Building a Scanner and Properties of RLs

CSCI 3136: Principles of Programming Languages

Agenda

- Announcements
  - ~~Assignment 1 is out and due May 24~~
  - Assignment 2 is out and due May 31
  - Professor Zeh will give the May 30 lecture
- Readings:
  - Today: 2.2.1
  - Next: 2.2.1
  - Note: I recommend using alternative texts for this part of the course:
  - E..g, Hopcorft et al, “Introduction to Automata Theory”
- Lecture Contents
  - Minimization
  - Scanner Implementations
  - Properties of Regular Languages

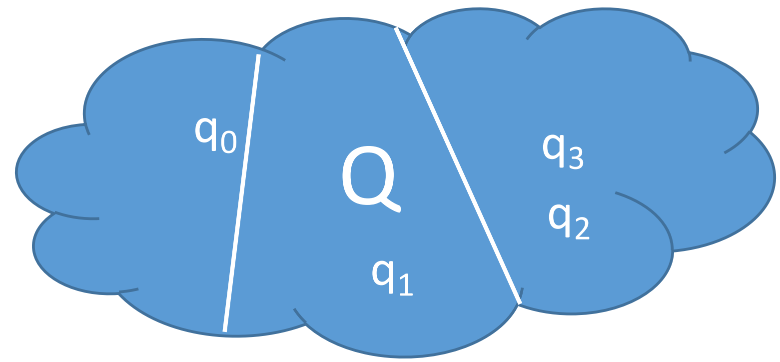
Learning Centre Office Hours

- Lauchlan and Kaari are the TAs for CSCI 3136
- Holds office hours in the Learning Centre on
  - Kaari: Thursday 1 – 3pm
  - Lauchlan: Monday 12 – 2pm
  - Lauchlan: Friday 11 - 12
- Email Lauchlan to meet outside of above hours: lauchlan@dal.ca
- The Learning Center is in the Goldberg CS Building (CS 233)

Minimization of Automata

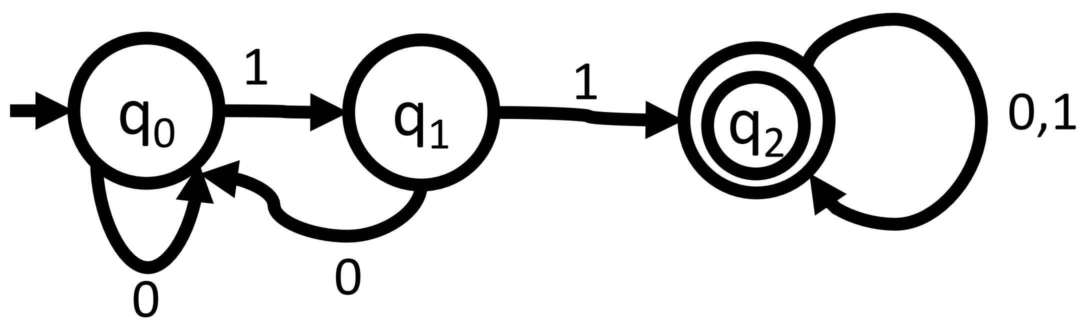
- **Motivation**: To build a scanner, we need to build a DFA
- The simpler a DFA is, the more efficient it is.
- So, we want to build the smallest DFA possible
- **Process**:
  - Build a DFA to recognize L
    - Specify L with a regular expression
    - Create an NFA that recognizes L
    - Convert NFA to DFA
  - Minimize it.
- A DFA is *minimal* if it has the minimum number of states necessary to recognize L

Equivalence Classes



- Start with a DFA $M = (Q,,\Sigma,\delta,q_0,F)$
- **Idea**: Divide Q into equivalence classes
- The classes represent the states of the minimal DFA
- **Definition**: q_1 and q_2 are *equivalent* (in the same class) means for all $\sigma \in \Sigma*$, $\delta(q_1,\sigma) \in F$ if and only if $\delta(q_2,\sigma) \in F$
- I.e., If there exists a string σ such that
  - $\delta(q_1,\sigma) \in F$
  - $\delta(q_2,\sigma) \notin F$

