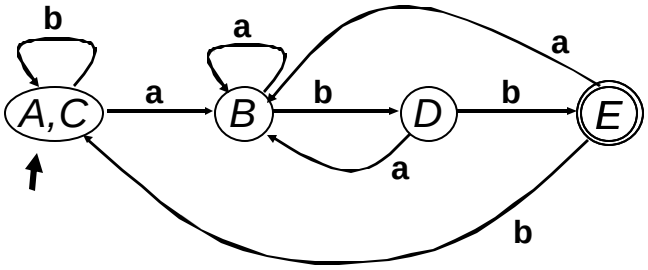
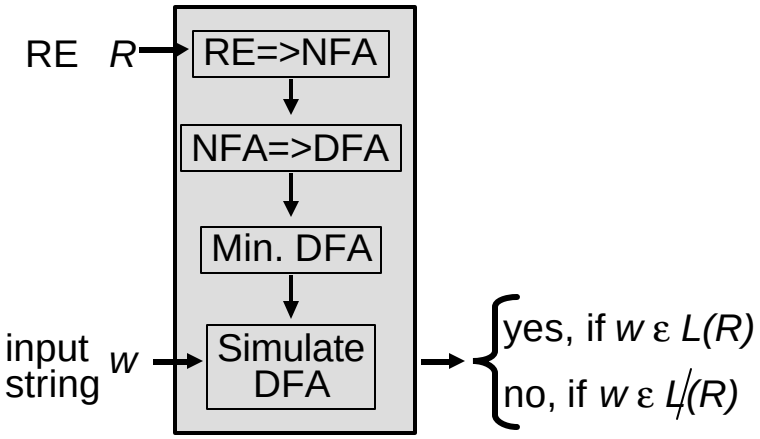
Lecture 9

DFA Minimization

The generated DFA may have a large number of states. The Hopcroft's algorithm can be used to minimize DFA states. The behind the algorithm is to find groups of *equivalent states*. All transitions from states in one group G_1 go to states in the same group G_2. Construct the minimized DFA such that there is one state for each group of states from the initial DFA. Here is the minimized version of the DFA created earlier; states A and C have been merged.



We can construct an optimized acceptor with the following structure:



Lexical Analyzers

Lexical analyzers (scanners) use the same mechanism but they have multiple RE descriptions for multiple tokens and have a character stream at the input. The lexical analyzer returns a sequence of matching tokens at the output (or an error) and it always return the longest matching token.

Lexical Analyzer Generators

The process of constructing a lexical analyzer can automated. We only need to specify Regular expressions for tokens and rules for assigning priorities for multiple longest match cases, e.g, "==" and "=", "==" is longer.

Two popular lexical analyzer generators are

- **Flex :** generates lexical analyzer in C or C++. It is more modern version of the original Lex tool that was part of the AT&T Bell Labs version of Unix.
- **Jlex:** written in Java. Generates lexical analyzer in Java

Using Flex

We will use for the projects in this course. To use Flex, one has to provide a specification file as input to Flex. Flex reads this file and produces an output file contains the lexical analyzer source in C or C++.

The input specification file consists of three sections:

C or C++ and flex definitions
%%
token definitions and actions
%%
user code

The symbols "%%" mark each section. A detailed guide to Flex is included in supplementary reading material for this course. We will go through a simple example.

The following is the Flex specification file for recognizing tokens found in a C++ function. The file is named "`lex.l`"; it is customary to use the ".l" extension for Flex input files.

```
%{
#include "tokdefs.h"
%}
D          [0-9]
L          [a-zA-Z_]
id   {L}({L}|{D})*
%%
"void" {return(TOK_VOID);}
"int"  {return(TOK_INT);}
"if"   {return(TOK_IF);}
Specification File lex.l
"else" {return(TOK_ELSE);}
"while"{return(TOK_WHILE)};
"<="   {return(TOK_LE);}
```

```
">="   {return(TOK_GE);}
"=="   {return(TOK_EQ);}
"!="   {return(TOK_NE);}
{D}+   {return(TOK_INT);}
{id}   {return(TOK_ID);}
[\n]|[\t]|[ ]  ;
%%
```

The file `lex.l` includes another file named "`tokdefs.h`". The content of `tokdefs.h` are

```
#define TOK_VOID 1
#define TOK_INT 2
#define TOK_IF 3
#define TOK_ELSE 4
#define TOK_WHILE 5
#define TOK_LE 6
#define TOK_GE 7
#define TOK_EQ 8
#define TOK_NE 9
#define TOK_INT 10
#define TOK_ID 111
```

Flex creates C++ classes that implement the lexical analyzer. The code for these classes is placed in the Flex's output file. Here, for example, is the code needed to invoke the scanner; this is placed in `main.cpp`:

```
void main()
{
     FlexLexer lex;
     int tc = lex.yylex();
     while(tc != 0) {
          cout << tc << ",” <<lex.YYText() << endl;
          tc = lex.yylex();
     }
}
```

The following commands can be used to generate a scanner executable file in windows.

```
flex lex.l
g++ –c lex.cpp
g++ –c main.cpp
g++ –o lex.exe lex.o main.o
```