Lexical Analysis

Amitabha Sanyal

(www.cse.iitb.ac.in/~as)

Department of Computer Science and Engineering,

Indian Institute of Technology, Bombay

September 2007

Recap

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);
}
```

Recap

The same program – as the compiler sees it (initially).

```
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

Recap

The same program – as the compiler sees it (initially).

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

How do you make the compiler see what you see?

Recap - Discovering the structure of the program

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 | ) | ; | ↩ | }
```

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

Recap - Discovering the structure of the program

Step 1:

b. Clean up – 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.

Recap - Discovering the structure of the program

Step 1:

b. Clean up – remove the ␣ and the ↩ characters.

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*.

Recap - 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 ) ; }
```

Recap - 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	)	;	}											

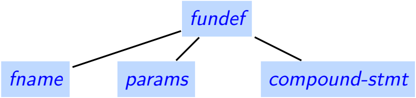
fundef

Recap - 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 ) ; }
```

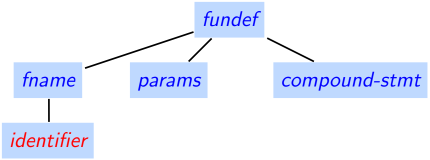


Recap - Discovering the structure of the program

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 ) ; }
```

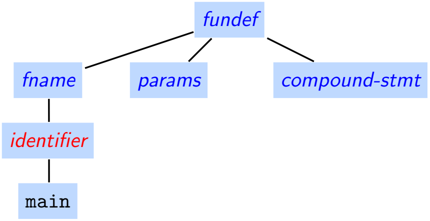


Recap - Discovering the structure of the program

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 ) ; }
```

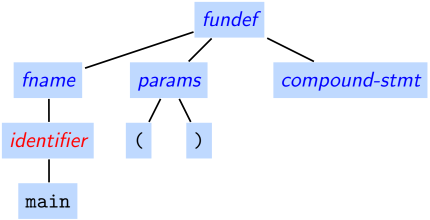


Recap - Discovering the structure of the program

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 ) ; }
```

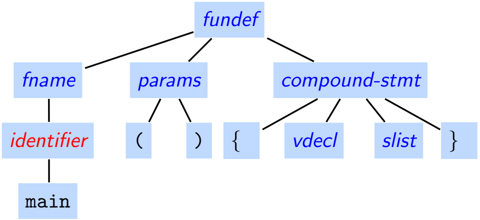


Recap - Discovering the structure of the program

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 ) ; }
```

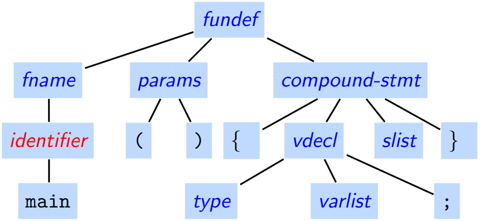


Recap - Discovering the structure of the program

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 ) ; }
```

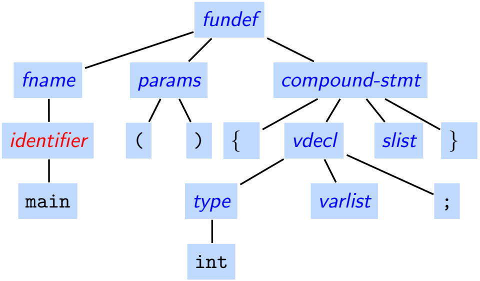


Recap - Discovering the structure of the program

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 ) ; }
```

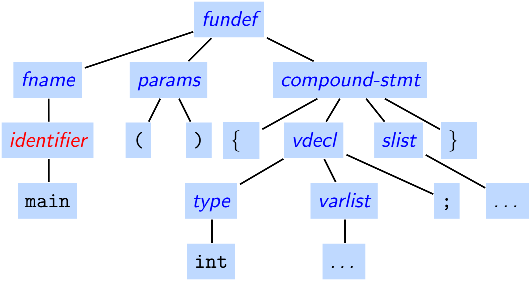


Recap - Discovering the structure of the program

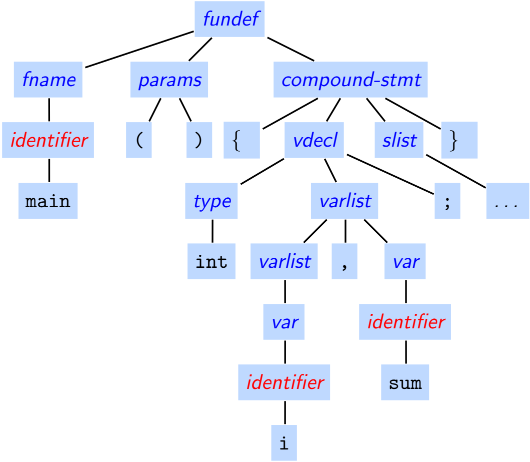
Step 2:

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 ) ; }
```



Recap - Discovering the structure of the program



This is *syntax analysis* or *parsing*.

Lexemes, Tokens and Patterns

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

Lexemes, Tokens and Patterns

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

Distinguish between

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", <=`.

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.
  Examples –
  identifier = {`i, sum, buffer`, ...}
  int_constant = {`1, 10, ...`}
  addop = {`+, -`}

Lexemes, Tokens and Patterns

Things that are not counted as lexemes –

Lexemes, Tokens and Patterns

Things that are not counted as lexemes –

- white spaces – tab, blanks and newlines

Lexemes, Tokens and Patterns

Things that are not counted as lexemes –

- white spaces – tab, blanks and newlines
- comments

Lexemes, Tokens and Patterns

Things that are not counted as lexemes –

- white spaces – tab, blanks and newlines
- comments

These too have to be detected and ignored.

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

Recap - Lexemes, Tokens and Patterns

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

Variants in different languages.

Recap - 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.

Recap - 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 the first character is an alphabet. It has a length of at most 31.

Recap - 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 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 the first character is an alphabet or an underline. It has a length of at most 31. Any characters after the 31st character are ignored.

Recap - 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 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 the first character is an alphabet or an underline. It has a length of at most 31. Any characters after the 31st character are ignored.

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

Recap - Basic concepts and issues

A pattern is used to

- *specify tokens* precisely
- *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:**
   2.1 Integer Constants
   - Octal, Hex and Decimal
   - 4 byte and 8 byte representation

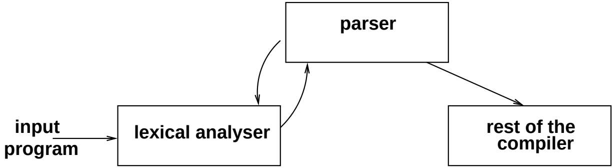
   2.2 Floating point constants
   - float - ends with `f`
   - double

   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`.

Recap - 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 intertwined.



Recap - Token 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, –>

Creating a Lexical Analyzer

Two approaches:

Creating a Lexical Analyzer

Two approaches:

1. *Hand code* – This is only of historical interest now.
    - Possibly more efficient.

