Building a Scanner and Properties of RLs

CSCI 3136: Principles of Programming Languages

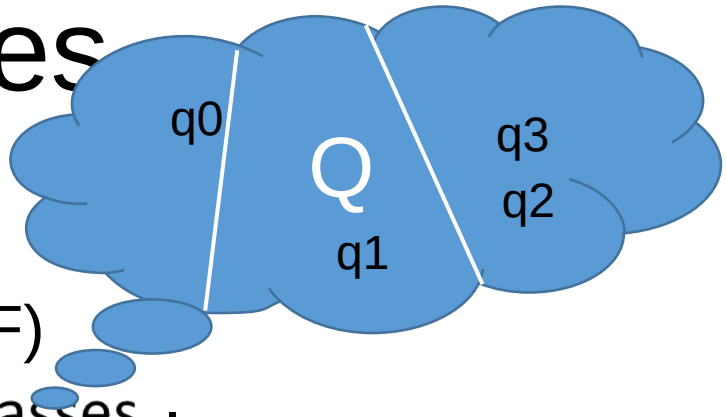
Agenda

- Minimization
- Scanner Implementations
- Properties of Regular Languages
- The Pumping Lemma

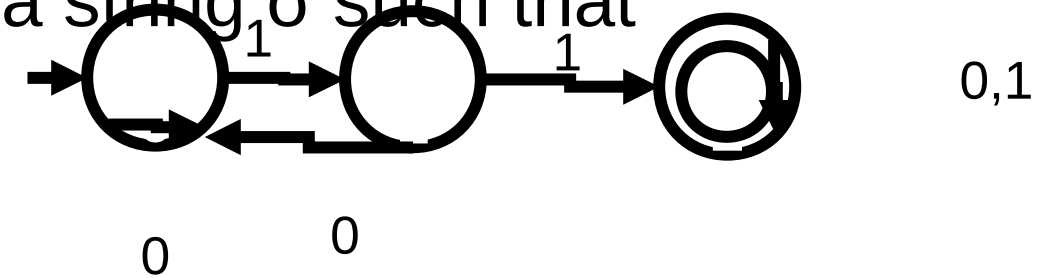
Minimization of Automata

- **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 ,
  - 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, Σ, δ, q0, 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
  - δ(q1,σ) E F
  - δ(q2,σ) ∉ F



Minimisation Procedure

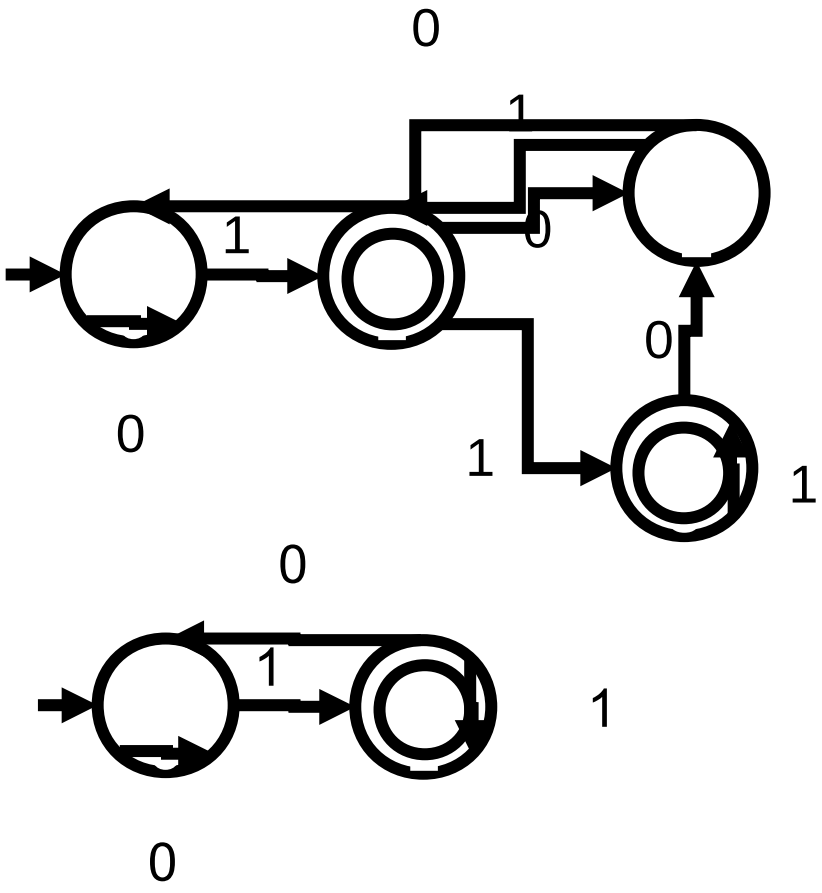
- Delete all unreachable states
  - Easily done with a graph traversal (BFS/DFS) from the start state
  - A state is unreachable if it only has transitions out (except for the start state)
  - Don't usually have to do this for DFAs created from NFAs
- Create a table with one row and one column for each state
- This table is used to identify "distinguishable" states
  - A pair of states is distinguishable if there exists a string σ such that:

