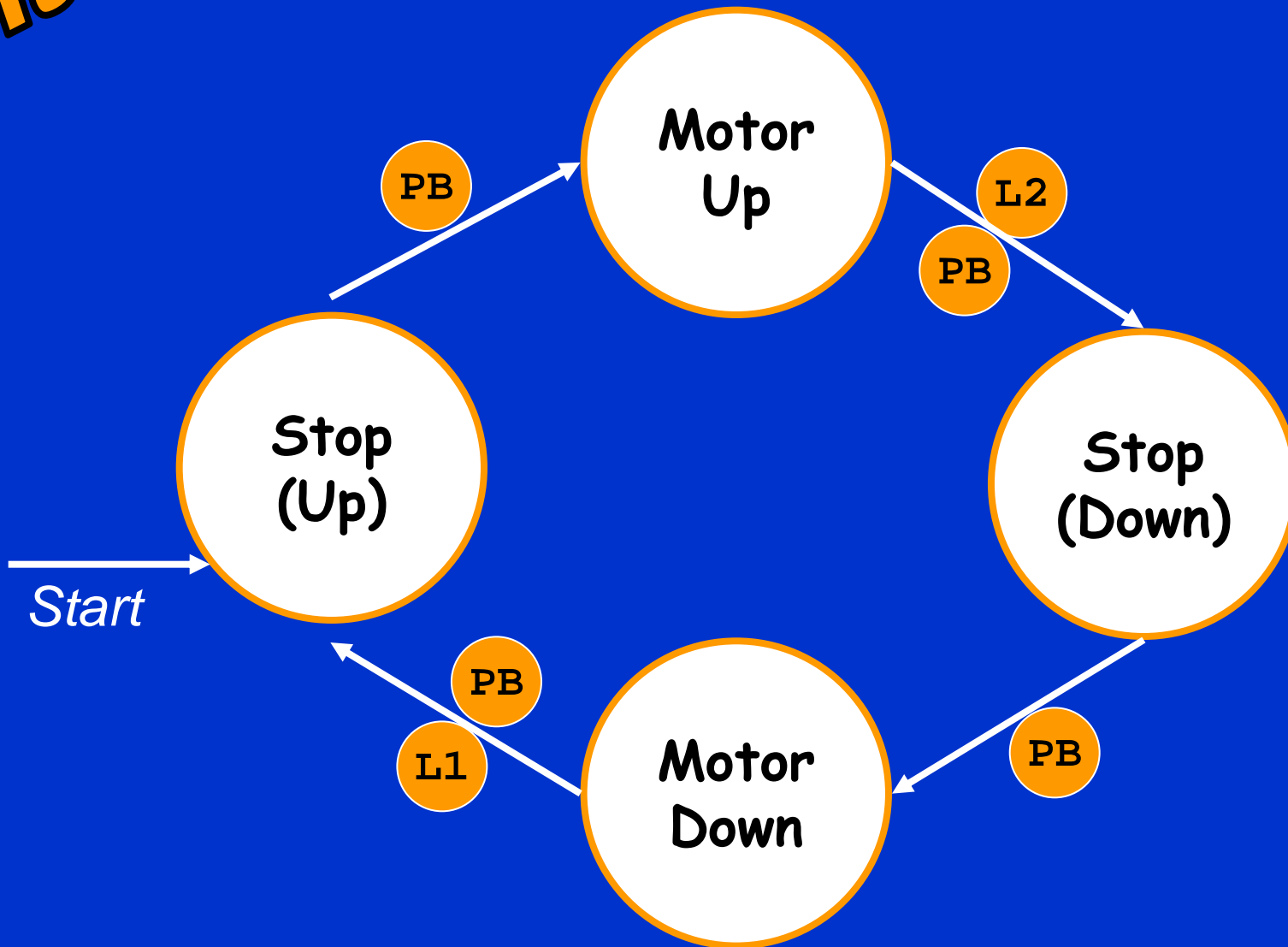


CS 3240

Presentation 4
Finite Automata

Recall

State Machine



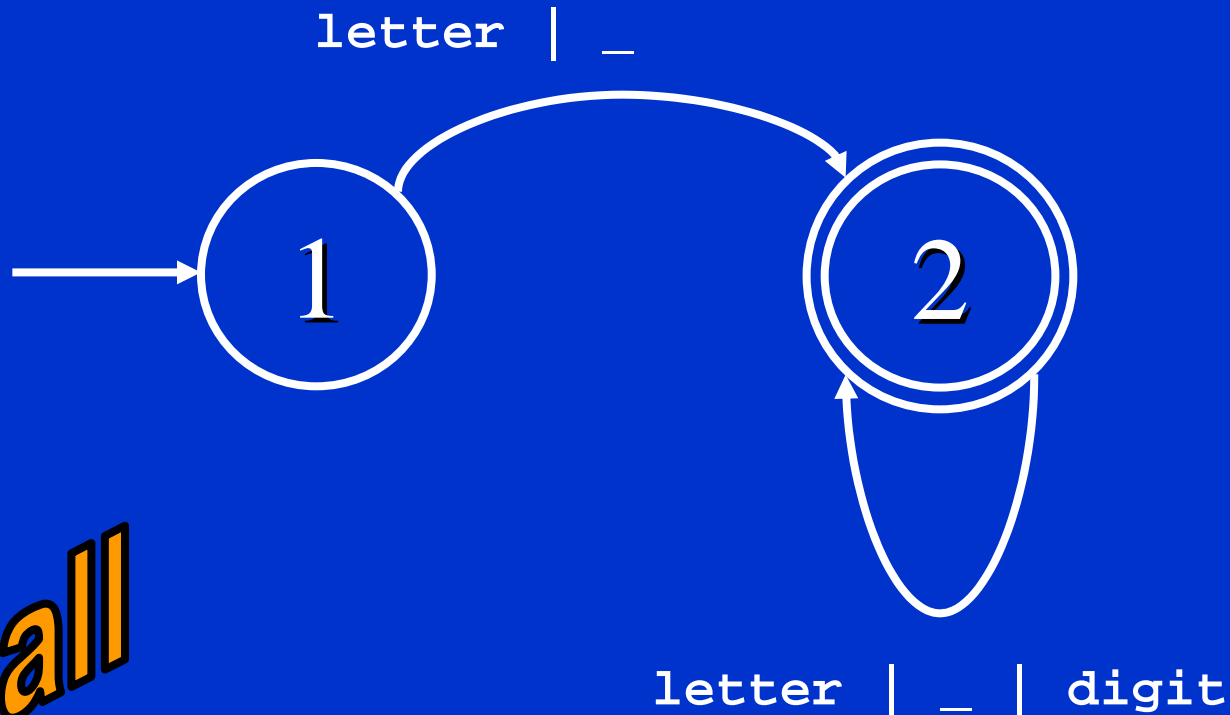
```
enum {L1, L2, PB, NONE} input;
enum {STOPUP, MOTORUP, STOPDOWN, MOTORDOWN} state;
while(1) {
    /* Get input here */
    switch(state) {
        case STOPUP:
            if(input == PB)
                state = MOTORUP;
            break;
        case MOTORUP:
            if (input == PB || input == L2)
                state = STOPDOWN;
            break;
        case STOPDOWN:
            if(input == PB)
                state = MOTORDOWN;
            break;
        case MOTORDOWN:
            if(input == PB || input == L1)
                state = STOPUP;
            break;
    }
    setMotor(state);
}
```

Warning!
Code not bulletproof
Do not use to control
actual apparatus!

Recall

Previews of Coming Attractions

`[a-zA-Z_][a-zA-Z_0-9]*`



Recall

Finite State Machines

- Deterministic Finite Automata
- Non-deterministic Finite Automata

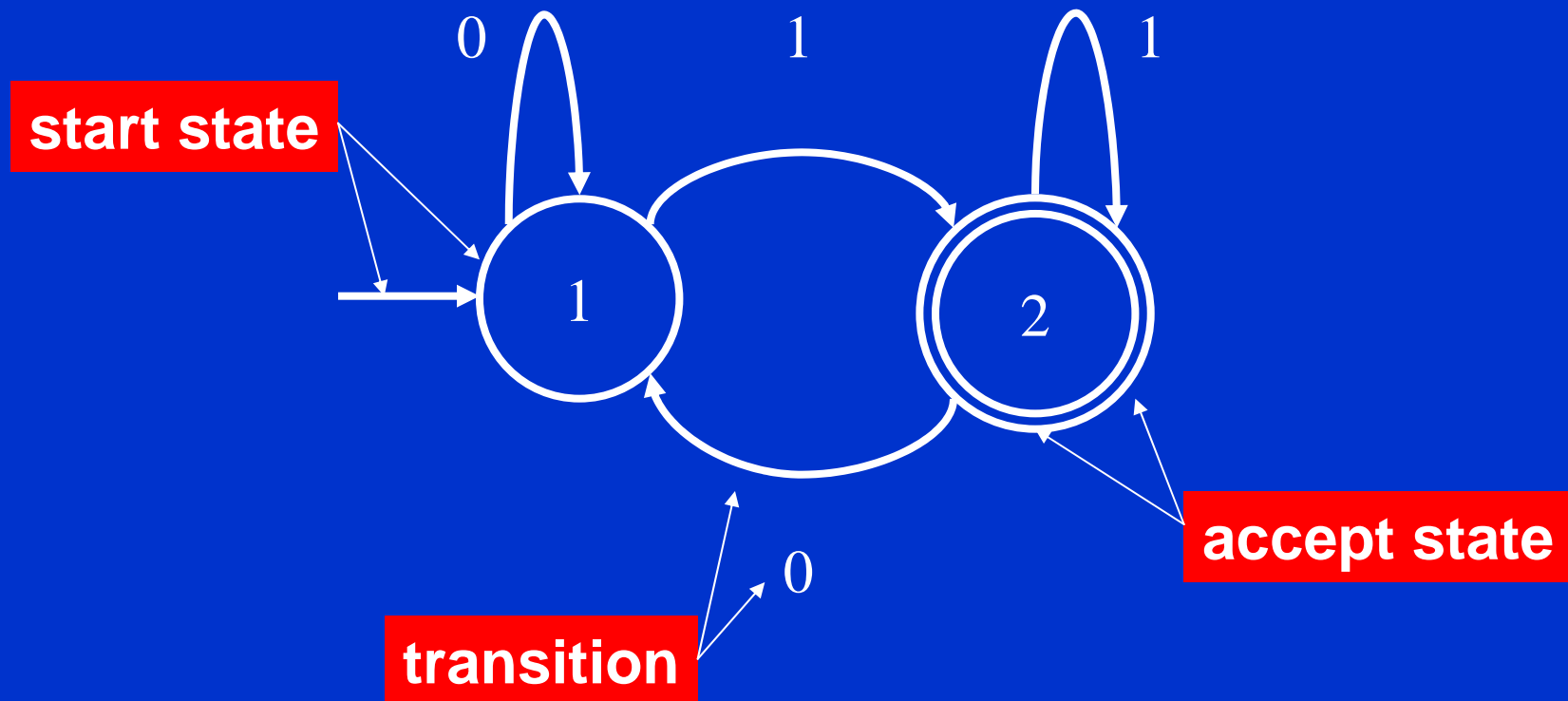
Deterministic Finite Automata

- Recall that theoreticians have developed a number of theoretical models to describe "computing"
 - Example: Turing Machine
- Simplest model is known as a DFA
- Deterministic: Machine will be in a state. Upon receipt of a certain symbol will go to a known state.
- Finite: The machines only have a certain number of states
 - The fascination here is that a machine with a finite number of states can "recognize" an infinite number of strings!
- Automata: (pl. of automaton)

