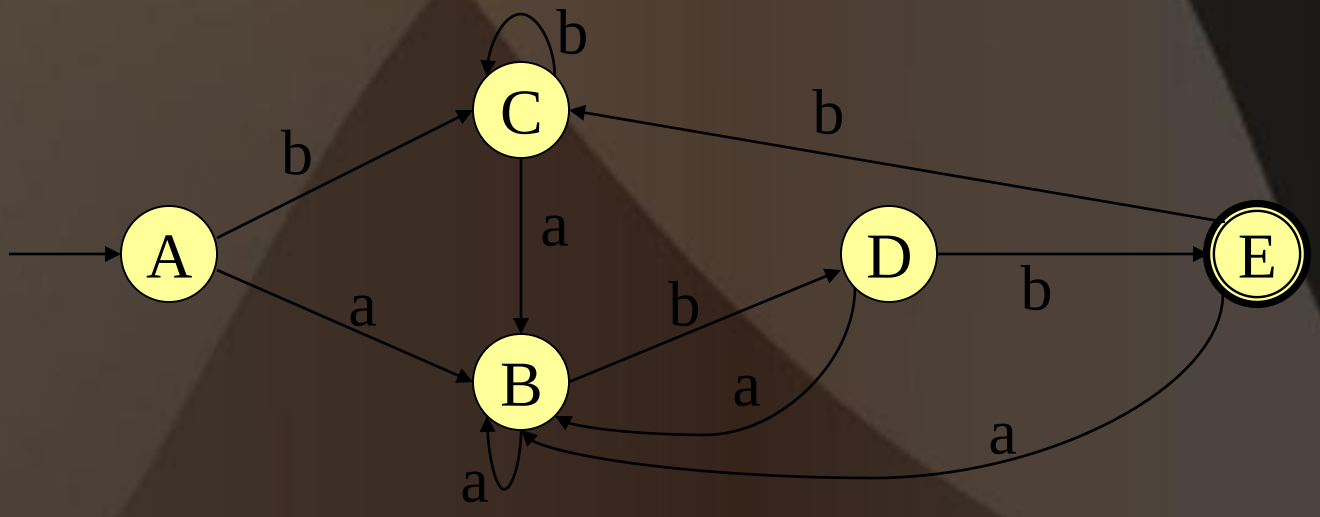
# *Lecture #6*

# Lexical Analysis - III

# *DFA Minimization*

- RE → NFA → DFA → ***Minimized DFA*** → Table-Driven Implementation
- State minimization (Technique 1)
  - Find groups of states where, for each input symbol, every state of such a group will have transitions to the same group

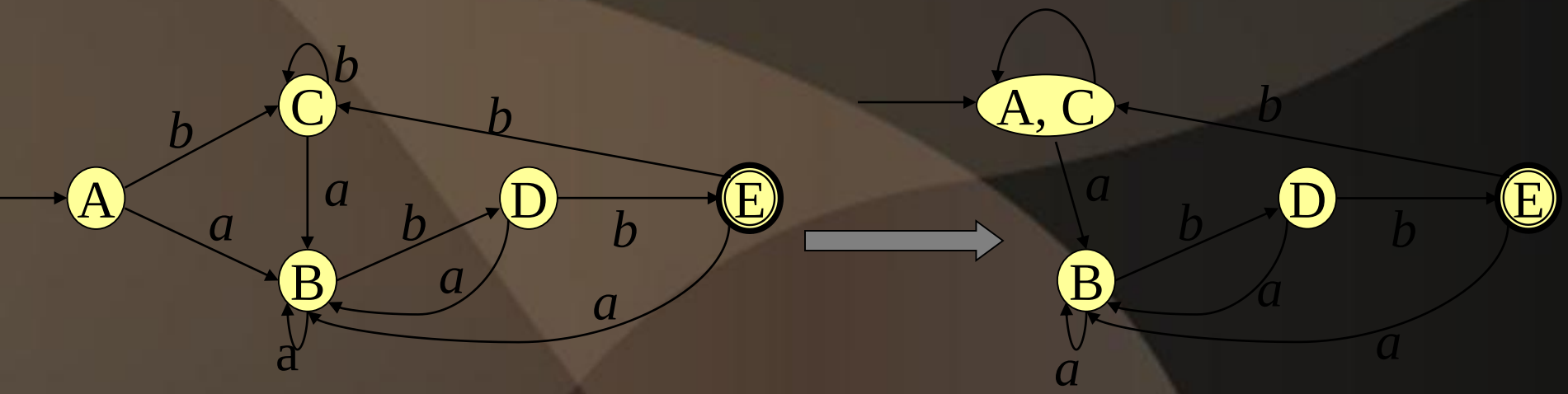


# *DFA Minimization – Technique 1*

Divide the states of the DFA into two groups: those containing final states and those containing non-final states.

