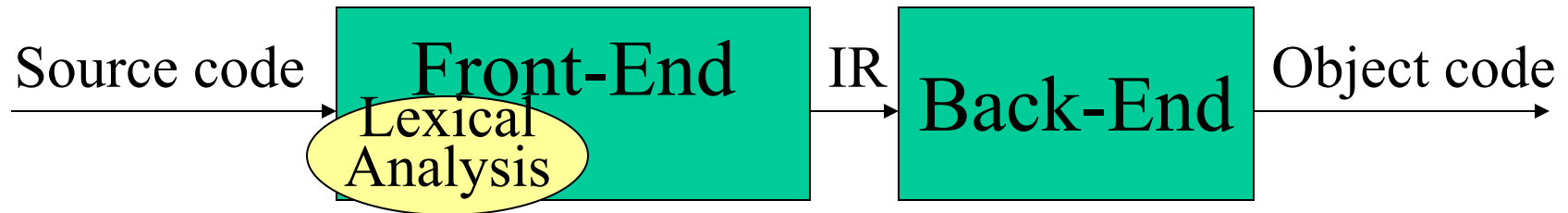


# Lecture 5: Lexical Analysis III: The final bits



(from the last 2 lectures) Lexical Analysis:

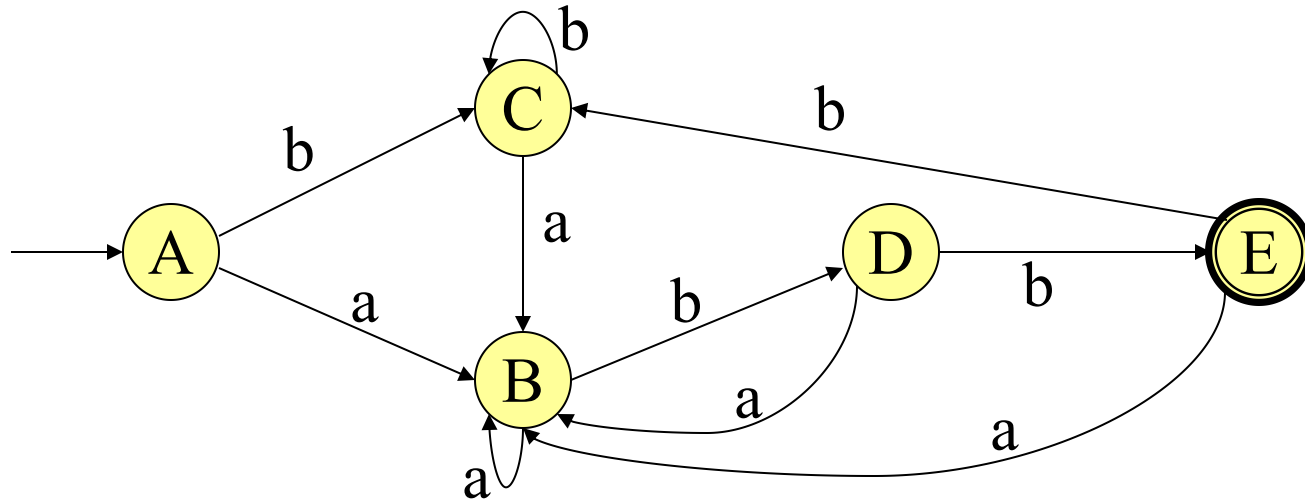
- Regular Expressions (REs) are formulae to describe a (regular) language.
- Every RE can be converted to a Deterministic Finite Automaton (DFA).
- DFAs can automate the construction of lexical analysers.
- Algorithms to construct a DFA from a RE (through a NFA) were given.

Today's lecture:

How to minimise the (resulting) DFA

Practical Considerations; lex; wrap-up

# DFA Minimisation: the problem



- Problem: can we minimise the number of states?
- Answer: yes, if we can find groups of states where, for each input symbol, every state of such a group will have transitions to the same group.

