CS4430 – Compilers I

Dr William Harrison

Spring 2017

Lexing 2: Theory of Scanning

HarrisonWL@missouri.edu

Announcements

- The web page for this class is up
  - The lecture slides are there and you should be able to get them now.
    - Lecture slides in PDF
  - https://harrisonwl.github.io/doc/cs4430.html

Today's Lecture

- **Continue discussion of front-end**
  - I.e., the earliest phases of the compiler
    - the front-end answers the question "is the input program really a program?"
  - The theory underlying lexing
    - allows us to understand what lexer generators do
- **Approach**
  - Start with really simple & concrete example
  - Consider the underlying theory
  - Learn some tools ("lex", "ScanGen", etc.)

What is Lexing again?

Turns a sequence of characters:

```
c,l,a,s,s,<SP>,F,o,o,<SP>,{,…
```

…into a sequence of words:

```
class,Foo,{,…
```

Synonyms for “word”: token, lexeme

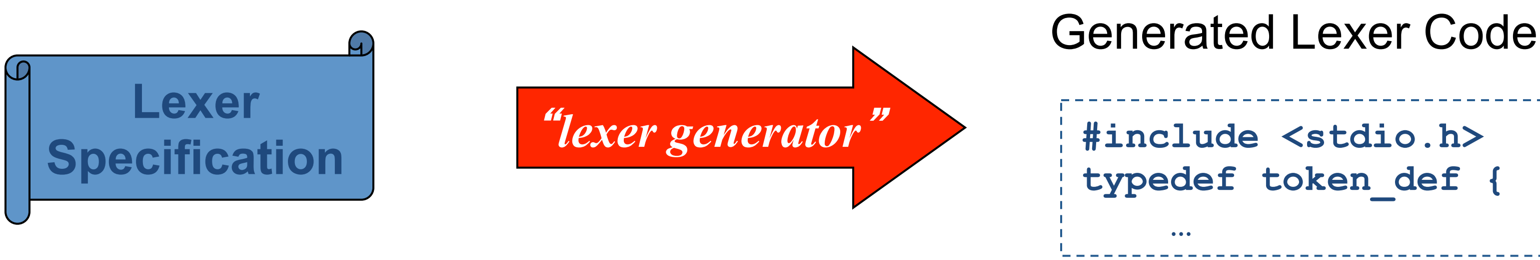
What is Lexing again?

Usually words/tokens/lexemes are represented symbolically:

Instead of: `class,Foo,{,...`

symbols: `CLASSDECL,ID("Foo"),LBRACK,...`

What we'd like – as much automated support as possible



- Most parts of a front-end are <u>generated</u> rather than written by hand
  - front-end issues are *quite* well-understood
- Tools for lexers: lex, ScanGen,…
- Tools for parsers: yacc, parsec, JLex, CUP, SableCC,…

Example: Micro programs

```
begin
  x := 7 + y;
  read(y,z);
end
```

Tokens for Micro

Micro Source

```
begin
  x := 7 + y;
  read(y,z);
end
```

C tokens

```c
typedef enum token_types {
    BEGIN, END, READ, WRITE,
    ID, INTLITERAL,
    LPAREN,RPAREN,SEMICOLON,
    COMMA,ASSIGNOP,
    PLUSOP,MINUSOP,SCANEOF
 } token;
```

Tokens for Micro

Micro Source

```
begin
  x := 7 + y;
  read(y,z);
end
```

```
--
-- define what a token is
--
data Token = BEGIN | END | READ | WRITE | ID String
          | INTLITERAL Int | LPAREN | RPAREN
          | SEMICOLON | COMMA | ASSIGNOP | PLUSOP
          | MINUSOP | SCANEOF
          deriving Show
```

```
data Maybe a = Just a | Nothing

-- Want to write:
scan :: String -> Maybe [Token]
scan = ……
```

Writing a scanner in Haskell

Here's its input-output behavior

input

"begin\n x:=7+y;\n read(y,z);\n end"

output

```
ghci> scan "begin\n x:=7+y;\n read(y,z);\n end"

  Just [BEGIN,ID "x", ASSIGNOP, INTLITERAL 7, PLUSOP, …, END, SCANEOF]

ghci> scan "begin\n x:=7+y & \n read(y,z);\n end"   -- illegal symbol

  Nothing
```

Formal definition of syntax – why is it necessary?

- Most languages allow float constants
  - e.g., 0.1, 10.01
- Should constants of the form ".1" or "10." be allowed as well?
- Consider lexing "1..10"

Formal definition of syntax – why is it necessary?

- Most languages allow float constants
  - e.g., 0.1, 10.01
- Should constants of the form ".1" or "10." be allowed as well?
- Consider scanning "1..10"
  - is it a range (i.e., 1, 2, …,10)?
  - or, two floats next to one another?
    - i.e., "1." followed by ".10"

Lexical Rules

- How would we specify the lexical rules of a computer language?
  - Regular expressions!
- Regular expressions are a simple formalism for accepting or rejecting strings.
- We specify lexers by writing a regular expression for each possible token.
  - Execute them concurrently
  - Have disambiguation rules

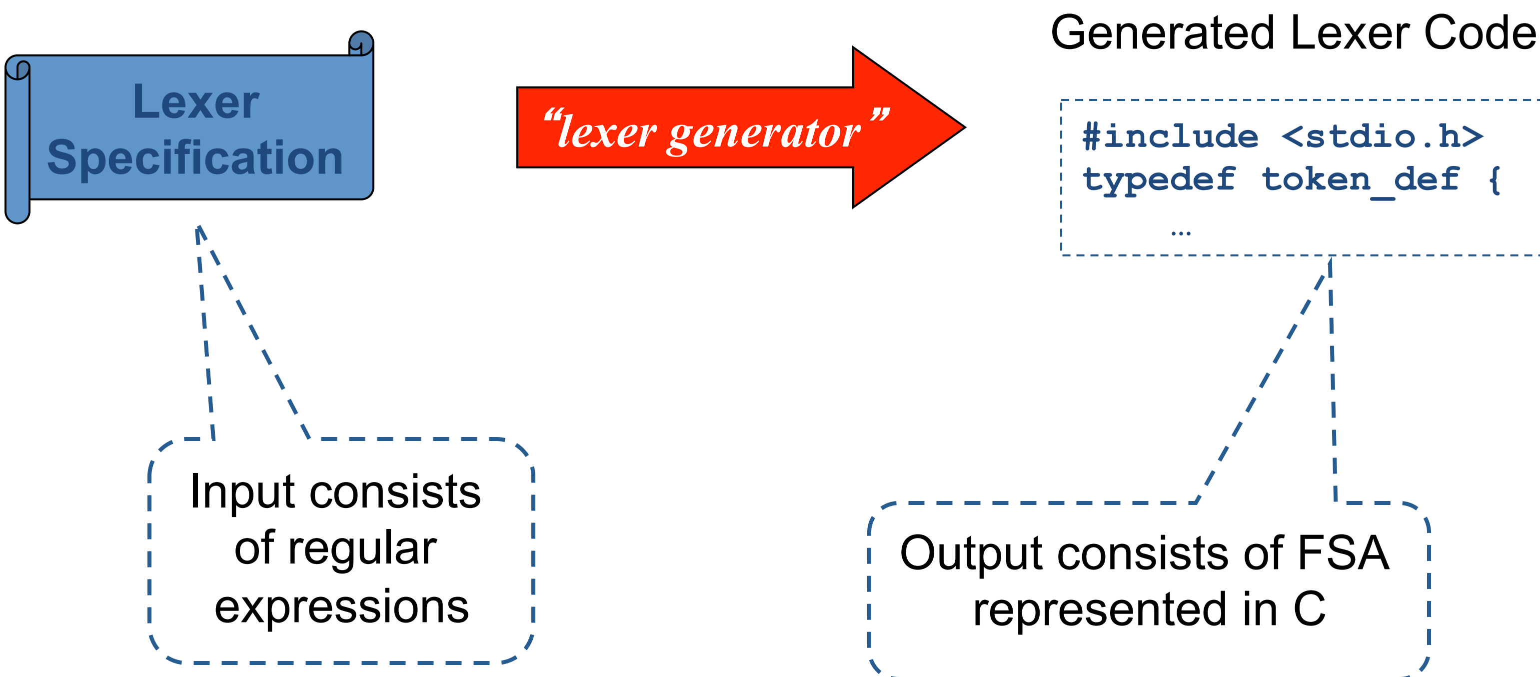
Formal tools for lexical syntax (I)