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

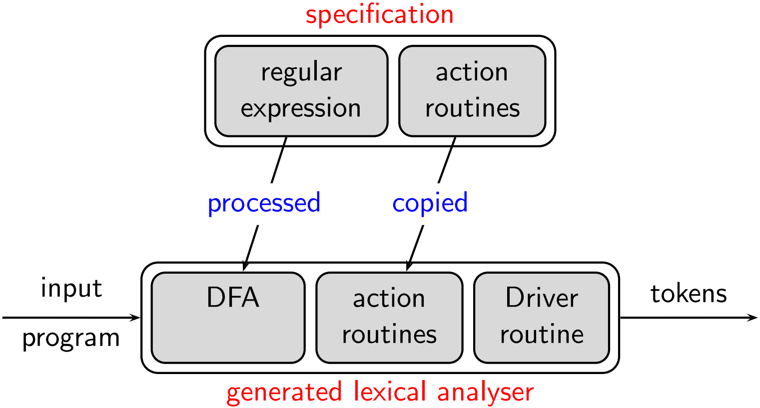
Inputs to the lexical analyser generator:

- A specification of the tokens of the source language, consisting of:
  - 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*.

Automatic Generation of Lexical Analysers

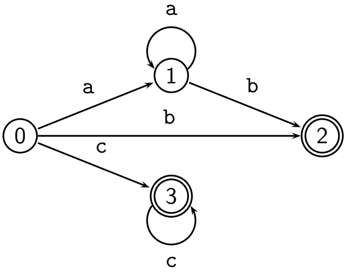


Example of Lexical Analyser Generation

Suppose a language has two tokens

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

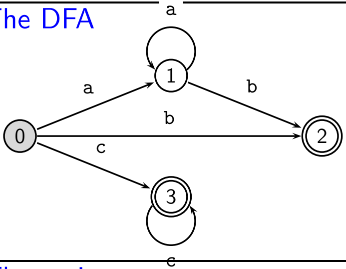
From the description, construct a structure called a deterministic finite automaton (DFA).



Example of Lexical Analyser Generation

Now consider the following together:

The DFA



The driver routine

```
void nexttoken ()
  {state = 0; c = nextchar();
   while (valid(nextstate[state,c]))
    {state = nextstate[state,c];
     c = nextchar();}
   if (!final(state))
     {error; return;}
   else
     {unput(c);action();return;}}
```

The actions

```
void action();
{
 switch(state)
  2:{printf("Token 1 found");
     break;}
  3:{printf("Token 2 found");
     break;}
}
```

The input and output

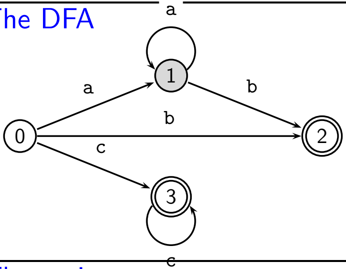
Input: aabadbcc⟵

Output:

Example of Lexical Analyser Generation

Now consider the following together:

The DFA



The driver routine

```
void nexttoken ()
  {state = 0; c = nextchar();
   while (valid(nextstate[state,c]))
    {state = nextstate[state,c];
     c = nextchar();}
   if (!final(state))
     {error; return;}
   else
     {unput(c);action();return;}}
```

The actions

```
void action();
{
 switch(state)
  2:{printf("Token 1 found");
     break;}
  3:{printf("Token 2 found");
     break;}
}
```

The input and output

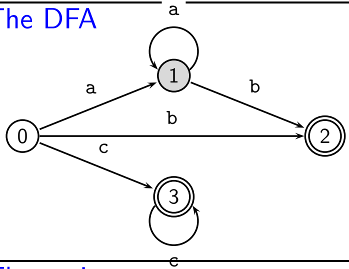
Input: aabadbcc⟵

Output:

Example of Lexical Analyser Generation

Now consider the following together:

The DFA



The driver routine

```
void nexttoken ()
  {state = 0; c = nextchar();
   while (valid(nextstate[state,c]))
    {state = nextstate[state,c];
     c = nextchar();}
   if (!final(state))
     {error; return;}
   else
     {unput(c);action();return;}}
```

The actions

```
void action();
{
 switch(state)
  2:{printf("Token 1 found");
     break;}
  3:{printf("Token 2 found");
     break;}
}
```

The input and output

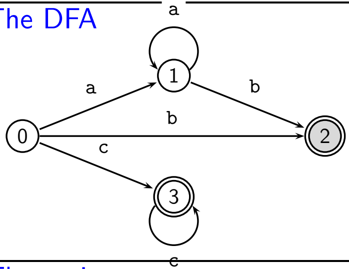
Input: aabadbcc⟵

Output:

Example of Lexical Analyser Generation

Now consider the following together:

The DFA



The driver routine

```
void nexttoken ()
  {state = 0; c = nextchar();
   while (valid(nextstate[state,c]))
    {state = nextstate[state,c];
     c = nextchar();}
   if (!final(state))
     {error; return;}
   else
     {unput(c);action();return;}}
```

The actions

```
void action();
{
 switch(state)
  2:{printf("Token 1 found");
     break;}
  3:{printf("Token 2 found");
     break;}
}
```

The input and output

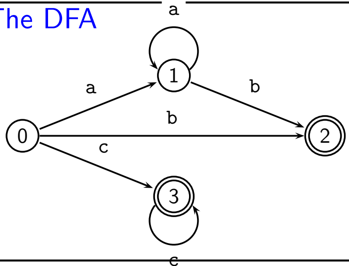
Input: aabadbcc⟵

Output:

Example of Lexical Analyser Generation

Now consider the following together:

The DFA



The driver routine

```
void nexttoken ()
  {state = 0; c = nextchar();
   while (valid(nextstate[state,c]))
    {state = nextstate[state,c];
     c = nextchar();}
   if (!final(state))
     {error; return;}
   else
     {unput(c);action();return;}}
```

The actions

```
void action();
{
 switch(state)
  2:{printf("Token 1 found");
     break;}
  3:{printf("Token 2 found");
     break;}
}
```

The input and output

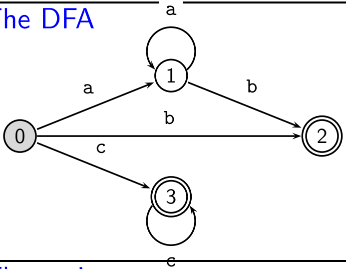
Input: aabadbcc⟵

Output: Token 1 found

Example of Lexical Analyser Generation

Now consider the following together:

The DFA



The driver routine

```
void nexttoken ()
  {state = 0; c = nextchar();
   while (valid(nextstate[state,c]))
    {state = nextstate[state,c];
     c = nextchar();}
   if (!final(state))
     {error; return;}
   else
     {unput(c);action();return;}}
```

