Plan for Today and Thursday

Important Logistics

- PA1 peer reviews, due Thursday! Need github repository ID for permissions.
- HW3, due Sunday night. NO LATE period.
- Midterm, Tuesday in class. Examples online. HW3. 1-side 8.5x11" note sheet.

Lexical Analysis

Regular Expressions to NFAs

NFAs to DFAs

Context Free Grammars

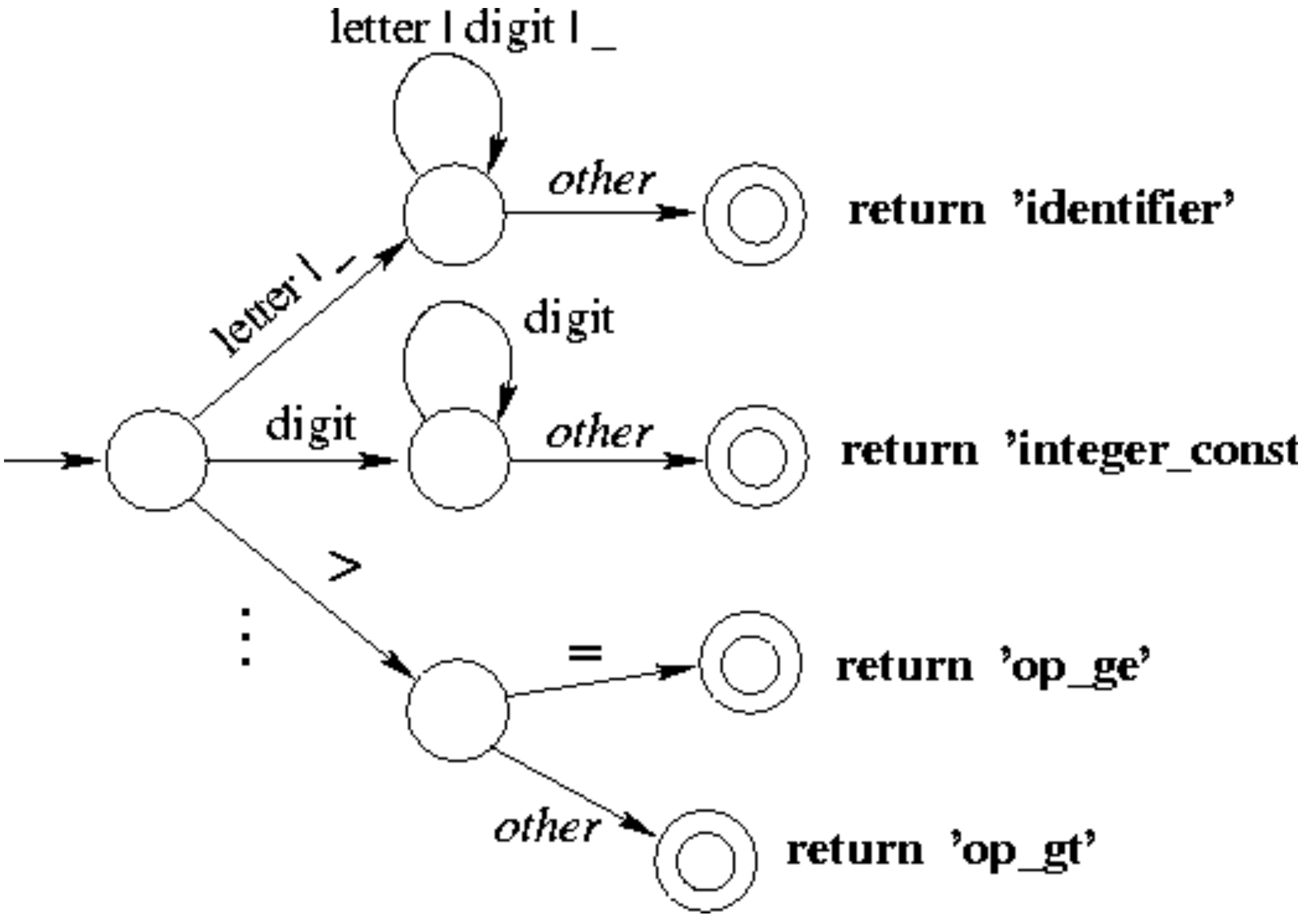
- models for specifying programming languages
- example grammars
- Derivations and parse trees. ← GOAL FOR TODAY