- Regular expressions (RE)
  - these are patterns which may be applied to a sequence of characters
  - resulting in either a <u>success</u> or a <u>failure</u>
- A "token" (e.g., an identifier) may be defined as any string on which a particular RE succeeds

Formal tools for lexical syntax (II)

- Finite State Automata (FSA)
  - these are machines which may be applied to a sequence of characters
  - resulting in either a <u>success</u> or a <u>failure</u>
- Given an RE, one may <u>automatically</u> construct an FSA which recognizes precisely the same tokens
  - …and vice versa
  - FSAs are easier to program that REs

Lexer generators



Regular expressions

A **regular expression** is anything of the following form

- $\varnothing$
- λ
- any string s
- if A,B are regular expressions, then so are:
  - A | B
  - A . B
  - A*

What regular expressions match

Each regular expression defines a set of strings that it matches

- $\emptyset$ matches {}
  - i.e., it fails to match any string
- λ matches {“”}
  - i.e., it matches only one string – the empty string
- string s matches only { s }
  - e.g., the set of matches of reg-exp `beavis` is just the singleton set { `beavis` }

What regular expressions match

Each regular expression defines a set of strings that it matches

- ∅ matches {}
  - i.e., it fails to match any string
- λ matches {“”}
  - i.e., it matches only one string – the empty string
- string s matches onl
  - e.g., the set of mat singleton set { `bea`

WARNING: Some texts uses λ as meaning either the regular expression or the empty string. This can be a bit confusing.

What regular expressions match

Let A and B be regular expressions.

- match(A | B) = match(A) ∪ match(B)
  - i.e., any string matched by A or matched by B
  - "|" is called alternation
- match(A B) = { a b : a ε match(A), b ε match(B) }
  - If `beavis` ε match(A), `butthead` ε match(B), then `beavisbutthead` ε match(A B)
  - called sequencing

What regular expressions match

Let A be a regular expression. Then set match(A*) is defined by:

- “” ε match(A*)
- If strings a ε match(A), a’ ε match(A*), then aa’ ε match(A*)

match(A*) may be thought of as all strings $a_1 \ldots a_n$ where each a_i ε match(A*) and $0 \leq n$

What regular expressions match

Example:

match((beavis)*) = { “”, beavis, beavisbeavis, …}

A* is called the “Kleene closure” of A after Stephen Kleene (1909-1994), famous 20th century logician and mathematician.

Some shorthand, etc.

- Given two regular expressions *M* and *N*
  - Can write *M* | *N* to mean either *M* <u>or</u> *N*
  - Can use parentheses (*M*)

(0 | 1 | 2 | 3 | 4) (0 | 1) (0 | 2 | 4 | 8) { ...}

Shorthand:

Use square brackets for list of alternative

- i.e., (X | Y) matches the same strings as [XY]

Can use x-y for consecutive ranges

[0-4][01][0248] { ... }

Some shorthand, etc.

Common abbreviations:

- period '.' matches anything
- (^a) is a shorthand matching anything but an 'a'
- Empty regular expression sometimes written $\in$

((X Y) | $\in$) (B | A) (0 | 1 | $\in$)   { ...}

Some shorthand, etc.

Question:

How might you write "optional" regular expressions

M? matches anything M matches or the empty string

match((XY)?[BA][01]?) = { … }

Question:

Given a regular expression *M* define *M*+

where *M*+ means one or more of *M*

match([0-9]+) = { … }

Ambiguities

- A syntax tends to be defined by multiple regular expressions
  - can lead to ambiguity
- For input "begin", reg-exp [a-z]+ matches
  - each initial prefix: "b", "be", "beg", etc.
  - Which token(s) do we choose?
- For input "begin", both of the following reg-exps succeed:
  - begin
  - [a-z]+
  - Which reg-exp applies?

Disambiguation Rules

- **Longest Match**
  - The longest string that matches a regular expression is taken as the next token.
  - Example

    [a-z]+ keeps looking for lower case letters, it does not always stop after the first letter.
  - Sometimes called the "maximal munch" rule.
- **Rules priority**
  - If two regular expressions both match, the *first* regular expression determines the token type.
  - Example
    - "if" is tokenized as a IF, not as an ID.

Example Lexer Specification

```
if                                              { return IF; }
[a-z][a-z0-9]*                                  { return ID; }
[0-9]+                                          { return NUM; }
([0-9]+“.”[0-9]*) | ([0-9]*“.”[0-9]+)           { return REAL; }
\“[^\”]*\”                                      { return STRING; }
(“- -”[a-z]* “ \n”) | (“ ”|“\n”|“\t”)+          { /* do nothing */ }
.                                               { error(); }
```

Next Time

- Continue learning the formal concepts behind generators
  - lexers: regular expressions & finite automata
  - parsers: context-free grammars (CFGs)

Keyword Pragmatics

- How do you know what are keywords?
  - Look at the reserved words for the language
- Example: Java
  - Reserved keywords → lexical keywords
    - In Java "if, then, else, …" are reserved keywords
  - Reserved literals → may or may not map to individual lexical entities.
    - In Java "null, true, false' are reserved literals.
  - Suggestion: map reserved literals to identifiers, and handle the reserved status later in the compiler.

