More on scanning: NFAs and Flex

Last time

- Scanners: the first step in compilation
  - Divides the program into tokens, or smallest meaningful units
  - This makes later parsing much simpler
- Theory end of things: tokenizing is equivalent to specifying a DFA, which recognizes a regular language

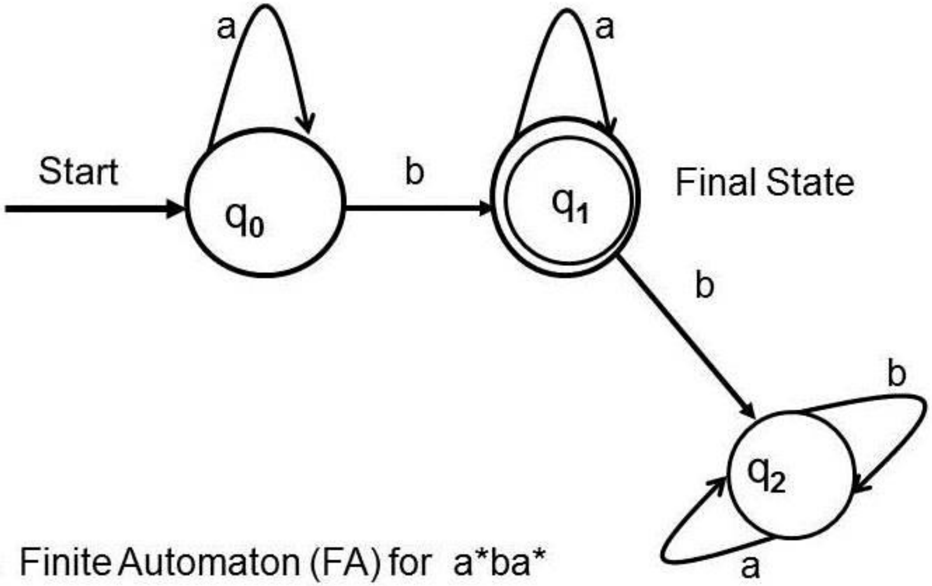
Scanning with Regular Languages

- Unsigned integers in Pascal:
  - Examples: 4, or 82.3, or 5.23e-26
- Formally:

$$
\begin{aligned}
\textit{digit} &\longrightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
\textit{unsigned\_integer} &\longrightarrow \textit{digit}\ \textit{digit}\,^* \\
\textit{unsigned\_number} &\longrightarrow \textit{unsigned\_integer}\ ((\ .\ \textit{unsigned\_integer}) \mid \epsilon) \\
&\qquad (((\texttt{e} \mid \texttt{E})\ (\texttt{+} \mid \texttt{-} \mid \epsilon)\ \textit{unsigned\_integer}) \mid \epsilon)
\end{aligned}
$$