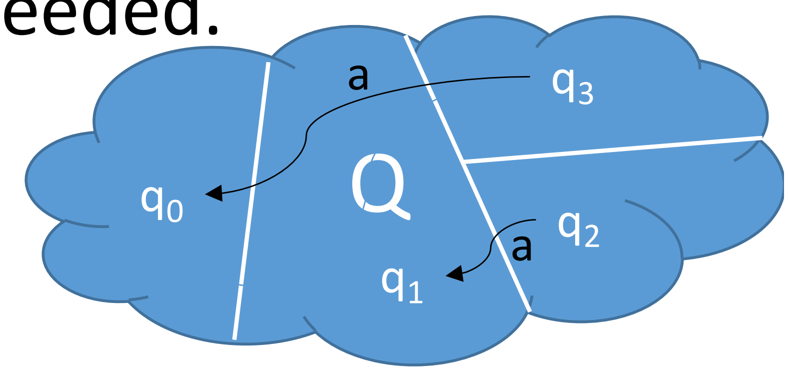
  then the two states are not in the same class.
- Example: q_0 and q_1 are in different classes



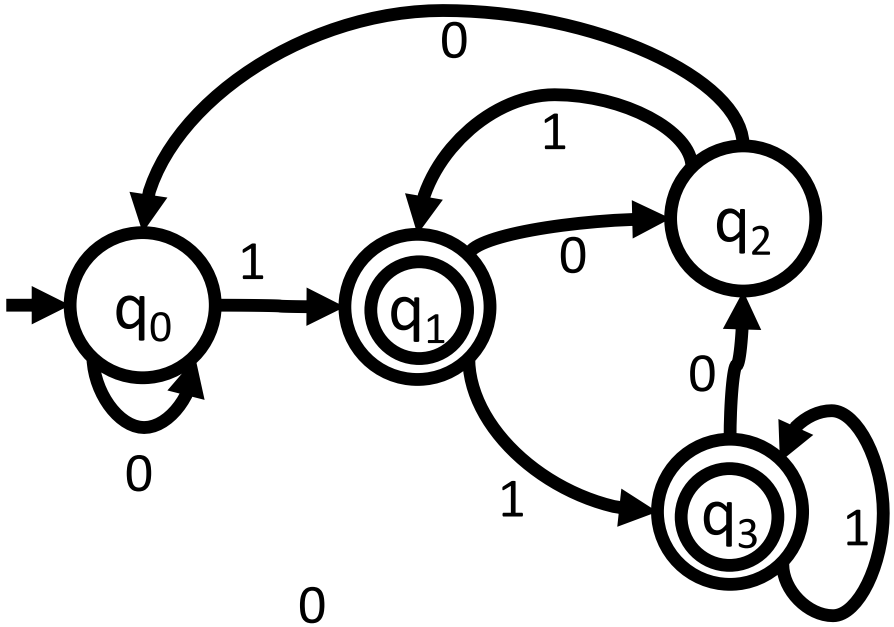
Minimization Procedure

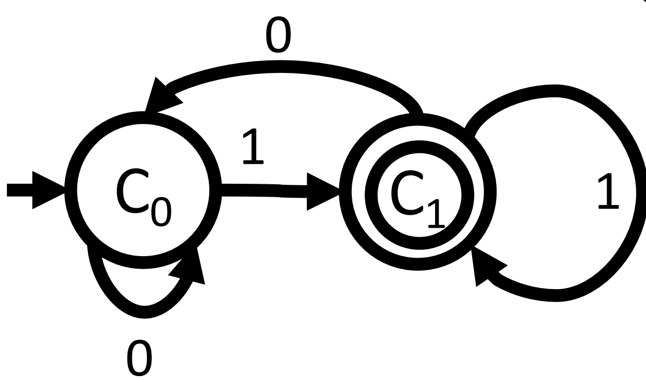
- Initially all states are either accepting or not
- **If** there is a class C and character $a \in \Sigma$ such that
  $\{\delta(q_i,a) | q_i \in C\}$ are in $k > 1$ equivalence classes
  - If there exists $q, r \in C$ where $q' = \delta(q,a)$ and $r' = \delta(r,a)$
  - Such that q' and r' are not in the same equivalence class

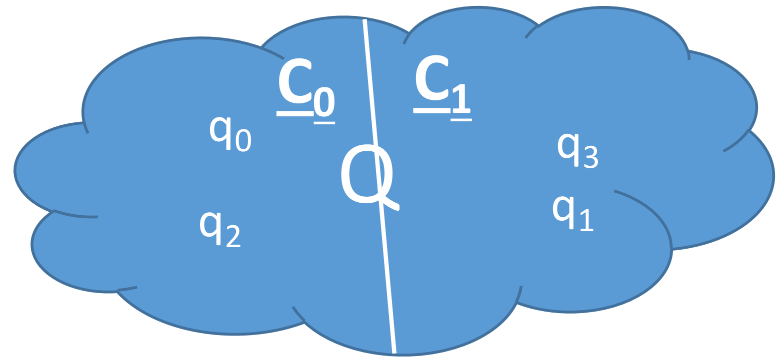
  Then Split C into k classes C_j such that
  $\delta(q_i, a)$, where $q_i \in C_k$, are in the same equivalence class.
- Repeat until no more splits are needed.