Recursive Descent Parsing / Predictive Parsing

Syntax-directed translation

- Used syntax-directed translation to generate code

Structure of a Scanner Automaton



Implementing finite state machines

Table-driven FSMs (e.g., *lex*, *flex*):

- Use a table to encode transitions:

$$\text{next\_state} = T(\text{curr\_state}, \text{next\_char});$$

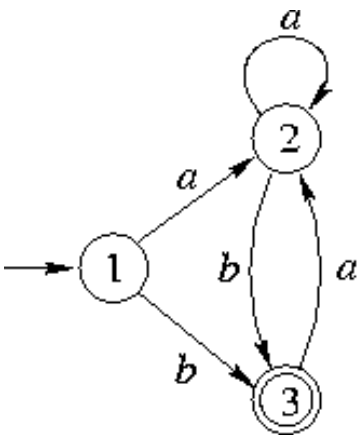
- Use one bit in state no. to indicate whether it's a final (or error) state. If so, consult a separate table for what action to take.

T	*next input character*		
Current state			

Table-driven FSMs: Example

```
int acceptString()
{  char ch;
   int currState = 1;  ch = nextChar();

   while (ch!=EOF) {
     currState= T [currState, ch];
   } /* while */
   if (IsFinal(currState)) {
      return 1; /* success */
   }
}
```

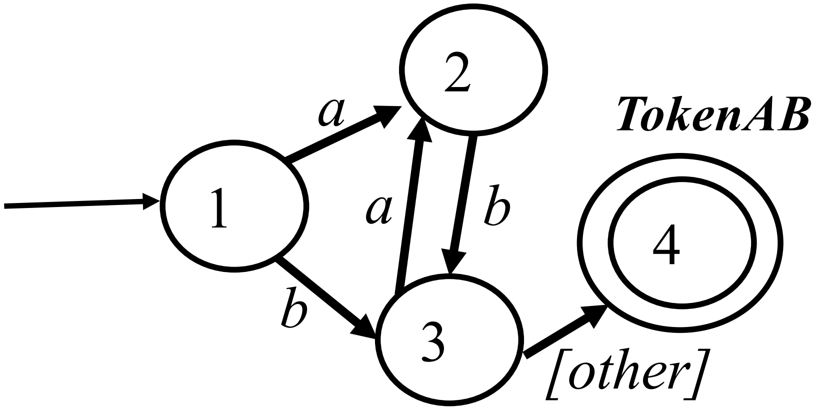


		input	
T		a	b
state	1	2	3
	2	2	3
	$3^{(\text{final})}$	2	

Table-driven FSMs: Determines if full string is in language

```
Token scanner()
{  char ch;
   int currState = 1;  ch = nextChar();

   while (not IsFinal(currState)) {
     nextState = T [currState, ch];
     if (consume(currState,ch)) {
       ch = NextChar( );
     }
     if (ch == EOF) { return 0; } /* fail */
     currState = nextState;
  } /* while */
  if (IsFinal(currState)) {
    return finalToken(currState); /* success */
  }
}
```



T

	input	
state	*a*	*b*
1	2	3
2	2	3
3	2	[other]
4(final)		

Go see http://www.cs.arizona.edu/classes/cs453/fall16/Recit/LexerStart-take2.hs

Table-Driven FSM for Numbers

```
-- Produce tokens until the input string
-- has been completely consumed.
lexer :: String -> [Token]
lexer []    = []
lexer input =
  let (tok,remaining) = driveTable 0 “” input
  in if tok==WhiteSpace then lexer remaining
     else tok : lexer remaining

-- From given state consume characters
-- from the string until token is found.
driveTable :: Int->String->String-
>(Token,String)
driveTable curr [] = (UnexpectedEOF, "")
driveTable curr (c:rest) =
  let (next,consume)   = nextState curr c
      (nextTokStr,remaining)= nextStrings ...
      (done,tok)       = final next nextTokStr
  in if done then (tok,remaining)
     else driveTable next nextTokStrnremaining
```

Draw FSM on board

- State 0
  - Digit goto state 1
- State 1
  - Digit goto state 1
  - Other goto state 2
- State 2 is a final state for TokenNUM

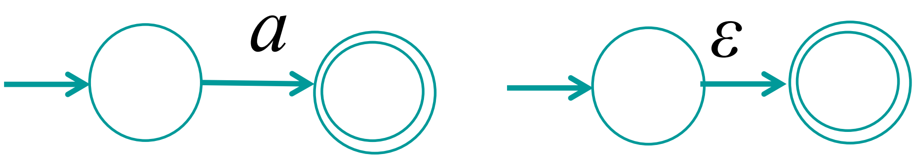
How should we define nextState and final functions?