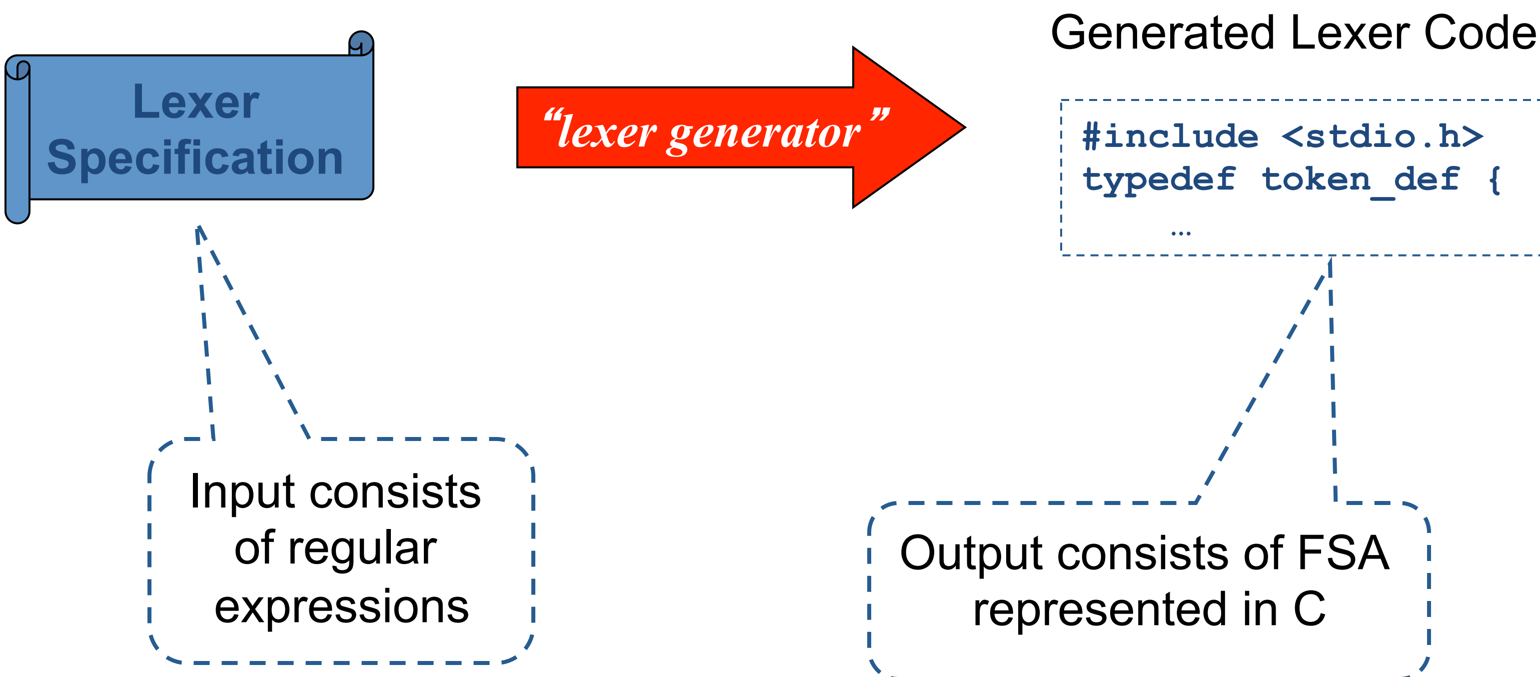
Introduction to Lexing

THEORY OF SCANNING

Today's Lecture

- **Continue discussion of front-end**
  - I.e., the earliest phases of the compiler
    - the front-end answers the question "is the input program really a program?"
  - The theory underlying lexing
    - allows us to understand what lexer generators do
- **Approach**
  - What is behind "scanner generators"?
  - Last time: Regular expressions
  - This time: Finite Automata

Next time: Lexer generators



Review: Regular Expressions describe sets of strings

Let S be an alphabet (i.e., a set of symbols)

1. $\emptyset$, λ, $a \in S$ are all regular expressions (primitive r.e's)

2. If r1 and r2 are r.e's, then so are

   r1 | r2 (alternation)

   r1 · r2 (concatenation)

   r1* (Kleene closure)

   (r1) (parentheses)

3. Only 1- 3 give regular expressions

* sometimes "·" is omitted

RE's describe sets of strings (i.e., *languages*)

- L(∅) describes the empty set of strings over S: { }
- L(λ) describes the set with just the empty string: {"" }
  (Note that these are **not** the same set!)

•L(a) = {a}

- L(r1 | r2) = L(r1) ∪ L(r2)
- L(r1 · r2) = { s1 s2 : s1 ∈ L(r1) & s2 ∈ L(r2) }
  Ex: if "abc" ∈ L(r1) and "de" ∈ L(r2), then
  "abcde" ∈ L(r1 · r2)

Kleene closure

$$L(r^*) = L(\lambda) \cup L(r) \cup L(r \cdot r) \cup L(r \cdot r \cdot r) \cup \ldots$$

Ex: What is $L(a^*)$?

$$\begin{aligned} L(a^*) &= \{\text{""}\} \cup L(a) \cup L(a \cdot a) \cup L(a \cdot a \cdot a) \cup \ldots \\ &= \{\text{""}, \quad a, \quad aa, \quad aaa, \quad \ldots\} \\ &= \text{set of strings "a...a" (possibly 0 length)} \end{aligned}$$

Ex: What is $L(a \cdot (a^*))$?

Some shorthand

- Frequently, we drop the "L" from "L(r)"
  That is, we treat a regular expression as a set

- We abbreviate $r1 \cdot r2$ by dropping the $\cdot$

Ex: So, ab is used as both a regular expression and a string

Further examples

S = {a,b}. Write a RE with language {ab, aba, abb, abaa,…}.

(ab) (a|b)*

Must start with "ab"

May have some number of "a" or "b" following.

Yet more examples: C identifiers

"An identifier is a sequence of letters and digits. The first character must be a letter; the underscore _ counts as a letter. Upper and lower case letters are different. Identifiers may have any length…"

From Kernighan and Ritchie, 2nd Edition, Appendix A, pp192.

Question: how would we write that as a regular expression?

C identifiers, cont'd

letters = (a | b | … | z | A | … | Z | _)

digits = (0 | 1 | … | 9)

identifier = letters (letters | digits)*

Finite State Automata

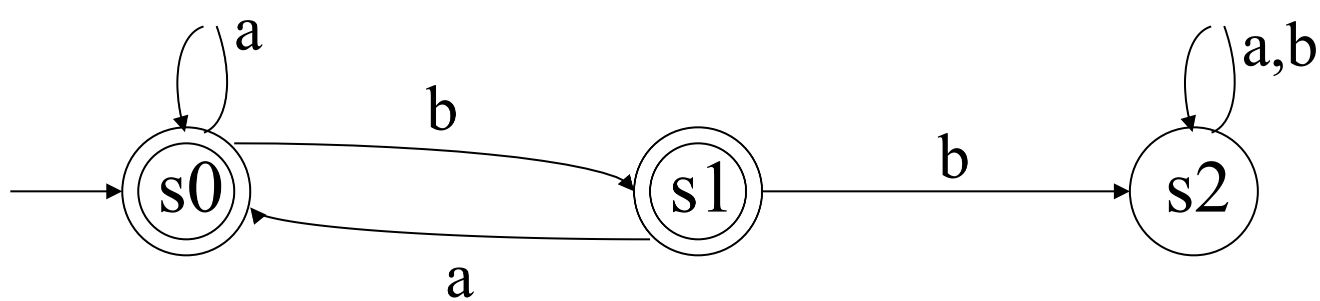
Some essentially equivalent terms

- Finite State Automaton (FSA)
- Deterministic Finite State Automaton (DFA)
- Finite State Machine (FSM)
- Deterministic Finite State *Accepter* (DFA)

"Deterministic" means that the transition relation between states is a *function*: that is, given a state and an input, there is one and only next state.

There are "non-deterministic" automata as well (we'll consider them in a moment).