Example 1

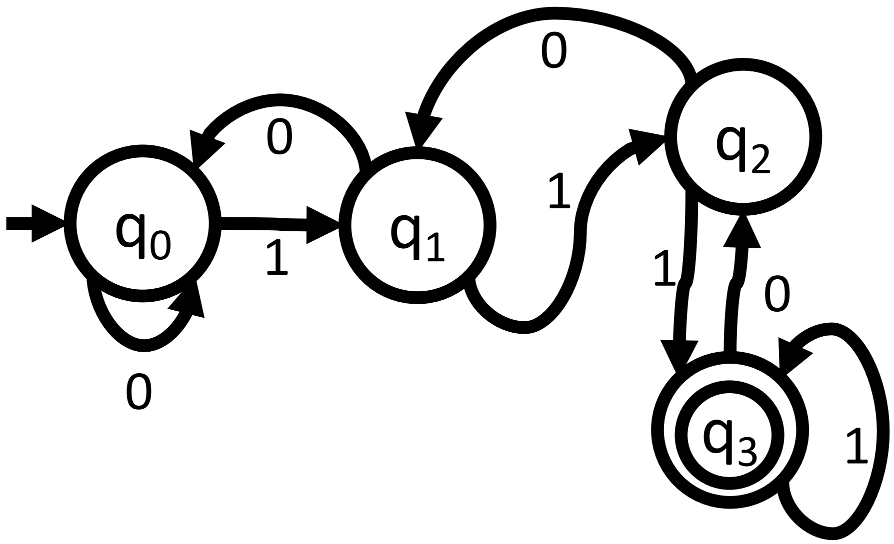


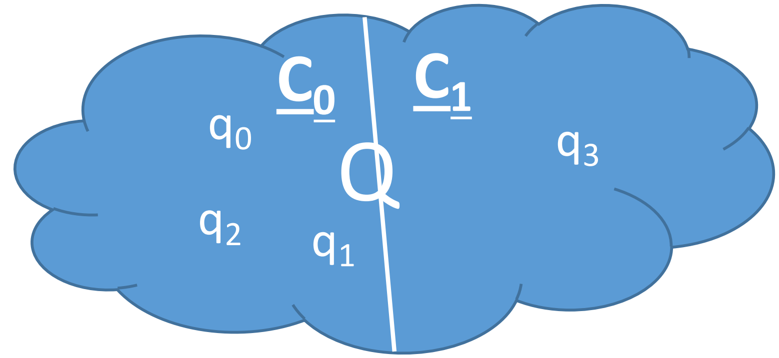




	Q	**0**	**1**
C_0	q_0	C_0	C_1
	q_2	C_0	C_1
C_1	q_1	C_0	C_1
	q_3	C_0	C_1

Example 2





	Q	0	1	0	1	0	1	
C_0	q_0	C_0	C_0	C_0	C_0			
	q_1	C_0	C_0	C_0	C_2			C_3
	q_2	C_0	C_1					C_2
C_1	q_3							

Implementing a Scanner

- Scanners produce a token stream from a character stream
  - Token = (token type, value)
- Scanners operate in one of two modes:
  - **Complete pass mode**: produces the entire token stream at once
  - **Iterative mode:** produces the next token when requested by parser
- Note: Scanners typically produce the longest possible valid tokens

  Example: abc42 can either be tokenized as
  - (identifier, “abc”) (int, 42)
  - (identifier, “abc42”)

  The latter is what will be produced.