From regular expressions to NFAs

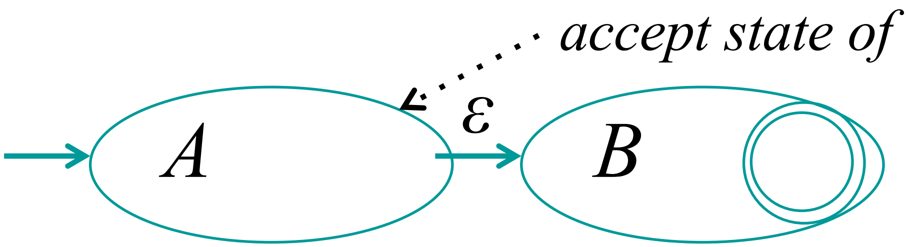
regexp

simple letter “a”

empty string

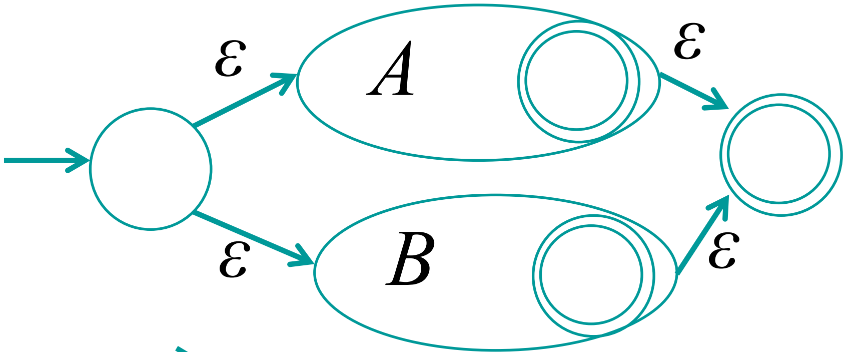


accept state of the NFA for A

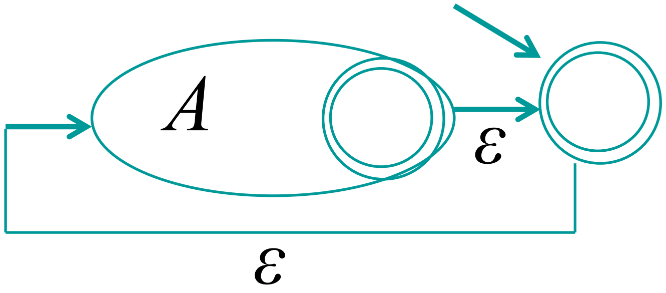


AB concat the NFAs

A|B split merge them



A* build a loop



The Problem

DFAs are easy to execute (table driven interpretation)

NFAs

- are easy to build from reg. exps,
- but hard to execute
- we would need some form of guessing, implemented by back tracking

To build a DFA from an NFA

- we avoid the back track by taking all choices in the NFA at once,
- a move with a character or ε gets us to a set of states in the NFA,
- which will become one state in the DFA.

We keep doing this until we have exhausted all possibilities.

- This mechanism is called transitive closure
- (This ends because there is only a finite set of subsets of NFA states. How many are there?)