The actions

```
void action();
{
 switch(state)
  2:{printf("Token 1 found");
     break;}
  3:{printf("Token 2 found");
     break;}
}
```

The input and output

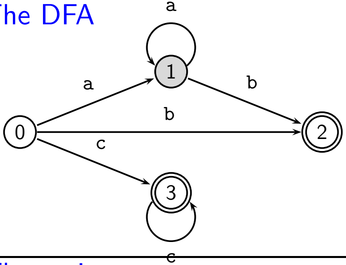
Input: aabadbcc⟵

Output: Token 1 found

Example of Lexical Analyser Generation

Now consider the following together:

The DFA



The driver routine

```
void nexttoken ()
  {state = 0; c = nextchar();
   while (valid(nextstate[state,c]))
    {state = nextstate[state,c];
     c = nextchar();}
   if (!final(state))
     {error; return;}
   else
     {unput(c);action();return;}}
```

The actions

```
void action();
{
 switch(state)
  2:{printf("Token 1 found");
     break;}
  3:{printf("Token 2 found");
     break;}
}
```

The input and output

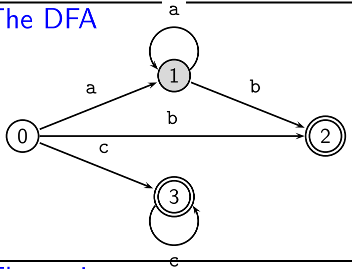
Input: aabadbcc↩

Output: Token 1 found

Example of Lexical Analyser Generation

Now consider the following together:

The DFA



The driver routine

```
void nexttoken ()
  {state = 0; c = nextchar();
   while (valid(nextstate[state,c]))
    {state = nextstate[state,c];
     c = nextchar();}
   if (!final(state))
     {error; return;}
   else
     {unput(c);action();return;}}
```

The actions

```
void action();
{
 switch(state)
  2:{printf("Token 1 found");
     break;}
  3:{printf("Token 2 found");
     break;}
}
```

The input and output

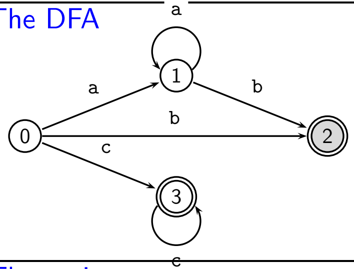
Input: aabadbcc⟵

Output: Token 1 found

Example of Lexical Analyser Generation

Now consider the following together:

The DFA



The driver routine

```
void nexttoken ()
  {state = 0; c = nextchar();
   while (valid(nextstate[state,c]))
    {state = nextstate[state,c];
     c = nextchar();}
   if (!final(state))
     {error; return;}
   else
     {unput(c);action();return;}}
```

The actions

```
void action();
{
 switch(state)
  2:{printf("Token 1 found");
     break;}
  3:{printf("Token 2 found");
     break;}
}
```

The input and output

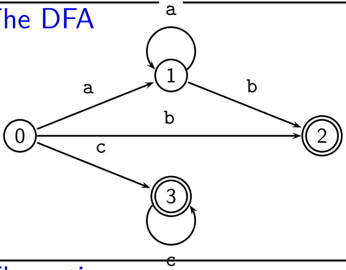
Input: aabadbcc⟵

Output: Token 1 found

Example of Lexical Analyser Generation

Now consider the following together:

The DFA



The driver routine

```
void nexttoken ()
  {state = 0; c = nextchar();
   while (valid(nextstate[state,c]))
    {state = nextstate[state,c];
     c = nextchar();}
   if (!final(state))
     {error; return;}
   else
     {unput(c);action();return;}}
```

The actions

```
void action();
{
 switch(state)
  2:{printf("Token 1 found");
     break;}
  3:{printf("Token 2 found");
     break;}
}
```

The input and output

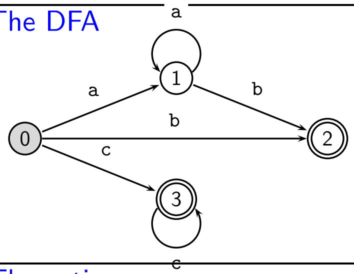
Input: aabadbcc⟵

Output: Token 1 found
Token 1 found

Example of Lexical Analyser Generation

Now consider the following together:

The DFA



The driver routine

```
void nexttoken ()
  {state = 0; c = nextchar();
   while (valid(nextstate[state,c]))
    {state = nextstate[state,c];
     c = nextchar();}
   if (!final(state))
     {error; return;}
   else
     {unput(c);action();return;}}
```

The actions

```
void action();
{
 switch(state)
  2:{printf("Token 1 found");
     break;}
  3:{printf("Token 2 found");
     break;}
}
```

The input and output

Input: aabadbcc↩

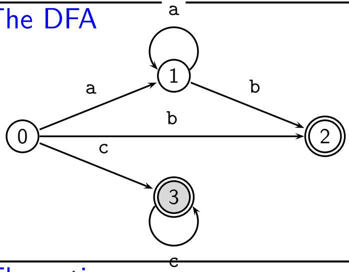
Output: Token 1 found

Token 1 found

Example of Lexical Analyser Generation

Now consider the following together:

The DFA



The driver routine

```
void nexttoken ()
  {state = 0; c = nextchar();
   while (valid(nextstate[state,c]))
    {state = nextstate[state,c];
     c = nextchar();}
   if (!final(state))
     {error; return;}
   else
     {unput(c);action();return;}}
```

The actions

```
void action();
{
 switch(state)
  2:{printf("Token 1 found");
     break;}
  3:{printf("Token 2 found");
     break;}
}
```

The input and output

Input: aabadbcc⟵

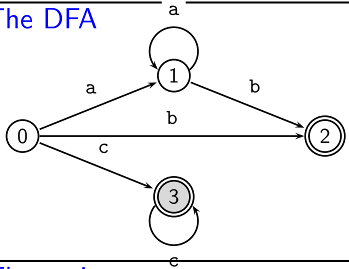
Output: Token 1 found

Token 1 found

Example of Lexical Analyser Generation

Now consider the following together:

The DFA



The driver routine

```
void nexttoken ()
  {state = 0; c = nextchar();
   while (valid(nextstate[state,c]))
    {state = nextstate[state,c];
     c = nextchar();}
   if (!final(state))
     {error; return;}
   else
     {unput(c);action();return;}}
```

The actions

```
void action();
{
 switch(state)
  2:{printf("Token 1 found");
     break;}
  3:{printf("Token 2 found");
     break;}
}
```

The input and output

Input: aabadbcc⟵

Output: Token 1 found
Token 1 found
Token 2 found

Example of Lexical Analyser Generation

In summary:

- The DFA, the driver routine and the action routines taken together, constitute the lexical analyser.
-
  - actions are supplied as part of specification.
  - driver routine is common to all generated lexical analyzers

  The only issue – how are the patterns, specified by regular expressions, converted to a DFA.

