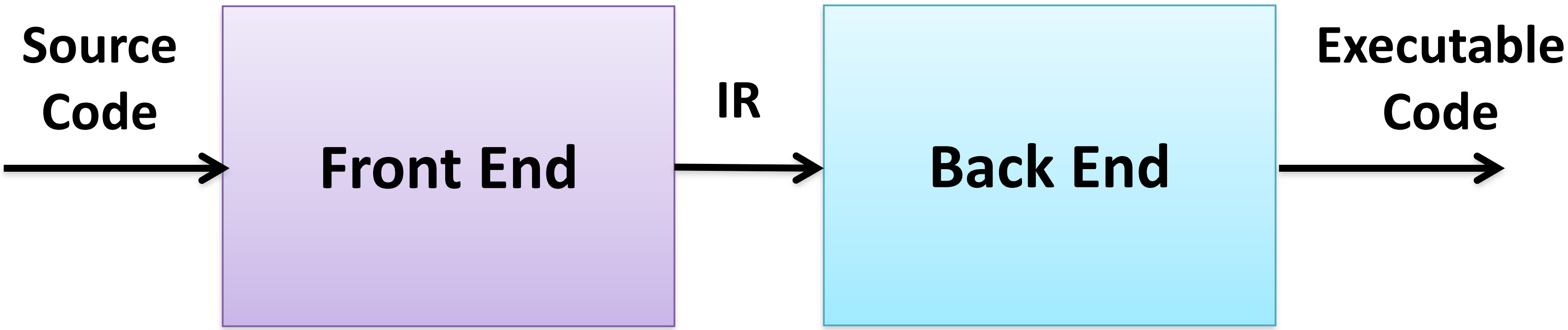
Introduction to Compilers, Part 1

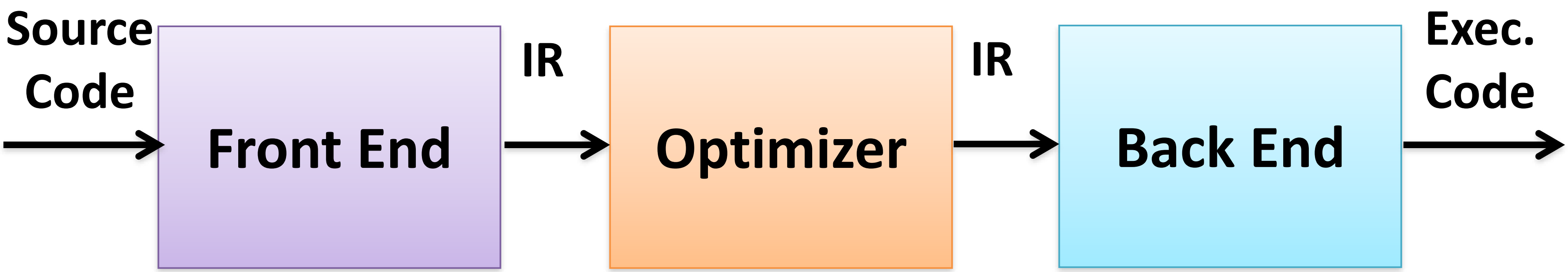
ITP 435
Week 10, Lecture 1

Compiler as a Black Box

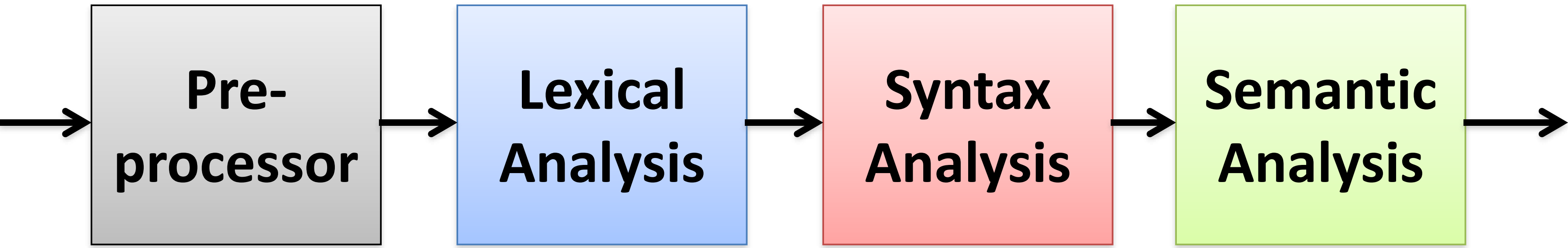
Basic "Two Stage" Compiler



Three Stage "Optimizing" Compiler



Front End in a C++ Compiler



Preprocessor

- Processes all # directives to generate the final C++ code which will be compiled
- Example 1:

```
#include "dbg_assert.h"
// dbg_assert.h code is essentially copy/pasted at this line
```

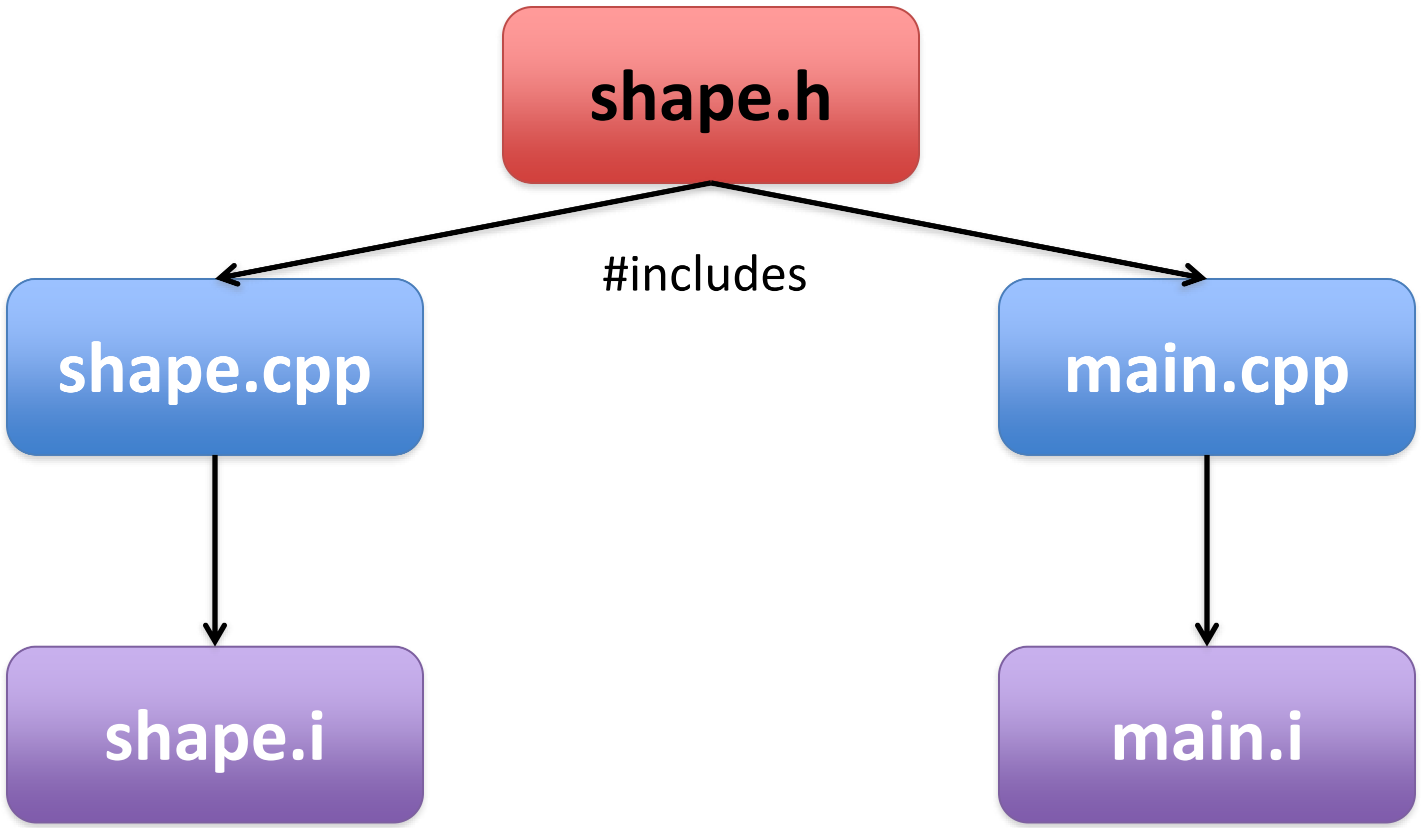
- Example 2:

```
// Compile this only in a debug build
#ifdef _DEBUG
// Random debug code…
#endif
```