Example IF and ID

let : [a-z]
dig : [0-9]

tok : if | id

if : “i” “f”

id : let (let | dig)*

Notes to read through later, Definitions: edge(s,c) and closure

edge(s,c): the set of all NFA states reachable from state s following an edge with character c

closure(S): the set of all states reachable from S with no chars or ε

$$closure(S) = T = S \cup (\bigcup_{s \in T} edge(s,\varepsilon))$$

```
T=S
 repeat T’ =T;
    forall s in T’ {  T’ =T;    $T = T' \cup (\bigcup_{s \in T'} edge(s,\varepsilon))$   }
 until T’ ==T
```

This transitive closure algorithm terminates because there is a finite number of states in the NFA

Notes to read through later, DFAedge and NFA Simulation

Suppose we are in state DFA d = {s_i, s_k,s_l}

By moving with character c from d we reach a set of new NFA states, call these DFAedge(d,c), a new or already existing DFA state

$$DFAedge(d,c) = closure(\bigcup_{s \in d} edge(s,c))$$

NFA simulation:

```
  let the input string be $c_1 \ldots c_k$
  d=closure({s1})  // $s_1$ the start state of the NFA
  for  i from 1 to k
     d = DFAedge(d,$c_i$)
```

Notes to read through later ,
Constructing a DFA with closure and DFAEdge

state d_1 = closure(s_1)   the closure of the start state of the NFA

make new states by moving from existing states with a character c, using DFAEdge(d,c); record these in the transition table

make accepts in the transition table, if there is an accepting state in d, decide priority if more than one accept state.

Instead of characters we use non-overlapping (DFA) character classes to keep the table manageable.

Suggested Exercise

Build an NFA and a DFA for integer and float literals

dot: “.”

dig: [0-9]

int-lit: dig$^+$

float-lit: dig* dot dig+

Regular Expressions: repetition and choice

let : “a” | “b” | “c”

word : let$^+$

What regular expressions cannot express:

nesting, e.g. matching parentheses: ( ) | (( )) | ((( ))) | …
to any depth

Why? A DFA has only a finite # states and thus cannot
encode that it has seen N “(“-s and thus now must
see N “)”-s for the parentheses to match (for any N).

For that we need a recursive definition mechanism:

S : “(“ S “)” | ε

Context Free Grammars

CFG: set of productions of the form

Non-terminal → phrase | phrase | phrase …

phrase: string of terminals and non-terminals

terminals: tokens of the language

non-terminals: represent sets of strings of tokens of the language

Example:

stmt → ifStmt | whileStmt

ifStmt → IF OPEN boolExpr CLOSE Stmt

whileStmt → WHILE OPEN boolExpr CLOSE Stmt

Syntax and Semantics

Regular Expressions define what correct tokens are

Context Free Grammars define what correctly formed programs are

But… are all correctly formed programs meaningful?