  In two steps:

  - Convert regular expression into NFA.
  - Convert NFA to DFA.

Example of Lexical Analyser Generation

Consider a language with the following tokens:

- *begin* – representing the lexeme `begin`
- *integer* – Examples: `0`, `-5`, `250`
- *identifier* – Examples: `a`, `A1`, `max`

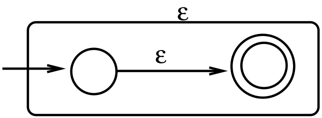
Converting Regular Expressions to NFA

In two parts;

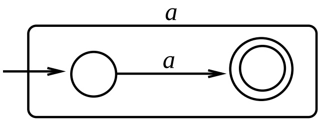
- First convert the regular expression corresponding to each token into a NFA.
  - Invariant: A single final state corresponding to each token.
- Join the NFAs obtained for all the tokens.

Converting Regular Expressions to DFA

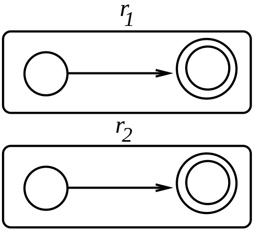
RE for ϵ



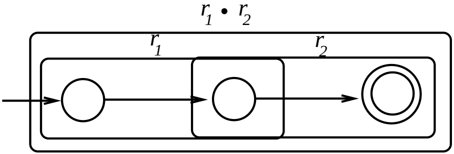
RE for a



Converting Regular Expressions to NFA

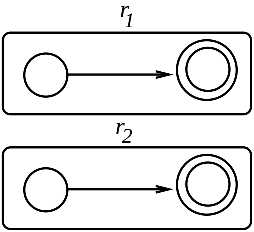


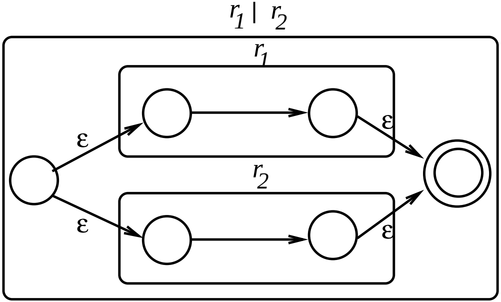
RE for $r_1 \bullet r_2$



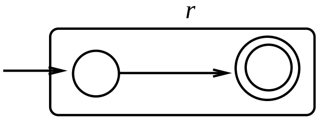
Converting Regular Expressions to NFA

RE for $r_1|r_2$

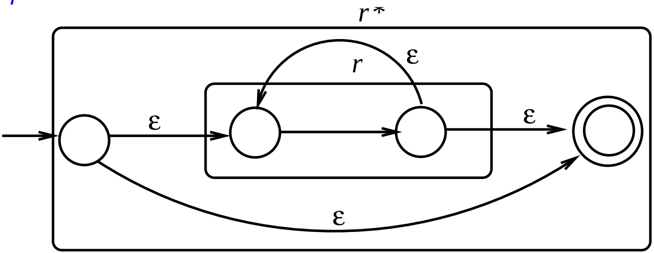




Converting Regular Expressions to NFA

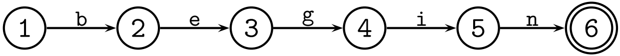


