To use the pumping lemma to prove that a given set $A$ is nonregular, we need to establish that (¬P) holds of $A$. Because of the alternating "for all/there exists" form of (¬P), we can think of this as a game between you and a demon. You want to show that $A$ is nonregular, and the demon wants to show that $A$ is regular. The game proceeds as follows:

1. The demon picks $k$. (If $A$ really is regular, the demon's best strategy here is to pick $k$ to be the number of states of a DFA for $A$.)
2. You pick $x, y, z$ such that $xyz \in A$ and $|y| \geq k$.
3. The demon picks $u, v, w$ such that $y = uvw$ and $v \neq \epsilon$.
4. You pick $i \geq 0$.

You win if $xuv^iwz \notin A$, and the demon wins if $xuv^iwz \in A$.

The property (¬P) for $A$ is equivalent to saying that you have a *winning strategy* in this game. This means that by playing optimally, you can always win no matter what the demon does in steps 1 and 3.

If you can show that you have a winning strategy, you have essentially shown that the condition (¬P) holds for $A$, therefore by Theorem 11.2, $A$ is not regular.

We have thus reduced the problem of showing that a given set is nonregular to the puzzle of finding a winning strategy in the corresponding demon game. Each nonregular set gives a different game. We'll give several examples in Lecture 12.

*Warning*: Although there do exist stronger versions that give necessary and sufficient conditions for regularity (Miscellaneous Exercise 44), the version of the pumping lemma given here gives only a necessary condition; there exist sets satisfying (P) that are nonregular (Miscellaneous Exercise 43). You cannot show that a set *is* regular by showing that it satisfies (P). To show a given set *is* regular, you should construct a finite automaton or regular expression for it.

**(Pumping lemma for CFLs)** *For every CFL $A$, there exists $k \geq 0$ such that every $z \in A$ of length at least $k$ can be broken up into five substrings $z = uvwxy$ such that $vx \neq \epsilon$, $|vwx| \leq k$, and for all $i \geq 0$, $uv^iwx^iy \in A$.*