DFA's

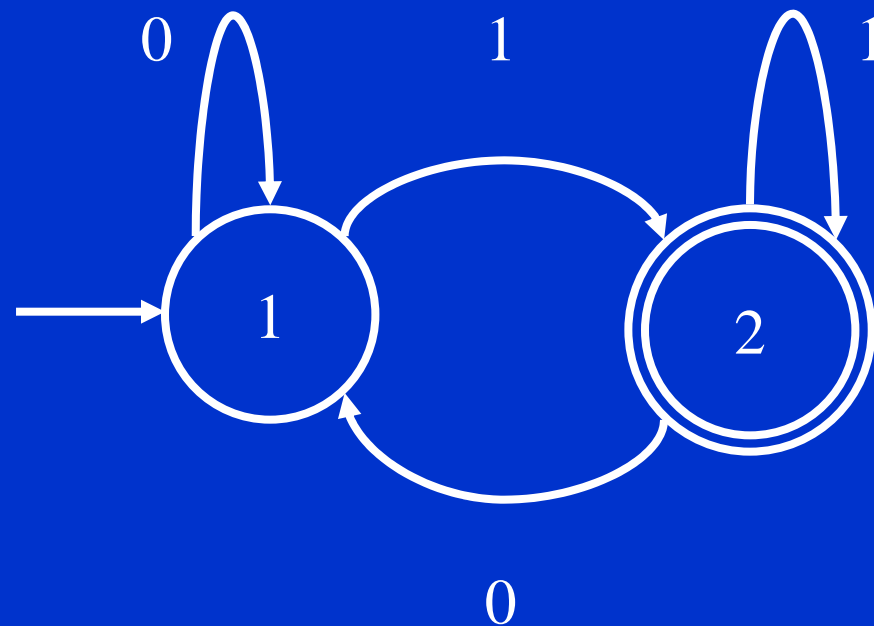
- DFA's recognize strings.
- If the input ends and the DFA is in an accept state then the string is "recognized"
- A "language" can be described as a set of strings
- A language is called a regular language if some finite automaton recognizes it.
- There is a precise mathematical definition of exactly what is meant by a finite automaton

Parts of a DFA



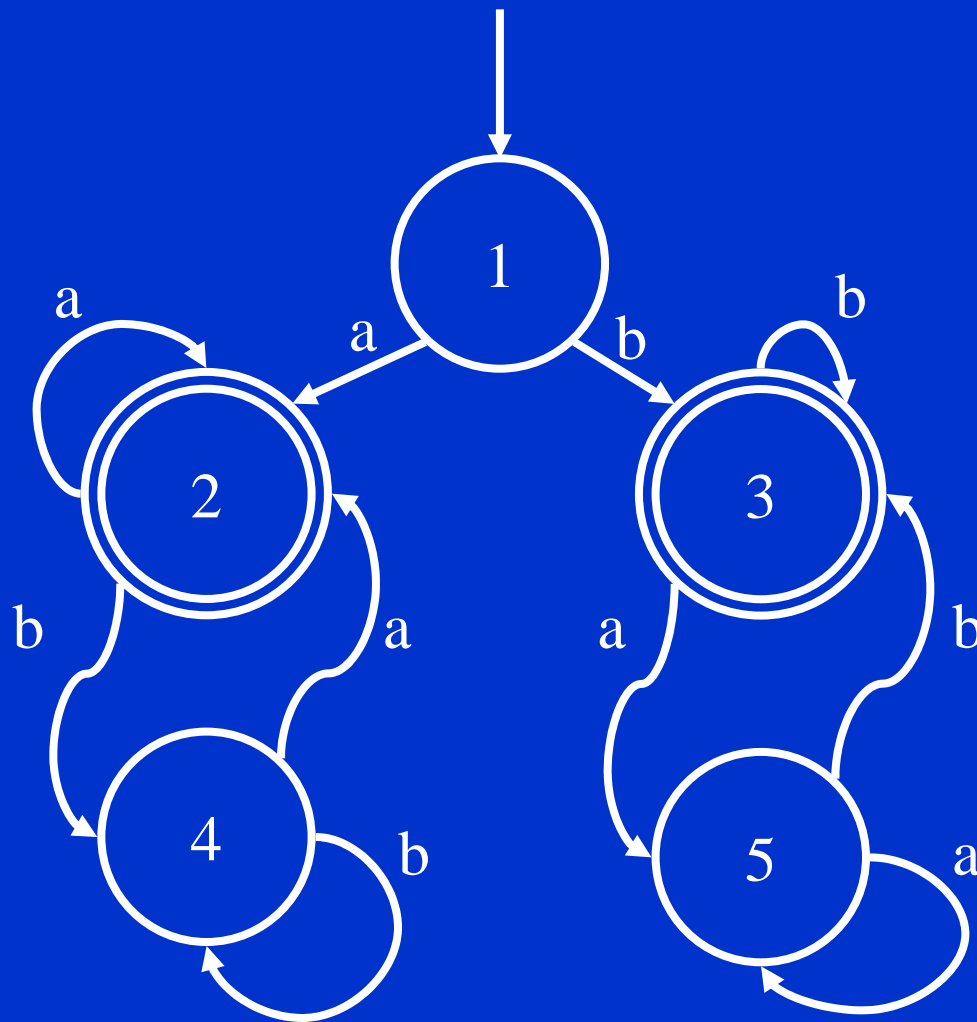
Note: The alphabet for this example is $\{0, 1\}$. Each state has a transition for every symbol in the alphabet

DFA Examples



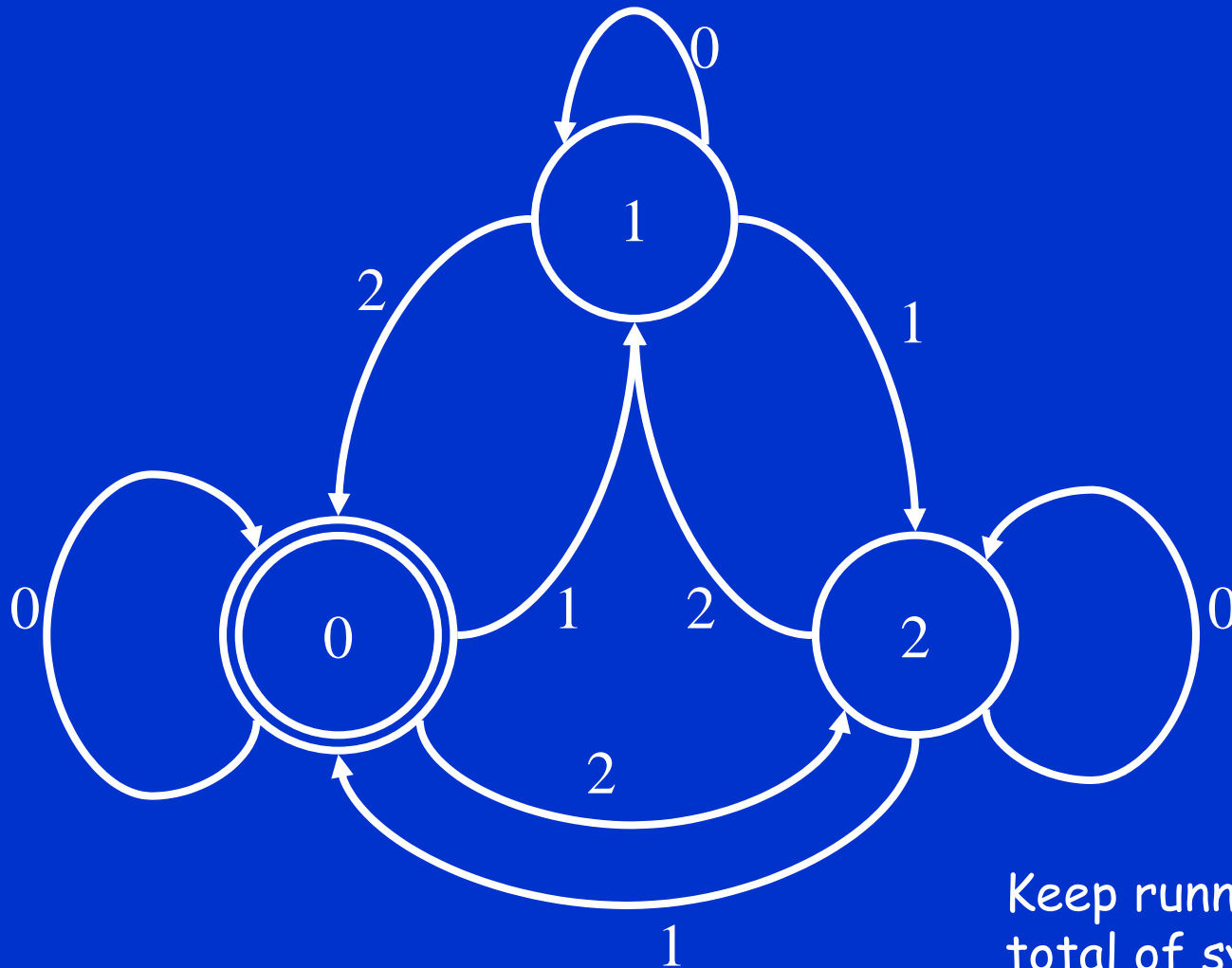
Accept all strings
that end in 1

DFA Examples



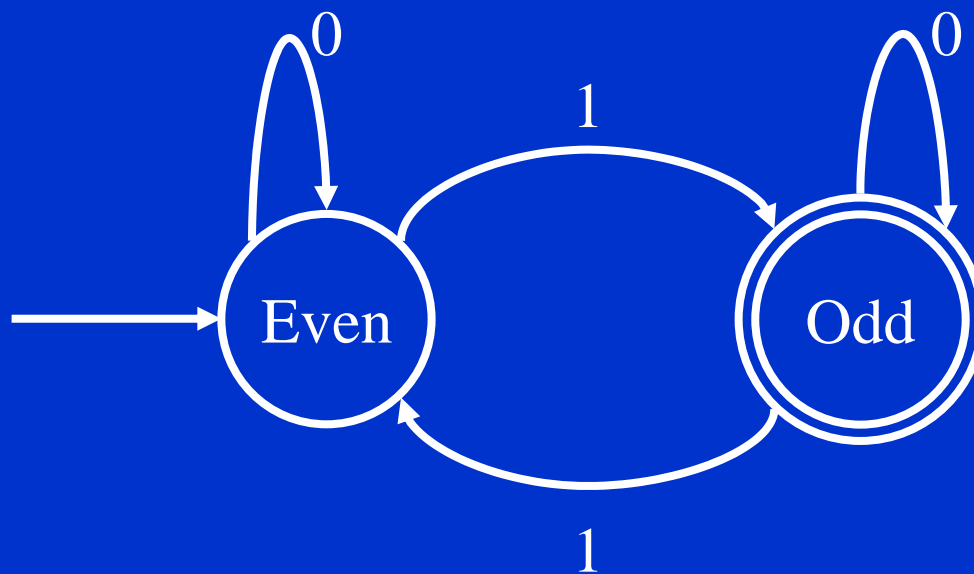
Accept strings of 'a's and 'b's that begin and end with same symbol