RE for r

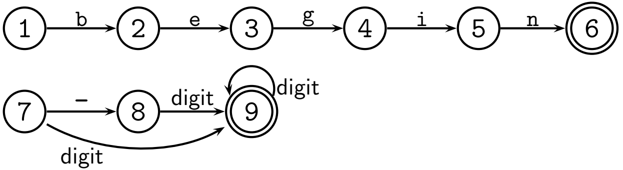


Converting NFA to DFA

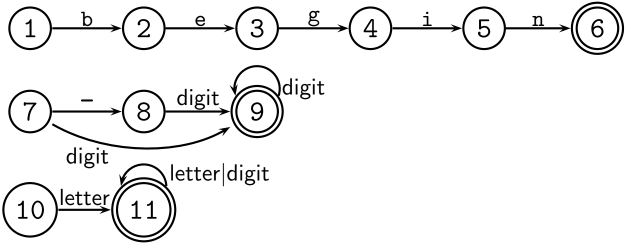
Converting NFA to DFA



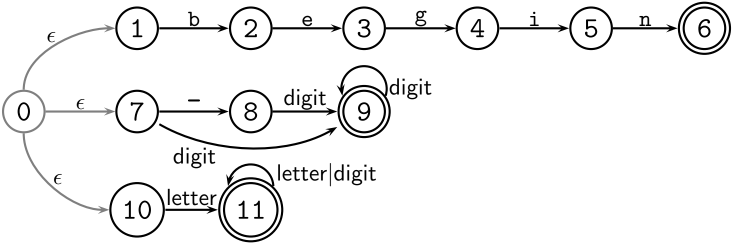
Converting NFA to DFA



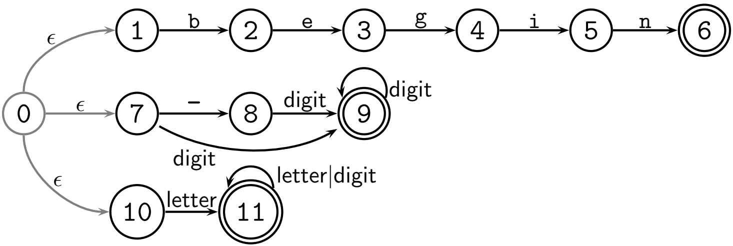
Converting NFA to DFA

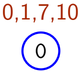


Converting NFA to DFA

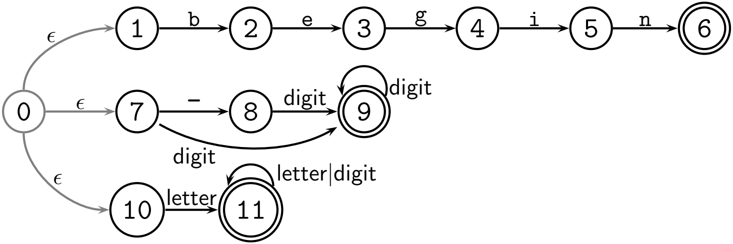


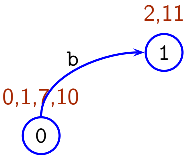
Converting NFA to DFA



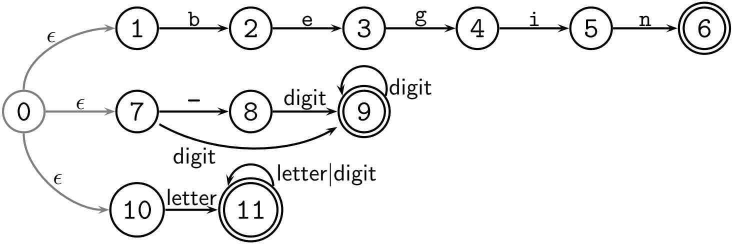


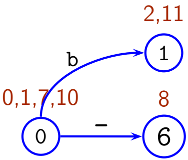
Converting NFA to DFA



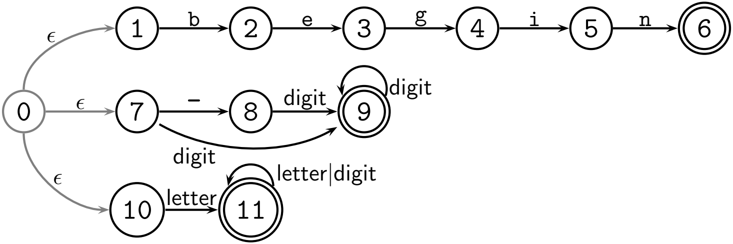


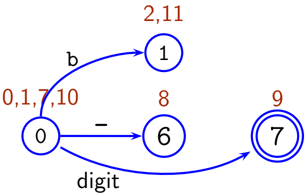
Converting NFA to DFA



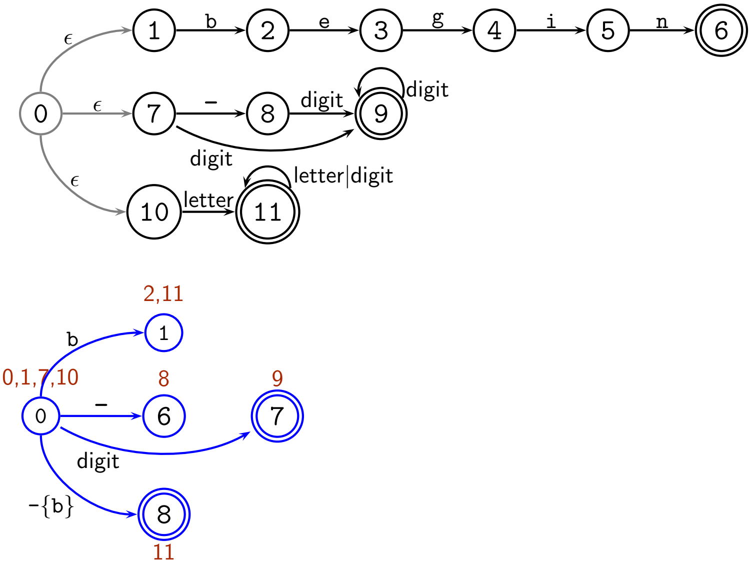


Converting NFA to DFA

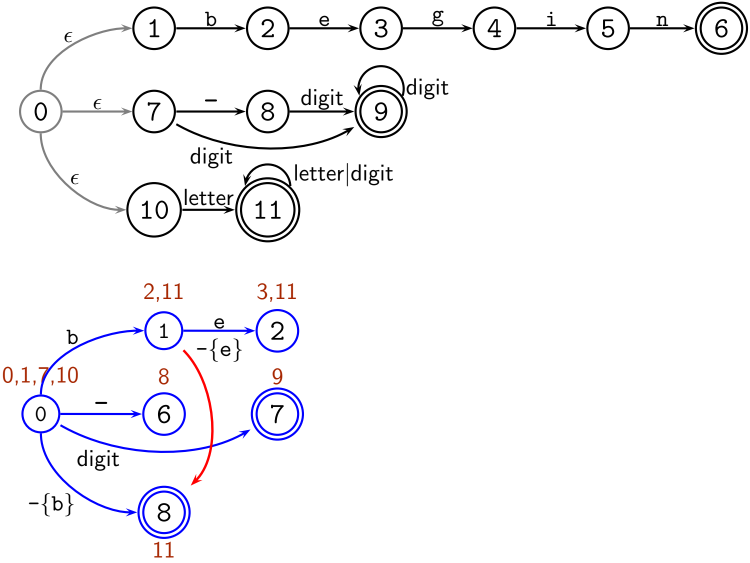




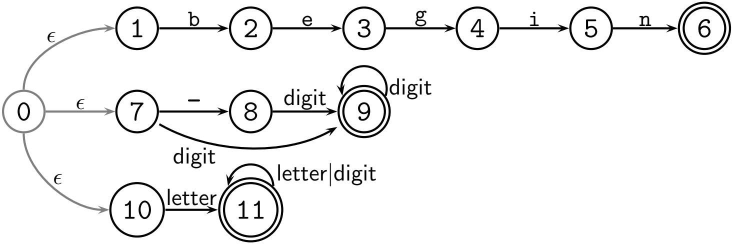
Converting NFA to DFA

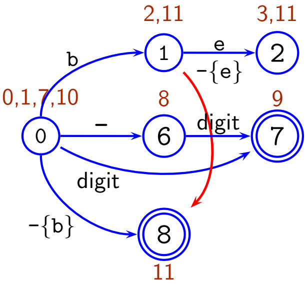


Converting NFA to DFA

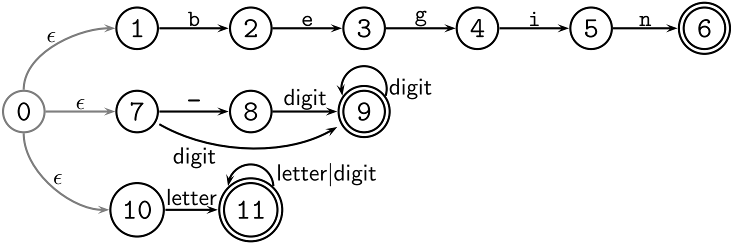


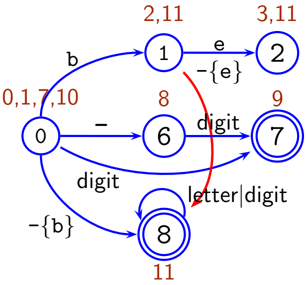
Converting NFA to DFA



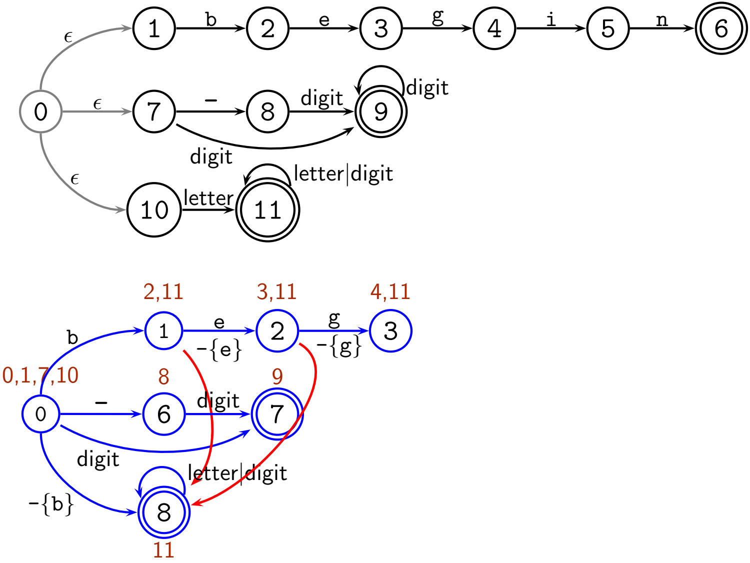


Converting NFA to DFA

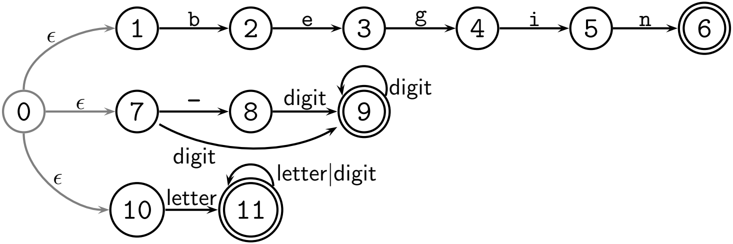