- Example 3:

```
// Replaces MAX_POOL_SIZE with 256 in code
// (Breaks Rule #2 in Effective C++)
#define MAX_POOL_SIZE 256
```

Preprocess Output in Visual Studio



Lexical Analysis aka Scanning

- Reads through the file and makes sure each “token” is valid
- Checks for any words or symbols which couldn’t be valid C++
- Example 1:
  ```
  © = 5; // Error: © is not a valid symbol
  ```
- Example 2:
  ```
  int 12xyz; // Error: Not a keyword or number, and variable names
  // can’t start with numbers.
  ```
- Example 3:
  ```
  ( { * { { | % agagaga // Success: These are all valid tokens
  ```
- Natural Language Analogies:
  ```
  The dog 0wn3d the cat. // Error: 0wn3d not a valid word
  cat dog. The owned // Success: All valid words/tokens
  ```

Lexical Analysis

- Example of token generation for main.cpp:

```
int main(int argc, char* argv[])
{
    return 0;
}
```

1.	int
2.	main
3.	(
4.	int
5.	argc
6.	,
7.	char
8.	*

9.	argv
10.	[
11.	]
12.	)
13.	{
14.	return
15.	0
16.	;
17.	}

Syntax Analysis aka Parsing

- Makes sure series of tokens follows the grammar rules.
- Does NOT check if types match, variables are defined, etc.
- Example 1:

```
bool Function1()
{
   return true // Syntax Error: Semi-Colon Missing
}
```

- Example 2:

```
if test != true // Syntax Error: Missing Parenthesis
```

- Example 3:

```
Shape myShape;
int a = myShape; // Success: Syntactically Correct
```

- Natural Language Analogies:

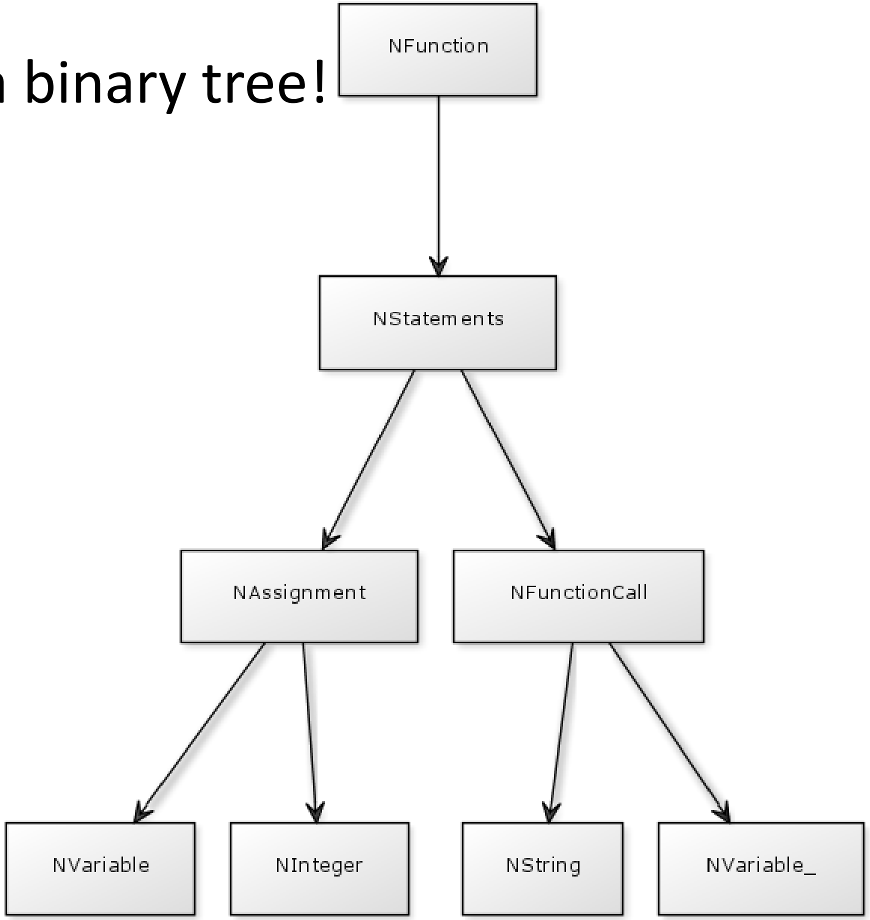
```
The cat own dog. // Error: Wrong conjugation, missing article
The dog flew with its wings. // Success: Syntactically Correct
```

Syntax Analysis

- The output of syntax analysis is an IR, the simplest of which is an Abstract Syntax Tree (AST)
- Note the AST is *not* necessarily a binary tree!
- It's more like a b-tree in a sense

- Example:

```
void function()
{
    int i = 0;
    printf("%d", i);
}
```



Semantic Analysis

- Makes sure the meaning of the code makes sense.
- Checks that functions/variables are declared in scope, types are correct, and everything of this nature.
- Example 1:

```
Shape myShape;
int a = myShape; // Error: Cannot assign Shape to int
```

- Example 2:

```
int a, b, c;
d = a; // Error: Variable "d" undefined
```

- Natural Language Analogies

```
The dog flew with its wings. // Error: wings not a member of dog
```

Semantic Analysis

- Two options:
  1. Traverse through the tree checking for semantic errors (really complicated)
  2. Check for semantic errors as the initial AST is being built (most common approach)

- We won't be doing this part for PA6, because we'll assume semantically valid programs

Back End

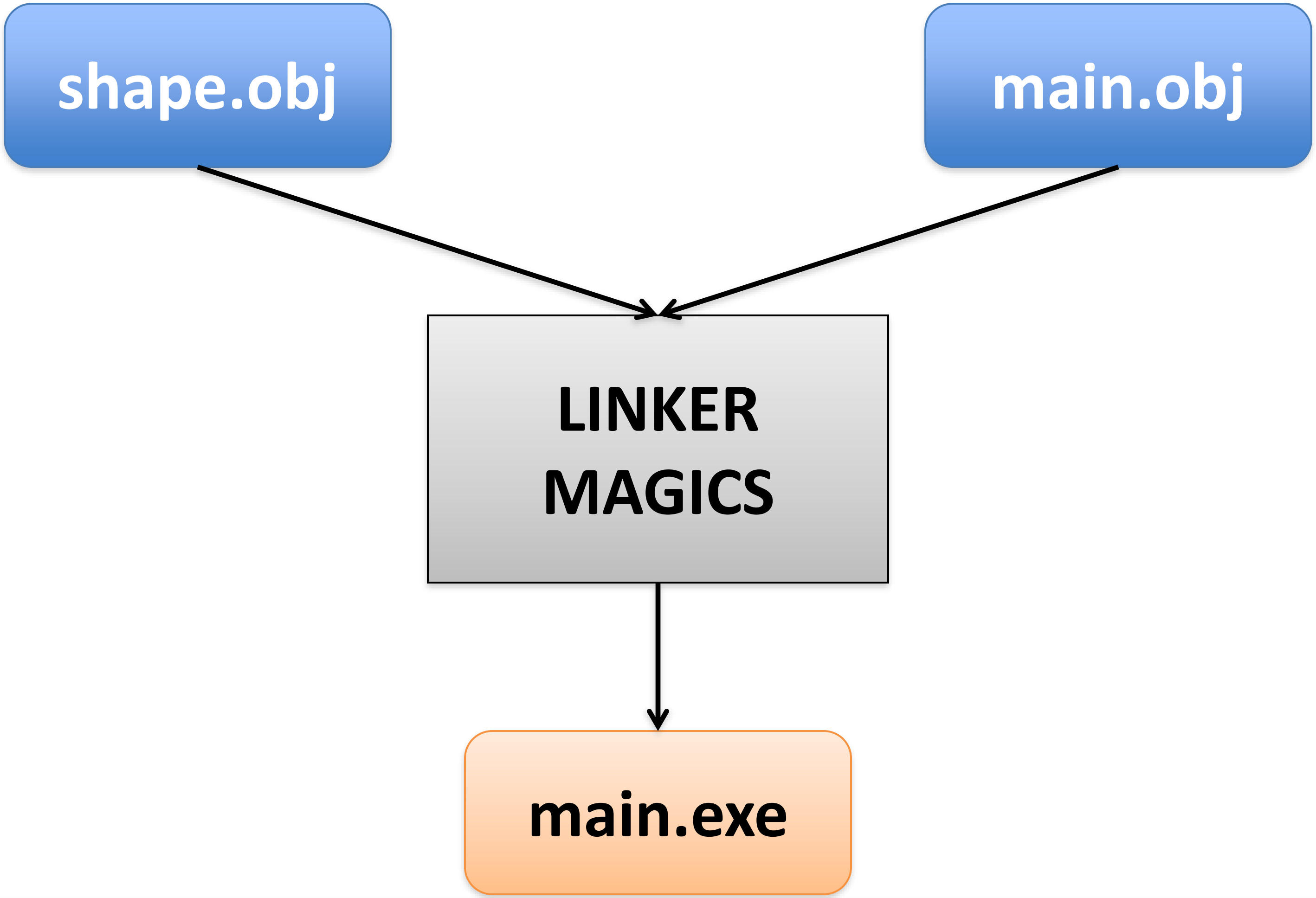
Instruction Selection
Register Allocation
Instruction Scheduling