Implementation Options

- **Case-based (ad-hoc)**
  - Implemented by hand
  - Transitions are implemented using switch or if statements
- **Table-based (ad-hoc)**
  - Implemented by hand
  - Transitions are implemented using table lookup
- **Table-based (generated)**
  - Generated from a series of REs (e.g., lex, flex)
  - Transitions are implemented using table lookup

Case-based Scanners

- Use a state variable to keep track of what you have seen so far.
- Use a nested switch statement to decide on transition
  - Outer switch switches on state
  - Inner switches switch on character
- Each inner case represents a transition
- If no transition is possible
  - Current token is done
  - Reset state to NEW_TOKEN
  - Return completed token
  - Reset current token to empty
- Repeat until all input is read
- Note: The code is long and tedious but simple

```
while input available:
  c = next_char()
  switch(state):
  case NEW_TOKEN:
    switch(c) # switch on character
    case 'C':
      state = STATE_2
      token.add(c)
      break
      …
  case STATE_1:
    switch(c) # switch on character
      …
  case STATE_2:
    switch(c) # switch on character
      …
  …
```

Case-based Scanners Example

- Use a state variable to keep track of what you have seen so far.
- Use a nested switch statement to decide on transition
  - Outer switch switches on state
  - Inner switches switch on character
- If the next character is part of the next token,
  - Return current token if using iterative mode
  - Save current token and reset state if using complete pass mode
- Keep doing this as long you have input

```
state = NEW_TOKEN
while input available:
  c = next_char()
  switch(state):
  case NEW_TOKEN:
    switch(c):
    case [ \n\t\r]: # white space
       break
    case [0-9]:
      state = NUMBER
      token.add(c)
      break
    case [a-z]:
      state = WORD
      token.add(c)
      break
…
```

Table-based Scanners

State	Typ	0	1	…	a	b	c	…	z	=	…
0	New	1	1	1	3	3	3	3	3		
1	Int	1	1	1	X	X	X	X	X		2
2	Dbl				X	X	X	X	X		
3	ID	X	X	X							
4	KW	X	X	X							
…	…										

```
next_token():
  state = 0
  init(token)
  while input available:
    c = next_char()
    ns = Tab[state][c]
    if ns == X:
       undo(c)
       break

    state = ns
    token.add( c )

  typ = Table[state][TYP]
  return (token, typ)
```

Table based (auto generated)

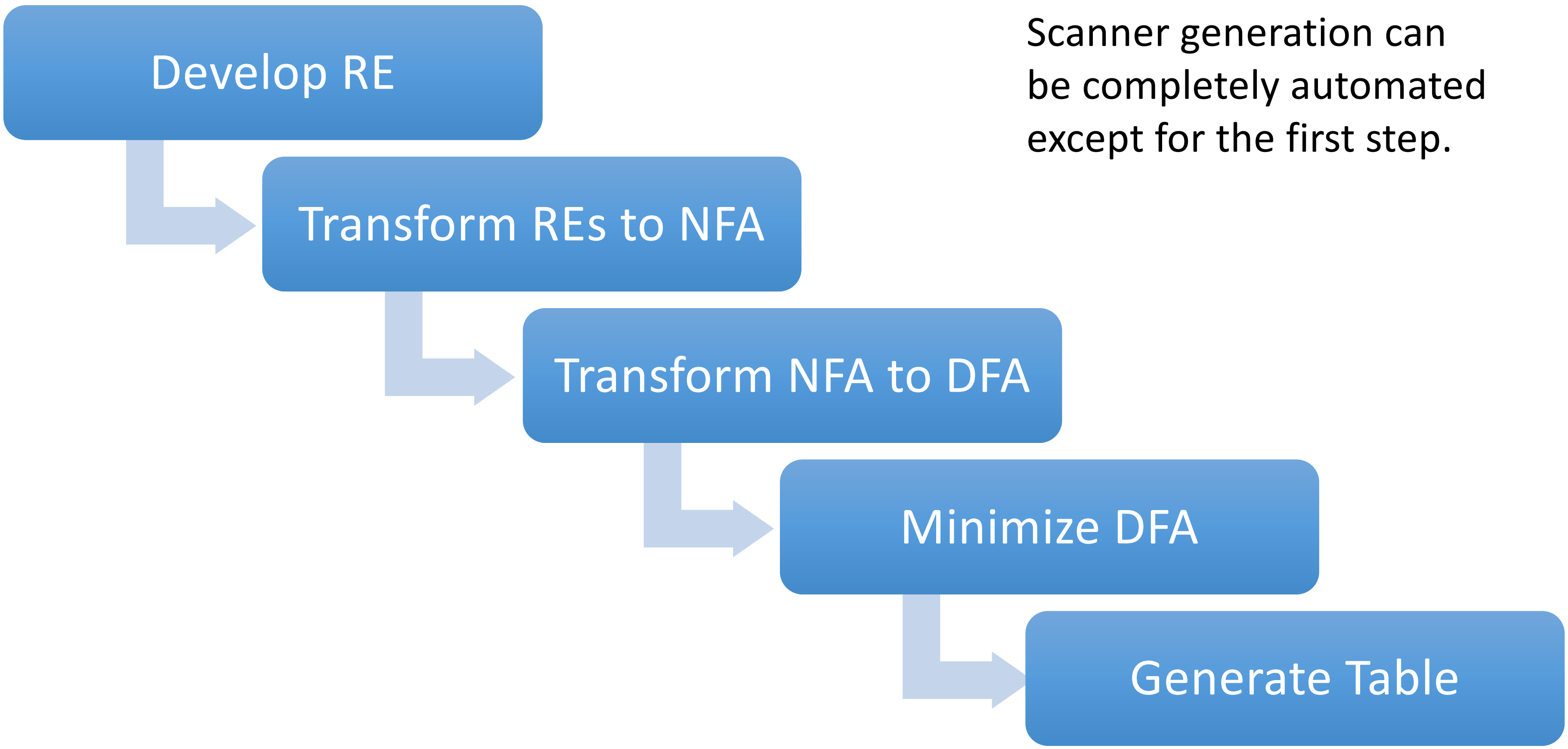
```
%{
  #include <string.h>
  #include "y.tab.h"
%}
D        [0-9]
NUM      {D}+
INT      "-"?{NUM}
ID       [A-Za-z_][A-Za-z0-9_]*
WS       [\t\ \r\n]
%%
<<EOF>>  { return(END_OF_FILE);}
while    { return(KW_WHILE);}
if       { return(KW_IF);}
...
```