Finite State Machine



- States
- Transitions between states
- The *language* of the FSM

Finite State Automata

- A Finite State Automaton (FSA) consists of five elements:

  (1) a finite set of inputs (an alphabet A)

  (2) a finite set S of states

  (3) a subset Y of S (Yes states aka "final states")

  (4) an initial state s0 of S

  (5) a next-state function F: S × A --> S
- A FSA, M, is a *pentuple*:

  M = (A, S, Y, s0, F)

An Example

(1) $A = \{a, b\}$

(2) $S = \{s0, s1, s2\}$

(3) $Y = \{s0, s1\}$

(4) s0, the initial state

(5) $F: S \times A \rightarrow S$ is

$F(s0,a) = s0$, $F(s1, a) = s0$, $F(s2,a) = s2$

$F(s0,b) = s1$, $F(s1,b) = s2$, $F(s2,b) = s2$

An Example - II

- State table:

F	a	b
s0	s0	s1
s1	s0	s2
s2	s2	s2

inputs

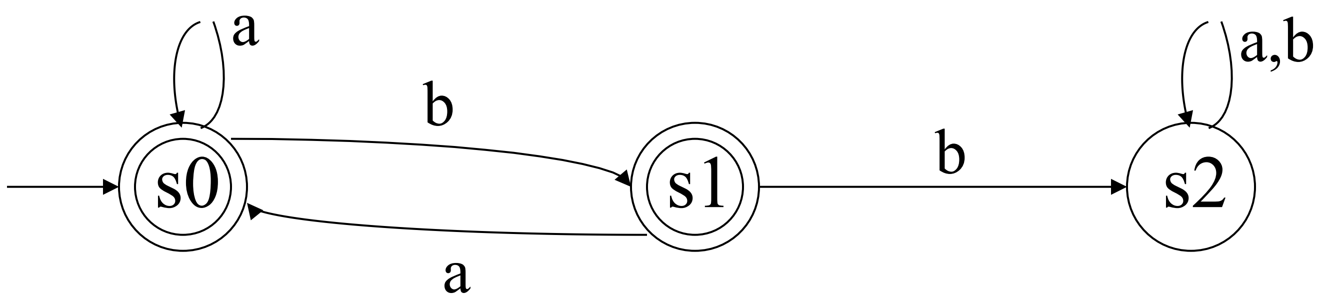
present state

next states

- A transition to a next state is determined by the current state and the input

A State Diagram

- A state diagram shows the initial state, the final (yes, or accepting) states and the transitions among states
- The symbol strings that cause the automaton to start in the initial state and end in a final state are the *language* accepted by the automaton A. Written "L(A)".



Example : Find a DFA recognizing all strings on A={a,b} starting with "ab"

That is, all strings like: ab, aba, abb, abaa, abab,…

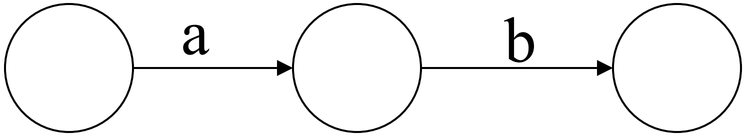
Recall that a DFA is a pentuple: ({a,b} , S, Y, s0, F)

This problem boils down to determining:

- the set of states S
- the set of "yes" states Y
- the initial state s0
- the transition function F

Example 1 (cont’d)

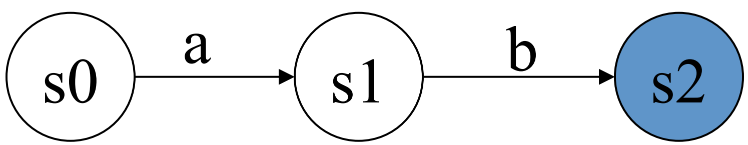
Starting with “ab” means that our DFA will resemble the following:



Example 1 (cont’d)

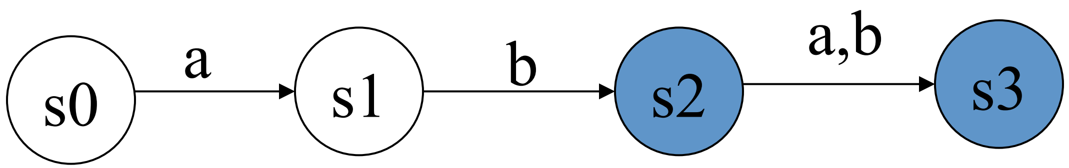
Furthermore, we know

- The initial state is s0,
- Y = {s2,…?…} (blue),
- F(s0,a) = s1, F(s1,b) = s2



Example 1 (cont' d)

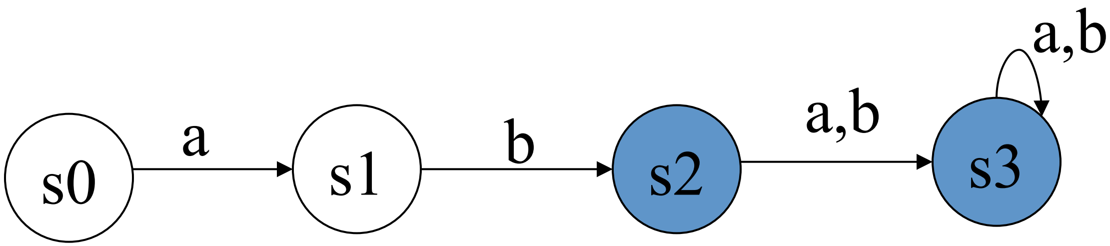
Now, if we' re in s2, and we see an "a" or a "b" then we should proceed to another "yes" state



- The initial state is s0,
- Y = {s2,s3,…?…},
- F(s0,a) = s1, F(s1,b) = s2, F(s2,a) = s3, F(s2,b) = s3

Example 1 (cont' d)

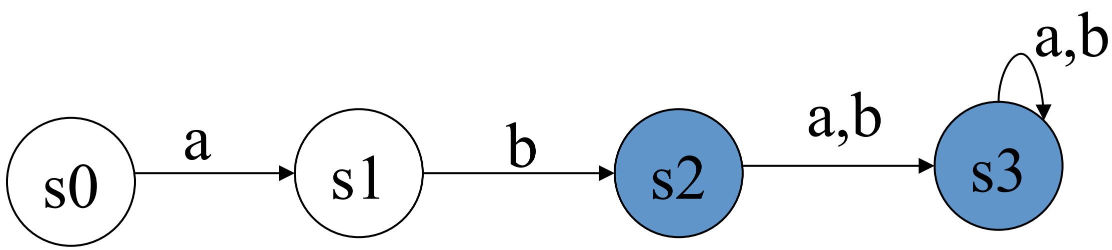
Now, if we' re in s3, and we see an "a" or a "b" then we should remain there.