Linker

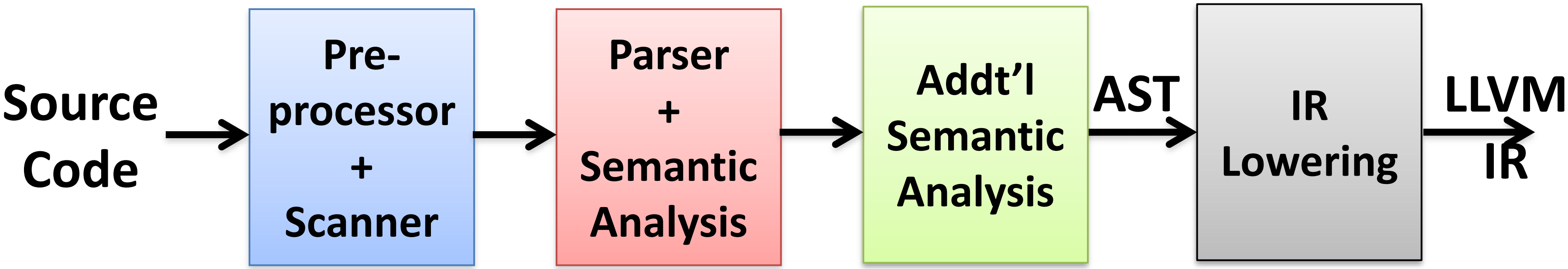
- Takes all object files and combines them into an executable
- Any external ".lib" libraries are also included
- Ensures external symbols are implemented somewhere
- Moves code to final memory locations
- Natural Language Analogy:

```
// If there is no Chapter 27, this is a linker error:
The dog ate the cat, as outlined in Chapter 27.
```

Linking

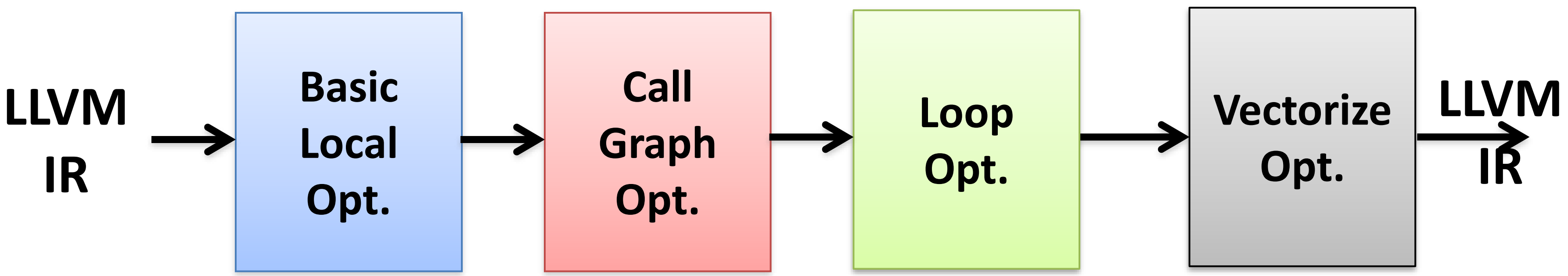


Stages of a Modern Compiler (Clang Frontend)



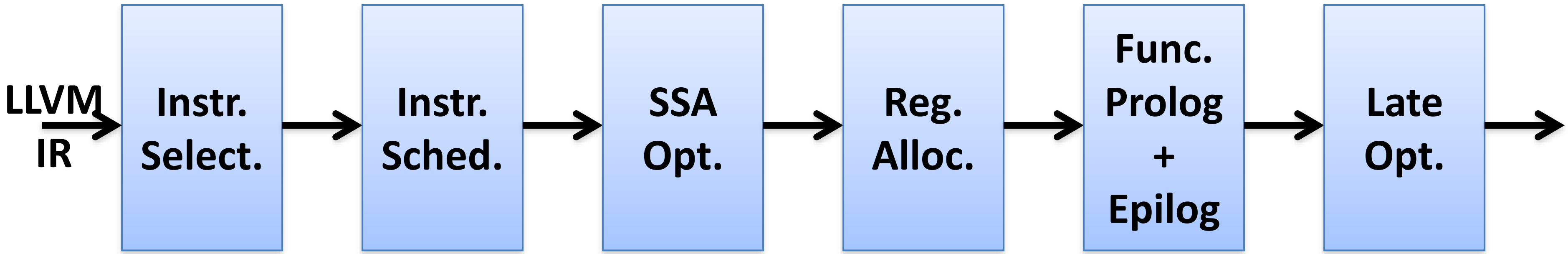
- Clang – C Language Front End for LLVM (default C/C++ compiler on Mac OS X and FreeBSD)

Stages of a Modern Compiler (Clang Optimizer)



- LLVM includes ~100 optimization passes – this is (approximately) the default grouping of passes with the -O2 compiler flag

Stages of a Modern Compiler (Clang Backend)



Lexer or Scanner

- Code that performs the lexical analysis
- Writing this code manually is not good because:
  - Very error prone
  - Pain to write
  - Pain to maintain
- The process of creating one from scratch is a hugely tedious endeavor

Scanner Problems

- A C++ scanner needs to recognize the correct keyword token in here:

```
_new
new_
_new_
new
newnew
new_new
```

- Writing code to handle this in addition to all the other keywords would be painful

lex to the rescue!

- Developed in 1975 in Bell Labs
- One of the two developers was Eric Schmidt (yes, that one)

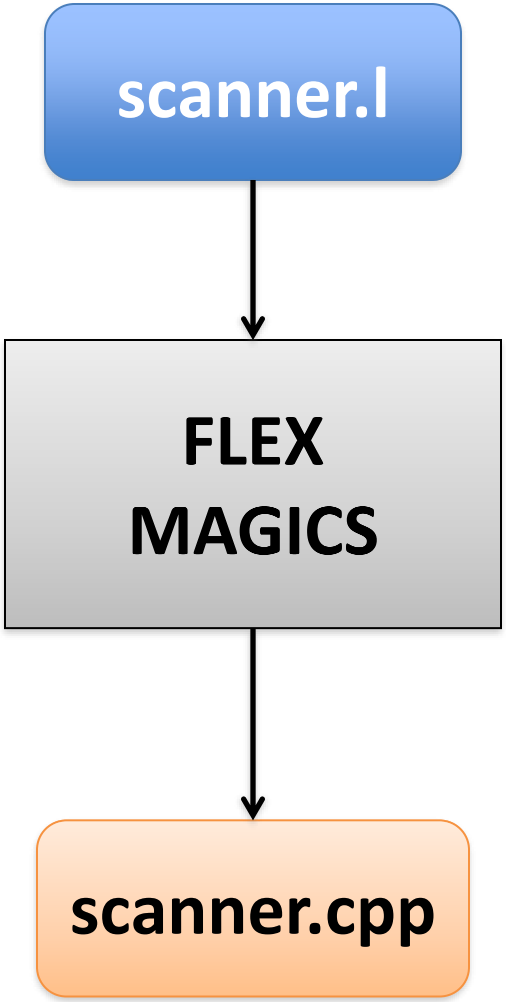
flex

- Fast lexical analyzer
- Written in 1987 by Vern Paxson

- Released as a faster, better, and open source version of lex

- Flex (and lex) generate a standalone scanner C file that we add to the project and then can use it for lexical analysis.

flex workflow



Regular Expressions

- Token definitions are given to lex using ***regular expressions***

- Think of a regular expression as a *pattern* that flex can then properly *match* to the correct token

- (There is a much more formal definition of a regular expression…)

Really Basic Regular Expressions

- A string of normal characters is matched directly

```
// Matches endline
\n

// Matches word new
new

// Quotes are optional, but I usually use them
// Matches delete
"delete"
```

[] operator

- [] Allows the regular expression to match any of the characters inside the brackets.

```
// Matches aac, abc, or acc
a[abc]c

// A hyphen means any characters within that range
// Matches aac, abc, acc, adc, ..., azc
a[a-z]c

// You can combine multiple ranges
// Matches the same as above, plus aAc, ..., aZc
a[a-zA-Z]c
```

* operator – The "Kleene Closure"

- The * or (***Kleene Closure***) means that the preceding character can appear 0 or more times.

```
// Matches ac, abc, abbc, abbbc, ...
ab*c

// Can be combined with square brackets
// Matches a followed by zero or more digits
a[0-9]*
```

+ operator

- Like the * operator, except the element **must** appear at least one time (with the * operator, it can be 0).

```
// Matches abc, abbc, abbbc, ..., but not ac
ab+c

// Can be combined with square brackets
// Matches a followed by one or more digits
a[0-9]+
```

| operator

- Used for “or”

- For example:

```
a | b | c
```

- Is equivalent to

```
[abc]
```

Parenthesis

- Parenthesis can be used to enforce precedence

```
// Matches ab, abab, ababab, ...
(ab)+
```

. operator

- Match anything, any character you want, for that slot:

```
// Matches any single character a, b, c, {
.

// Matches a followed by any random character
a.
```

But I don't want that operator!!!