- A script such as the one on the left is fed into a scanner generator

  E.g., flex, lex, etc

- The script contains regular expressions and actions.
- The generator generates code that performs the specific actions when the corresponding token is encountered.

Generating a Scanner



Scanner generation can be completely automated except for the first step.

Note: These are extended DFAs

- Token type, value and location are returned, not (accept/reject)
- A different accepting state is used for each token type
- Keyword and identifiers are treated separately (both look the same)
    - Keywords are encoded in REs or stored in a hash-table.
- Backtracking to last accepted state is done to find longest token
- **Question: How do we ensure that our tokens are representable by REs?**

Are All Tokens Regular?

- Our theory/scanners only work for regular languages
- Tokens are typically regular (How do we know?)
- What happens when we combine tokens?
- Do the languages remain regular?
- It depends...

Properties of Regular Languages

If R and S are regular languages then so are:

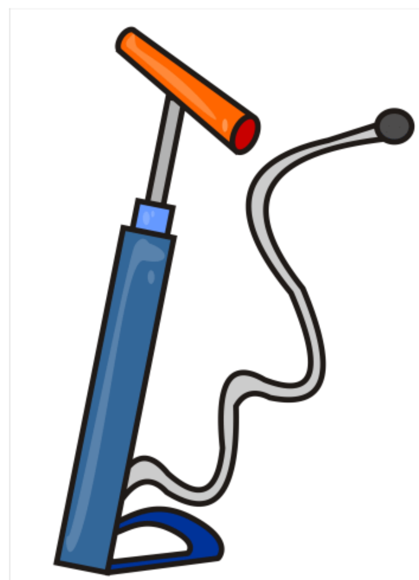
- RS, R ∪ S, $R*$
  by definition, RLs are closed under concatenation, union, and Kleene-*
- $\overline{R}$ = $\Sigma^* \setminus$ R (complement of R)
  Switch accepting and rejecting states of the DFA (or NFA).
- $R^r = \{\sigma^r | \sigma \in R\}$ (reverse of R)
  RE for R written backwards or reverse transitions in DFA …
- R∩S (intersection of R and S)
  $R \cap S = \overline{\overline{R} \cup \overline{S}}$
- R \ S : symmetric difference of R and S
  $R \setminus S = R \cap \overline{S}$

Examples:

- Show that $L = \{a^p \mid p \text{ is not prime}\}$ is not regular.
  - Recall that $\{a^p \mid p \text{ is prime}\}$ is not regular
- Show that $L = \{a^p b^q \mid p \text{ or } q \text{ is prime}\}$ is not regular.
- Show that $L = \{a^p a^* a^p \mid p \text{ is prime}\}$ is regular.