# DFA minimisation: the algorithm

(Hopcroft's algorithm: simple version)

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

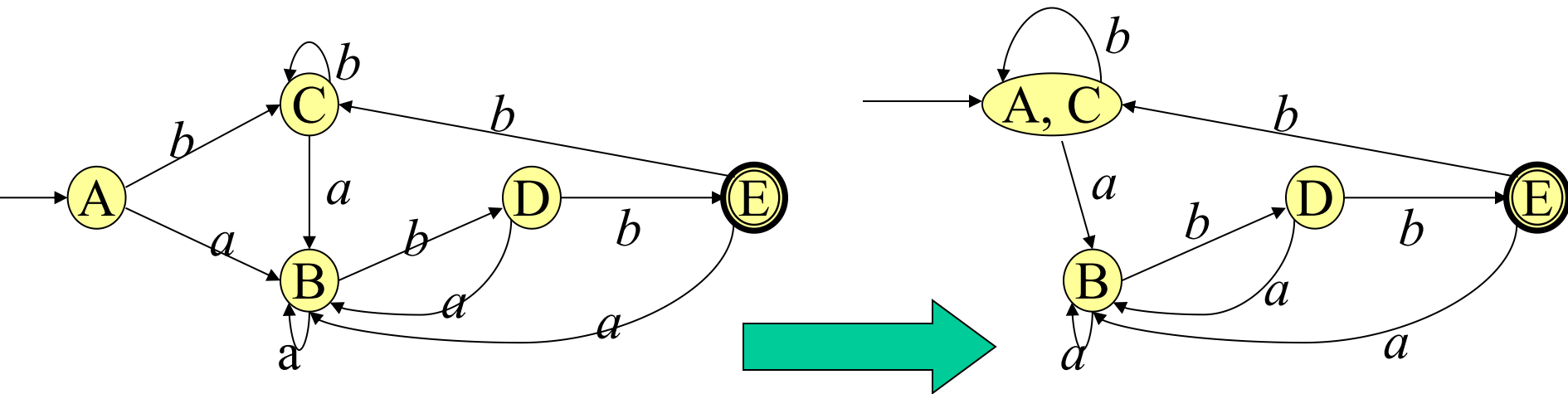
**if** for any two states of the group and a given input symbol, their transitions do not lead to the same group, these states must belong to different groups.

For the curious, there is an alternative approach: create a graph in which there is an edge between each pair of states which cannot coexist in a group because of the conflict above. Then use a graph colouring algorithm to find the minimum number of colours needed so that any two nodes connected by an edge do not have the same colour (we'll examine graph colouring algorithms later on, in register allocation)

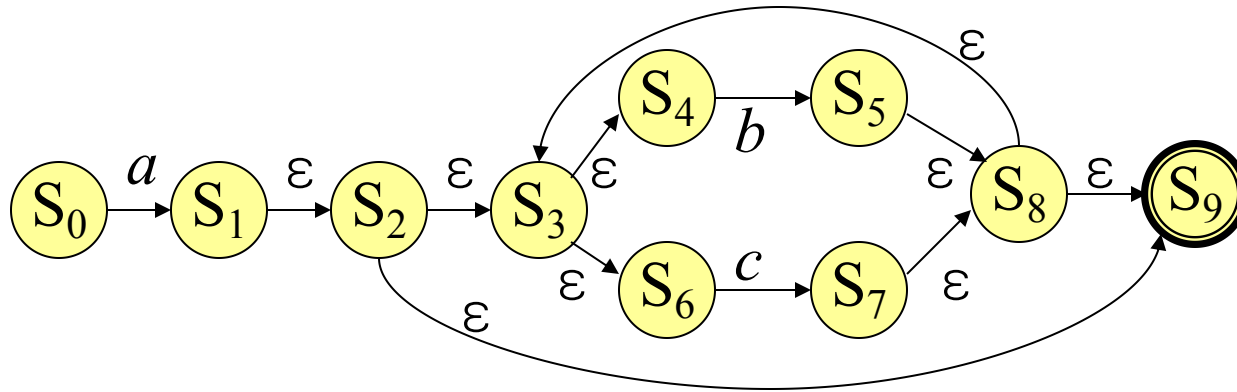
# How does it work? Recall $(a \mid b)^* abb$