- If you have a character that is part of your string, for instance, you want a + symbol, you have two options:

```
// Use quotes around the literal
"+"

// Use the backslash to escape the symbol
\+
```

Sample Regular Expressions

- What matches an integer token?
- One or more number.

```
[0-9]+
```

- Decimal token?
- One or more number, followed by a period, followed by 0 or more numbers.

```
[0-9]+\.[0-9]*
```

Sample Regular Expressions, Cont'd

- What matches a C++ identifier (name of variable, function, class, etc.)?
- A letter or underscore, followed by zero or more letters, numbers, or underscores:

```
[a-zA-Z_][a-zA-Z0-9_]*
```

In-class Activity

flex input file format

- There are other sections, but the main section (which defines the tokens) will typically look something like this:

```
%%
"new"                      { return TNEW; }
"delete"                   { return TDELETE; }
[a-zA-Z_][a-zA-Z0-9_]*     { return TIDENTIFIER; }
[0-9]+                     { return TINTEGER; }
"+"                        { return TPLUS; }
"-"                        { return TMINUS; }
%%
```

flex token order

- The order is very important.
- Tokens at the top will be matched first.
- Always put generic identifier tokens below any keywords which would also otherwise satisfy the identifier token!!!
- eg. Don’t do this:

```
[a-z]+
"new"
```

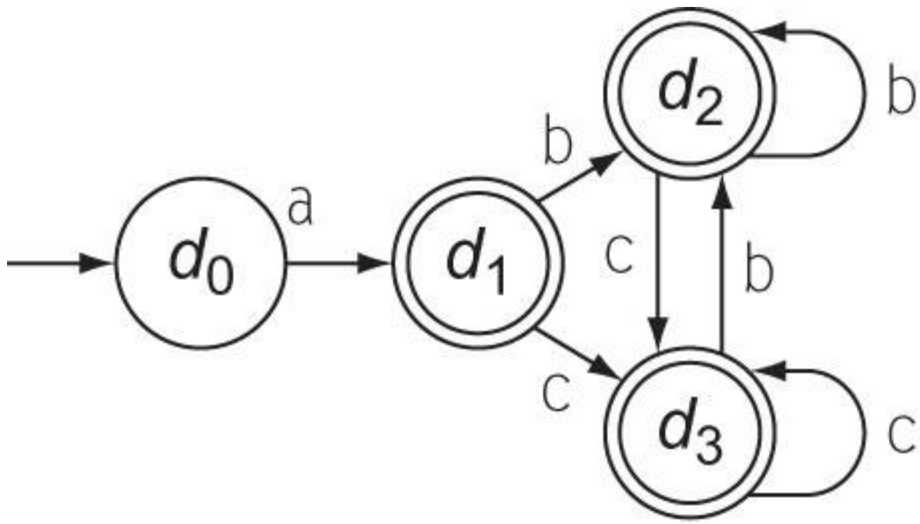
Running the Scanner

- Flex, by default, scans from stdin
- We can change it to scan from a file instead, by setting the FILE* yyin to what we want it to point to
- The function to get the token id of the next token is yylex()
- We typically do not call this directly. Bison (which we will talk about next week) will call it for us!

How does a scanner work?

Scanner Implementation Details

- Most standalone scanners, including lex, implement convert regular expressions into *deterministic finite automata* (**DFA**s)

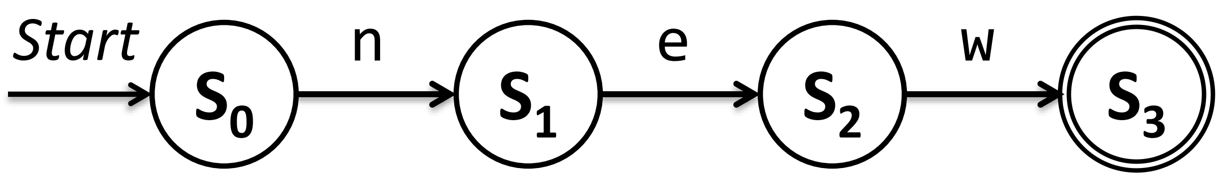


- This is a DFA for the regular expression: `a(b|c)*`

A Simple Example

- Create a scanner that accepts the string new

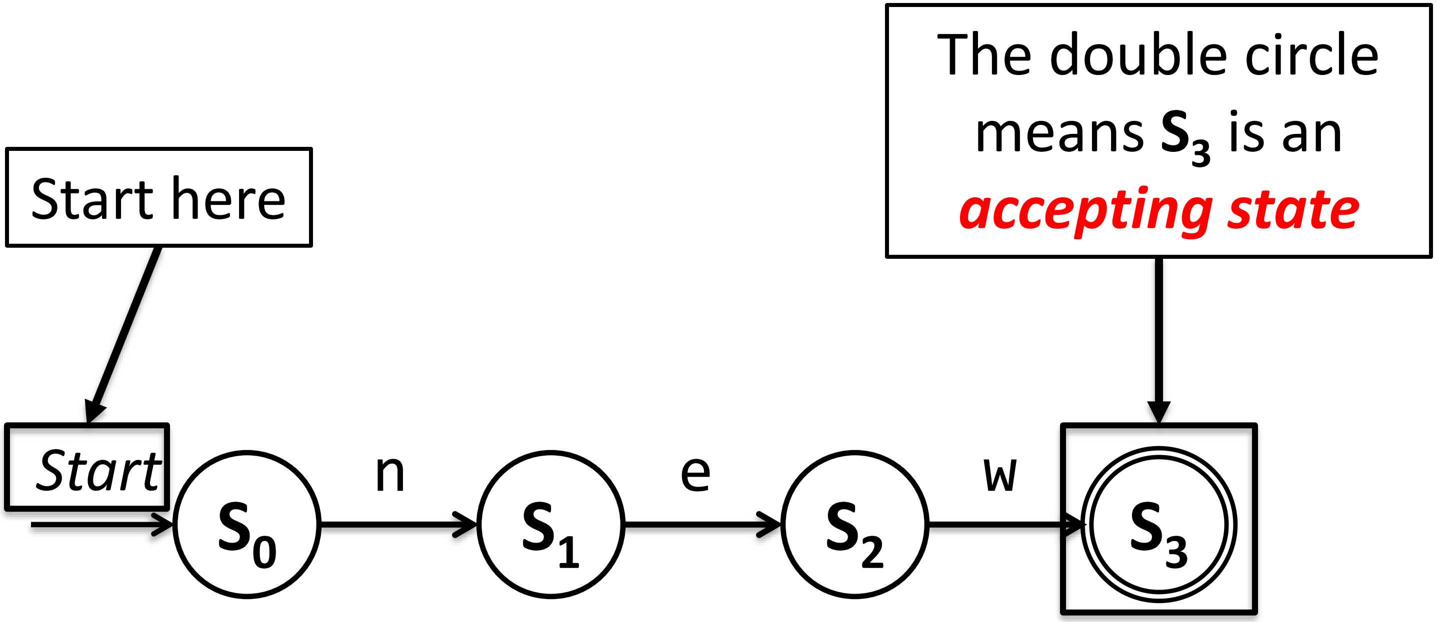
1. Represent this as a finite state machine:



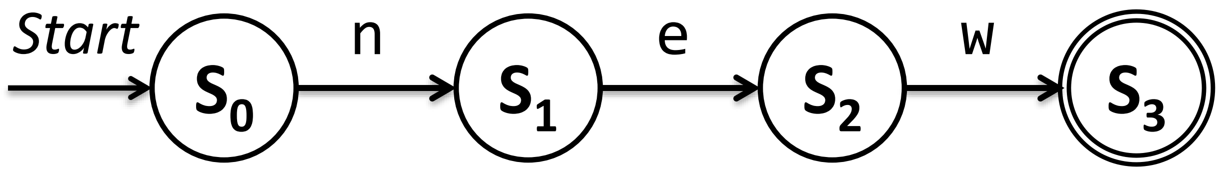
- (Specifically, this type of state machine is called a ***deterministic finite automaton*** or DFA)

A Simple Example

- Create a scanner that accepts the string new



Making a State Transition Table



2. Once you have a DFA, you can convert it to a table representing the states:

	Action on Input…			
State	**n**	**e**	**w**	**(Other)**
$\mathbf{S_0}$	goto $\mathbf{S_1}$	error	error	error
$\mathbf{S_1}$	error	goto $\mathbf{S_2}$	error	error
$\mathbf{S_2}$	error	error	goto $\mathbf{S_3}$	error
$\mathbf{S_3}$*	error	error	error	error

* = Accepting State

Writing Code

	Action on Input…			
State	**n**	**e**	**w**	**(Other)**
S_0	goto S_1	error	error	error
S_1	error	goto S_2	error	error
S_2	error	error	goto S_3	error
S_3*	error	error	error	error

* = Accepting State

3. It's fairly trivial to write code that can read in an arbitrary state transition table and operate on it

Writing Code

3. It's fairly trivial to write code that can read in an arbitrary state transition table and operate on it

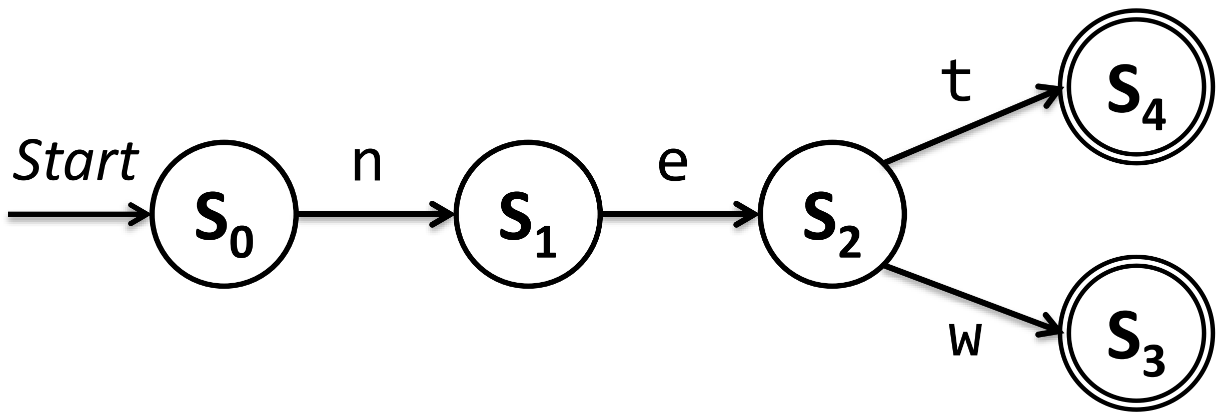
```
// Pseudocode adapted from EaC, p. 32
c = nextChar();
state = 0;

while (c != EOF && state != error) {
   // Transition to the next state
   state = transition(state, c);
   c = nextChar();
}

if (state is accepting) {
   // Do whatever is done when accepted
   // (such as report the token)
   accept(state);
} else {
   reportError();
}
```

Another Example

- For simple cases, it's not too tough to just create the DFA diagram off the top of your head…
- Create a DFA that accepts the string `new` or `net`:



A More Realistic Example

- Let’s do an “identifier” where it:
  - Must begin with a lowercase letter
  - Can be followed by zero or more lowercase letters or numbers

- ***Q:*** *How could this be represented in flex?*

A More Realistic Example

- Let's do an "identifier" where it:
  - Must begin with a lowercase letter
  - Can be followed by zero or more lowercase letters or numbers

- ***A:** A **regular expression**!*

A More Realistic Example

- Let's do an "identifier" where it:
  - Must begin with a lowercase letter
  - Can be followed by zero or more lowercase letters or numbers

- ***A:*** *A* ***regular expression****!*

Specifically, in flex syntax you could write:

```
[a-z][a-z0-9]*
```

- …but there's no one step process to convert a regular expression into a DFA ☹

Regular Expression Cheat Sheet

- Technically, you only really need four operations to express all regular expressions:

Operation	Examples	Note
Concatenation	a b a•b	"a followed by b" (the dot is optional)
Or	[ab] a|b	"a or b" (two different ways)
Kleene Closure	a*	"zero or more instances of a"
Parenthesis	(a|b)•c	"a or b, followed by c" (to enforce precedence)

Regular Expression Cheat Sheet

- But usually, we throw in at least a couple of more options…

Operation	Examples	Note
Concatenation	a b a·b	"a followed by b" (the dot is optional)
Or	[ab] a|b	"a or b" (two different ways)
Kleene Closure	a*	"zero or more instances of a"
Parenthesis	(a|b)·c	"a or b, followed by c" (to enforce precedence)
Positive Closure	a+	"one or more instances of a"
Range	[a-z]	"a single character from a to z"
Wildcard	.	"any single character" (period)

Back to the Identifier Example

- An “identifier” where it:
  - Must begin with a lowercase letter
  - Can be followed by zero or more lowercase letters or numbers

- Rewrite the regular expression slightly to only use the “required four” plus also ranges:

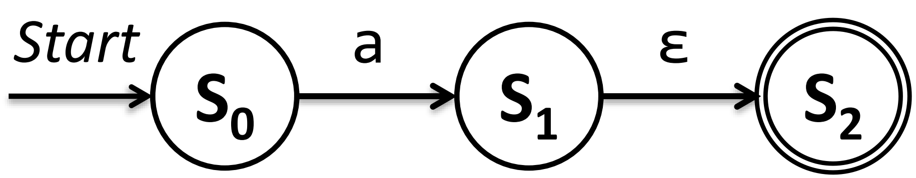
```
[a-z]([a-z] | [0-9])*
```

- We can’t convert this to a DFA in one step, but there is a ***multistep*** process…

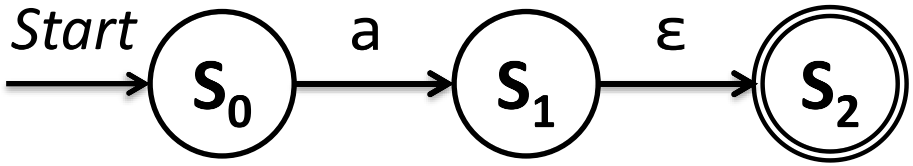
Nondeterministic Finite Automata

- In a ***nondeterministic finite automata (NFA)***, a single input may lead to multiple possible states
- Specifically, an NFA usually has ***ε-transitions*** (“empty” or epsilon transitions)

- An example:



Why is this an NFA?



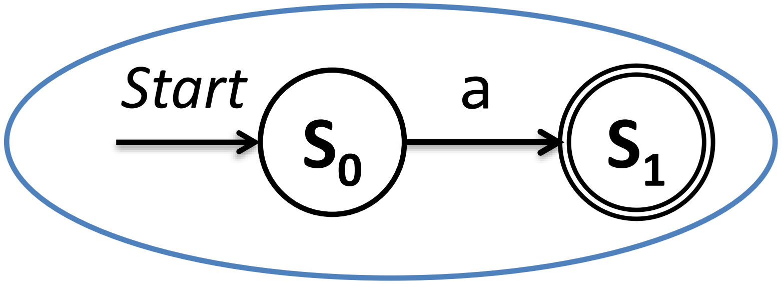
- An ε-transition will or won't be taken, in a nondeterministic manner

- So in the above example, if the state machine is in $\mathbf{S_0}$ and gets an a, there are two possibilities:
  - It goes to $\mathbf{S_1}$ and doesn't take the ε-transition
  - It goes to $\mathbf{S_1}$ and ***does*** take the ε-transition, ending up in $\mathbf{S_2}$

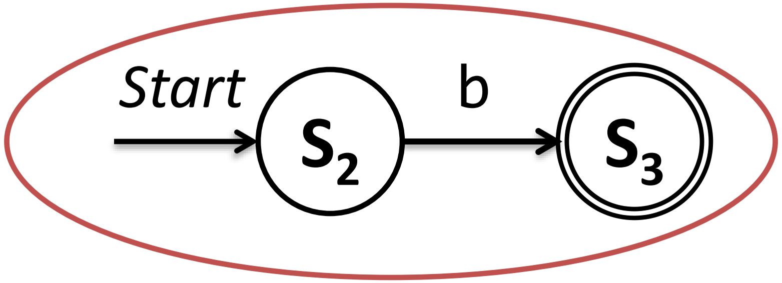
- Thus, ***nondeterministic***!