DFAs

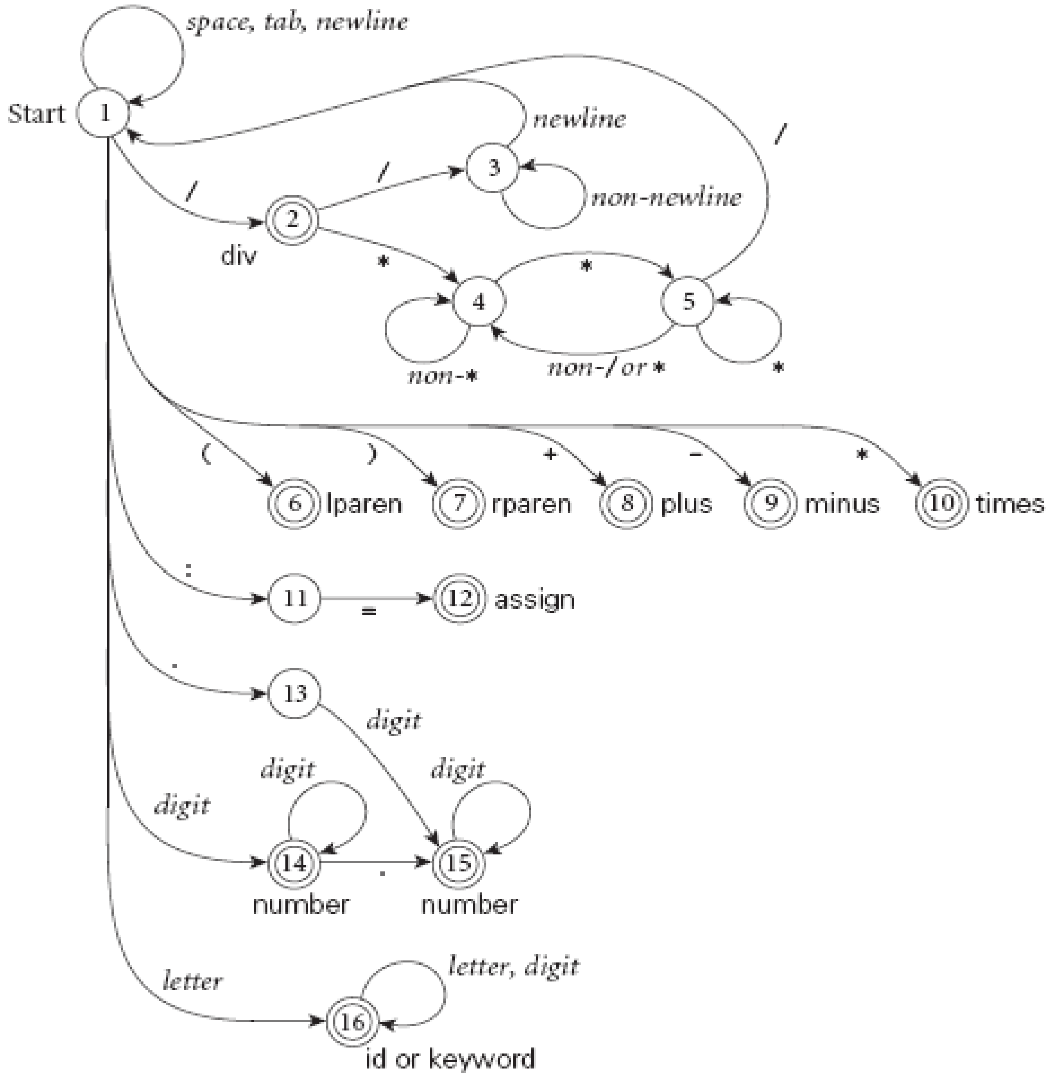
- More often, we'll just draw a picture (like in graph theory)
- Example:



A Finite Automaton (FA) for a*ba*

Scanning with DFAs

- Pictorial representation of a scanner for calculator tokens, in the form of a finite automaton



Regular expression recap

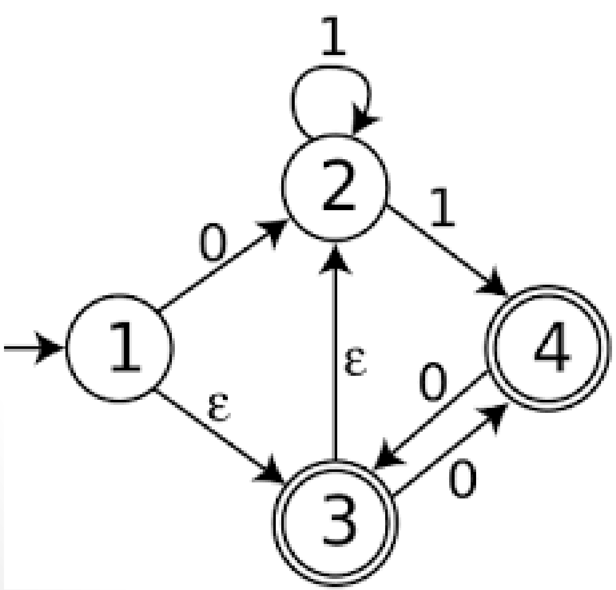
- Write a DFA that recognizes any 0,1 string that has the number of 0’s in the string equal to 0 mod 3:

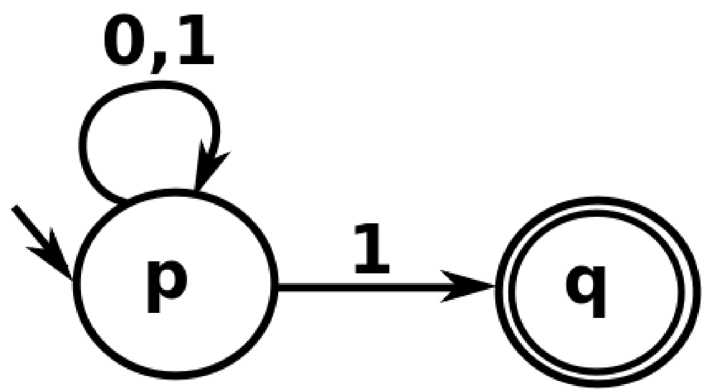
NFAs

- Nondeterministic finite automata (NFA) are a variant of DFAs.
- NFAs allow for ambiguity:
  - If a character is read, there can be multiple arrows showing where to go
  - Empty string transitions allow state transitions without reading an input character
- DFAs have no ambiguity- must have exactly one transition for each input character in each state

NFA Examples

- NFAs accept a string when there is *any* path to an accept state.
- What do the following NFAs accept?





More NFAs

- Some things are easier with NFAs than DFAs:

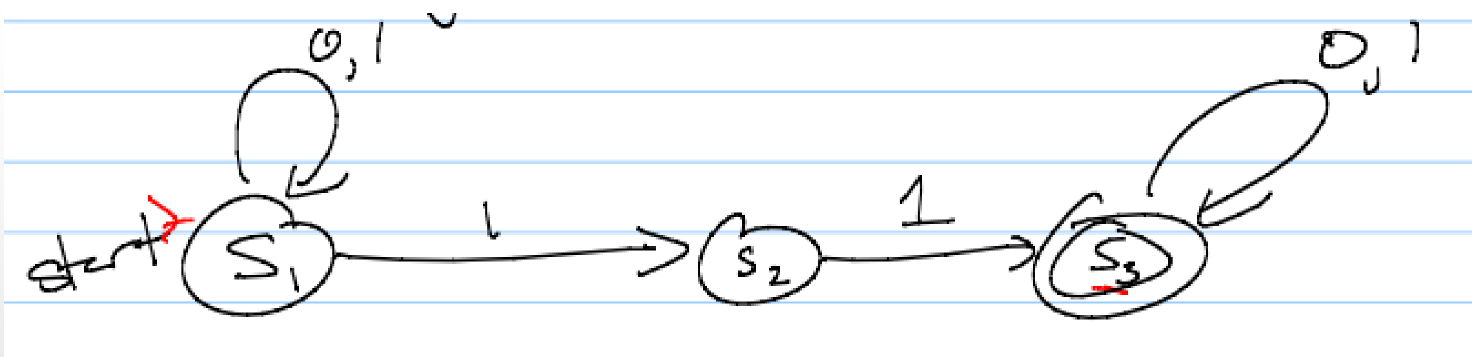
```
unsigned_number ->
     unsigned_int (ε|.unsigned_int)


unsigned_int -> [0-9]
```

Are NFAs more powerful?

- DFAs take exactly one transition each input
- We can think of an NFA as modeling “all possible” transitions simultaneously
- Can NFAs describe more languages than DFAs? E.g. could they recognize the language of balanced parentheses?
- Theorem: Every NFA has an equivalent DFA.
- Both just recognize the regular languages, even though NFAs seem more powerful!