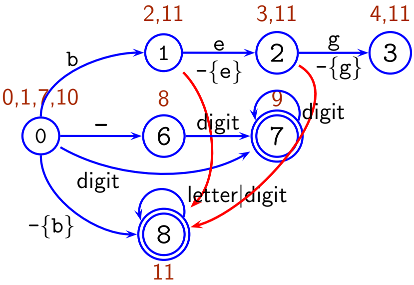


Converting NFA to DFA

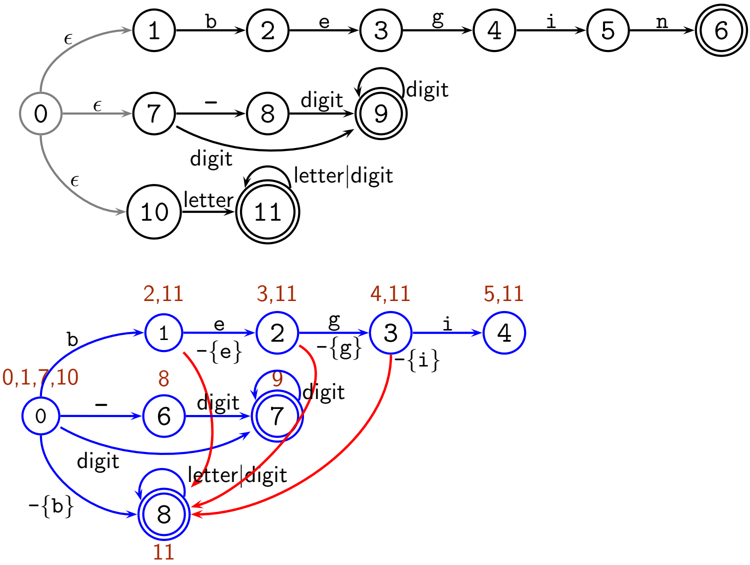


Converting NFA to DFA

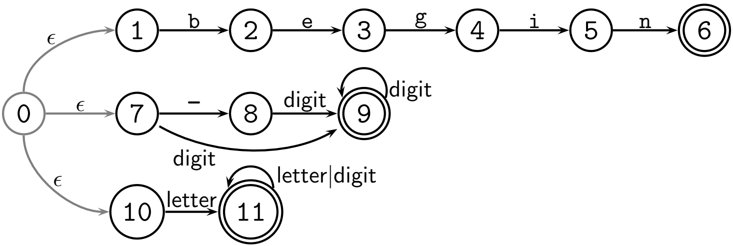


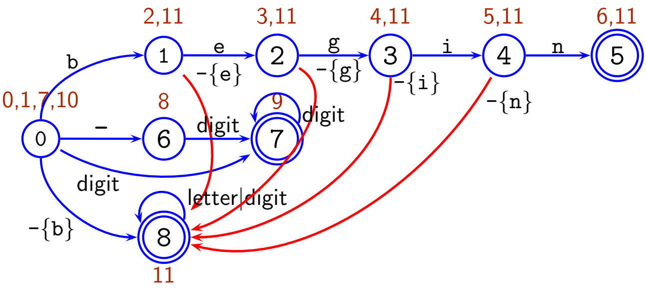


Converting NFA to DFA

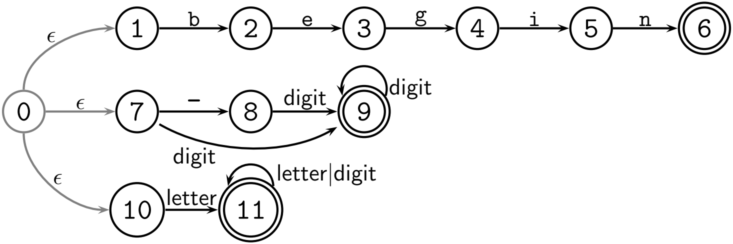


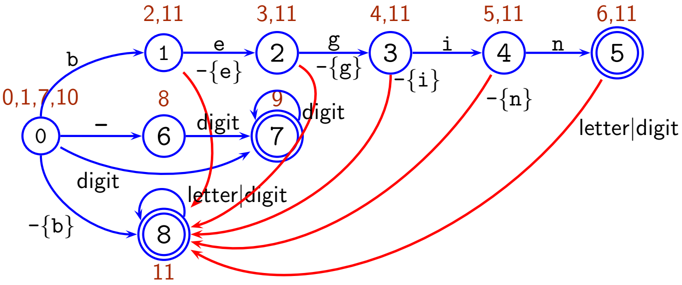
Converting NFA to DFA





Converting NFA to DFA





LEXICAL ERRORS

Primarily of two kinds:

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

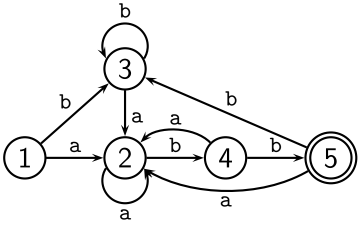
Handling Lexical Errors

issuing an error message, the action taken on detection of an error are:

1. Issue an appropriate error message.
2. - 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.
     - pass the character to the parser which has better knowledge of the context in which error has occurred. more possibilities of recovery - *replacement* instead of *deletion*.
   - Error of the third type – wait till end of file an issue error message.

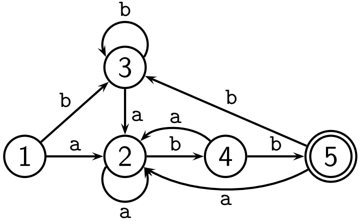
MINIMIZING THE NUMBER OF STATES

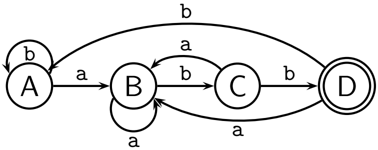
- The DFA constructed for $(b|\epsilon)(a|b)^*abb$.