- The initial state is s0,
- Y = {s2,s3,…?…},
- F(s0,a) = s1, F(s1,b) = s2, F(s2,a) = s3, F(s2,b) = s3, F(s3,a)=s3, F(s3,b)=s3

Example 1 (cont'd)

Notice that this accepts the language: {ab, aba, abb, …}

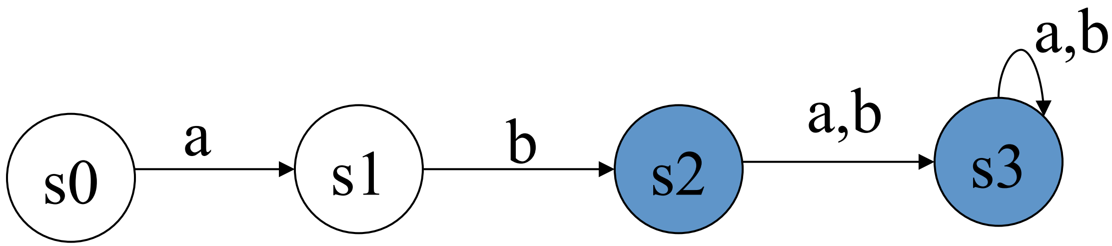


- The initial state is s0,
- Y = {s2,s3},
- F(s0,a) = s1, F(s1,b) = s2, F(s2,a) = s3, F(s2,b) = s3, F(s3,a)=s3, F(s3,b)=s3

So, are we done?

Example 1 (cont'd)

Notice that this accepts the language: {ab, aba, abb, …}



- The initial state is s0,
- Y = {s2,s3},
- F(s0,a) = s1, F(s1,b) = s2, F(s2,a) = s3, F(s2,b) = s3,

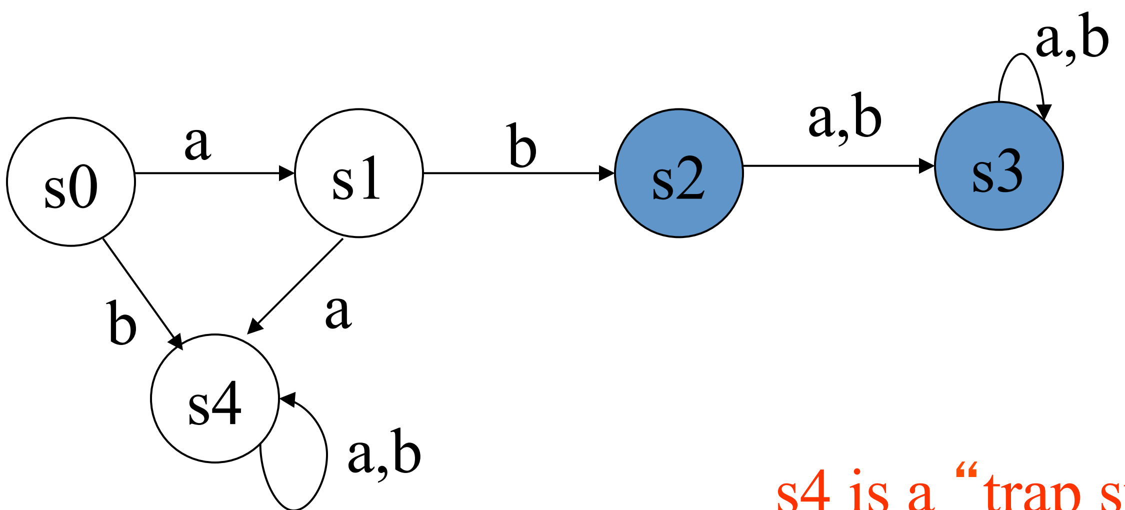
F(s3,a)=s3, F(s3,b)=s3

So, are we done?

What are F(s0,b) and F(s1,b)? Recall that F is a *function*.

Example 1 (cont’d)

This kind of “error” state (like s4) is usually left out

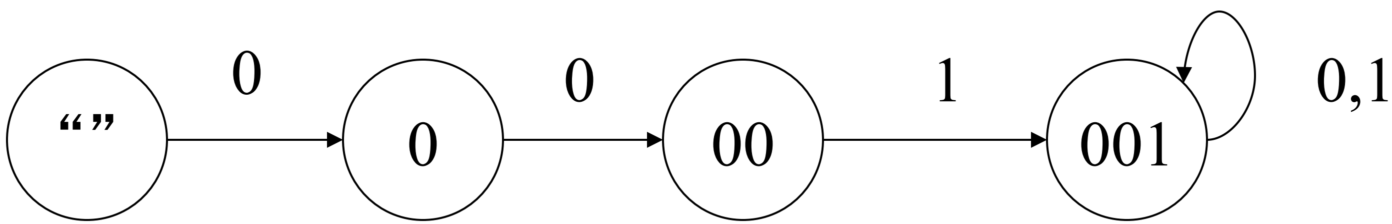


s4 is a “trap state”

- The initial state is s0,
- Y = {s2,s3,s4},
- F(s0,a) = s1, F(s1,b) = s2, F(s2,a) = s3, F(s2,b) = s3, F(s3,a)=s3, F(s3,b)=s3, F(s0,b) = s4, F(s1,a) = s4, F(s4,_) = s4

Harder example

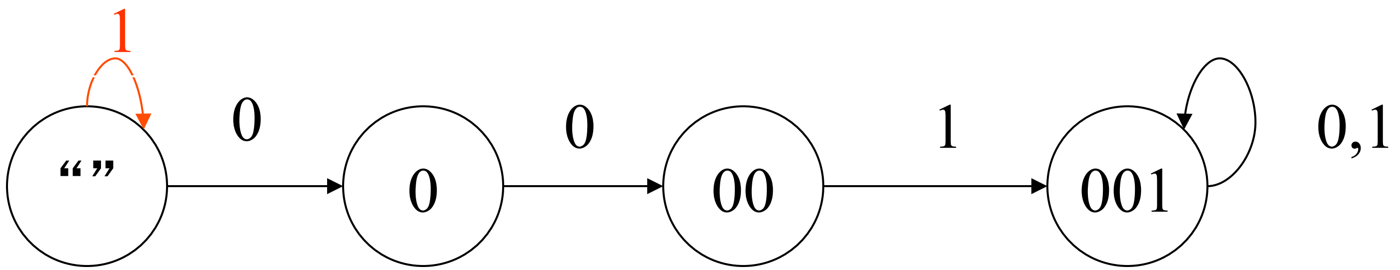
Find a DFA that recognizes all strings over {0,1} except those containing “001”



First, “set the trap”

Harder example

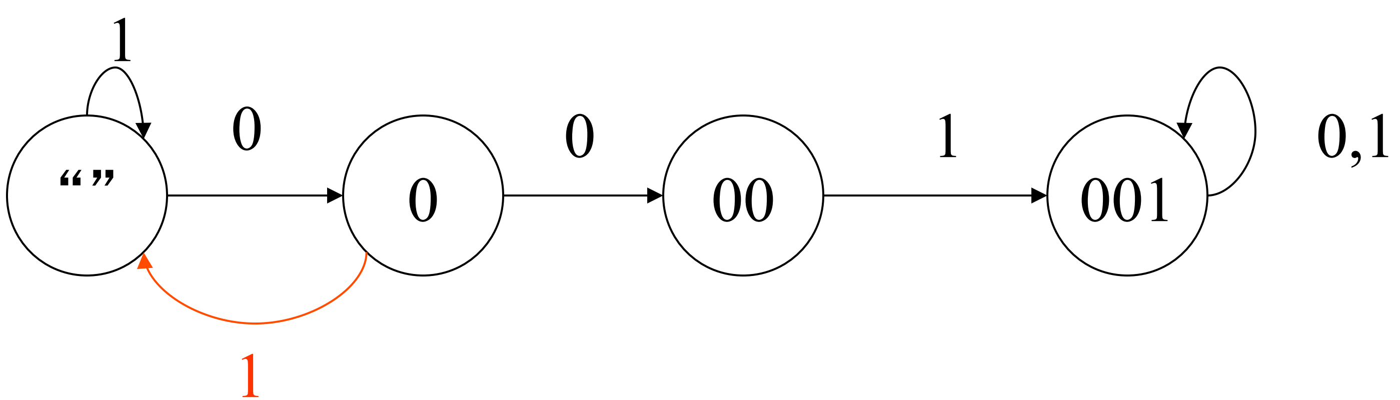
Find a DFA that recognizes all strings over {0,1} except those containing "001"