Converting NFAs to DFAs

- To convert, mimic set of possible states given an input
- A state is an accept state if any state in it is an accept state – that means the string could have ended in an accept state, and so is in the language



Why do we care?

- You may ask: why do we care about NFAs?
- Well, in terms of defining a parser, we usually start with regular expressions.
- We then need a DFA (since NFAs are harder to code).
  - However, getting from a regular expression to a DFA in one step is difficult.
  - Instead, programs convert to an NFA, and THEN to a DFA.
  - Somewhat un-intuitively, this winds up being easier to code.

Constructing NFAs

- The construction process for NFAs is pretty easy.
- Recall how a regular expression is defined:
  - A single character or ε
  - Concatenation
  - An "or"
  - Kleene star
- So all we need to do is show how to do each of these in an NFA (and how to combine them)

Constructing NFAs

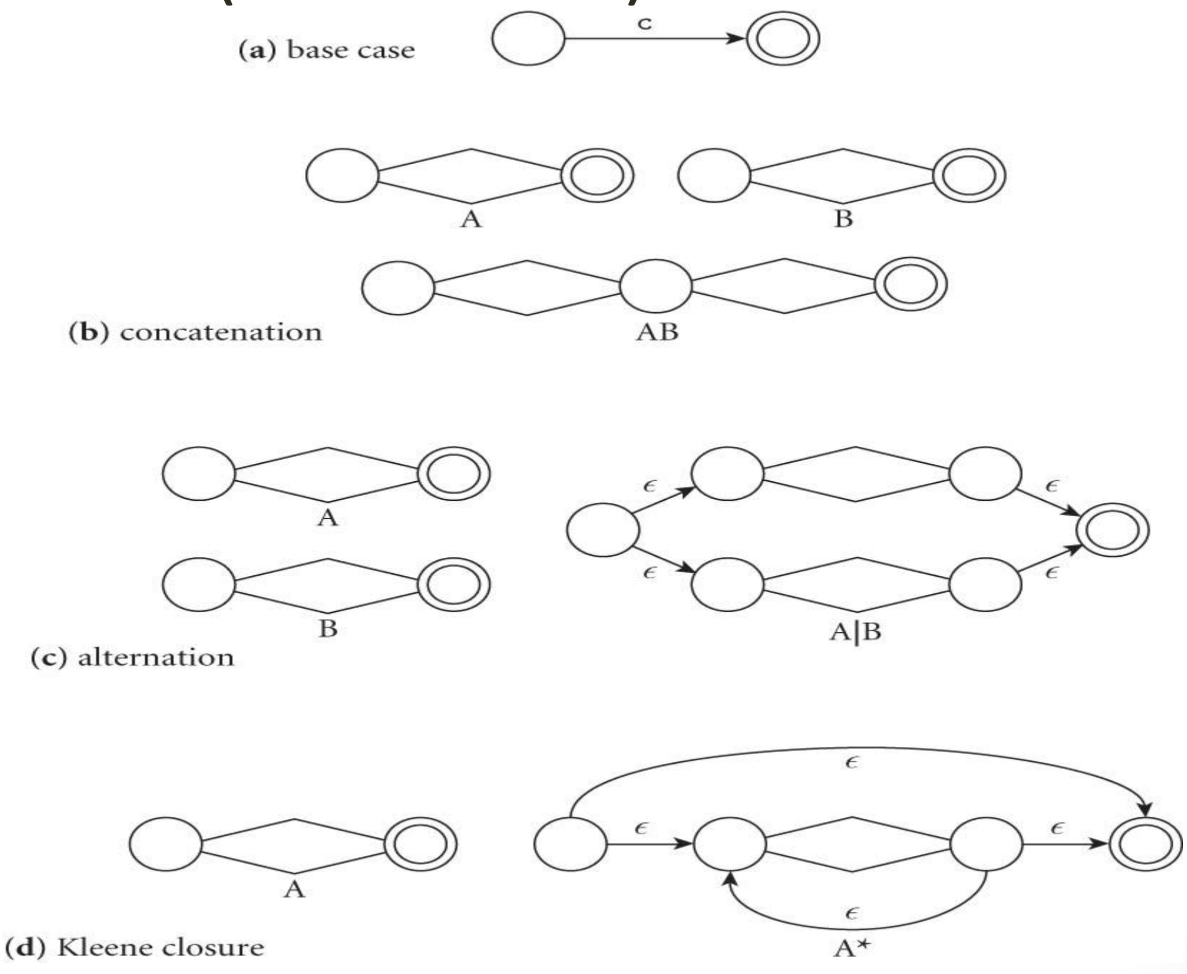
- Easy first step: What is the NFA for a single character, or for the empty string?
- Now: what if I have NFAs for 2 regular expressions, and want to concatenate?