    $\delta(q_1,\sigma) \in F$ AND $\delta(q_2,\sigma) \notin F$
  - That is, states are distinguishable if they are not in the same equivalence class.
  - Mark all pairs of states (in the table) if one is final and the other is not – obviously they are distinguishable, one is final and the other isn't

Minimisation (Continued)

- For every unmarked pair, q1 and q2, (which could be equivalent):
- Examine the transitions for all *a* E Σ
  - if δ(q1,*a*) AND δ(q2,*a*) both go to the same state, q3, then leave the pair as unmarked
  - if, for any *a*, they go to different resulting states then mark the pair as distinguishable
- Idea: if any character causes a transition to different places, the states are distinguishable, if all characters produce the same result, the pair are identical (indistinguishable)

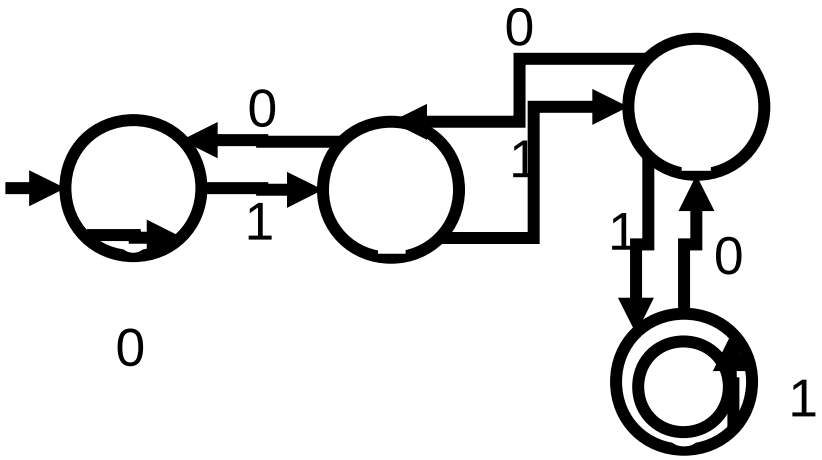
Example 1



	q0	q1	q2	q3
q0	NA	ΔF		ΔF
q1	ΔF	NA	ΔF	
q2		ΔF	NA	ΔF
q3	ΔF		ΔF	NA

Example 2

	q0	q1	q2	q3
q0	NA			
q1	Δ1	NA		
q2	Δ0	Δ0	NA	
q3	ΔF	ΔF	ΔF	NA



All states are marked as "distinguishable" and thus, the DFA must already be minimal.

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

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 if and switch statements to decide whether the next character is part of the current token or the beginning of the next token
- 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 a you have input

```
while input available:
    c = next_char()
    switch(state):
    case NEW_TOKEN:
        switch(c):
        case [ \n\t\r]: # white space
                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:
```