Nonregular Languages

- Problem: Not all languages are regular!
  - E.g. $L = \{0^n1^n | n >= 0\}$ is not regular.
- Intuition: We need to keep track of how many 0's we encounter.
- A DFA has a finite number of states, so beyond that number we cannot keep track.
- How do we prove this formally?
  - The Pumping Lemma!



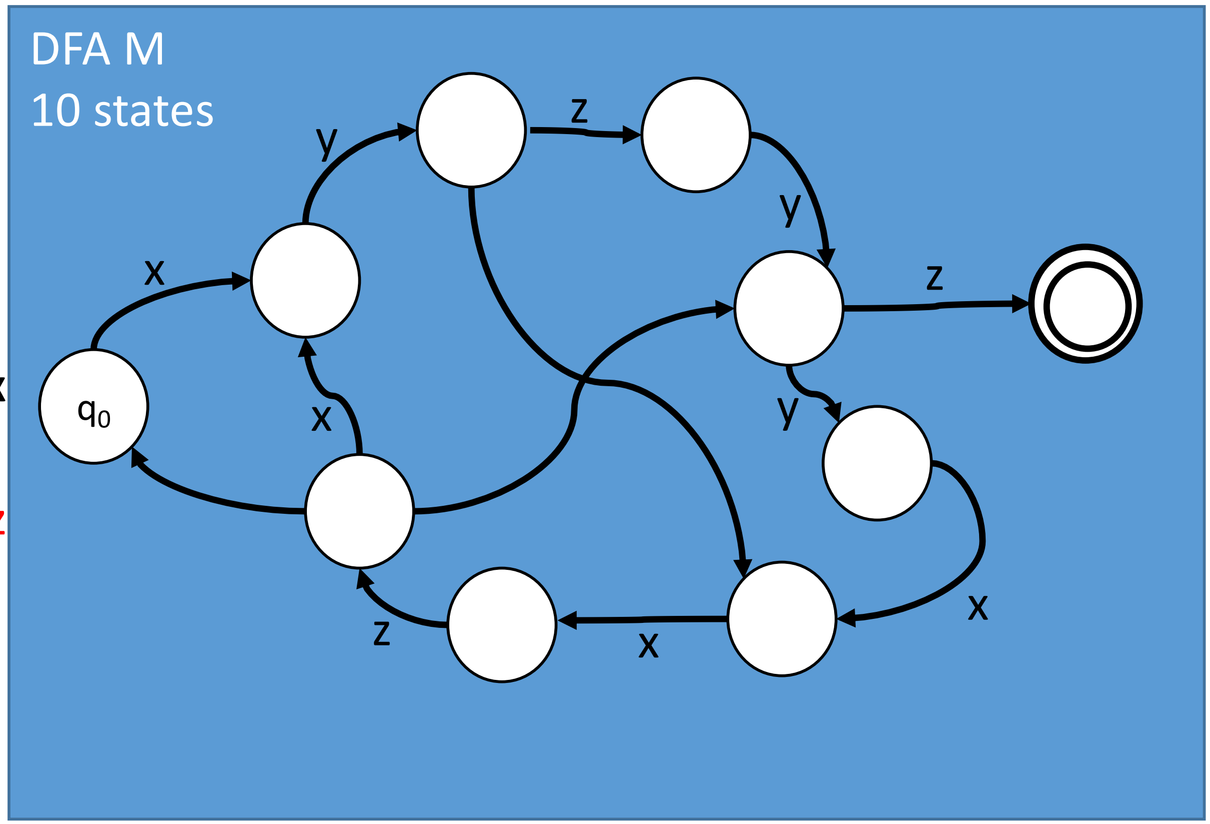
Intuition

xyzyz

xyzyyxxzx

xyzyyxxzxyzyz

x(yzyyxxzx)^{i}yzyz



Any string of this form is accepted!

Observation: If a DFA with n states reads a string of length n or greater, one or more states will be visited multiple times (there will be a cycle).