DFA Examples



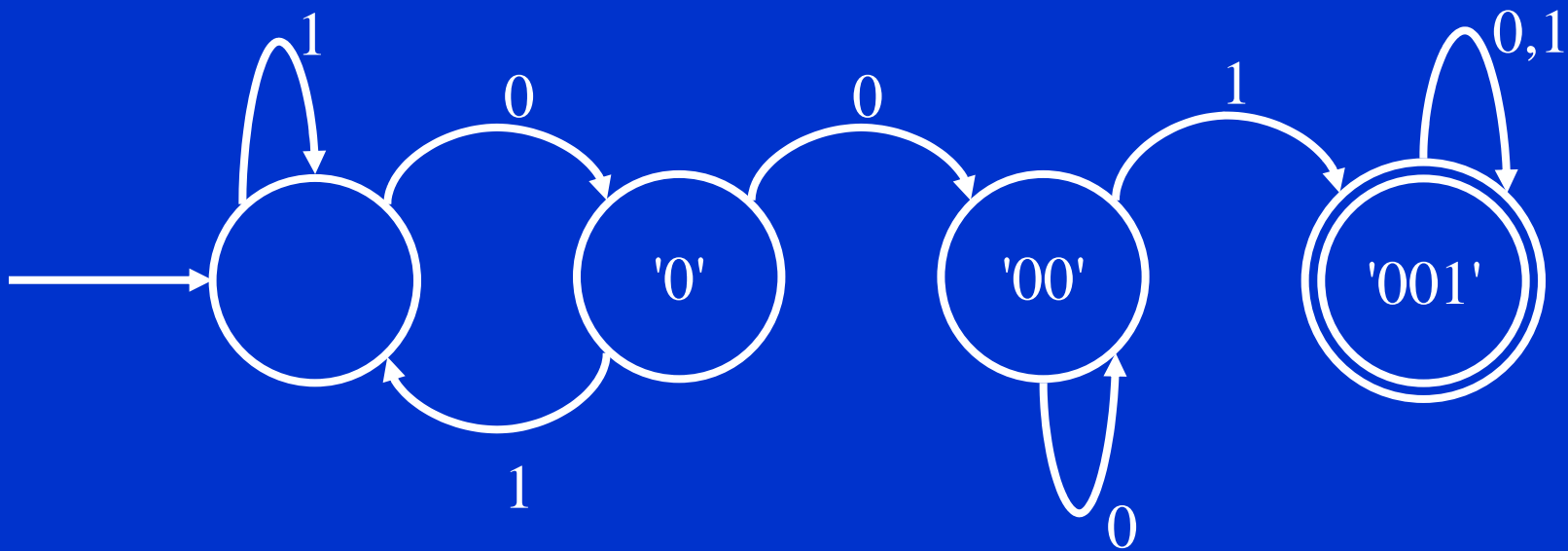
Keep running count of total of symbols read in mod 3. Accept on 0.

DFA Examples



Strings with an odd number of ones.

DFA Examples



Strings containing
the substring 001

Can DFA's be designed to
accept any string?

No!

Examples

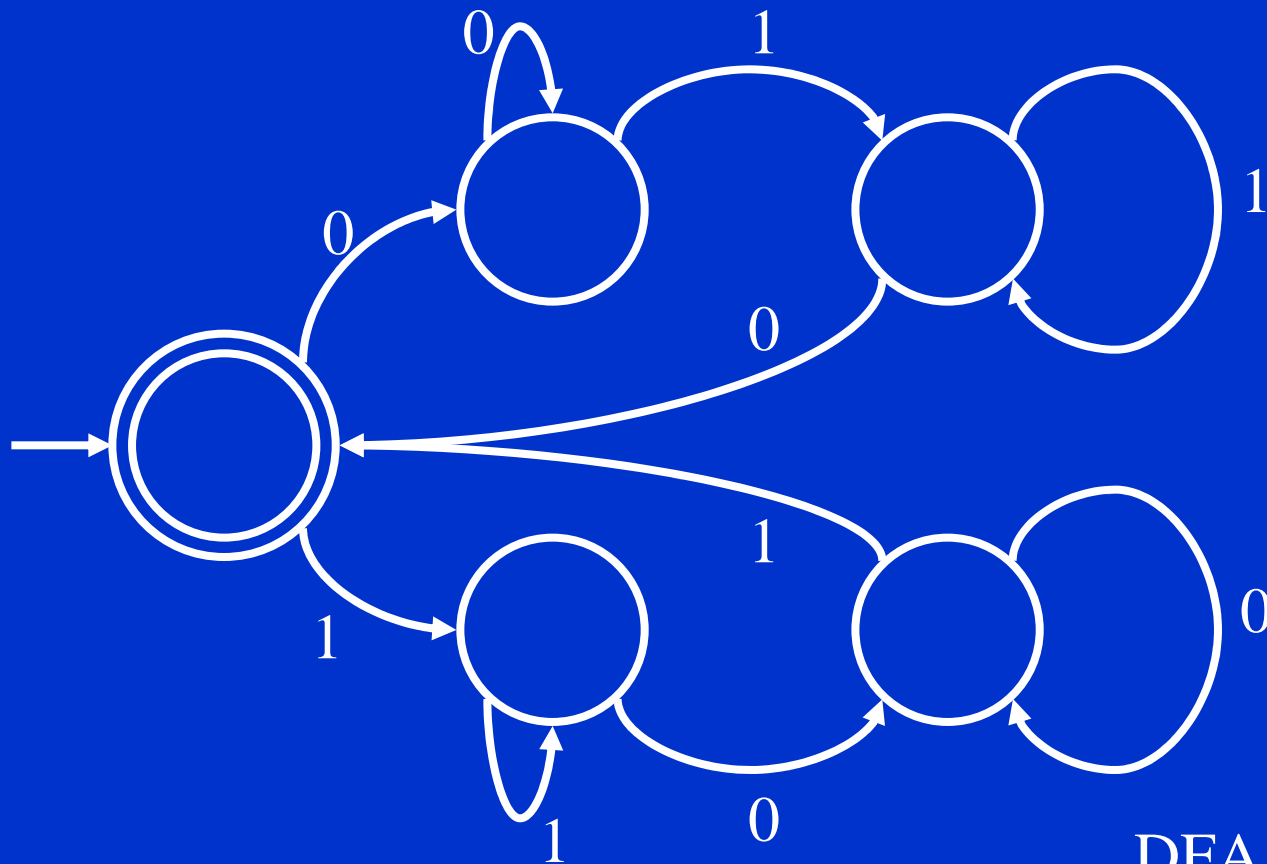
- Design a DFA to recognize strings that start out with k zeros followed by k ones.
- Design a DFA to recognize strings with an equal number of ones and zeros.

Why?

Properties of strings to be remembered

Do not have memory to remember a lot of information

Equal number of 01 and 10s

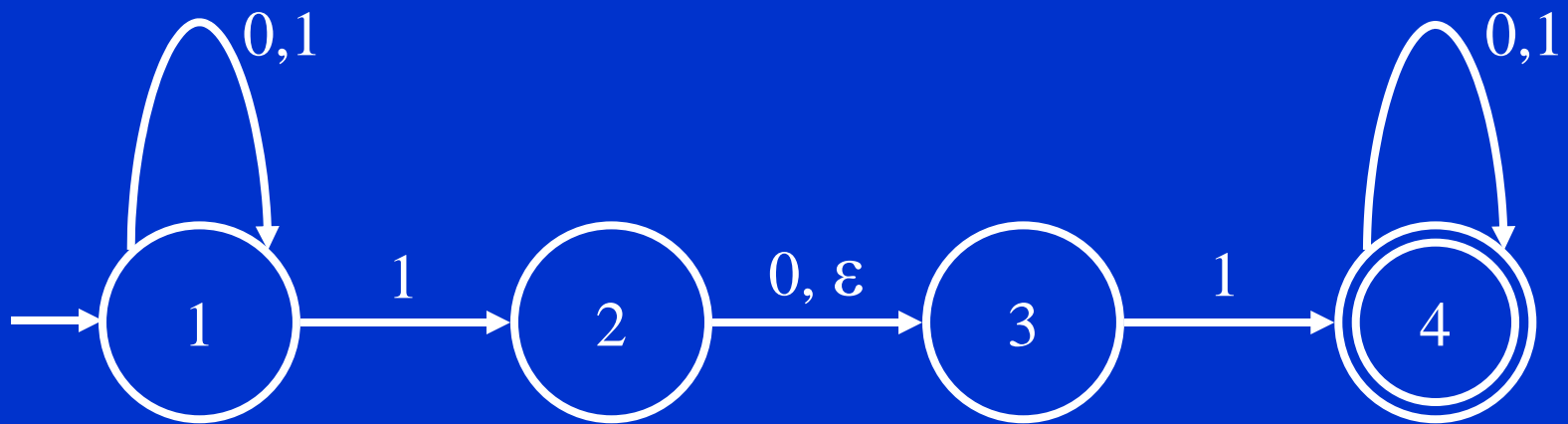


DFA to recognize strings with an equal number of strings "01" and "10"

The NFA

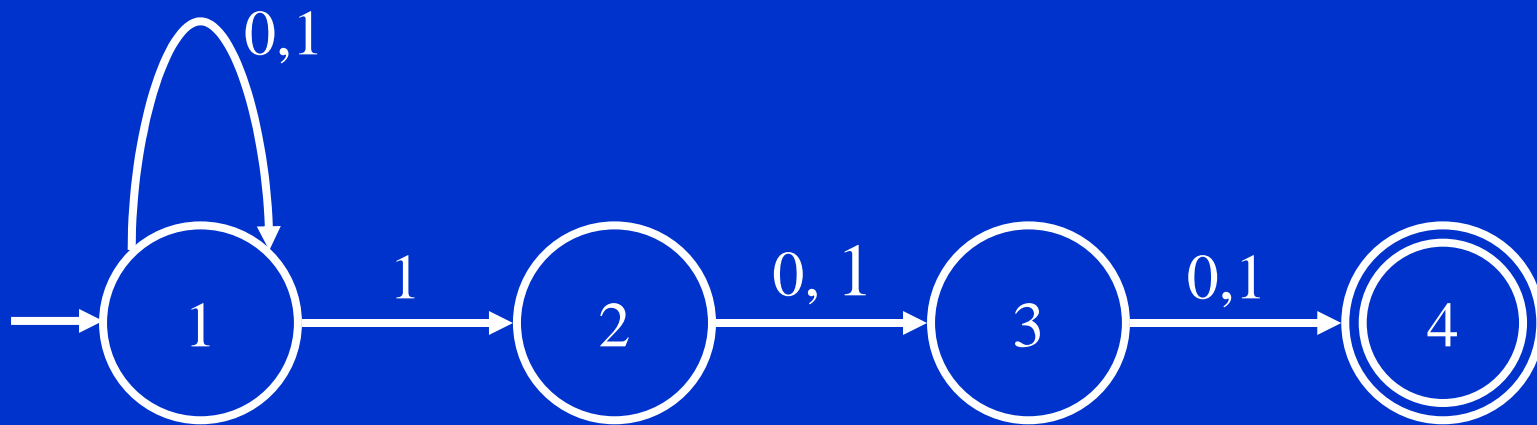
- Nondeterministic Finite Automaton
- Compact way of expressing complex DFA
- Will seem bizarre at first!
- Recall DFA "Rules"
 - Each DFA state must have an exit arrow for every possible symbol in alphabet
 - Transition arrows are labeled only with symbols from alphabet
 - DFA is always in just one state