Thompson's Construction

- ***Thompson's Construction*** is a method to convert a regular expression to an NFA
- Suppose you have two DFA/NFAs:

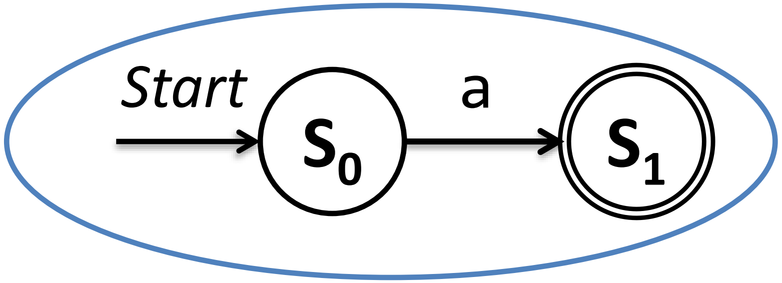


DFA that accepts a

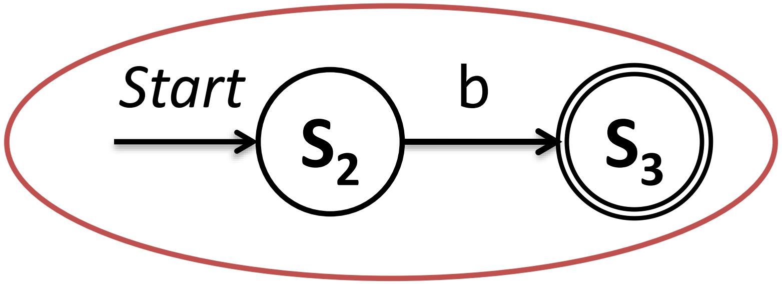


DFA that accepts b

Thompson's Construction – Concatenation

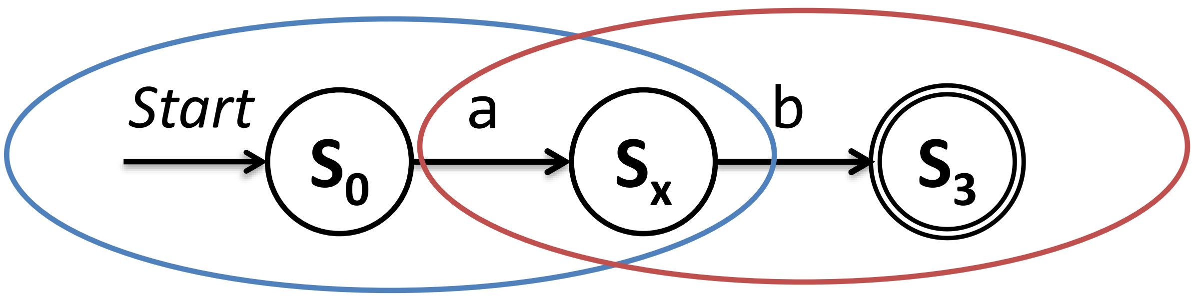


DFA that accepts a



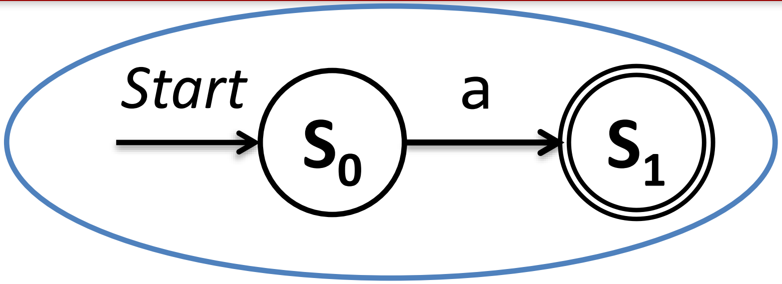
DFA that accepts b

- "a followed by b" merges the accepting state of a with the start state of b…

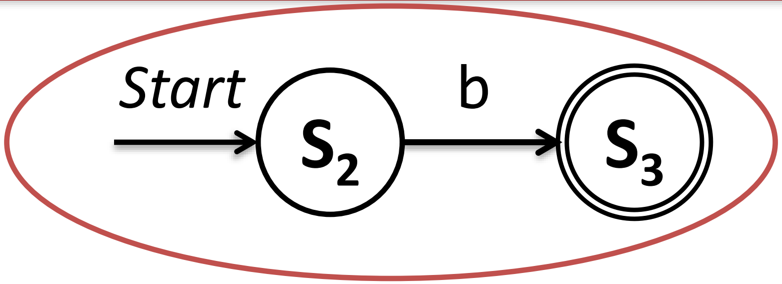


DFA that accepts a • b

Thompson's Construction – Or

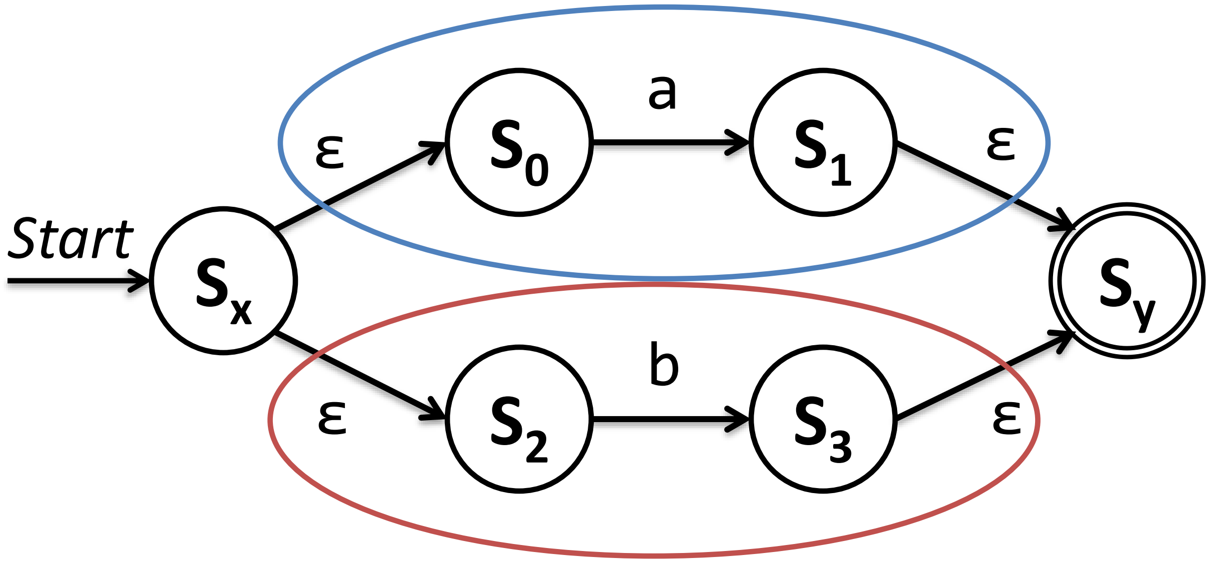


DFA that accepts a



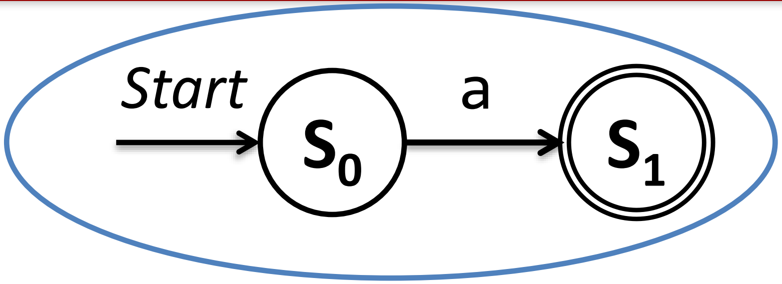
DFA that accepts b

- "a or b":