**while** there are group changes
    **for each** group
        **for each** input symbol
            **if** for any two states of the group and a given input symbol, their transitions do not lead to the same group, these states must belong to different groups.

# DFA Minimization – Technique 1

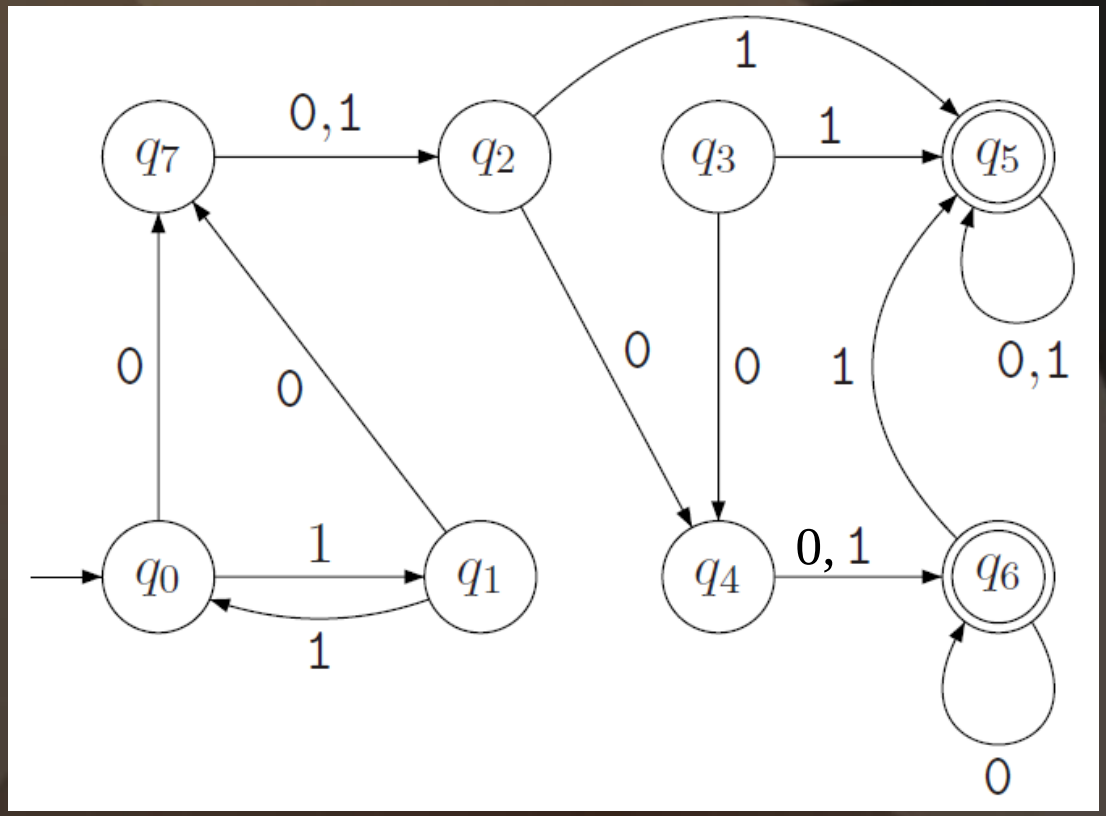



| Iteration | Current groups | Split on a | Split on b |
|---|---|---|---|
| 0 | {E}, {A,B,C,D} | None | {A,B,C}, {D} |
| 1 | {E}, {D}, {A,B,C} | None | {A,C}, {B} |
| 2 | {E}, {D}, {B}, {A, C} | None | None |