Next, fill in all of the transitions

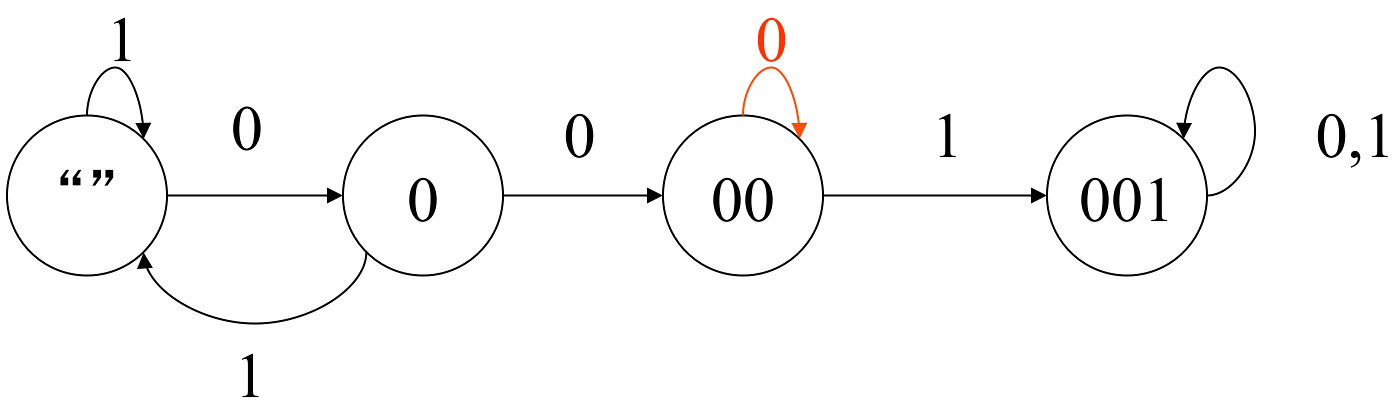
Harder example

Find a DFA that recognizes all strings over {0,1} except those containing "001"



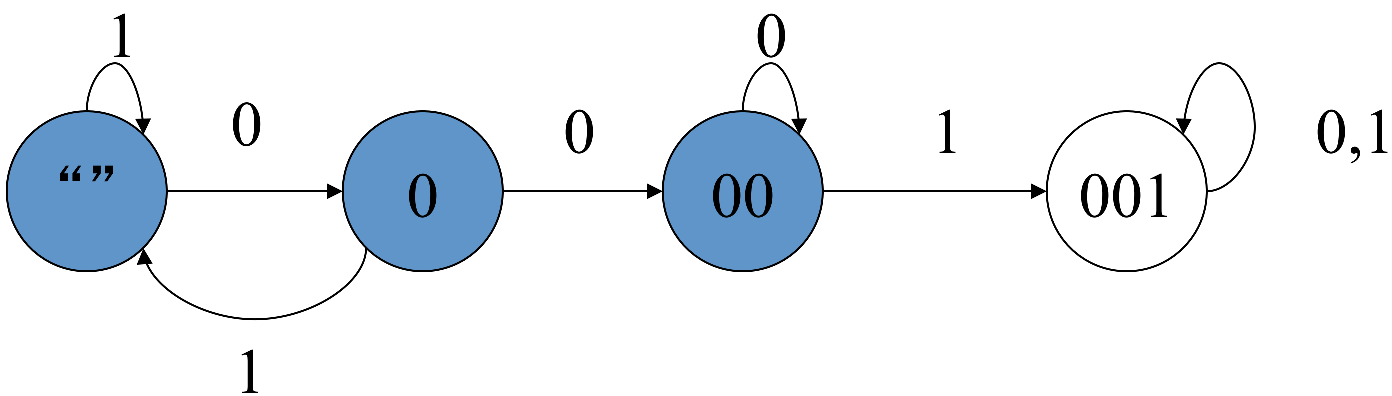
Harder example

Find a DFA that recognizes all strings over {0,1} except those containing “001”



Harder example

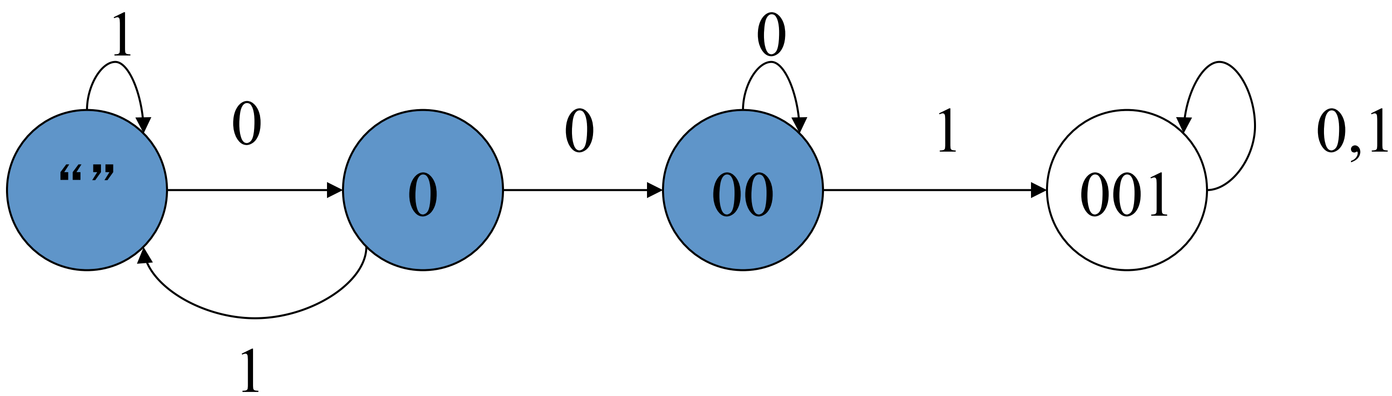
Find a DFA that recognizes all strings over {0,1} except those containing “001”