| *Rule applied* | *Sentential forms in a leftmost derivation of $x$ in $G$* | *Configurations of $M$ in an accepting computation of $M$ on input $x$* |
|---|---|---|
| | $S$ | $(q,$ [ [ [ ] [ ] ], $S)$ |
| (iii) | $[SB$ | $(q,$ [ [ ] [ ] ], $SB)$ |
| (iv) | $[\,[SBSB$ | $(q,$ [ ] [ ] ], $SBSB)$ |
| (ii) | $[\,[\,[BBSB$ | $(q,$ ] ] [ ] ], $BBSB)$ |
| (v) | $[\,[\,[\,]BSB$ | $(q,$ ] [ ] ], $BSB)$ |
| (v) | $[\,[\,[\,]\,]SB$ | $(q,$ [ ] ], $SB)$ |
| (ii) | $[\,[\,[\,]\,]\,[BB$ | $(q,$ ] ], $BB)$ |
| (v) | $[\,[\,[\,]\,]\,[\,]B$ | $(q,$ ], $B)$ |
| (v) | $[\,[\,[\,]\,]\,[\,]\,]$ | $(q,$ $\epsilon$, $\epsilon)$ |

```
type bConst = True | False

type bExpr = Constant of bConst
           | Variable of string
           | And of bExpr * bExpr
           | Or of bExpr * bExpr
           | Not of bExpr

let sample1 = And(Not(Variable "a"),Not(Variable "b"))

let sample2 = Or(Not(Variable "a"),And(Variable "b",Constant(True)))

let sample3 = And(Variable "a", Not(Variable "a"))

(* Awww yeee, using that function from a long time ago to get rid of duplicates *)
let rec vars (bexpr) = getUniqueSet(
  match bexpr with
    | Constant c -> []
    | Variable v -> [v]
    | And (a1, a2) -> append(vars(a1), vars(a2))
    | Or (o1, o2) -> append(vars(o1), vars(o2))
    | Not n -> vars(n);;
  )

let rec subst (bexpr,var,sub) = match bexpr with
  | Constant c -> Constant c
  | Variable v -> if v = var then sub else Variable v
  | And (a1, a2) -> And (subst(a1, var, sub), subst(a2, var, sub))
  | Or (o1, o2) -> Or (subst(o1, var, sub), subst(o2, var, sub))
  | Not n -> Not (subst(n, var, sub))

let evalConst(const) = match const with True -> true | False -> false

let rec evalBooleans (bexpr) = match bexpr with
  | Constant True -> true
  | Constant False -> false
  | And (a1, a2) -> evalBooleans(a1) && evalBooleans(a2)
  | Or (o1, o2) -> evalBooleans(o1) || evalBooleans(o2)
  | Not n -> not(evalBooleans(n))
  | _ -> false

let rec eval (bexpr) =
  if vars(bexpr) <> [] then None
  else if evalBooleans(bexpr) then Some True else Some False
```

```
type 'a dfa = {states : 'a list;
               alphabet : char list;
               start : 'a;
               delta : ('a * char * 'a) list;
               final : 'a list}

let isolatedBs =                (* langu
  {alphabet = ['a'; 'b'];       (* is b
   states = ["start"; "readb"; "sink"];
   start = "start";
   delta = [("start", 'a', "start");
            ("start", 'b', "readb");
            ("readb", 'a', "start");
            ("readb", 'b', "sink");
            ("sink", 'a', "sink");
            ("sink", 'b', "sink")];
   final = ["start";"readb"]}
let rec getTransition(deltas, state, input) = match deltas with
  | [] -> transitionError(input)
  | h::t ->
    let (start,tran,terminal) = h in
      if start = state && tran = input then
        terminal
      else
        getTransition(t, state, input)

let transition (dfa,state,input) =
  getTransition(dfa.delta, state, input);;

let rec extendedTransition (dfa, state, cs) = match cs with
  | [] -> dfa.start
  | [a] -> getTransition(dfa.delta, state, a)
  | h::t ->
    let newState = getTransition(dfa.delta, state, h) in
      extendedTransition(dfa, newState, t)

let accept (dfa, input) =
  setIn(extendedTransition(dfa, dfa.start, explode(input)), dfa.final)
```

```
let rec mapT f t =
  match t with
    EmptyTree -> EmptyTree
  | Node (v,l,r) -> Node (f v, mapT f l, mapT f r)

let rec foldT f t b =
  match t with
    EmptyTree -> b
  | Node (v,l,r) -> f v (foldT f l b) (foldT f r b)

let size t = foldT (fun v l r -> 1 + l + r) t 0
let sum t = foldT (fun v l r -> v + l + r) t 0

let max(a, b) = if a > b then a else b

let rec height t = match t with
  | EmptyTree -> 0
  | Node (v,l,r) -> 1 + max(height l, height r)

let height' t = foldT (fun v l r -> 1 + max(l, r)) t 0

let rec fringe t = match t with
  | EmptyTree -> []
  | Node (v,l,r) -> match l, r with
    | EmptyTree, EmptyTree -> [v]
    | EmptyTree, Node(v2, l2, r2) -> fringe r
    | Node(v1, l1, r1), EmptyTree -> fringe l
    | Node(v1, l1, r1), Node(v2, l2, r2) -> (fringe l) @ (fringe r)
```

```
type 'a nfa = {nfa_states :   'a list;
               nfa_alphabet : char list;
               nfa_start :    'a;
               nfa_delta :    ('a * char * 'a list) list;
               nfa_final :    'a list}
```

```
ec cross (xs, ys) = match xs with
] -> []
::t -> append(elementCross(h, ys), cross(t, ys));;

ec getTransition(deltas, state, input) = match deltas with
] -> transitionError(input)
::t ->
et (start,tran,terminal) = h in
 if start = state && tran = input then
   terminal
 else
   getTransition(t, state, input)

ransition (dfa,state,input) =
Transition(dfa.delta, state, input);;

ec deltasOfAlphabet(dfa1, dfa2, startState, alphabet) = match alphabet with
] -> []
ymbol::t ->
 let (first, second) = startState in
   (startState, symbol, (transition(dfa1, first, symbol), transition(dfa2, second, symbol))) ::
   deltasOfAlphabet(dfa1, dfa2, startState, t)

ec getDeltas(dfa1, dfa2, alphabet, crossed) = match crossed with
] -> []
::t -> append(deltasOfAlphabet(dfa1, dfa2, h, alphabet) , getDeltas(dfa1, dfa2, alphabet, t))

ec getUniqueSet(set) = match set with
] -> []
::t ->
 if setIn(h, t) then getUniqueSet(t)
 else h :: getUniqueSet(t);;

nion (dfa1, dfa2) =
lphabet = append(dfa1.alphabet, dfa2.alphabet);
tates = cross(dfa1.states, dfa2.states);
art = (dfa1.start, dfa2.start);
lta = getUniqueSet(getDeltas(dfa1, dfa2, append(dfa1.alphabet, dfa2.alphabet),
ross(dfa1.states, dfa2.states)));
nal = append(cross(dfa1.final, dfa2.states), cross(dfa2.final, dfa1.states))
```

```
let predicate_opt p = fun x -> if p x then Some(x) else None

let map_opt f = fun x -> match x with Some x -> Some(f x) | _ -> None

let comb_opt f = fun x y -> match x, y with Some x, Some y -> Some(f x y) | _ -> None

let default v = fun x -> match x with Some x -> x | _ -> v

let compose_opt f g = fun x ->
  match f x with
    Some a -> match g a with Some b -> Some b | _ -> None
    | _ -> None

let at_least n p xs = (List.fold_right (fun x y -> if p x then 1 + y else y) xs 0) > n

(* Assuming positive integers *)
let max_list xs = match List.fold_right (fun x y -> if x > y then x else y) xs 0 with
  | 0 -> None
  | a -> Some(a)

let map_funs fs x = List.fold_right (fun a b -> (a (x)) :: b) (fs) []

let map_cross fs xs = List.fold_right (fun a b -> (map_funs fs a) @ b) xs []
```

```
let rec fold_right comb xs base =
  match xs with
    [] -> base
  | x :: xs' -> comb x (fold_right comb xs' base)
```

```
ec nfa_hasFinal (nfa,states) = match states with
] -> false
::t -> if(setIn(h, nfa.nfa_final)) then true else nfa_hasFinal(nfa, t);;

ec getTransition(deltas, state, input) = match deltas with
] -> []
::t ->
et (start,tran,terminals) = h in
 if start = state && tran = input then
   terminals
 else
   getTransition(t, state, input)

ingleTransition (nfa,state,input) =
Transition(nfa.nfa_delta, state, input);;

ec nfa_transition (nfa,states,input) = match states with
] -> []
State::t -> append(singleTransition(nfa, aState, input), nfa_transition(nfa, t, input))

ec nfa_extendedTransition (nfa, states, cs) = match cs with
] -> [nfa.nfa_start]
a] -> nfa_transition(nfa, states, a)
::t ->
et newStates = nfa_transition(nfa, states, h) in
 nfa_extendedTransition(nfa, newStates, t)

fa_accept (nfa, input) =
a_hasFinal(nfa, nfa_extendedTransition(nfa, [nfa.nfa_start], explode(input)))
```