NFA



Accept strings containing either
101 or 11 as a substring

NFA

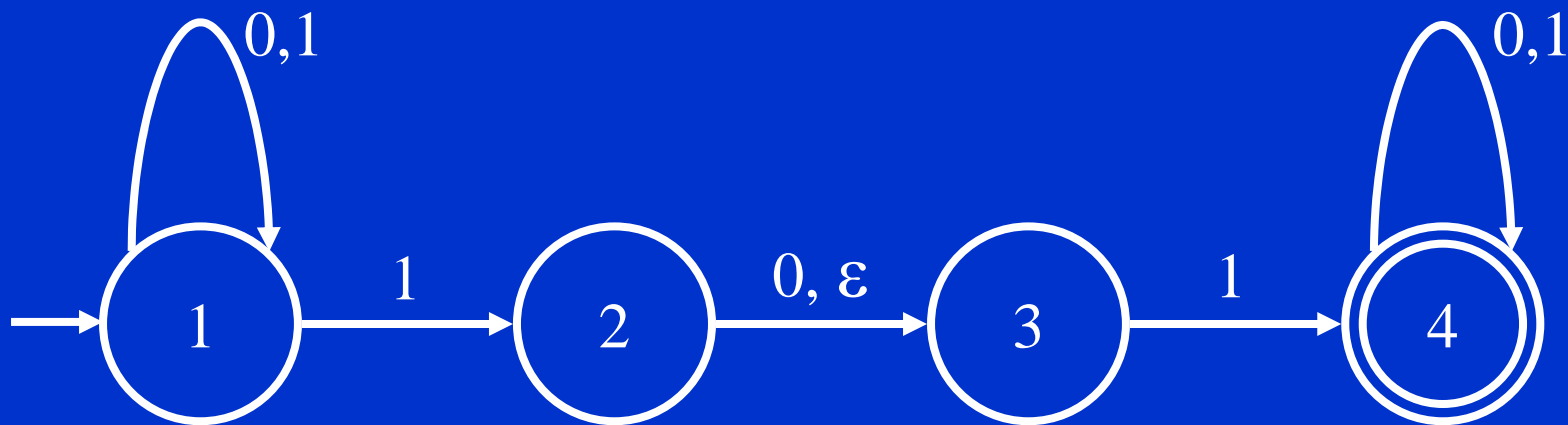


Accept strings containing a 1 in the third position from the end

Basic NFA Ideas

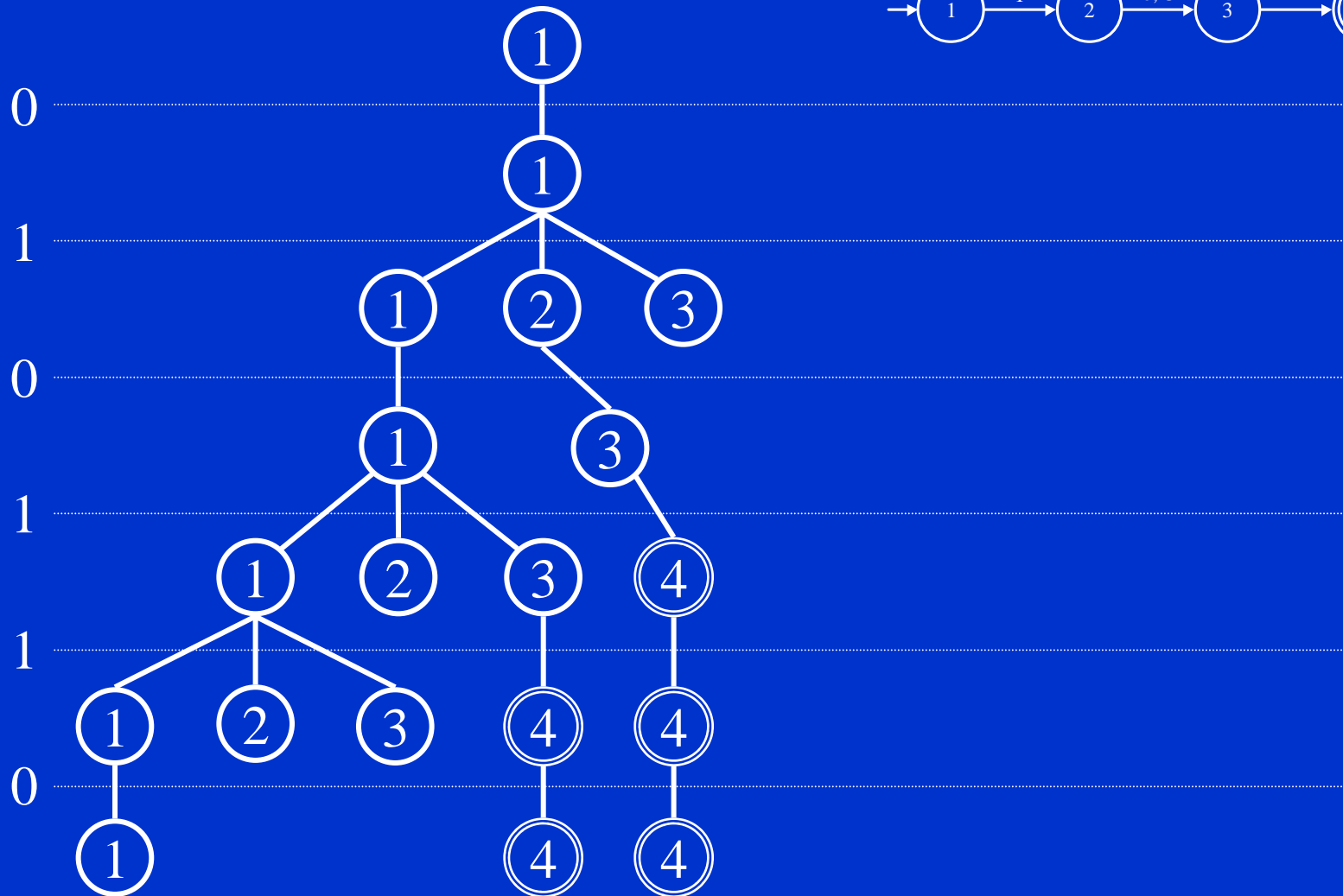
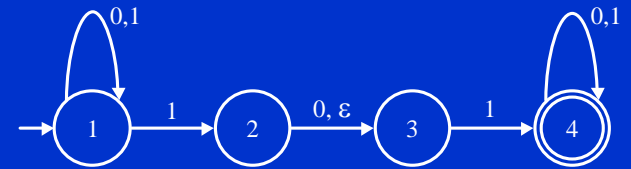
- Transition arrows may be labeled with some of the symbols from the alphabet
- Transition arrows can now have the symbol ε
- Computation
 - Start in start state
 - If any ε transitions, clone a machine for each
 - (I'm not making this up.)
 - Read a symbol, clone a machine for each matching transition
 - If a symbol is read and there is no way to exit from a state, that machine dies.
 - At end of input if any machine accepts then accept

Example: Read: 010110



Accept strings containing either
101 or 11 as a substring

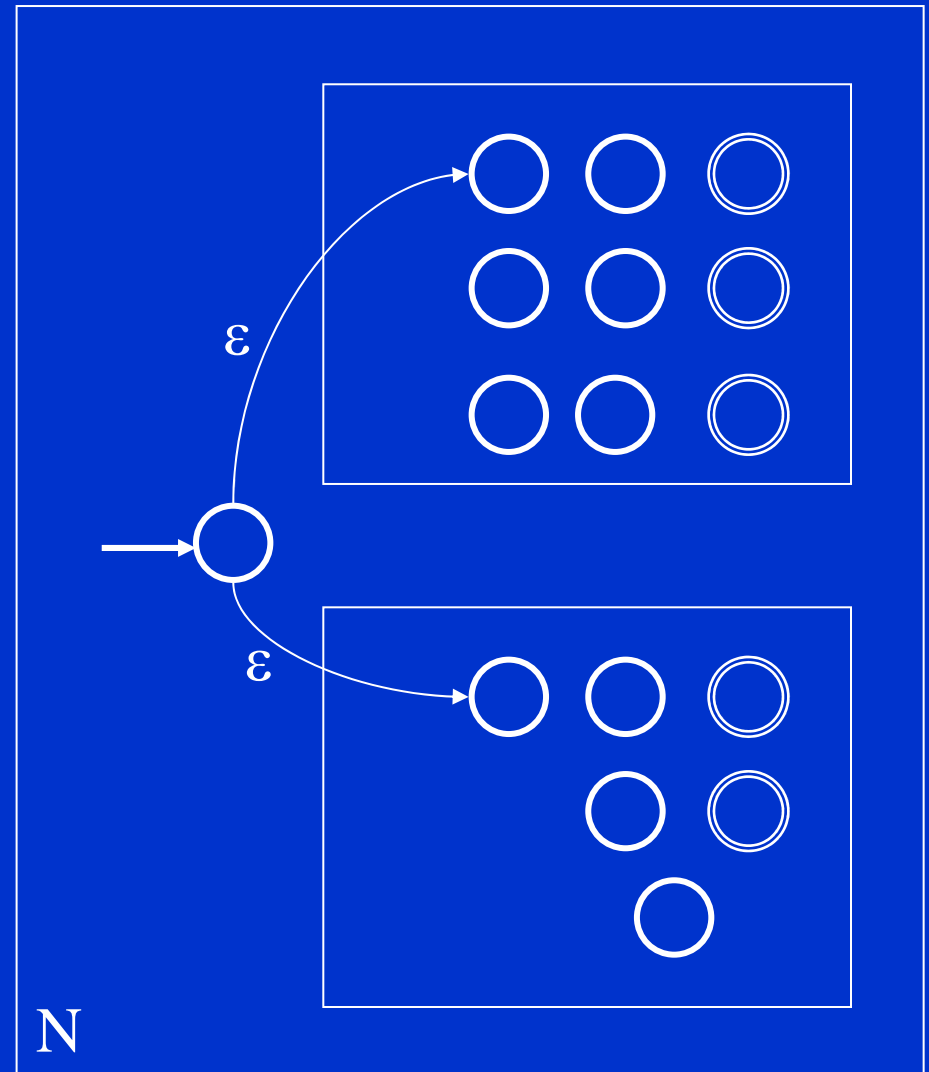
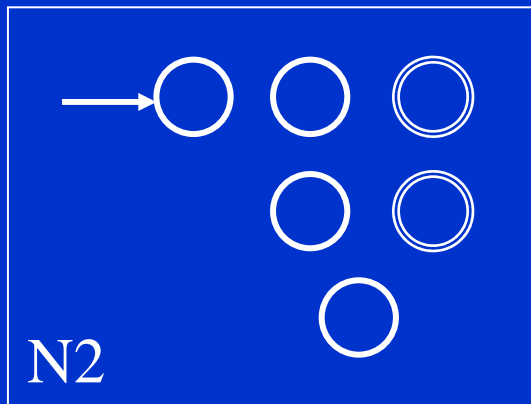
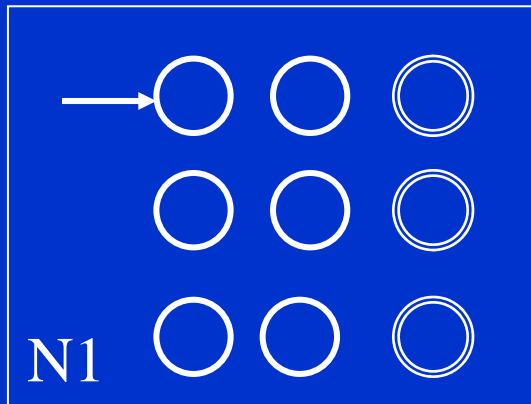
Read: 010110



