising on the time required to retrieve the nextstate information.

EFFICIENT REPRESENTATION OF DFA

THE FOUR ARRAYS SCHEME

A key observation leading to a efficient representation is the fact that for a DFA as that shown below, the states marked with # behave like state 11 on all symbols *except for one symbol***.**



Therefore information about state 11 can also be used for these states.

EFFICIENT REPRESENTATION OF DFA

This scheme uses four arrays called $DEFAULT$, $BASE$, $NEXT$ and $CHECK$. 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 $DEFAULT[s]$.**

The function which gives the next state can be described as:

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

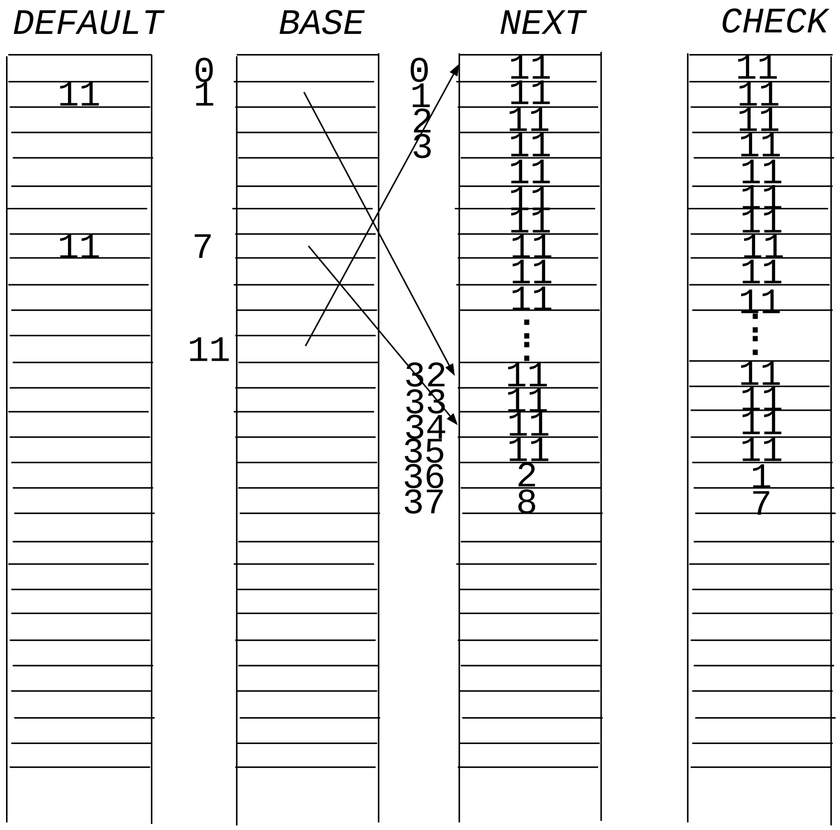
EFFICIENT REPRESENTATION OF DFA

EXAMPLE

Consider an alphabet with its symbols numbered as shown below:

`a–z`	**0–25**
`0–9`	**26–35**
`>`	**36**
`=`	**37**
`-`	**38**

The earlier DFA would be stored under the four arrays scheme as:



EFFICIENT REPRESENTATION OF DFA

NOTES:

1. **All the entries for state 11 have been stored in the array $NEXT$. The $CHECK$ array shows that the entries are valid for state 11.**

2. **State 1 has a transition on e(4), which is different from the corresponding transition on state 11. This differing entry is stored in $NEXT$[36]. Therefore $BASE$[1] is set to 36 – 4 = 32.**

3. **By a similar reasoning $BASE$[7] is set to 34.**

4. **To find nextstate[1,0], we first refer to $NEXT$[32+0], But since $CHECK$[32+0] is not 1 we have to refer to $DEFAULT$[1] which is 11. So the correct next state is found from $NEXT[BASE[11]+0] = 11$.**

5. **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 the existing entries.**

GRADED EXERCISES

1. Identify the lexemes that make up the tokens in the following programs. Give reasonable attribute values for the tokens.

   a. Pascal

```
function max ( i, j : integer ) : integer;
{ return maximum of integers i and j }
begin
    if i > j then max := i
    else max := j
end;
```

   b. Fortran 77

```
      FUNCTION MAX ( I, J )
C     RETURN MAXIMUM OF INTEGERS I AND J
          IF (I .GT. J) THEN
              MAX = I
          ELSE
              MAX = J
          END IF
      RETURN
```

2. Write a regular expression for the language consisting of all strings of 0's and 1's, in which the number of 1's is a multiple of 3.

3. Draw a DFA to accept binary representations of all numbers which are multiples of 3. Repeat the exercise for the sets $\{k \mid k \; mod \; 3 = 1\}$ and $\{k \mid k \; mod \; 3 = 2\}$.

4. Describe the sets denoted by the following regular expressions:

   a. (11|0)*(00|1)*

   b. (1|01|001)*(ϵ |0|00)

   c. (00|11|(01|10)(00|11)*(01|10))*

5. Define a comment as any sequence of characters enclosed between `(*` and `*)`, not containing the sequence `*)`. Comments may extend over more than one lines.

   Write an extended regular expression to specify a comment.

6. Repeat the above exercise for a string, defined as a sequence of characters enclosed between ' and '. If the character ' itself is a part of the string, it should be represented as ''.

7. Using the direct method, construct minimum DFAs for the following regular expressions:

   a. 10|(0|11)0*1

   b. 01(((10)*|111)*|0)*1

   c. ((0|1)(0|1))*|((0|1)(0|1)(0|1))*

8. Consider the following LEX-like specification:

   <u>AUXILLIARY DEFINITIONS</u>

```
letter = A |.. | Z
digit  = 0 |.. | 9
sign   = + | -
```

   <u>TRANSLATION RULES</u>

```
END                  {return 1}
FOR                  {return 2}
GOSUB                {return 3}
GOTO                 {return 4}
IF                   {return 5}
LET                  {return 6}
REM                  {return 7}
RETURN               {return 8}
letter(digit)?       {return 9}
sign digit digit*    {return 10}
```

a. Construct the minimum DFA which would recognize the tokens specified by the program.

b. Construct a four array representation for this DFA.