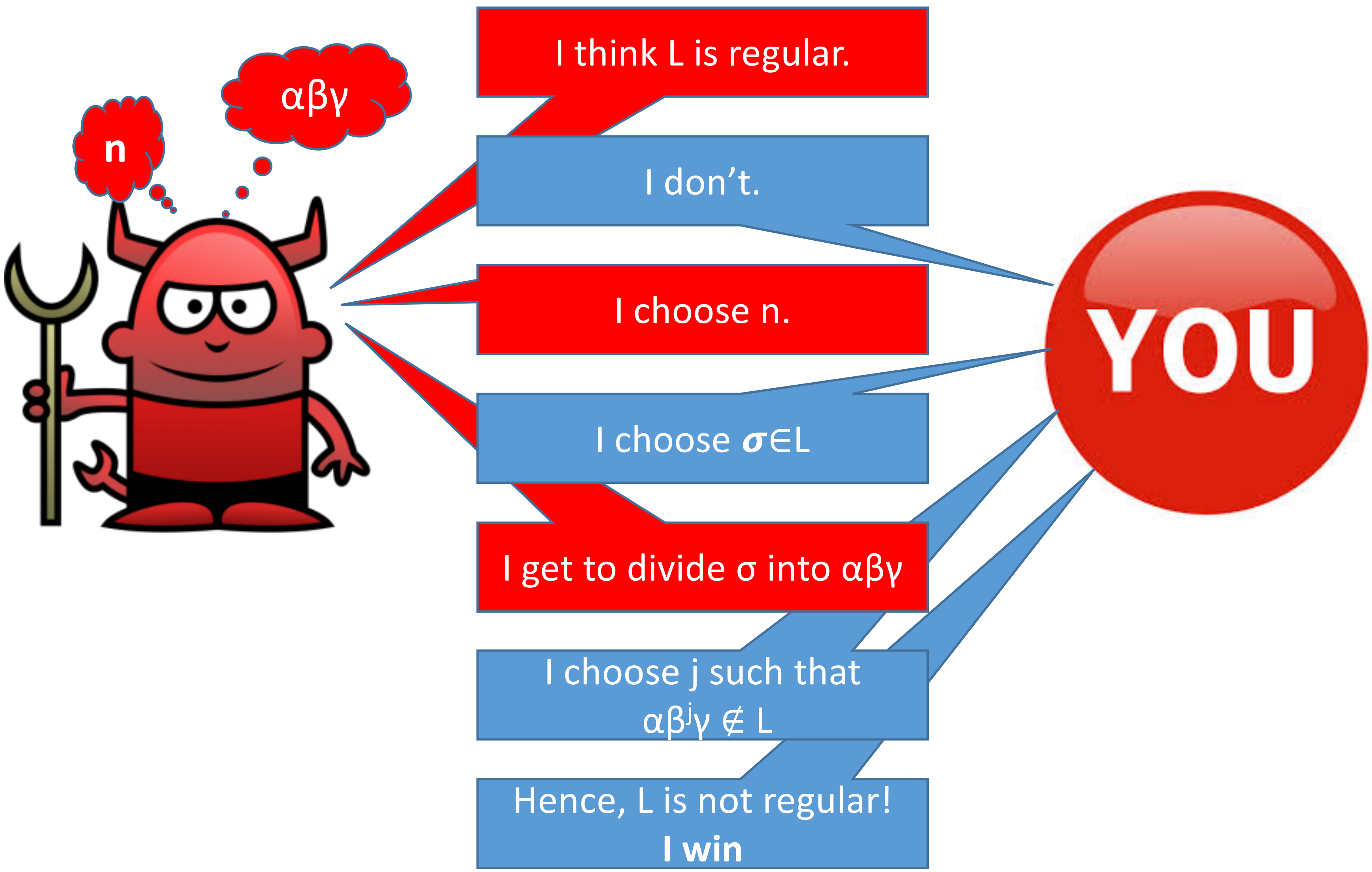
The Pumping Lemma

For every regular language L, there exists a constant n such that every $\sigma \in L$, where $|\sigma| \geq n$, can be divided into three substrings $\sigma = \alpha\beta\gamma$ with the following properties:

- $|\alpha\beta| \leq n$
- $|\beta| > 0$,and
- $\alpha\beta^k\gamma \in L, \forall k \geq 0$

- We can use this Lemma to show that a given language is **non-regular**.

Using the Pumping Lemma is like an Argument with the Devil



Applying the Pumping Lemma

Procedure: To show that L is not regular

- Convince yourself L is not regular (intuition)
- Assume that L is regular and that there is a constant n as stated by the Pumping Lemma
- Select $\sigma \in L$ such that
  - $|\sigma| > n$
  - $\sigma = \alpha\beta\gamma$ such that **for all α and β**
    - $|\alpha\beta| \leq n$
    - $|\beta| > 0$
- Show that there **exists a $j \geq 0$** such that $\alpha\beta^j\gamma \notin L$
- But according to Pumping Lemma, $\sigma \in L$.
- Contradiction!
- Therefore L is not regular.