Now, any string that doesn’t end up in “001” is accepted

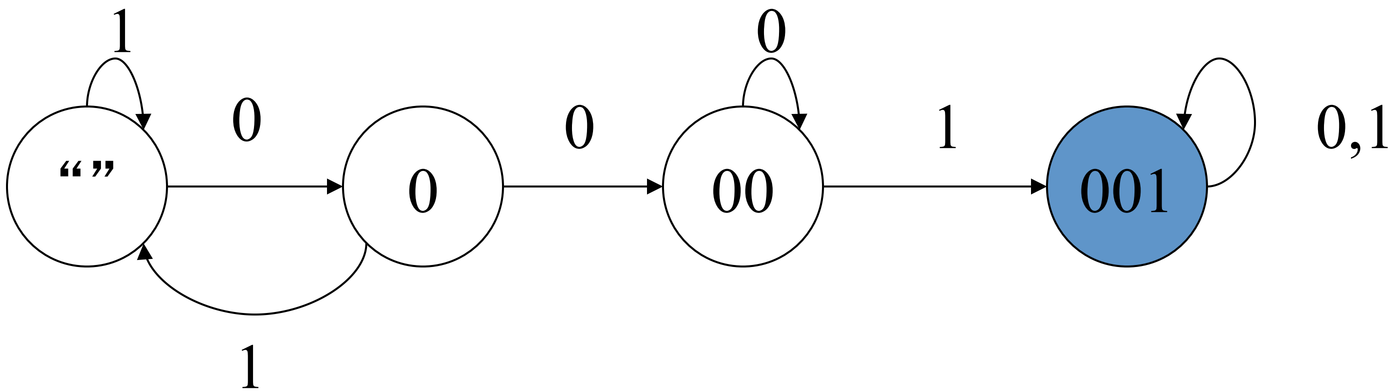
Question

Question: What if we wanted to find a DFA that recognized every string containing "001"? How would that differ from our previous DFA?



Answer

Reverse the "yes" states:



Finite State Machines

- A Finite State Machine (FSM) is a Finite State Automata with *output*
  - a FSM has an output alphabet, Z, and
  - an output function g: $S \times A \rightarrow Z$

(1) $A = \{a, b\}$-- the input alphabet

(2) $S = \{s0, s1, s2\}$-- the set of states

(3) $Z = \{x, y, z\}$-- the output alphabet

(4) s0, the initial state

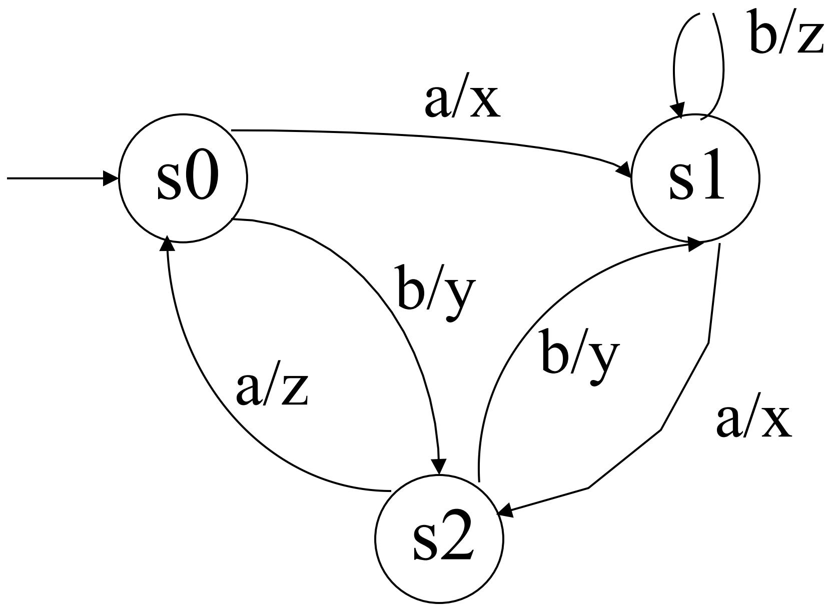
(5) f: $S \times A \rightarrow S$ -- the transition function

f(s0,a) = s0,	f(s1, a) = s0,	f(s2,a) = s2
f(s0,b) = s1,	f(s1,b) = s2,	f(s2,b) = s2

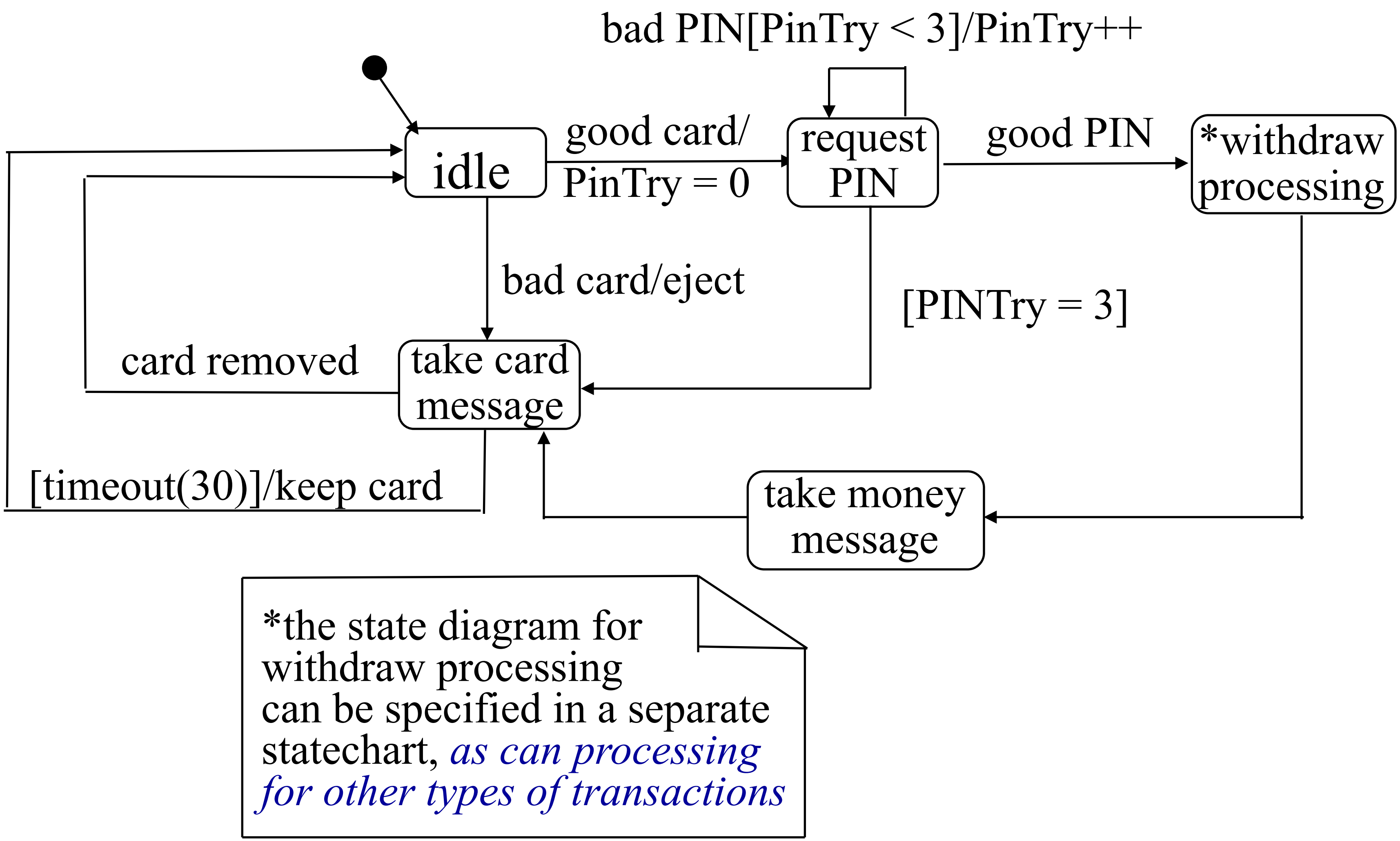
(6) g: $S \times A \rightarrow Z$ -- the output function