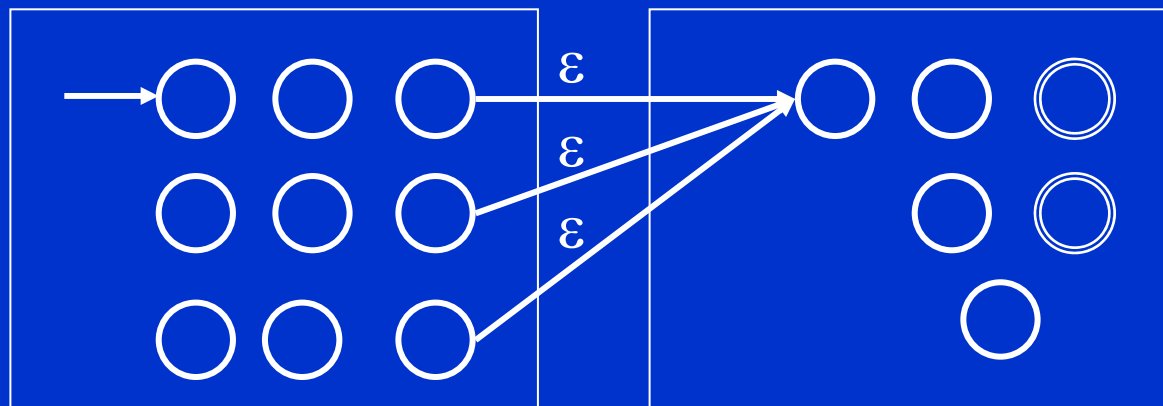
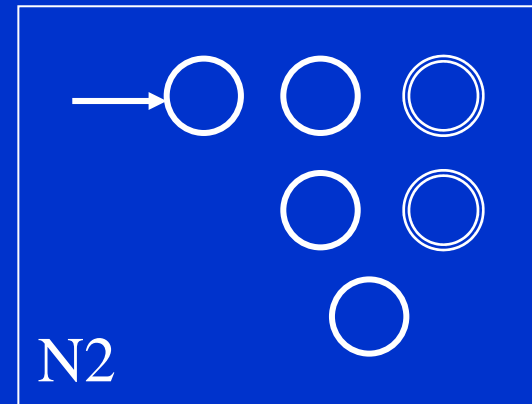
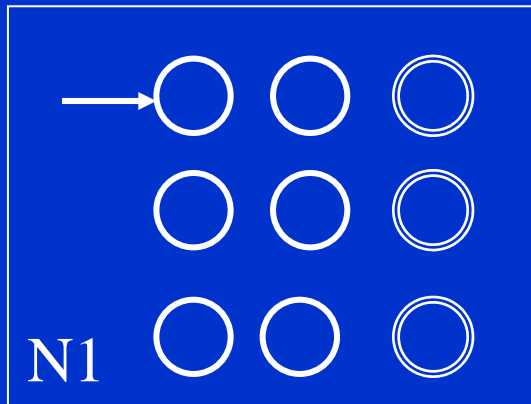
Why?

- NFAs can express simple automata that would be extremely complex in DFA form
- Theorems and algorithms exist that show that any DFA can be converted into an NFA and any NFA can be converted to a DFA
- Theorems also exist showing why union, concatenation and star operations on regular languages produce regular languages.

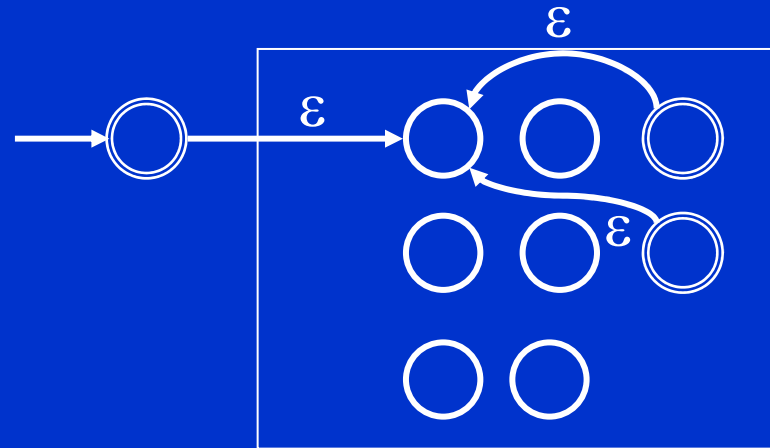
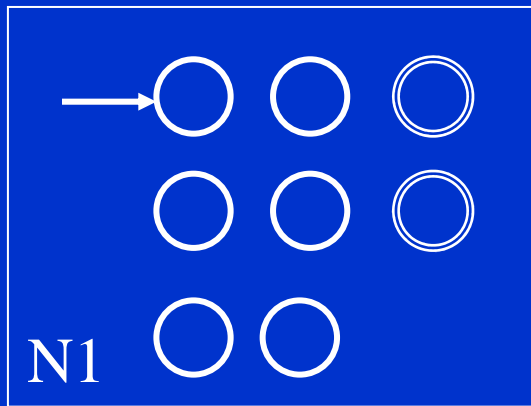
Union: $N = N1 \cup N2$



Concatenation: $N = N1 \cdot N2$



Star: $N = N1^*$



DFAs, NFAs, Regex

- DFA's are a mathematical concept representing a machine that can recognize strings in a language
- Languages recognizable by a DFA are regular
- NFA's are a technique to simplify DFA drawings and they can be shown equivalent to DFA's and vice versa

DFAs, NFAs, Regex

- Regular expressions also may be used to provide a description of a language
 - The value of the arithmetic expression $(5+3)*4$ is 32
 - The value of a regular expression is a language
- Regular expressions and [DN]FA are equivalent
- Regular expressions are common in many CS apps

Words to the Wise

- CS majors will see DFA's, NFA's and Regular Expressions in a theoretical sense in CS 3500 (CMPEs can take it too!)
- You can use regular expressions in practical CS ways all the time. Take the time to understand them.
 - Perl
 - Awk
 - Grep
 - VIM
 - emacs
 - etc.

Lexical Analysis

State Machines

- A lexical analyzer is a state machine
- A state machine is a virtual or real device which responds to inputs with certain outputs that depend on what internal state the machine is in. This state is also changed as a result of the inputs.
- State machines are very similar to finite automata

Symbology



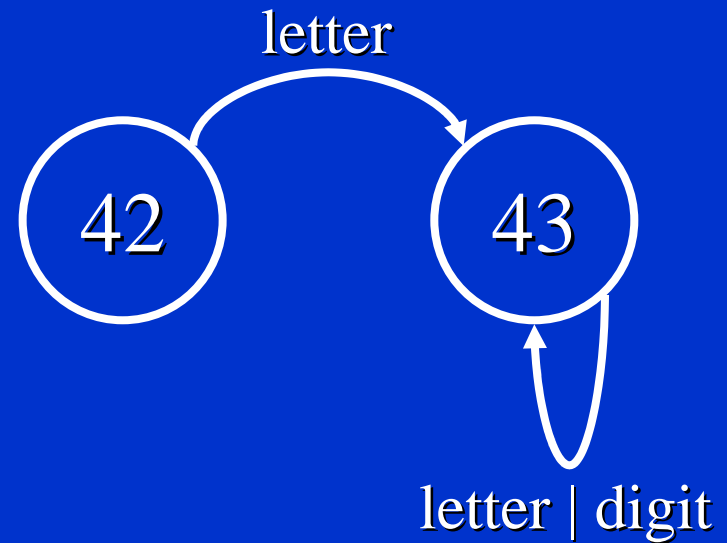
Start
State



State
42



End
State



Typical State
Transitions

Problem

- Create a state machine that will recognize identifiers
- Error states omitted for simplicity

Recognize Identifiers*



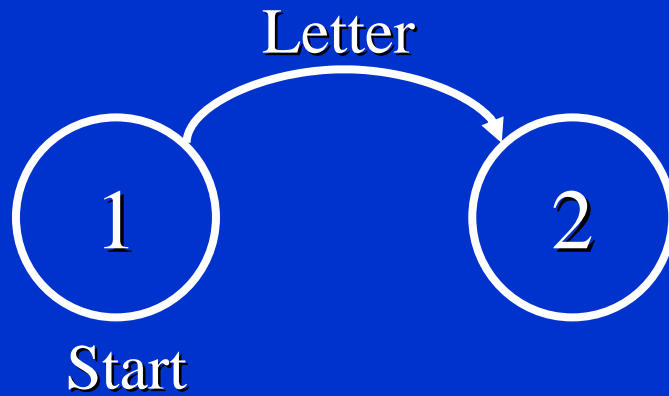
Begin is "Start State"

Read a character

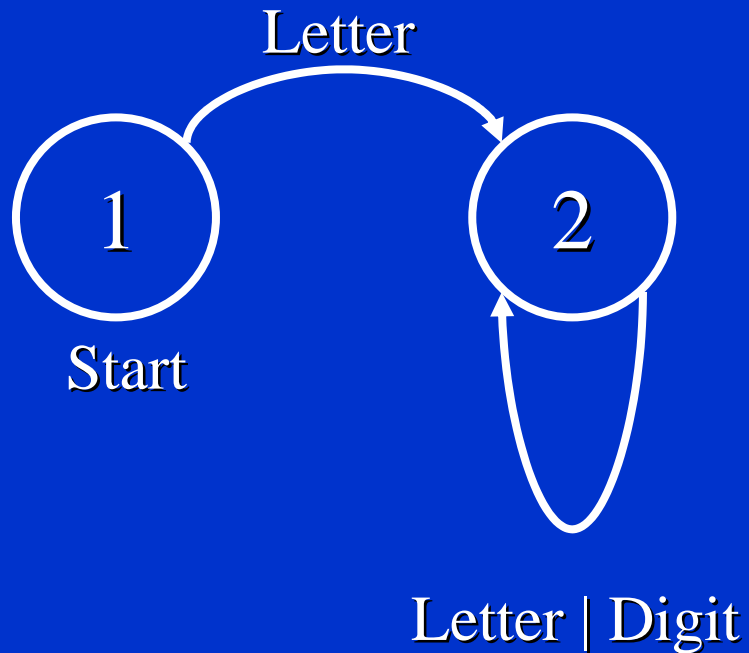
Change state depending on character read