$x$ matches $\alpha + \beta$ if $x$ matches either $\alpha$ or $\beta$:

$$L(\alpha + \beta) = L(\alpha) \cup L(\beta);$$

$x$ matches $\alpha \cap \beta$ if $x$ matches both $\alpha$ and $\beta$:

$$L(\alpha \cap \beta) = L(\alpha) \cap L(\beta);$$

$x$ matches $\alpha\beta$ if $x$ can be broken down as $x = yz$ such that $y$ matches $\alpha$ and $z$ matches $\beta$:

$$\begin{aligned} L(\alpha\beta) &= L(\alpha)L(\beta) \\ &= \{yz \mid y \in L(\alpha) \text{ and } z \in L(\beta)\}; \end{aligned}$$

$x$ matches $\sim\alpha$ if $x$ does not match $\alpha$:

$$\begin{aligned} L(\sim\alpha) &= \sim L(\alpha) \\ &= \Sigma^* - L(\alpha); \end{aligned}$$

$x$ matches $\alpha^*$ if $x$ can be expressed as a concatenation of zero or more strings, all of which match $\alpha$:

$$\begin{aligned} L(\alpha^*) &= \{x_1 x_2 \cdots x_n \mid n \geq 0 \text{ and } x_i \in L(\alpha),\ 1 \leq i \leq n\} \\ &= L(\alpha)^0 \cup L(\alpha)^1 \cup L(\alpha)^2 \cup \cdots \\ &= L(\alpha)^*. \end{aligned}$$

The null string $\epsilon$ always matches $\alpha^*$, since $\epsilon$ is a concatenation of zero strings, all of which (vacuously) match $\alpha$.