Syntax and Semantics

Regular Expressions define what correct tokens are

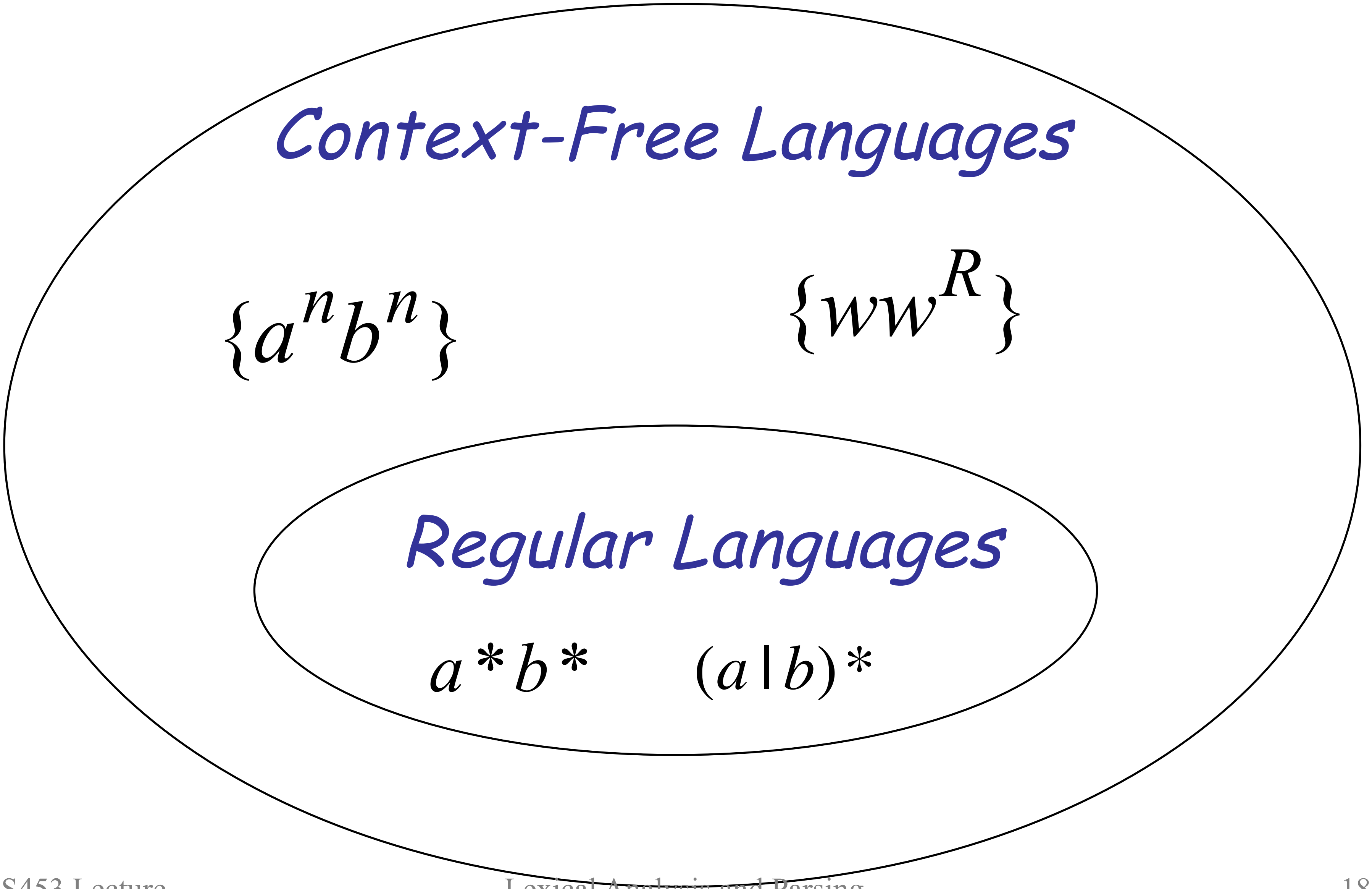
Context Free Grammars define what correctly formed programs are

But… are all correctly formed programs meaningful?

NO: the program can have semantic errors
some can be detected by the compiler: type errors, undefined errors
some cannot: run-time errors,
program does not compute what it is supposed to

The semantics of a program defines its meaning.
Here, we do syntax directed translation / interpretation

Our Next Class of Languages



Context-Free Languages

$\{a^n b^n\}$

$\{ww^R\}$

Regular Languages

a^*b^*

$(a|b)^*$

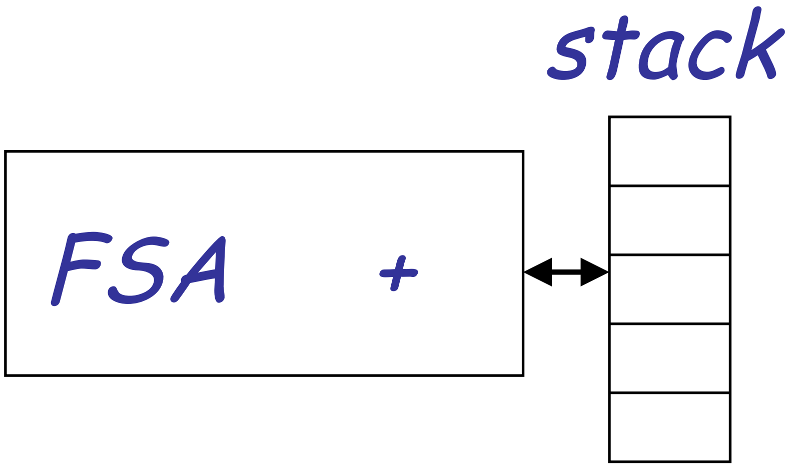
Context-Free Languages

Context-Free Grammars

Recursive definitions

Pushdown Automata

We will start here



Example

A context-free grammar G:

$$S \to aSb$$

$$S \to \varepsilon$$

A derivation:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$

Another derivation:

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaaSbbb \Rightarrow aaabbb$$

An Application of this Language

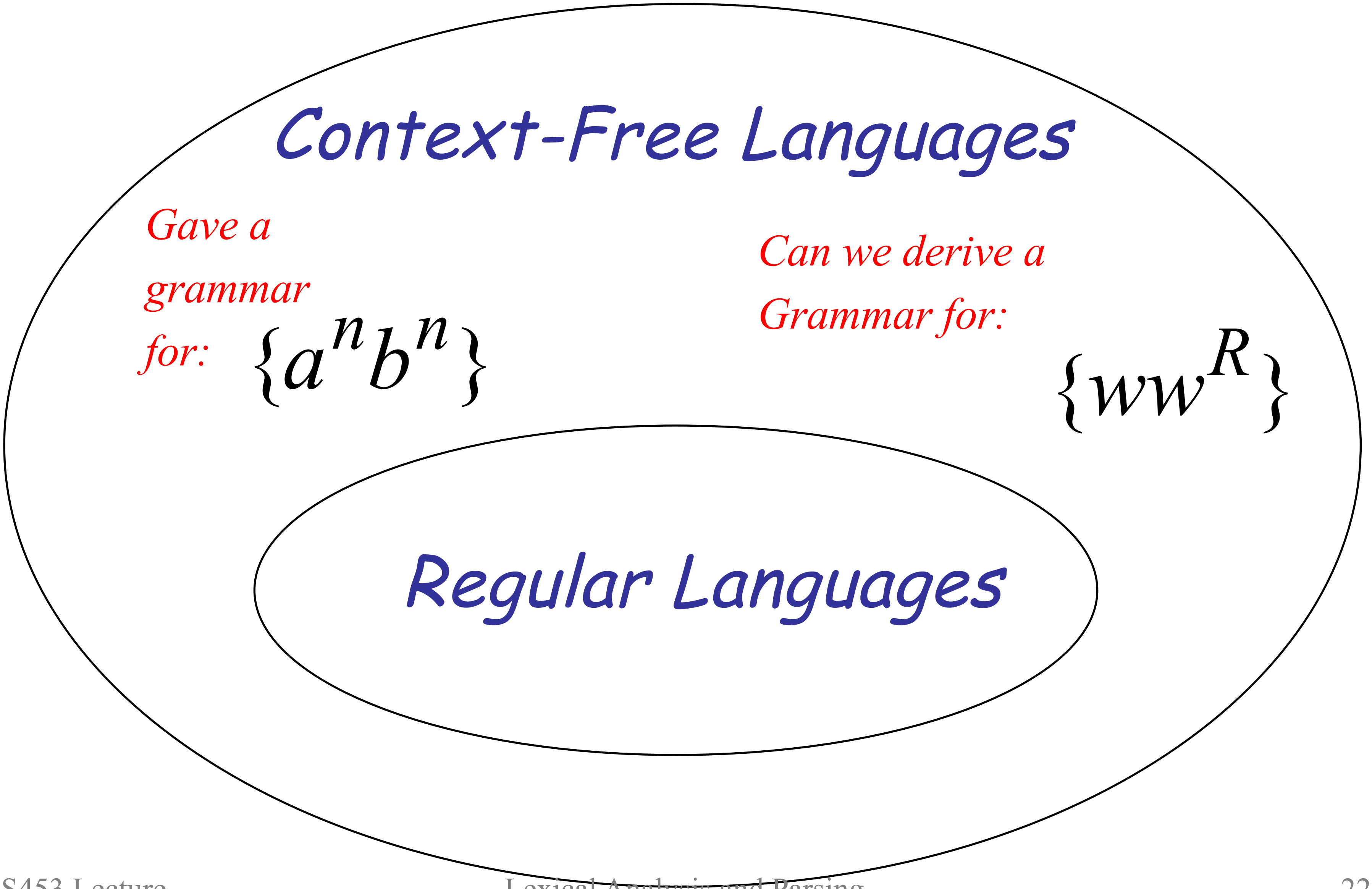
$$S \rightarrow aSb$$

$$S \rightarrow \varepsilon$$

$$L(G) = \{a^n b^n : n \geq 0\}$$

Describes parentheses: (((( ))))

Deriving another grammar



Context-Free Languages

Gave a grammar for: $\{a^n b^n\}$

Can we derive a Grammar for: $\{ww^R\}$

Regular Languages