*Underscore omitted for simplicity

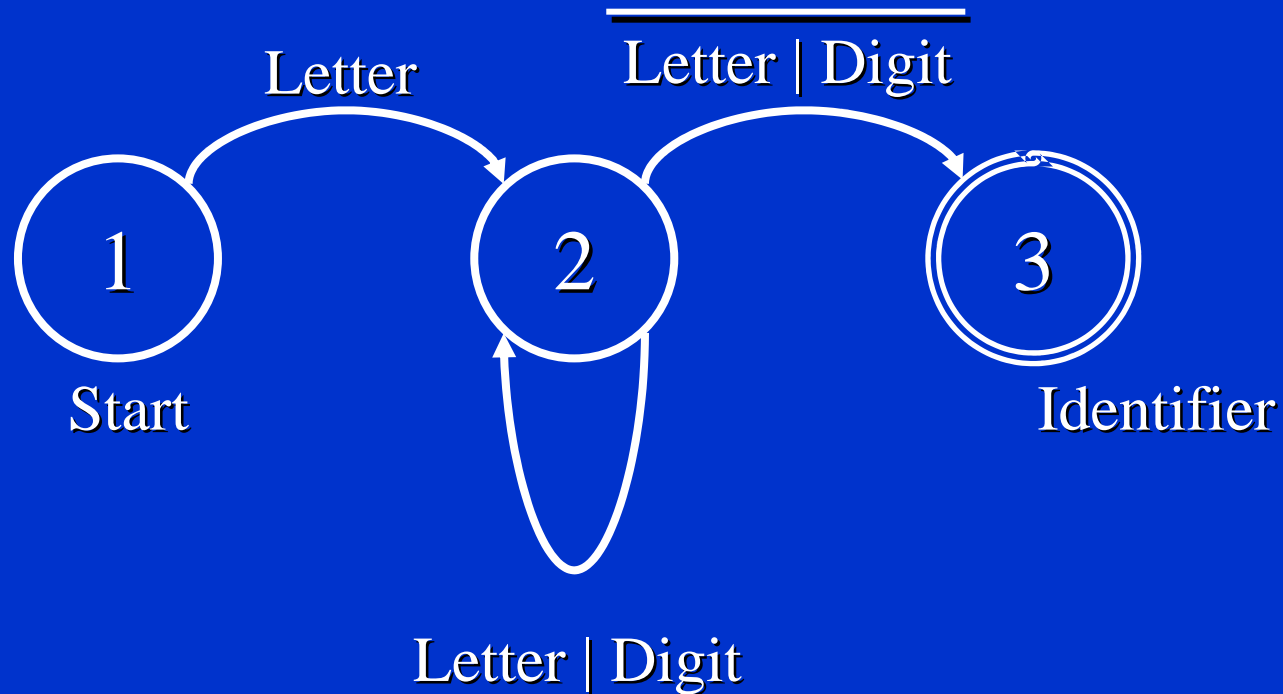
Recognize Identifiers



Recognize Identifiers



Recognize Identifiers



Notice the difference between this and pure finite automata

More Complex

- In a case like this:

```
    i      =      i      +      7      ;  
<ident> <oper> <ident> <oper> <literal> <special>
```

- the white space characters terminate each identification but what about a case like this:

```
i=i+7;
```

- The character which terminates the identification process is part of the next token!

What to do?

- We need to save the last character (i.e. the one that put us in the end state)
- How?
- Push back on input
- Save a character in a variable

Whitespace

- Very little whitespace is actually required by c

```
main() {return(EXIT_SUCCESS);}
```

- would compile!
- There are some places where c does require whitespace

```
int i;
```



Lookahead

- In some languages the language specification is such that the determination of what token type is being scanned cannot be determined until well after the beginning of the token
- Example: FORTRAN
 - Variables don't have to be explicitly declared
 - Implicit typing based on first character
 - Real numbers begin with A-H or O-Z
 - Integers begin with I-N
 - (Mimicked typical mathematical convention)

Lookahead (continued)

- Fortran (continued)
- Typical loop construct

```
DO 10 I = 1,10000
```

```
...
```

```
...
```

```
...
```

```
10 CONTINUE
```

- meaning do all statements from the DO statement up to and including statement 10 starting I with a value of 1 and ending with a value of 10000 (incrementing by 1 as default)

Lookahead (continued)

- Fortran (continued)

- Spaces have no meaning so

```
DO 10 I = 1, 100000
```

- would be equivalent to

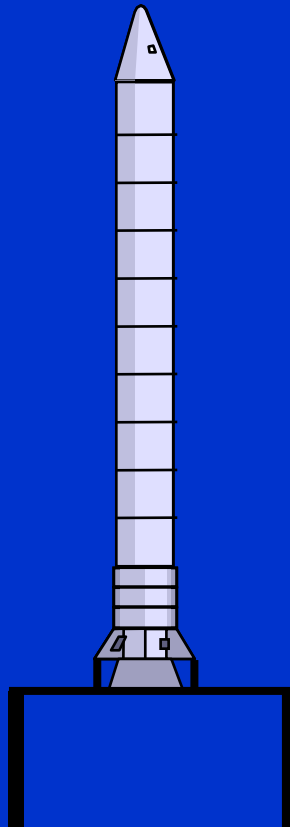
```
DO10I=1,100000
```

- which would be differentiated from

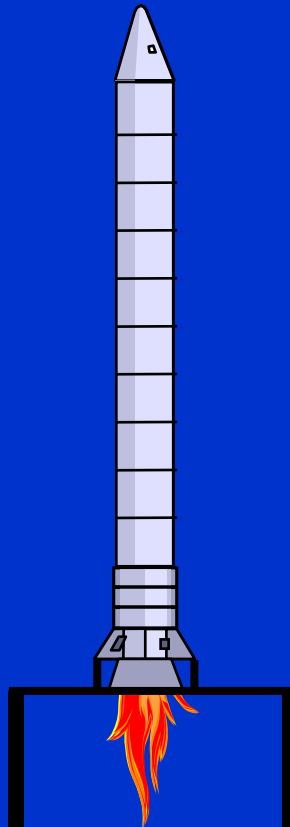
```
DO10I=1.100000
```

- when the compiler encountered the , or .

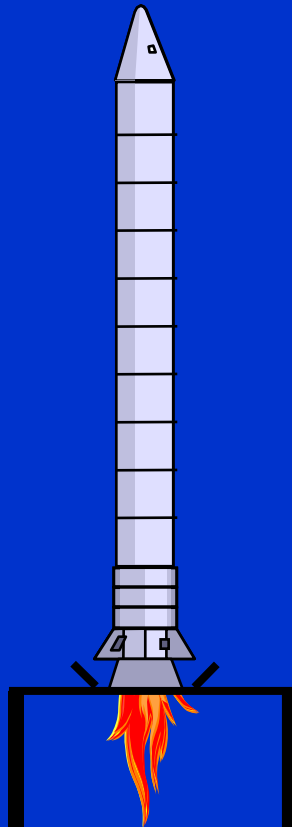
Result



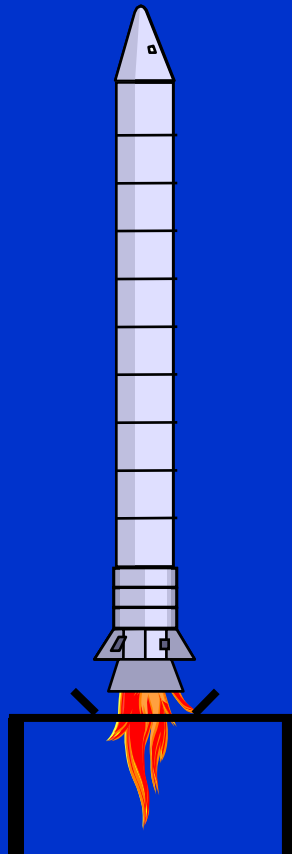
Result



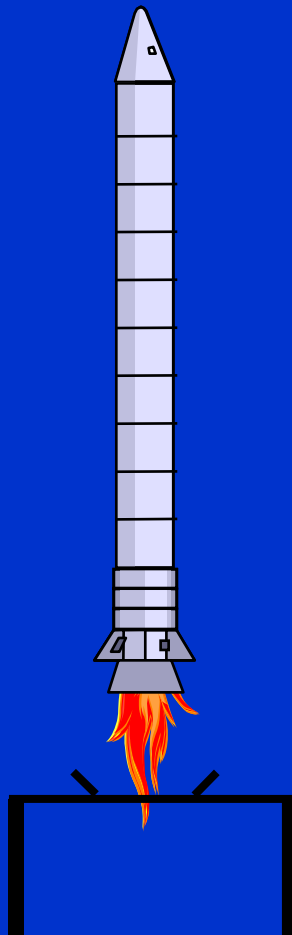
Result



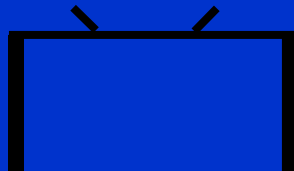
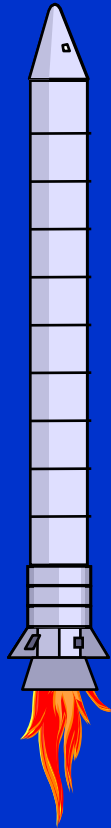
Result