$x$ matches $\alpha^+$ if $x$ can be expressed as a concatenation of one or more strings, all of which match $\alpha$:

$$\begin{aligned} L(\alpha^+) &= \{x_1 x_2 \cdots x_n \mid n \geq 1 \text{ and } x_i \in L(\alpha),\ 1 \leq i \leq n\} \\ &= L(\alpha)^1 \cup L(\alpha)^2 \cup L(\alpha)^3 \cup \cdots \\ &= L(\alpha)^+. \end{aligned}$$

rings containing at least three occurrences of $a$:

$$@a@a@a@;$$

rings containing an $a$ followed later by a $b$; that is, strings of the form $xybz$ for some $x, y, z$:

$$@a@b@;$$

l single letters except $a$:

$$\#\cap\sim a;$$

ngs with no occurrence of the letter $a$:

$$(\#\cap\sim a)^*;$$

ngs in which every occurrence of $a$ is followed sometime later by occurrence of $b$; in other words, strings in which there are either occurrences of $a$, or there is an occurrence of $b$ followed by no urrence of $a$; for example, $aab$ matches but $bba$ doesn't:

$$(\#\cap\sim a)^* + @b(\#\cap\sim a)^*.$$

he alphabet is $\{a, b\}$, then this takes a much simpler form:

$$\epsilon + @b.$$ □

**(Pumping lemma)** *Let $A$ be a regular set. Then the following prope holds of $A$:*

(P) *There exists $k \geq 0$ such that for any strings $x, y, z$ with $xyz \in A$ a $|y| \geq k$, there exist strings $u, v, w$ such that $y = uvw$, $v \neq \epsilon$, and all $i \geq 0$, the string $xuv^iwz \in A$.*

Informally, if $A$ is regular, then for any string in $A$ and any sufficiently lo substring $y$ of that string, $y$ has a nonnull substring $v$ of which you c pump in as many copies as you like and the resulting string is still in $A$

We have essentially already proved this theorem. Think of $k$ as the numb of states of a DFA accepting $A$. Since $y$ is at least as long as the numb of states, there must be a repeated state while scanning $y$. The string $v$ the substring between the two occurrences of that state. We can pump as many copies of $v$ as we want (or delete $v$—this would be the case $i =$ and the resulting string is still accepted.

## Games with the Demon

The pumping lemma is often used to show that certain sets are nonregul For this purpose we usually use it in its contrapositive form:

**(Pumping lemma, contrapositive form)** *Let $A$ be a set of strin Suppose that the following property holds of $A$.*

(¬P) *For all $k \geq 0$ there exist strings $x, y, z$ such that $xyz \in A$, $|y| \geq k$, a for all $u, v, w$ with $y = uvw$ and $v \neq \epsilon$, there exists an $i \geq 0$ such t $xuv^iwz \notin A$.*

*Then $A$ is not regular.*

<stmt> ::= <if-stmt> | <while-stmt> | <begin-stmt> | <assg-stmt>

<if-stmt> ::= **if** <bool-expr> **then** <stmt> **else** <stmt>

<while-stmt> ::= **while** <bool-expr> **do** <stmt>

<begin-stmt> ::= **begin** <stmt-list> **end**

<stmt-list> ::= <stmt> | <stmt> ; <stmt-list>

<assg-stmt> ::= <var> := <arith-expr>

<bool-expr> ::= <arith-expr><compare-op><arith-expr>

<compare-op> ::= < | > | ≤ | ≥ | = | ≠

<arith-expr> ::= <var> | <const> | (<arith-expr><arith-op><arith-expr>)

<arith-op> ::= + | − | ∗ | /

<const> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<var> ::= a | b | c | ... | x | y | z