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

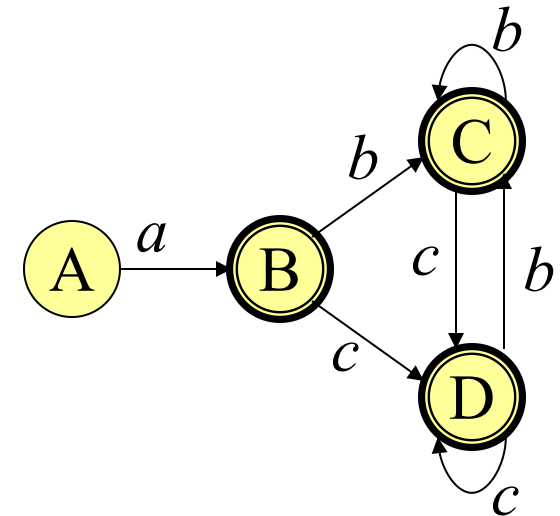
In each iteration, we consider any non-single-member groups and we consider the partitioning criterion for all pairs of states in the group.



# From NFA to minimized DFA: recall $a(b \mid c)^*$



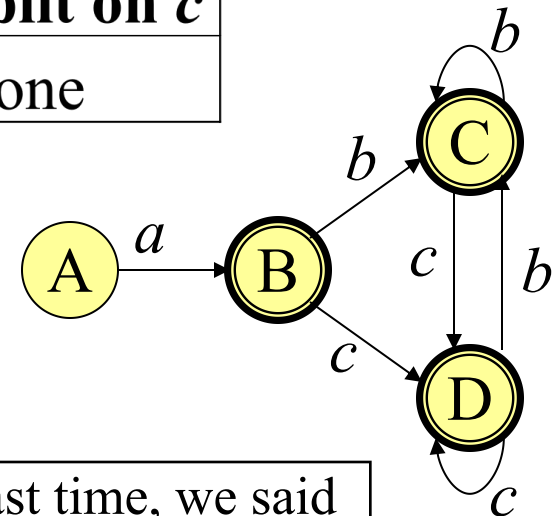
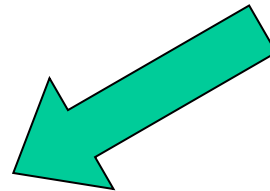
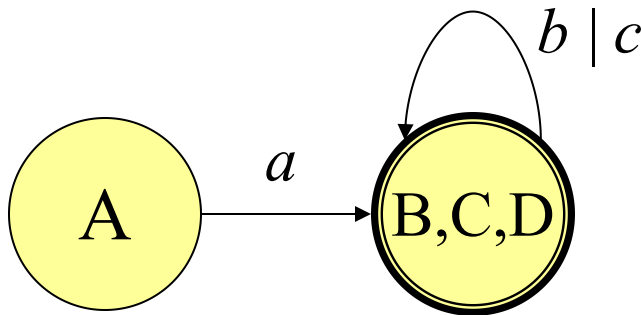
DFA states	NFA states	$\epsilon$ -closure(move(s,*))		
		<i>a</i>	<i>b</i>	<i>c</i>
A	S0	S1,S2,S3, S4,S6,S9	None	None
B	S1,S2,S3, S4,S6,S9	None	S5,S8,S9, S3,S4,S6	S7,S8,S9, S3,S4,S6
C	S5,S8,S9, S3,S4,S6	None	S5,S8,S9, S3,S4,S6	S7,S8,S9, S3,S4,S6
D	S7,S8,S9, S3,S4,S6	None	S5,S8,S9, S3,S4,S6	S7,S8,S9, S3,S4,S6



# DFA minimisation: recall $a(b \mid c)^*$

Apply the minimisation algorithm to produce the minimal DFA:

	Current groups	Split on $a$	Split on $b$	Split on $c$
0	{B, C, D} {A}	None	None	None



Remember, last time, we said that a human could construct a simpler automaton than Thompson's construction? Well, algorithms can produce the same DFA!

# Building fast scanners...

From a RE to a DFA; why have we been doing all this?