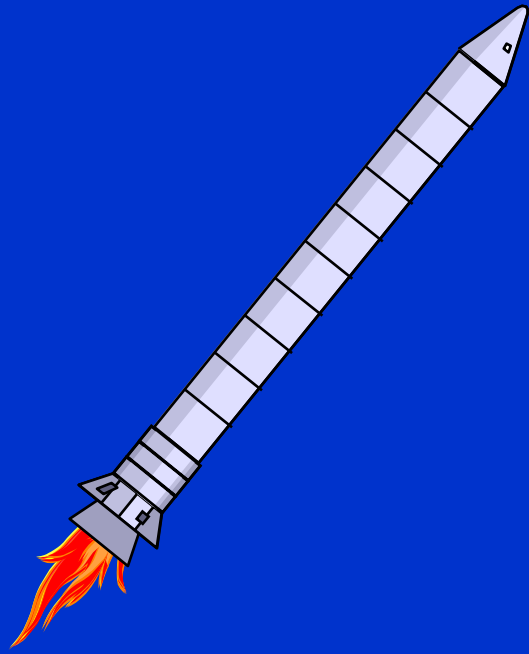
Result



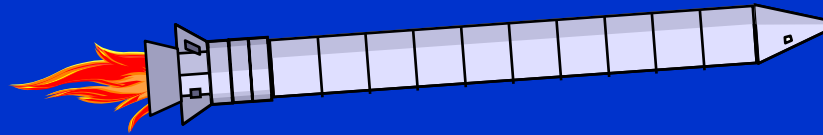
Result



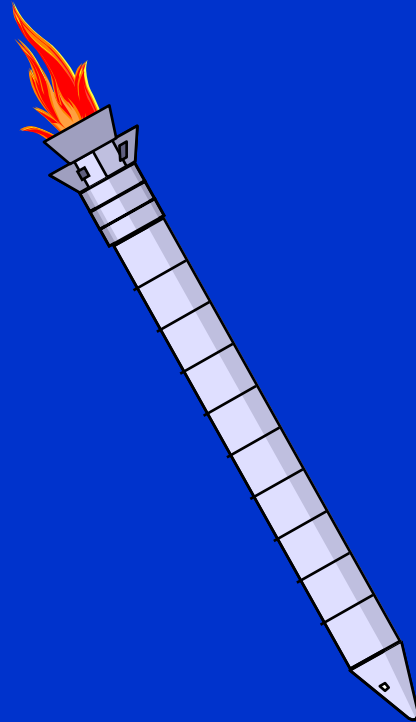
Result



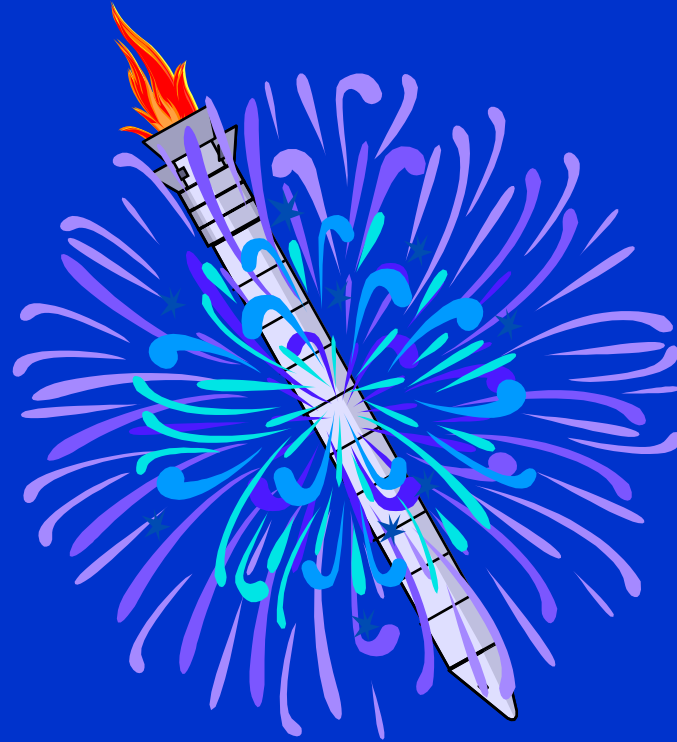
Result



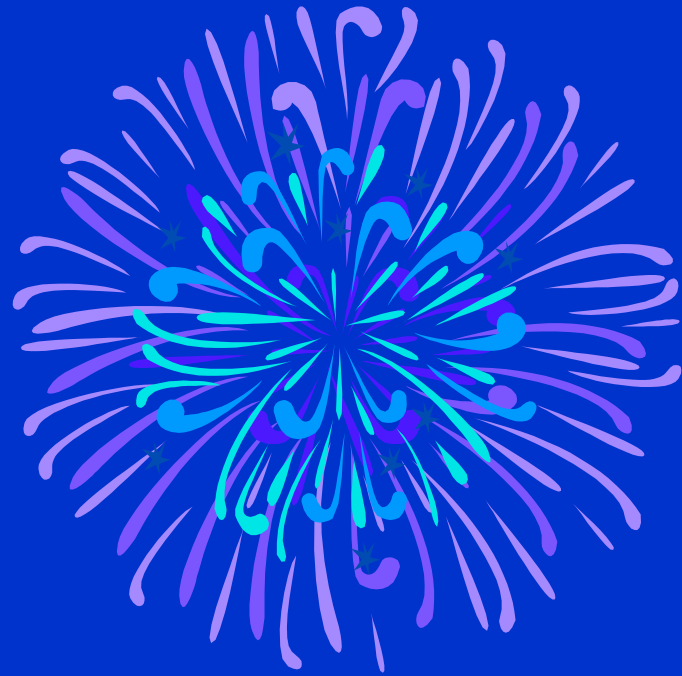
Result



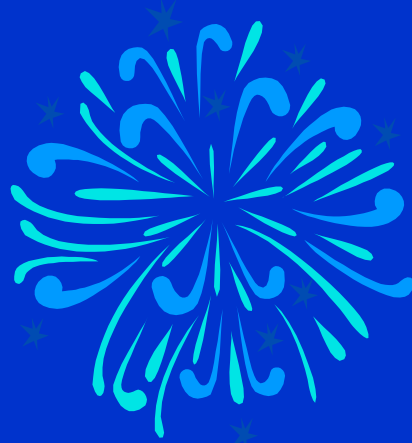
Result



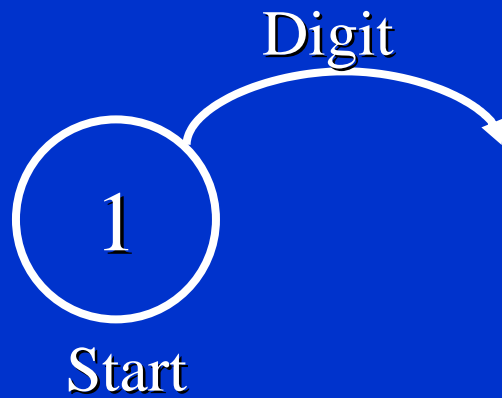
Result



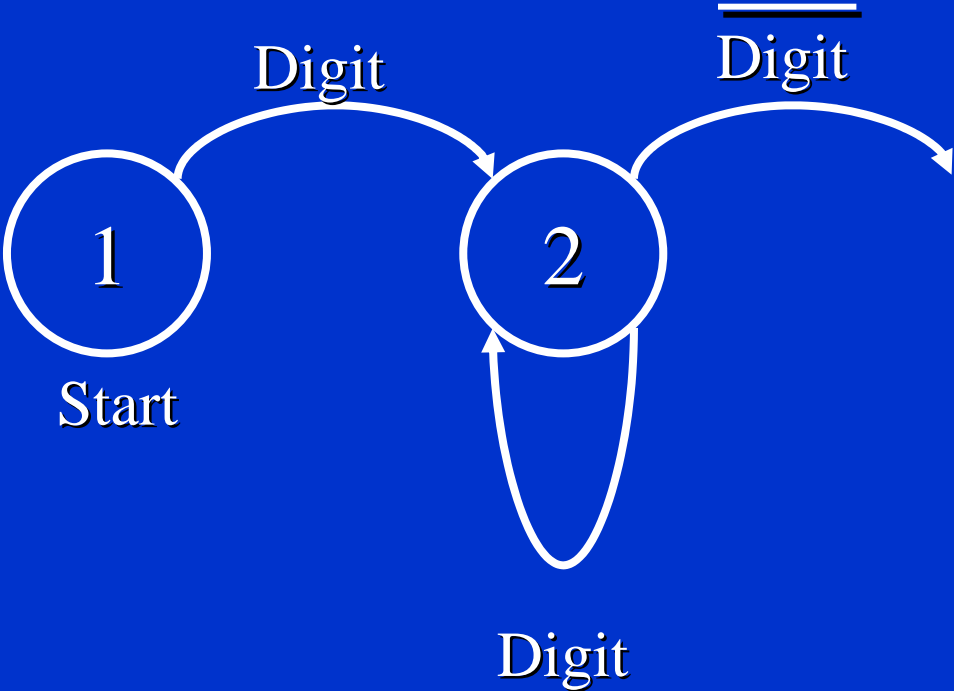
Result



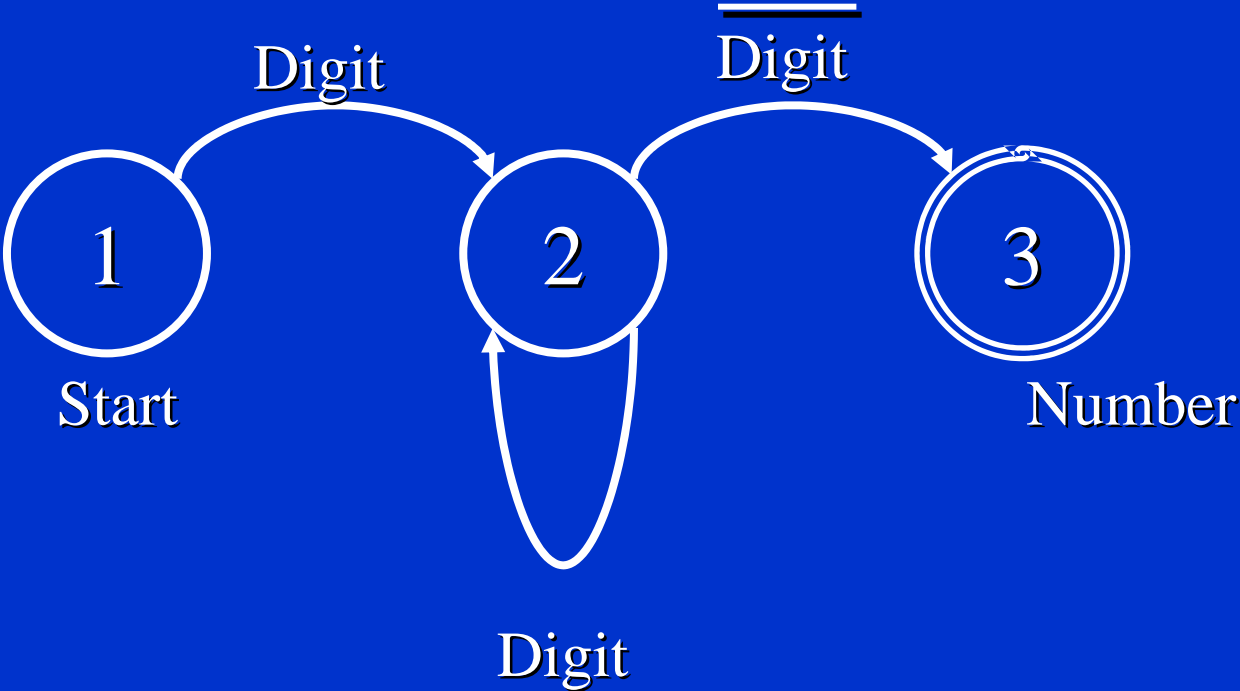
State Machine to Recognize Number



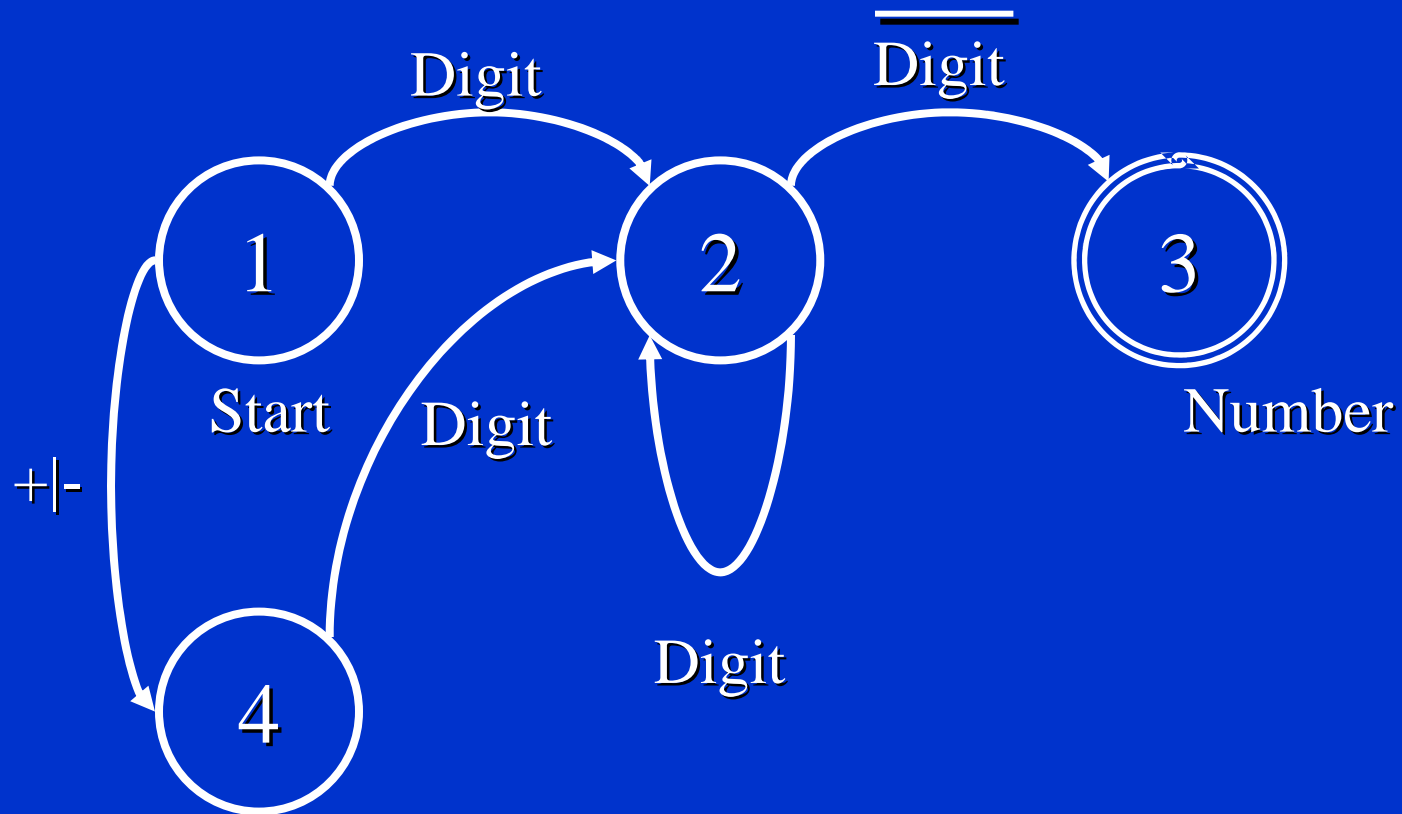
State Machine to Recognize Number



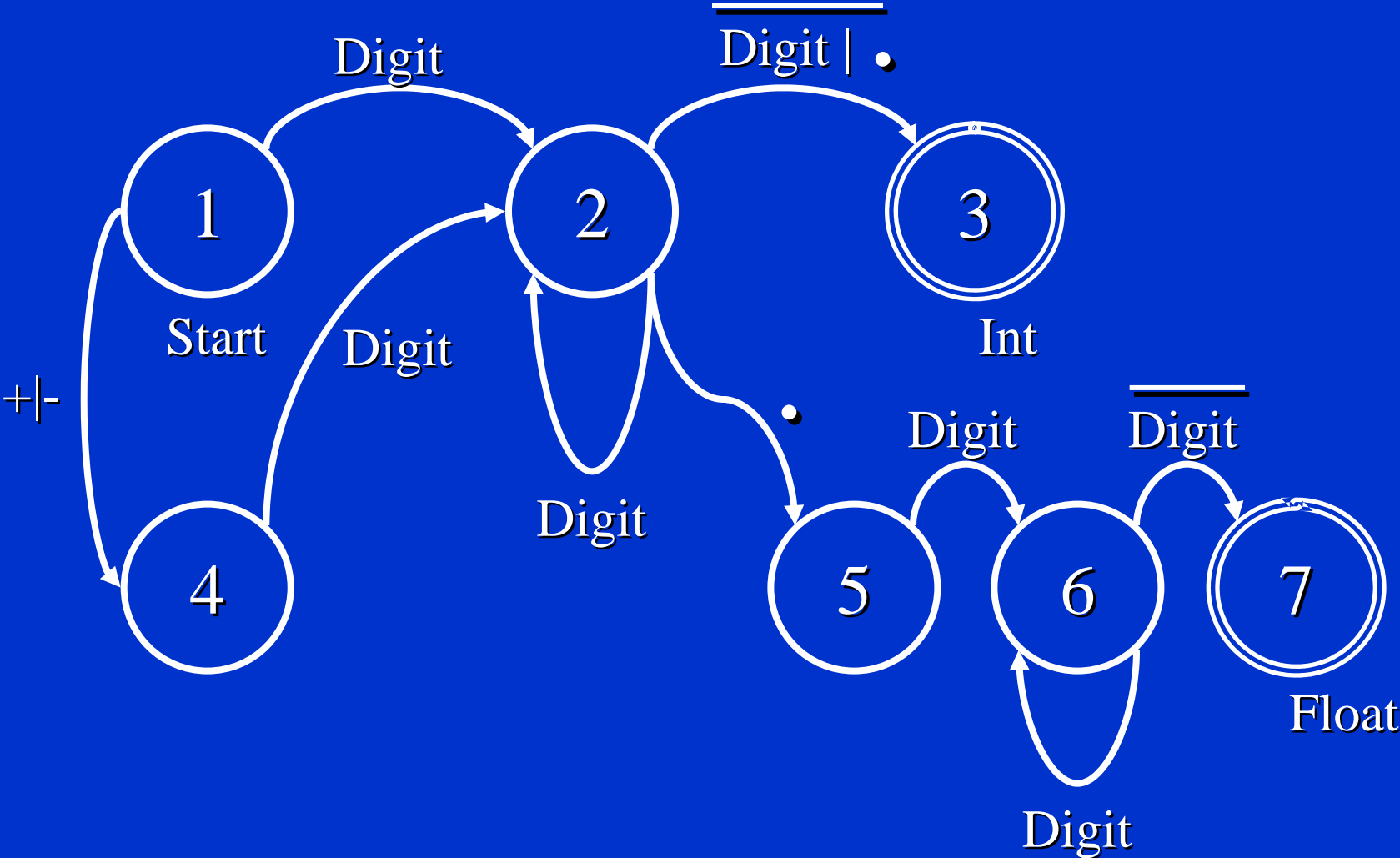
State Machine to Recognize Number



Modify to Recognize Sign



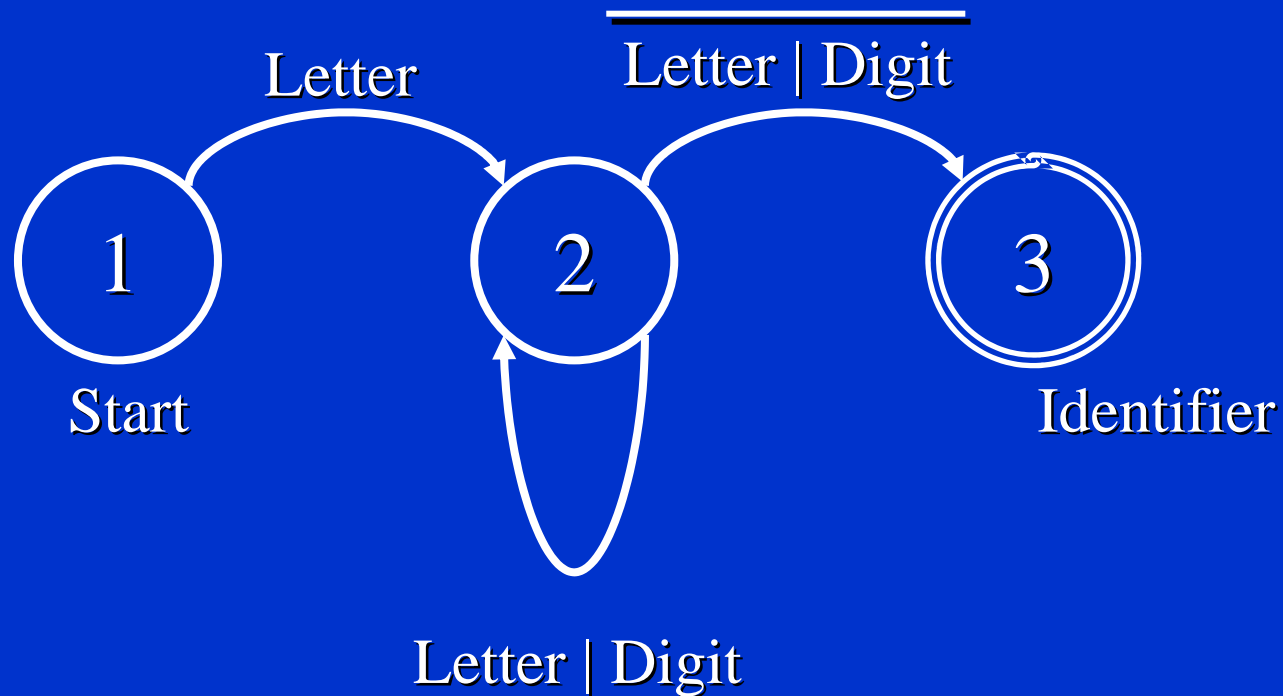
Modify to Recognize Decimals



Problem with State Machines?

- Drawings quickly get too big
- Need way to convert into code
- What is needed?
 - State variable
 - While Loop
 - Input a character
 - Switch

Recognize Identifiers



Example

```
enum {STATE1, STATE2, STATE3} state;
int inchar;
state = STATE1;
while((inchar=getchar())!=EOF) {
    switch(state) {
        case STATE1:
            if(inchar == 'a' || inchar == 'b' ||
               inchar == 'c' || inchar == 'd' ||
               inchar == 'e' || inchar == 'f' ||
               inchar == 'g' || inchar == 'h' ||
               inchar == 'i' || inchar == 'j' ||
               inchar == 'k' || inchar == 'l' ||
               /* How do you like it so far? */)

```

Example

```
enum {STATE1, STATE2, STATE3} state;
int inchar;
state = STATE1;
while((inchar=getchar())!=EOF) {
    switch(state) {
        case STATE1:
            if((inchar >= 'a' && inchar <= 'z') ||
                (inchar >= 'A' && inchar <= 'Z' ))

                /* Better? */
```

Example

```
enum {STATE1, STATE2, STATE3} state;
int inchar;
state = STATE1;
while((inchar=getchar())!=EOF) {
    switch(state) {
        case STATE1:
            if(toupper(inchar) >= 'A' &&
                toupper(inchar) <= 'Z')

                /* Now we're cookin'??? */
```

Example

```
enum {STATE1, STATE2, STATE3} state;
int inchar;
state = STATE1;
while((inchar=getchar())!=EOF) {
    switch(state) {
        case STATE1:
            if(isalpha(inchar))
                state = STATE2;