***Generate minimized DFA for the RE: (a | b)* c***

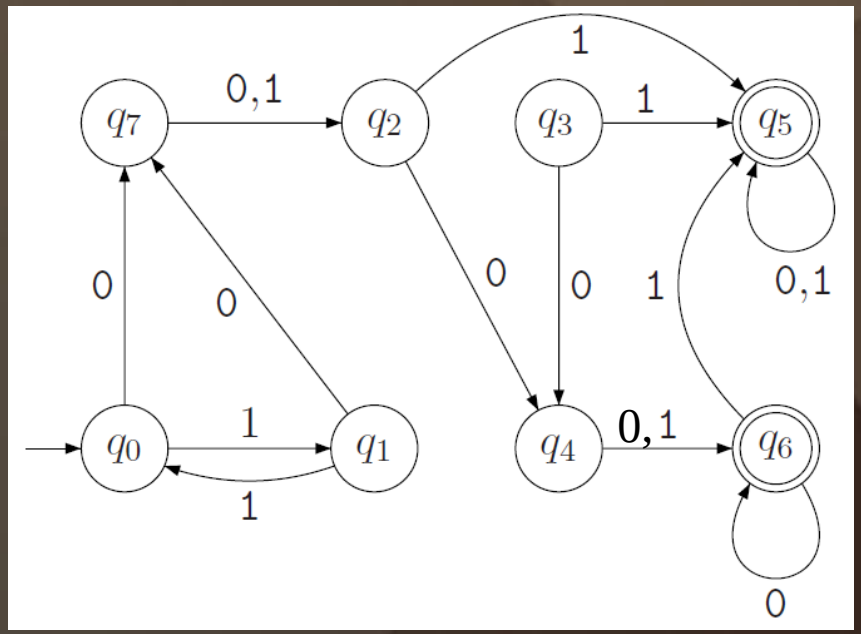
# *DFA Minimization –Technique 1*

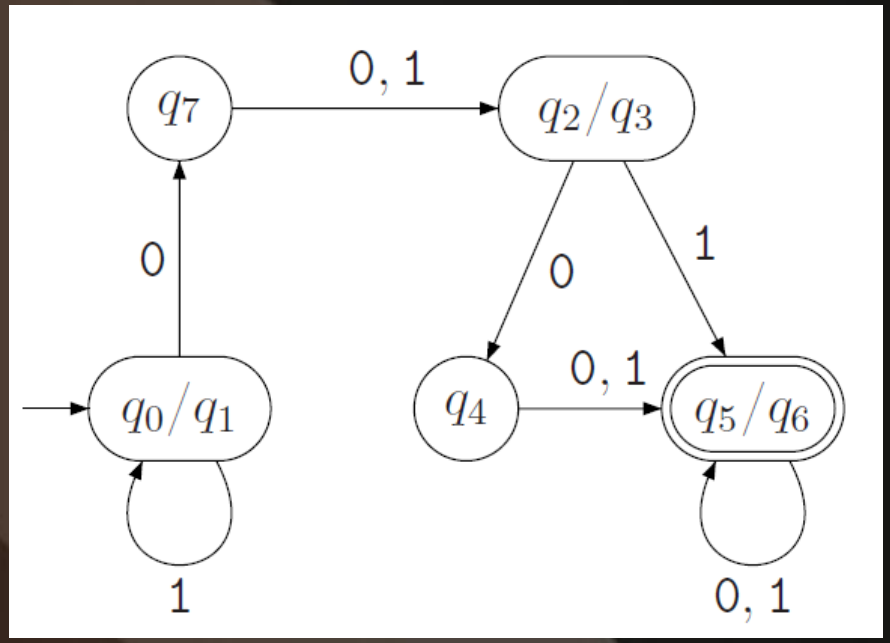
Apply the minimisation algorithm to produce the minimal DFA:



# *DFA Minimization –Technique 1*

Apply the minimisation algorithm to produce the minimal DFA:

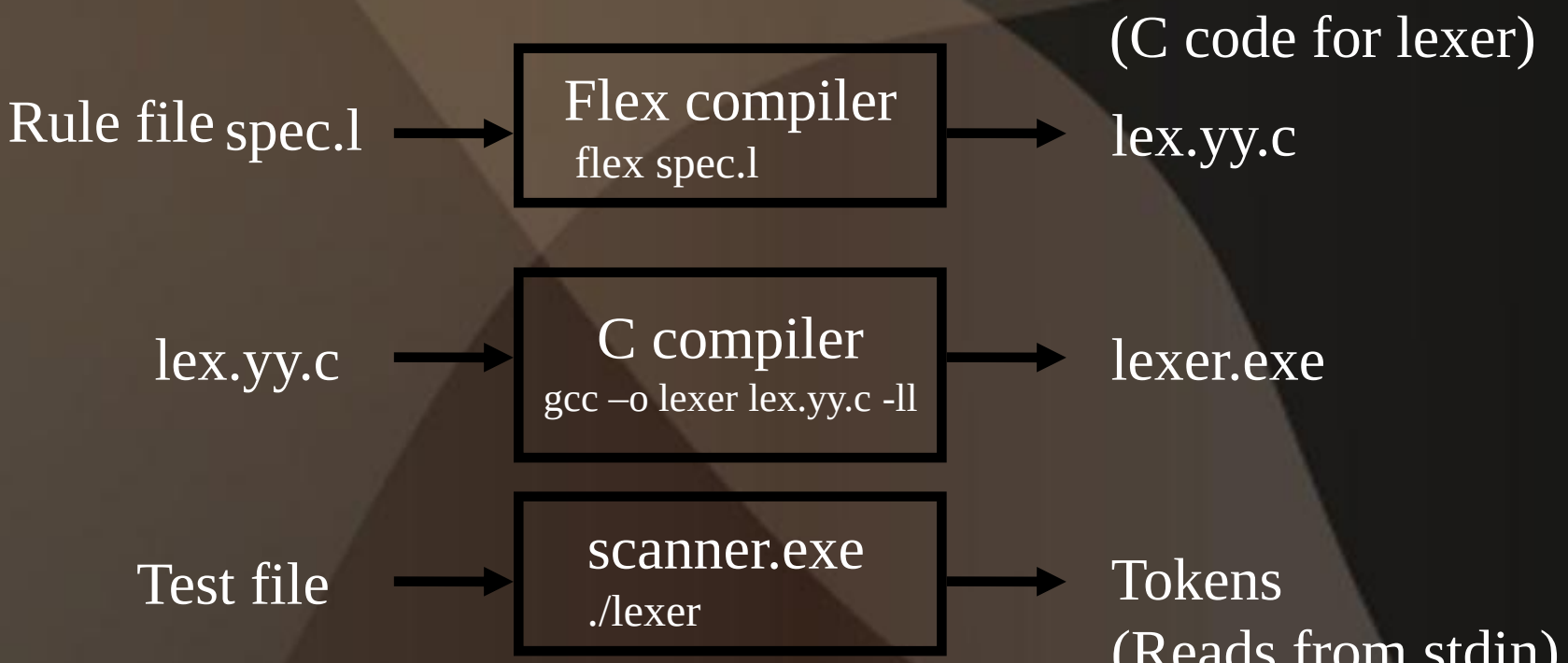


# *Flex*

- A Lexical Analyzer Generator
  - Generates a C scanner program directly, given the *regular expressions to match* and *C code for actions on each token*
  - Creates combined NFA for all patterns, converts it to a DFA, minimizes the DFA and generates C code to implement it
- Steps to using flex
  - Create a description or rules file for flex to operate on .
  - Run flex on the input file. Flex produces a C file called lex.yy.c with the scanning function yylex().
  - Run the C compiler on the C file to produce a lexical analyzer

# *Flex – Execution Steps*



# *Flex – Input File Structure*

- Flex input file consists of three sections separated by lines with %%

| |
|---|
| %{<br>Declarations<br>%}<br>Definitions |
| %%<br>Rules |
| %%<br>User subroutines |

- *Declarations and user subroutines (Optional)*: Ordinary C code, just copied to generated C file.
- *Definitions (Optional)*: options for scanner; give names to regular expressions for substitution in *rules*.
- *Rules (Required)*: Specify patterns to identify tokens and actions to perform upon recognizing each token

# *The First Flex Program*

- Prog1.l

```
%%
"My First Program" printf("WELL DONE\n");
.                  ;
%%
```

# *Flex Pattern Examples*

| | |
|---|---|
| abc | Match the string "abc" |
| [a-zA-Z] | Match any lower or uppercase letter. |
| dog.*cat | Match any string starting with dog, and ending with cat. |
| (ab)+ | Match one or more occurrences of "ab" concatenated. |
| [^a-z]+ | Matches any string of one or more characters that do not include lower case a-z. |
| [+-]?[0-9]+ | Match any string of one or more digits with an optional prefix of + or -. |

# *Definitions Section*

- The definitions section contains declarations of simple name definitions to simplify the scanner specification.
- Name definitions have the form:
  - `name definition`
  - Example:
    - `DIGIT    [0-9]`
    - `ID       [a-z][a-z0-9]*`

# *Rules Section*

$P_1$ $action_1$
$P_2$ $action_2$
...
$P_n$ $action_n$

where $P_i$ are regular expressions and $action_i$ are C program segments

# *Functions and Variables*

- yylex()
  - A function implementing the lexical analyzer and returning the token matched
- yytext
  - A global pointer variable pointing to the lexeme matched
- yyleng
  - A global variable giving the length of the lexeme matched
- yylineno
  - A global variable giving the current line number in the input file

# *Example 2*

```
%%
[0-9]+      printf (“?”);
.           ECHO;
```

%% marks the beginning of rules section
The “.” matches any single character
“ECHO” just prints the character unchanged

```
$ flex prog1.l
$ gcc –o prog1 lex.yy.c –ll
$ ./prog1
```

# *Example 2*

```
%{
   int numChars = 0, numWords = 0, numLines = 0;
%}
%%
\n              {numLines++; numChars++;}
[^ \t\n]+       {numWords++; numChars += yyleng;}
.               {numChars++;}
%%
int main(){
        yylex();
        printf("%d\t%d\t%d\n", numChars, numWords, numLines);
}

$ flex prog2.l
$ gcc -o prog2 lex.yy.c -ll
$ ./prog2 < prog2.l
```

# *Next Lecture*

Parsing