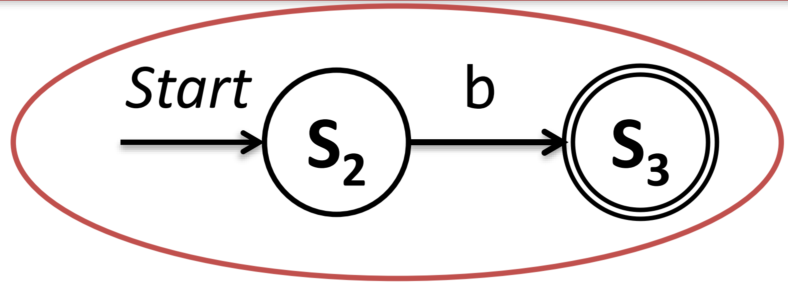


NFA that accepts a | b

Thompson's Construction – Kleene Closure

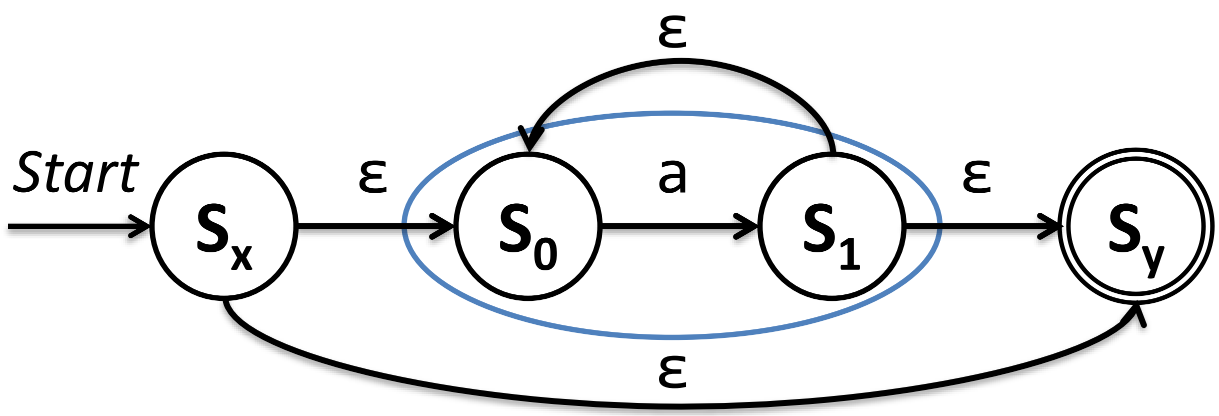


DFA that accepts a



DFA that accepts b

- "zero or more instances of a":



NFA that accepts a*

What about ranges?

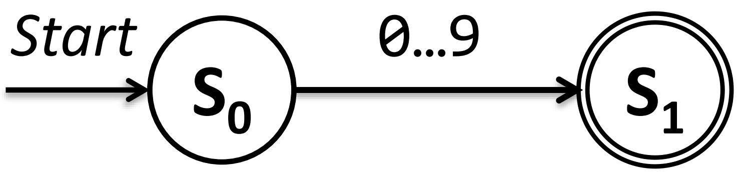
- Technically, if you have:

```
[0-9]
```

- You would have to represent it like this:

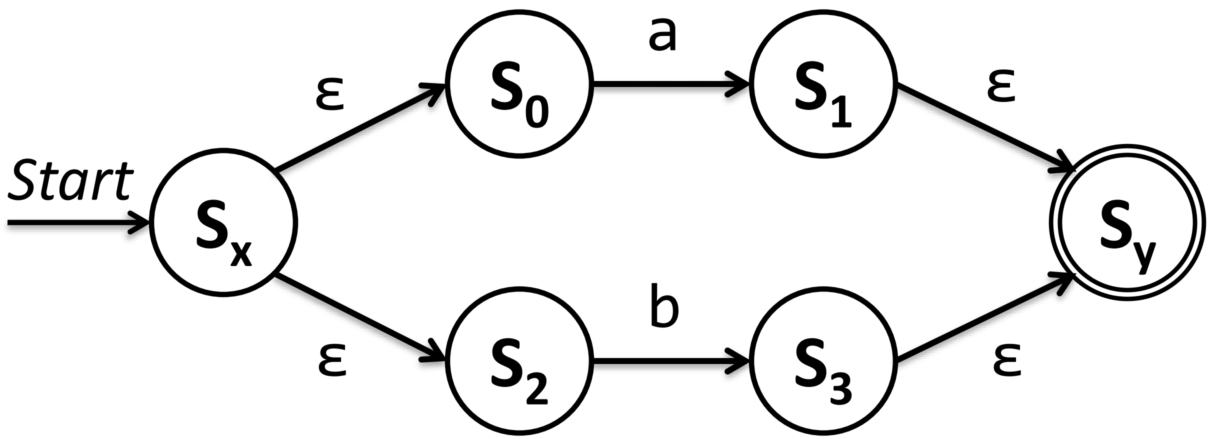
```
(0|1|2|3|4|5|6|7|8|9)
```

- But that's just tedious, so I'd recommend drawing it like this:



More Advanced Constructions

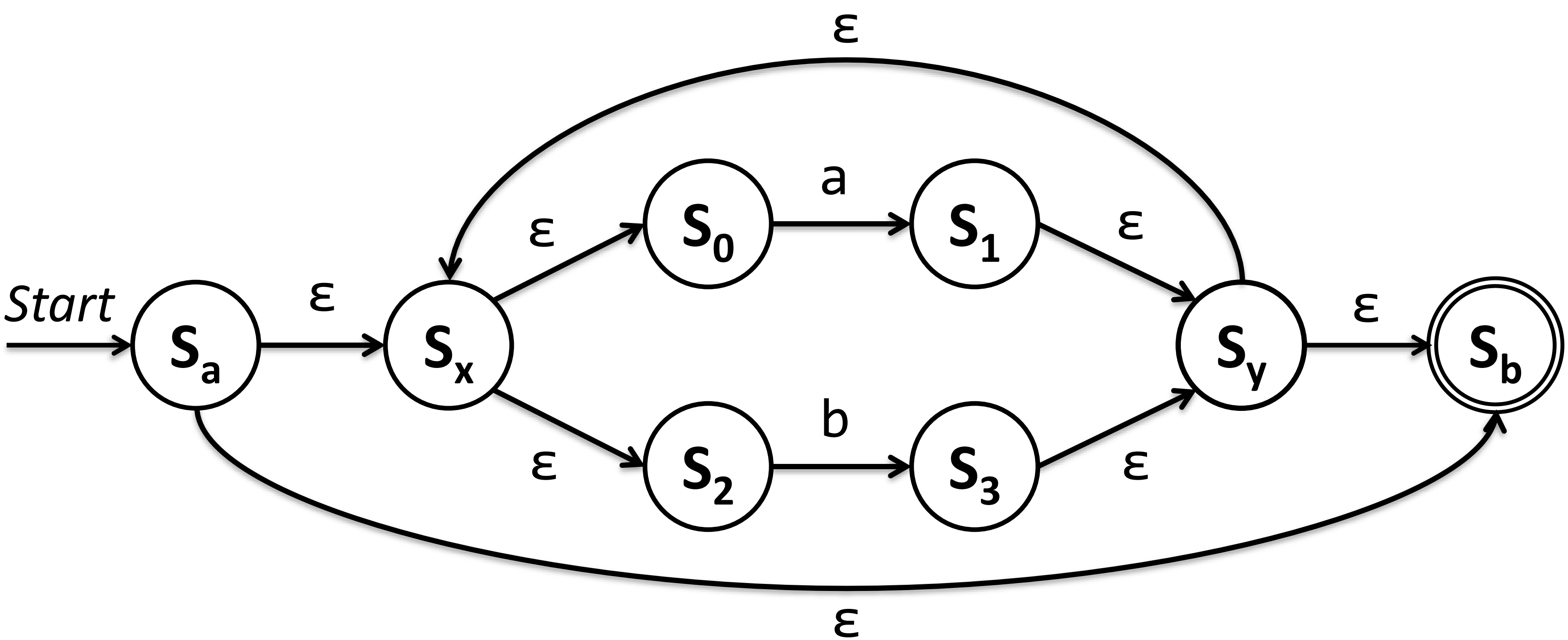
- Usually we'll need to combine multiple levels of construction, for example:
  (a|b)*