            else
                /* ERROR!!! */
            endif
            break;
```

Example

```
case STATE2:  
    if(isalpha(inchar) || isdigit(inchar))  
  
    /* Here we go again! */
```

Example

```
case STATE2:  
    if(isalnum(inchar))  
        state = STATE2;  
    else  
        state = STATE3;  
    break;
```

Example

```
case STATE2:
    if(isalnum(inchar))
        state = STATE2;
    else
        state = STATE3;
    break;
case STATE3:
    ungetc(stdin, inchar); /* Check return! */
    break;

/* Make sense? */
```

Example

```
case STATE2:  
    if(isalnum(inchar))  
        state = STATE2;  
    else  
        state = STATE3;  
    break;  
case STATE3:  
    ungetc(stdin, inchar);  
    break;
```

Why not?

Example

```
case STATE2:  
    if(isalnum(inchar))  
        state = STATE2;  
    else  
        state = STATE3;  
    break;  
case STATE3:  
    ungetc(stdin, inchar);  
    break;
```

The character that put us in State 3 (the termination state) was encountered in State 2 processing

Example

```
case STATE2:  
    if(isalnum(inchar))  
        state = STATE2;  
    else  
        state = STATE3; ← ungetc here  
    break;  
case STATE3:  
    ungetc(stdin, inchar);  
    break;
```

The character that put us in State 3 (the termination state) was encountered in State 2 processing

Example

```
case STATE2:
    if(isalnum(inchar))
        state = STATE2;
    else {
        state = STATE3;
        ungetc(stdin, inchar);
    }
    break;

/* Depending on situation should be able to
   break out of loop upon reaching STATE3 */

/* Anything Missing??? */
```

Example

```
case STATE2:
    if(isalnum(inchar))
        state = STATE2;
    else {
        state = STATE3;
        ungetc(stdin, inchar);
    }
    break;

default:
    /* Handle Error */
```

Lexical Analysis

- Tokens are assigned a number
- Each "class" of literal is assigned a number
 - int
 - float
 - string
 - etc
- A single code is assigned to all identifiers

Token Types

• Operators	*	+	-	/	%	
	0	1	2	3	4	...
• Special	;	{	}	()	
	10	11	12	13	14	...
• Keywords	if	else	for	while		
	20	21	22	23		...
• Literals	42	3.141592	"Hello\n"			
	30	31	32			
• Identifiers	[_A-Za-z][_A-Za-z0-9]*					
	40					

Questions?