Constructing NFAs

- A bit harder: what about an “or” or Kleene star?

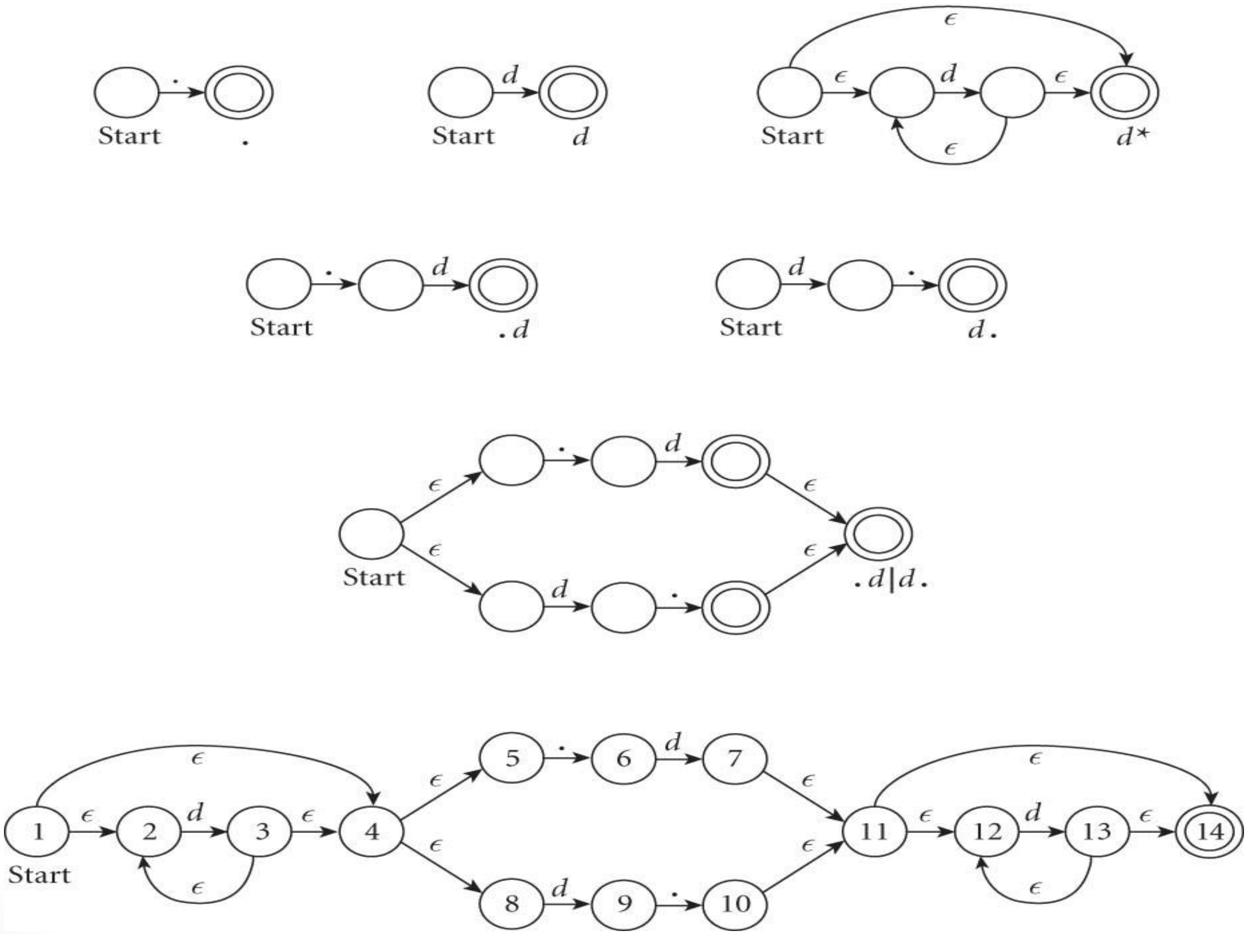
Constructing NFAs

- Final picture (2.7 in book):



An example: decimals

- Let d = [0-9], then decimals are: d* (.d | d.) d*



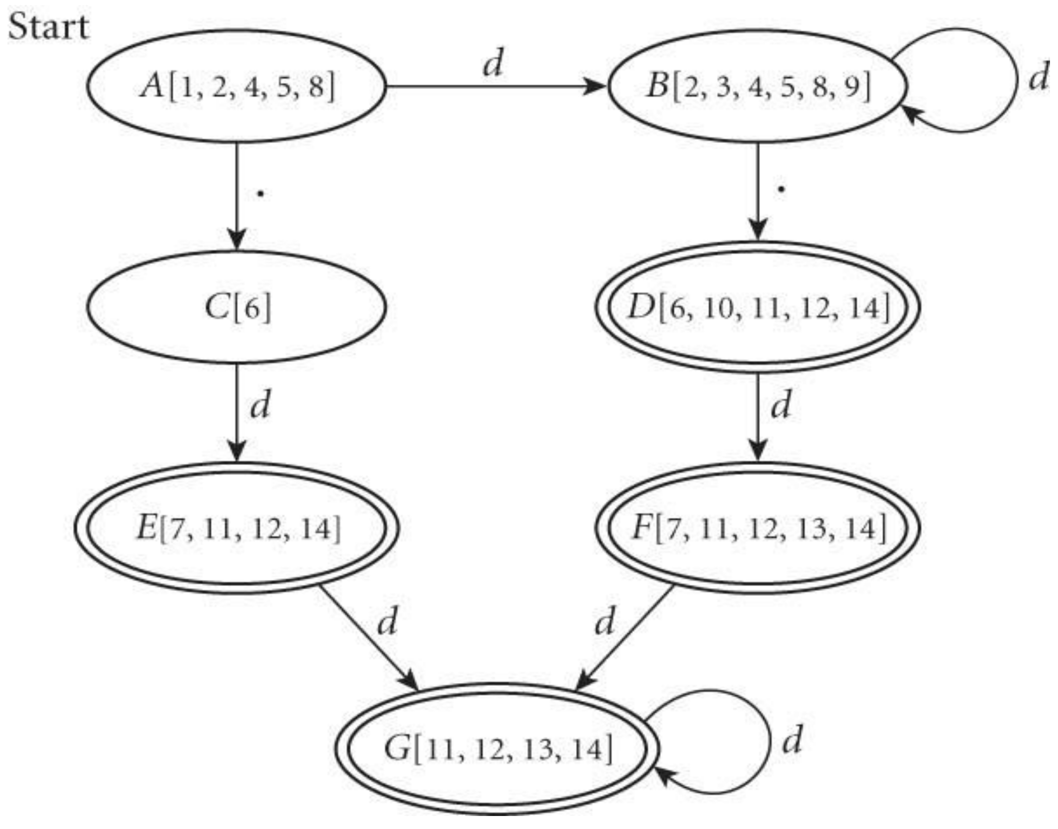
From NFAs to DFAs

- NFAs are hard to compute with, but easy to specify
  - "Guessing" the right transition if there are multiple transitions is hard to code
  - DFAs are unambiguous. Much easier to convert to DFA and then compute, even though it can get bigger.
  - (Side note: how much bigger?)
- A practical approach is to go:

  RegEx => NFA => DFA

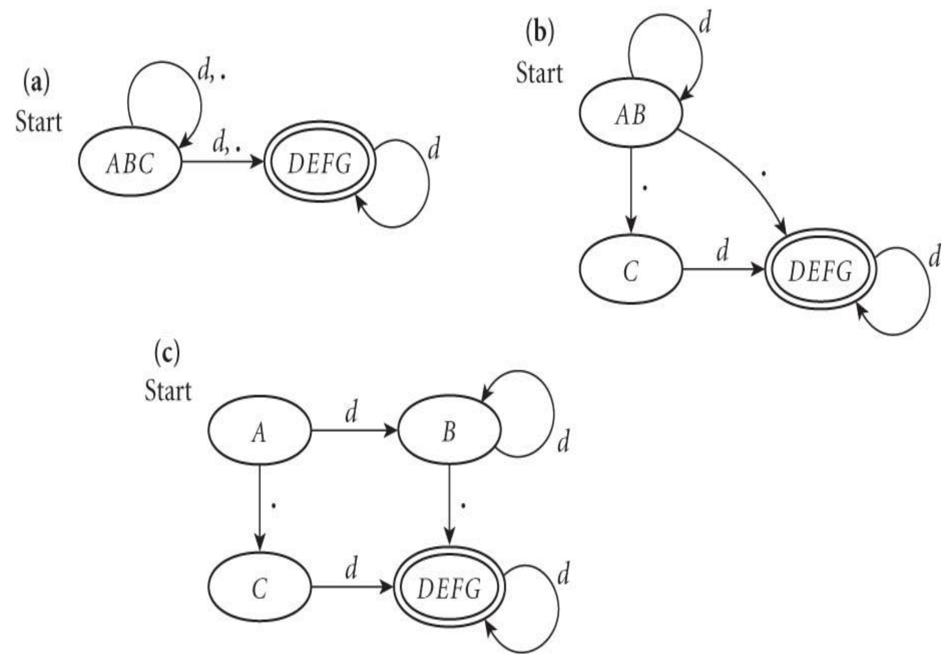
From NFAs to DFAs

- If we automate this conversion on our last NFA (of decimals), we get:



Minimizing DFAs

- In addition, scanners take this final DFA and minimize.
  - (We won't do this part by hand – I just want you to know that the computer does it automatically, to speed things up later.)



Coding DFAs (scanners)

- So, given a DFA, code can be implemented in 2 ways:
  - A bunch of if/switch/case statements
  - A table and driver
- Both have merits, and are described further in the book.
- We'll mainly use the second route in homework, simply because there are many good tools out there.

Scanners

- Writing a pure DFA as a set of nested case statements is a surprisingly useful programming technique
  - though it's often easier to use perl, awk, sed
  - for details see Figure 2.11
- Table-driven DFA is what lex and scangen produce
  - lex (flex) in the form of C code – this will be an upcoming homework
  - scangen in the form of numeric tables and a separate driver (for details see Figure 2.12)

Limitations of regular languages

- Certain languages are simply NOT regular.
- Example: Consider the language 0^n1^n
- How would you do a regular expression of DFA/NFA for this one?

Beyond regular expressions

- Unfortunately, we need things that are stronger than regular expressions.
- A simple example: we need to recognize nested expressions

```
expr -> id | number | -expr |
          (expr)| expr op expr
op -> + | - | * | /
```

- Regular expressions can't quite manage this, since could do ((((x + 7) * 2) + 3) - 1)

Next time

- Flex, a c-style scanner
- Later: parsing and CFGs, which are stronger than DFAs/scanning