NFA that accepts a|b

Add the Kleene Closure…

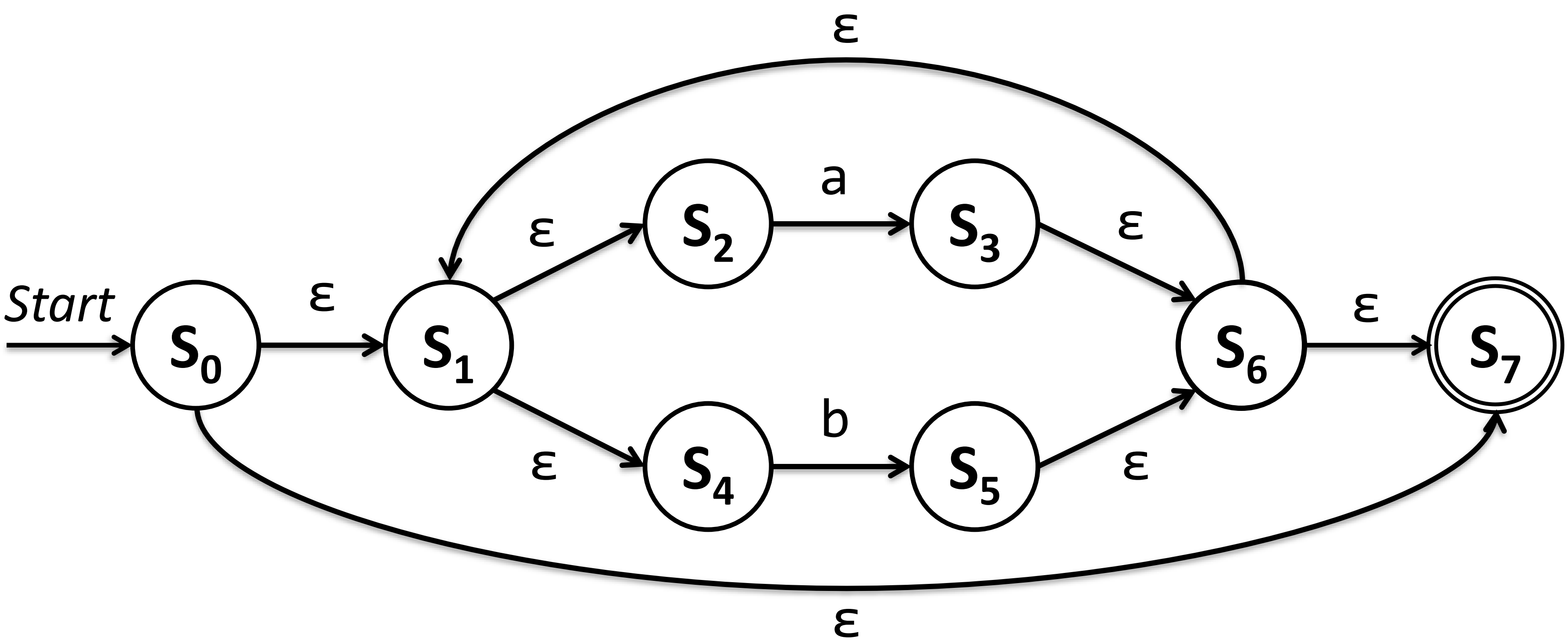
- (a|b)*



NFA that accepts (a|b)*

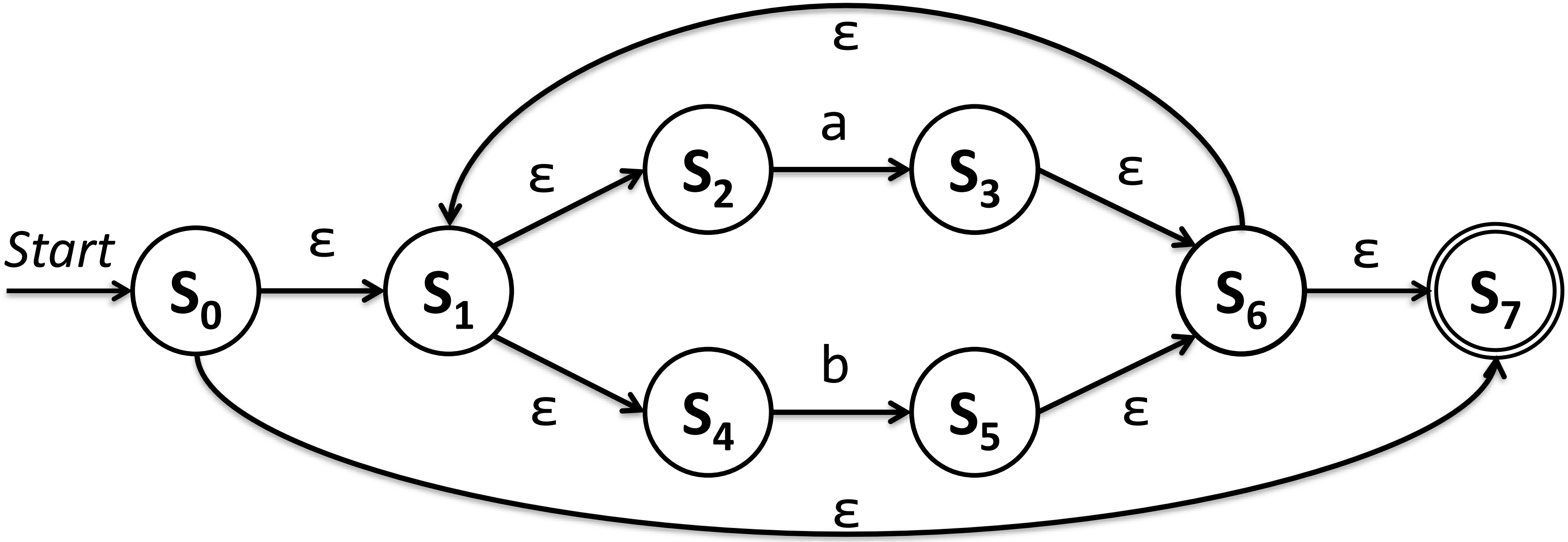
Renumber the states

- (Just for clarity)



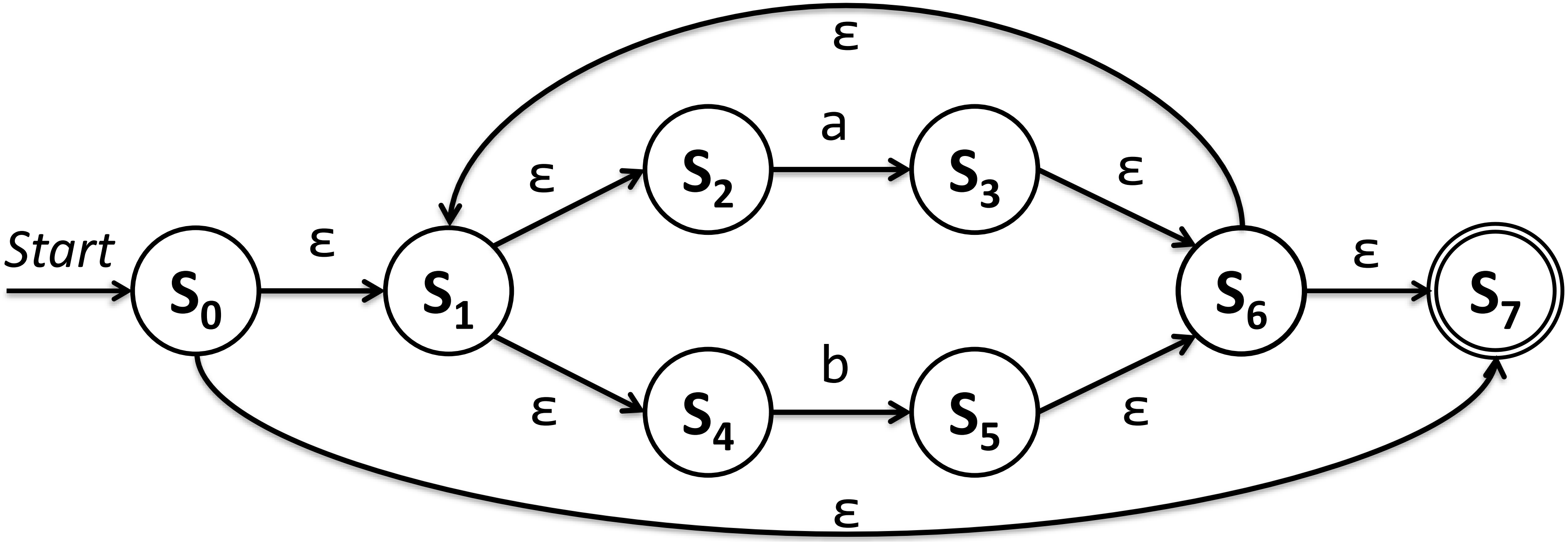
NFA that accepts (a|b)*

Now what?



- Once we have an NFA for (a|b)*, how do we convert this to a DFA???

Subset Construction



- The ***subset construction*** converts an NFA to a DFA

- The basic premise is that we take groupings of NFA states and convert them to a DFA state…luckily I'm not covering this!