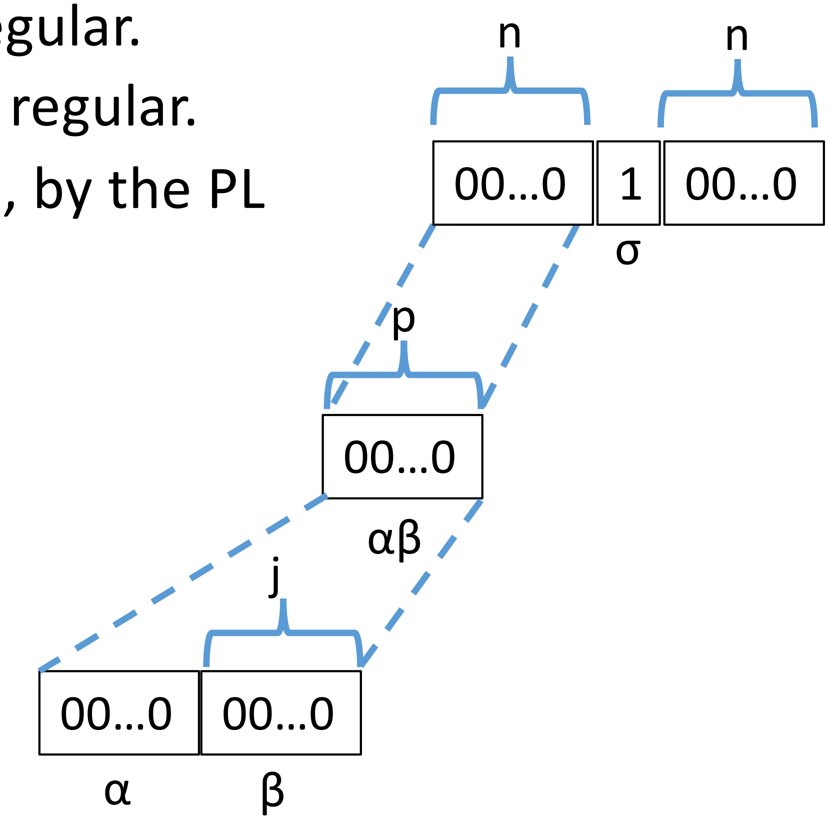
σ will (typically) be a function of n

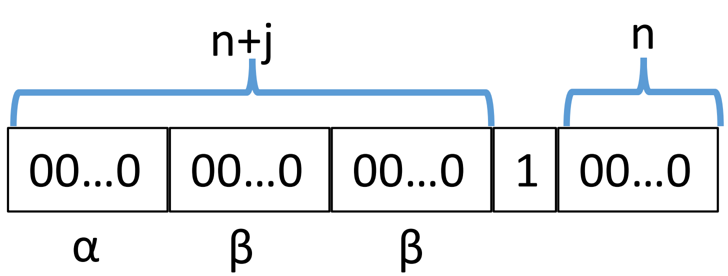
HARD PART

Example: Use the Pumping Lemma

Show that $L = \{0^m 1 0^m \mid m \geq 0\}$ is not regular.

- Proof by contradiction: Assume L is regular.
- If L is regular, then there exists an n, by the PL
- Select $\sigma = 0^n 1 0^n$
- Therefore, **for all α and β**
    - $\alpha\beta = 0^p$, because $p \leq n$
    - $\beta = 0^j$, $0 < j \leq p$
- By the PL, $\alpha\beta^2\gamma \in L$
- But $\alpha\beta^2\gamma = 0^{n+j}10^n \notin L$
- **Contradiction!**





Examples

- $L = \{a^i b^j | i < j\}$
- $L = \{a^p | p \text{ is prime}\}$
- $L = \{a^i b^j \mid i = j \bmod 3\}$

  This one is actually regular
- Note: We cannot use the Pumping Lemma to prove a language is regular.
- Question: How do you show a language is regular?
  - Construct a regular expression for the language
  - Construct an NFA that recognizes the language
  - Construct the language from known Regular Languages using closure properties of regular languages.