- If we know the transition table and the final state(s) we can build directly a recogniser that detects acceptance (recall from Lecture 3):

```
char=input_char();
state=0; // starting state
while (char != EOF) {
    state ← table(state,char);
    if (state == '-') return failure;
    word=word+char;
    char=input_char();
}
if (state == FINAL) return acceptance; else return failure;
```

But, table-driven recognisers can waste a lot of effort.

- Can we do better? Encode states and actions in the code.

How to deal with reserved keywords?

- Some compilers recognise keywords as identifiers and check them in a table.
- Simpler to include them as REs in the lexical analyser's specification.

# Direct coding from the transition table

Recall *Register*  $\rightarrow r \text{ digit digit}^*$  (Lecture 3, slides 10-11)

```
    word='' ; char='' ; goto s0;
s0: word=word+char;
    char=input_char();
    if (char == 'r') then goto s1 else goto error;
s1: word=word+char;
    char=input_char();
    if ('0' ≤ char ≤ '9') then goto s2 else goto error;
s2: word=word+char;
    char=input_char();
    if ('0' ≤ char ≤ '9') then goto s2
        else if (char== EOF) return acceptance
        else goto error
error: print "error"; return failure;
```

- fewer operations
- avoids memory operations (esp. important for large tables)
- added complexity may make the code ugly and difficult to understand



# Practical considerations

- Poor language design may complicate lexical analysis:
  - **if then then = else; else else = then** (PL/I)
  - **DO5I=1,25** vs **DO5I=1.25** (Fortran: urban legend has it that an error like this caused a crash of an early NASA mission)
  - the development of a sound theoretical basis has influenced language design positively.
- Template syntax in C++:
  - **aaaa<mytype>**
  - **aaaa<mytype<int>>** (>> is an operator for writing to the output stream)
  - The lexical analyser treats the >> operator as two consecutive > symbols. The confusion will be resolved by the parser (by matching the <, >)

# Lex/Flex: Generating Lexical Analysers

Flex is a tool for generating scanners: programs which recognised lexical patterns in text (from the online manual pages: `% man flex`)

- Lex input consists of 3 sections:
  - regular expressions;
  - pairs of regular expressions and C code;
  - auxiliary C code.
- When the lex input is compiled, it generates as output a C source file `lex.yy.c` that contains a routine `yylex()`. After compiling the C file, the executable will start isolating tokens from the input according to the regular expressions, and, for each token, will execute the code associated with it. The array `char yytext[]` contains the representation of a token.

# flex Example

```
%{
#define ERROR -1
int line_number=1;
%}
whitespace      [ \t]
letter          [a-zA-Z]
digit           [0-9]
integer         ({digit}+)
l_or_d          ({letter}|{digit})
identifier      ({letter}{l_or_d}*)
operator        [-+*/]
separator       [;,(){}]
%%
{integer}       {return 1;}
{identifier}    {return 2;}
{operator}|{separator} {return (int)yytext[0];}
{whitespace}    {}
\n              {line_number++;}
.               {return ERROR;}
%%
int yywrap(void) {return 1;}
int main() {
    int token;
    yyin=fopen("myfile","r");
    while ((token=yylex())!=0)
        printf("%d %s \n", token, yytext);
    printf("lines %d \n",line_number);
}
```

Input file ("myfile")

123+435+34=aaaa

329\*45/a-34\*(45+23)\*\*3

bye-bye

Output:

1 123

43 +

1 435

43 +

1 34

-1 =

2 aaaa

1 329

42 \*

1 45

47 /

2 a

45 -

1 34

42 \*

40 (

1 45

43 +

1 23

41 )

42 \*

42 \*

1 3

2 bye

45 -

2 bye

lines 4

# Conclusion

- Lexical Analysis turns a stream of characters in to a stream of tokens:
  - a largely automatic process.
    - REs are powerful enough to specify scanners.
    - DFAs have good properties for an implementation.
- Next time: Introduction to Parsing
- Reading: Aho2 3.9.6, 3.5 (lex); Aho1, pp. 141-144, 105-111 (lex); Grune pp.86-96; Hunter pp. 53-62; Cooper1 pp.55-72.
- Exercises: Aho2, p.166, p.186; Aho1 pp.146-150; Hunter pp. 71; Grune pp.185-187.