MINIMIZING THE NUMBER OF STATES

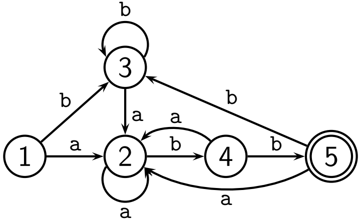
- The DFA constructed for $(b|\epsilon)(a|b)^*abb$.
- There is another DFA for the same regular expression with lesser number of states.

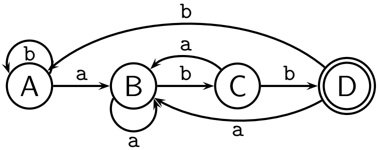




MINIMIZING THE NUMBER OF STATES

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

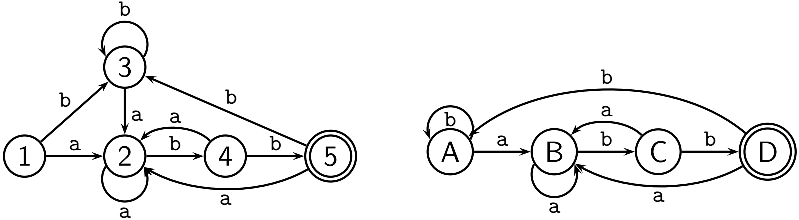




- For a typical language, the number of states of the DFA is in order of hundreds.

MINIMIZING THE NUMBER OF STATES

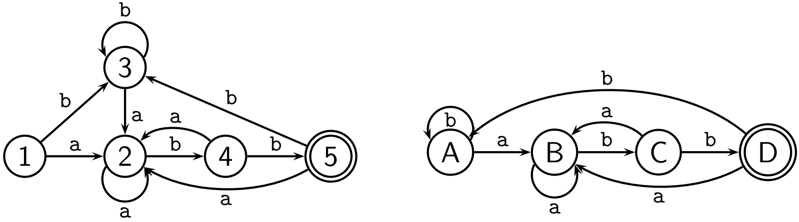
- The DFA constructed for $(b|\epsilon)(a|b)^*abb$.
- There is another DFA for the same regular expression with lesser number of states.



- 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.

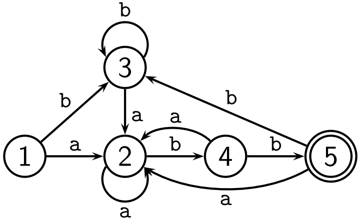
MINIMIZING THE NUMBER OF STATES

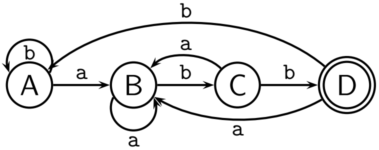
- The second DFA has been obtained by merging states 1 and 3 of the first DFA.



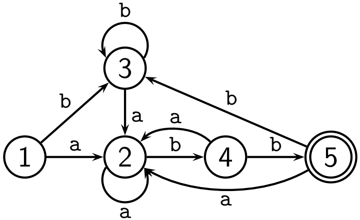
MINIMIZING THE NUMBER OF STATES

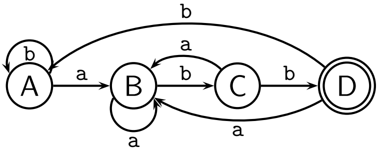
- The second DFA has been obtained by merging states 1 and 3 of the first DFA.
- Under what conditions can this merging take place?





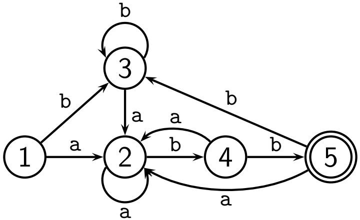
MINIMIZING THE NUMBER OF STATES

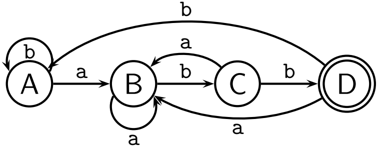




- The string bb takes both states 1 and 3 to a final state.

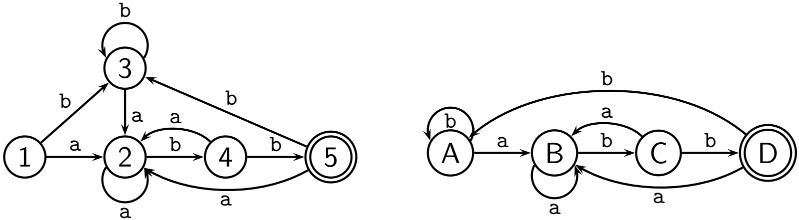
MINIMIZING THE NUMBER OF STATES





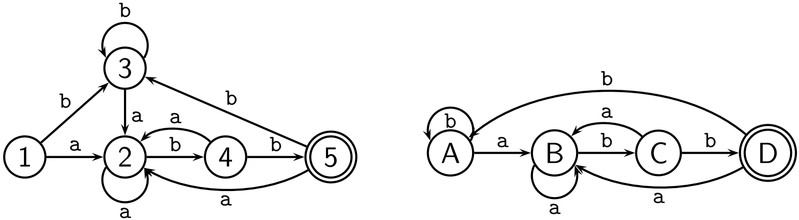
- The string bb takes both states 1 and 3 to a final state.
- The string aba takes both states 1 and 3 to a `non-final state`.

MINIMIZING THE NUMBER OF STATES



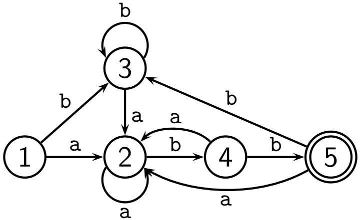
- The string bb takes both states 1 and 3 to a final state.
- The string aba takes both states 1 and 3 to a `non-final state`.
- The string ϵ takes both states 1 and 3 to a non-final state.

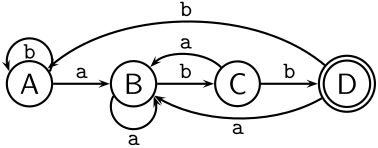
MINIMIZING THE NUMBER OF STATES



- The string bb takes both states 1 and 3 to a 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.

MINIMIZING THE NUMBER OF STATES



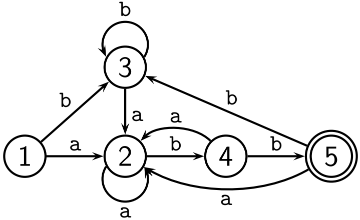


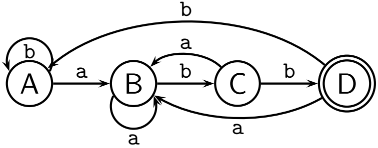
- The string bb takes both states 1 and 3 to a final state.
- The string aba takes both states 1 and 3 to a non-final state.
- The string ϵ takes both states 1 and 3 to a non-final state.
- The string bbabb takes both states 1 and 3 to a final state.