g(s0,a) = x,	g(s1, a) = x,	f(s2,a) = z
f(s0,b) = y,	f(s1,b) = z,	f(s2,b) = y

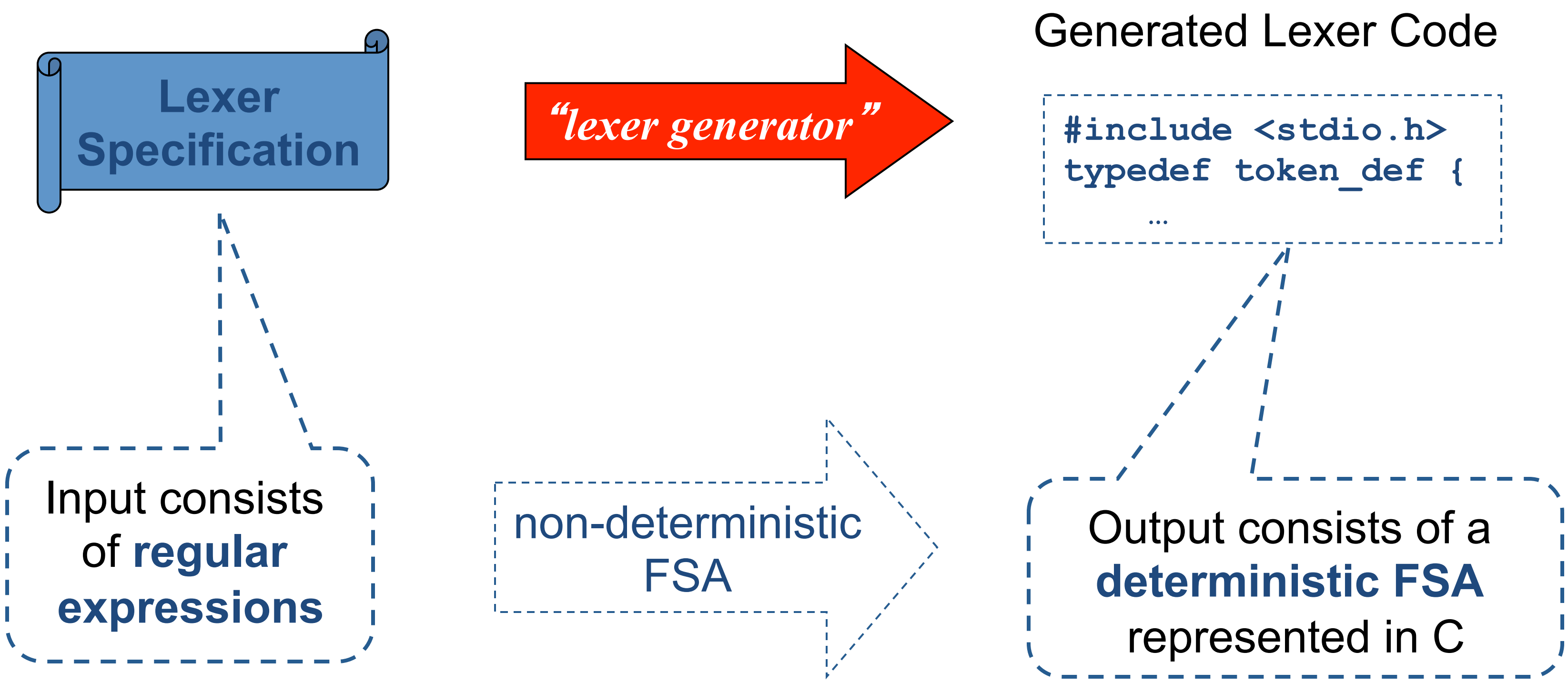
A FSM State Diagram



A PARTIAL STATE MACHINE FOR AN ATM



Lexer generators



Today's Lecture

- **We'll look at an example scanner generator**
  - Generate a scanner for Micro
- **Next time:** start parsing

Scanner Generators

- Take an input file specifying the lexical syntax of your language
    - usually in the form of regular expressions
    - …and including other helper functions, token definitions, etc.
- …and generate code for a scanner
- Many such generators
    - Lex, Flex, ScanGen
        - generate C code
    - JLex, Sable, Cup
        - generate Java
    - Lex was the first such scanner generator
- We'll create a scanner for Micro* using "Flex"
    - this is the GNU version of Lex and is freely available

* Actually, for a slightly different language

Flex Specification for the Micro language

The format of a **Flex** specification file is:

```
%{
        #include's
%}
        special "short-hand" definitions
%%
        lexical specification (i.e., regular expressions)
%%
        other C procedures (possibly including main)
```

`#include`'s

Flex specification includes "chunks" of C code, which, may use library functions:

```
%{
int yywrap(void){ } ;

/* call C library function atof() below */
#include <math.h>
%}
```

* these are enclosed by "`%{`" and "`}%`"

Special shorthand section

Flex allows the definition of abbreviations for frequently used regular expressions

```
DIGIT     [0-9]
ID        [a-z][a-z0-9]*
```

Recall Micro

Micro Source

```
begin
  x := 7 + y;
  read(y,z);
end
```

C tokens

```c
typedef enum token_types {
    BEGIN, END, READ, WRITE,
    ID, INTLITERAL,
    LPAREN,RPAREN,SEMICOLON,
    COMMA,ASSIGNOP,
    PLUSOP,MINUSOP,SCANEOF
 } token;
```

Lexical specification section

Typical scanner action consists of regular expression and action

```
begin | end | read | write
                          { printf( "keyword: %s\n", yytext ); }
```

regular expression

action: C code performed if this r.e. matches

Lexical specification section

Typical scanner action consists of regular expression and action

```
begin | end | read | write
                          { printf( "keyword: %s\n", yytext ); }
```

special variable "`yytext`" contains the string it just matched

Other cases

```
{ID}            printf( "An identifier: %s\n", yytext );
":="            printf( "Assignment: %s\n", yytext );
"+"|"-"|"*"|"/" printf( "An operator: %s\n", yytext );
"("             printf( "Left parenthesis: %s\n", yytext );
")"             printf( "Right parenthesis: %s\n", yytext );
","             printf( "Comma: %s\n", yytext );
";"             printf( "Semicolon: %s\n", yytext );
[" "\t\n]+      /* eat up whitespace */
.               printf( "Unrecognized: %s\n", yytext );
```

C procedures section

To make a standalone lexer, define main here, as in:

```
main( argc, argv )
int argc;
char **argv;
{
    ++argv, --argc;   /*  skip over program name */
    if ( argc > 0 )
            yyin = fopen( argv[0], "r" );
    else
            yyin = stdin;

    yylex();

}
```

C procedures section

To make a standalone lexer, define main here, as in:

```
main( argc, argv )
int argc;
char **argv;
{
    ++argv, --argc;   /*  skip over program name */
    if ( argc > 0 )
            yyin = fopen( argv[0], "r" );
    else
            yyin = stdin;

    yylex();

}
```

Special variables **yylex** and **yyin**

Running flex

- Simply apply it to the file with the flex specification:
  - flex micro.flex
- This generates a C file containing your lexer
  - called “lex.yy.c”
- Compile as usual (if it’s standalone)
  - gcc lex.yy.c