Conclusion:

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

MINIMIZING THE NUMBER OF STATES





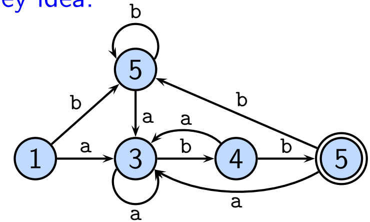
- States 1 and 3 are said to be *indistinguishable*.
- Minimimization strategy:
  - Find indistinguishable states.
  - Merge them.
- Question: How does one find indistingushable states?

MINIMIZING THE NUMBER OF STATES

Key idea:

MINIMIZING THE NUMBER OF STATES

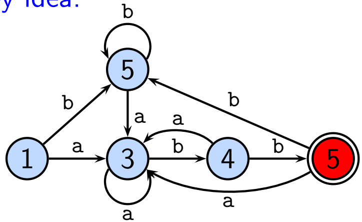
Key idea:



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

MINIMIZING THE NUMBER OF STATES

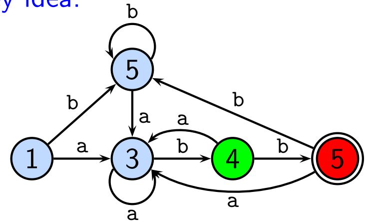
Key idea:



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

MINIMIZING THE NUMBER OF STATES

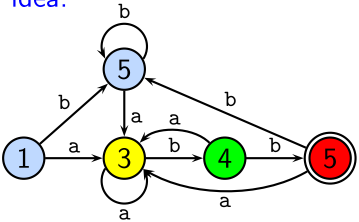
Key idea:



- 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

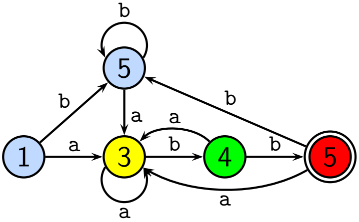
Key idea:

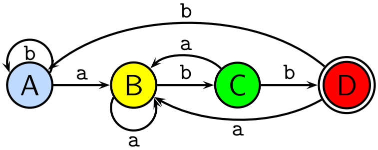


- The string b distinguishes 3 from other states in the blue partition.

MINIMIZING THE NUMBER OF STATES

Key idea:





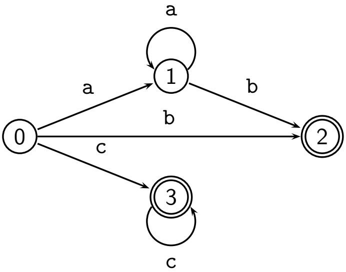
- No other partition possible. Merge all states in the same partition.

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.



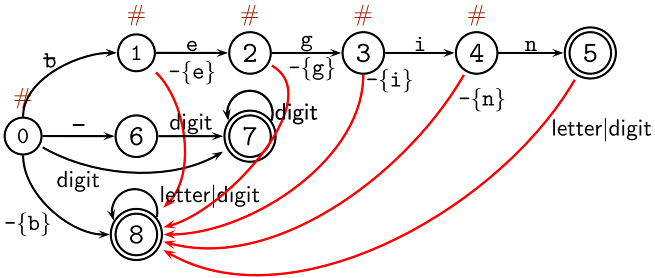
x

	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.

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 one symbol*.

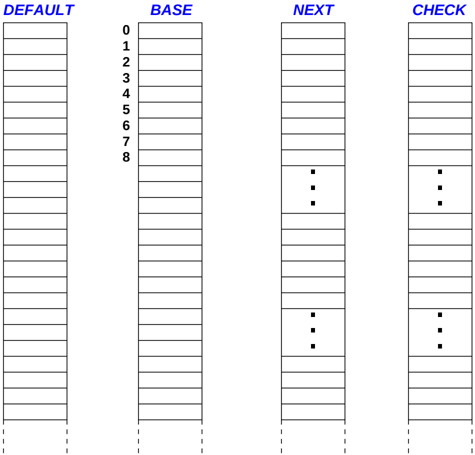


Therefore information about state 8 can also be used for these states.

Four Arrays Representation of DFA

Symbols and their numbering

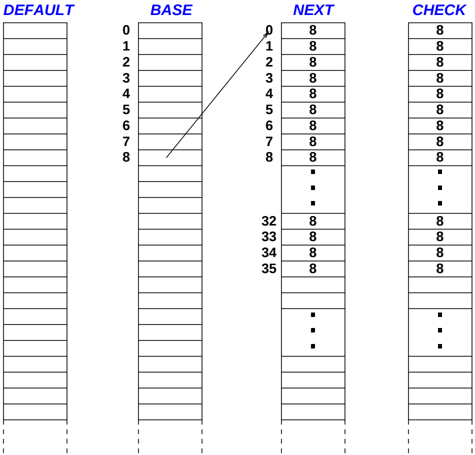
a–z	0–25
0–9	26–35
-	36



Four Arrays Representation of DFA

Symbols and their numbering

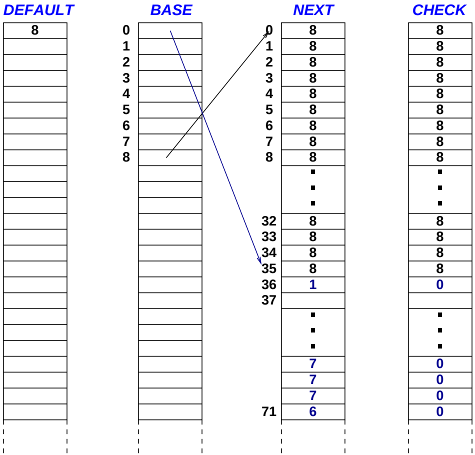
a–z	0–25
0–9	26–35
-	36



Four Arrays Representation of DFA

Symbols and their numbering

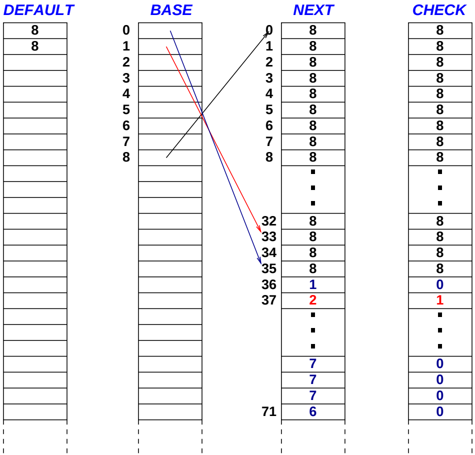
a–z	0–25
0–9	26–35
-	36



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 is different from the corresponding transition on state 8. This differing entry is stored in $NEXT[37]$. Therefore $BASE[1]$ is set to $37 - 4 = 33$.
- By a similar reasoning $BASE[0]$ is set to 36.
- 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.