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_]*
```

- 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

Generating a Scanner

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

Flex just needs the programmer to describe the PL using REs.

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)
  - Key words 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**

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, $R \cup S$, 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).
- $R^r = \{\sigma \mid \sigma \in R\}$ (reverse of R)
  RE for R written backwards or reverse transitions in DFA …
- $R \cap S$ (intersection of R and S
  $R \cap S = \overline{\overline{R} \cup \overline{S}}$
- $R \setminus S$: symmetric difference of R and S
  $R \setminus S = R \cap \overline{S}$

RS, R ∨ S, RX
by definition, RLs are closed under concatenation, union, and Kleene-*
= ΣX \ R (Complement of R)
Switch accepting and rejecting states of the DFA (or NFA).
Rr = {σr | σ∈R} (reverse of R)
RE for R written backwards or reverse transitions in DFA …
R∩S (intersection of R and S
R \ S: symmetric difference of R and S

Nonregular Languages

- Problem: Not all languages are regular!

  E.g. L = {0n1n|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 t

- How do we prove this formally

  The Pumping Lemma!

Finiteness

- The language L = {0i1i | i > 0} is not regular.
- A regular language is recognized by a DFA.
- Strings in L can be very long if i is very large.
- We need to somehow count how many 0s we have (using states).
- At some point, we can make i > number of states and the DFA will fail.
- A DFA has a finite number of states .. this is important.

Intuition: The Pumping Lemma

- Every Regular Language has a DFA recognizer.
- This recognizer has n states.
- If a string in the language is longer than n, the recognizer has to have a cycle.
- Thus, we can break the string into 3 parts:
  1. characters before the cycle
  2. the cyclic part
  3. characters after the cycle
- "Pumping" means "repeating the cycle"

The Pumping Lemma

For every regular language L, there exists a constant n (the number of states in its DFA) such that every σ E L, where |σ| ≥ n, can be divided into three substrings σ = αβγ with the following properties:

- |αβ|≤n
- |β| > 0, (this is the cyclic part) and
- αβkγEL,′I k≥0

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

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 σ E L such that
  - |σ| > n
  - σ = αβγ such that <u>for all α and β</u>
    - |αβ| ≤ n
    - |β| > 0
  - There <u>exists a j ≥ 0</u> such that αβjγ Z L

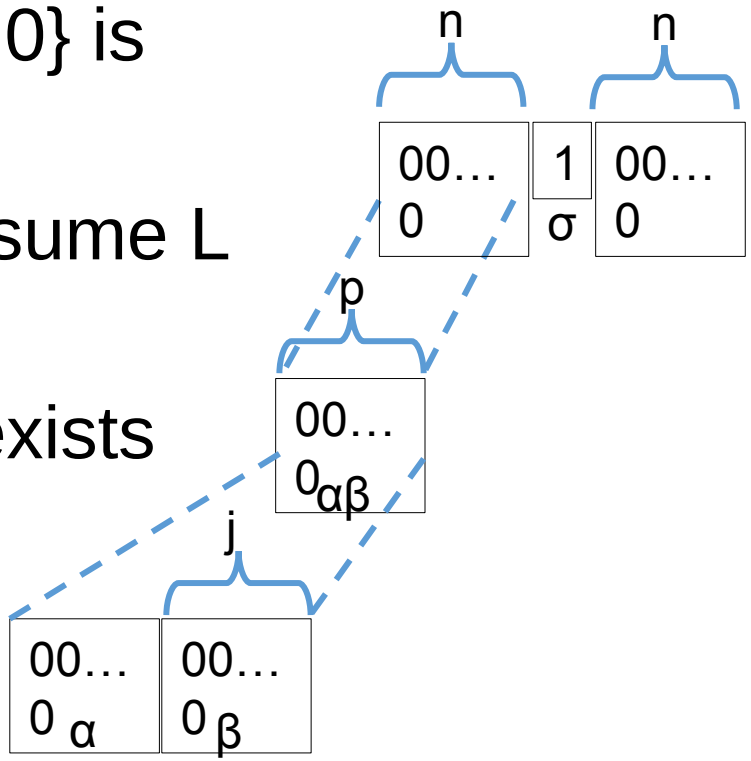
HARD PART

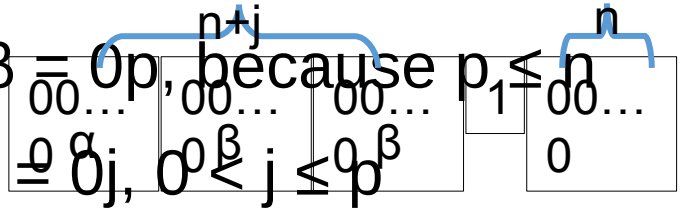
Example: Use the Pumping Lemma

Intuition: DFA can only keep track of n + 1 things (at most).

Show that L = {0m10m|m ≥ 0} is not regular.

- Proof by contradiction: Assume L is regular.
- If L is regular, then there exists an n, by the PL
- Select σ = 0n10n
- Therefore, **for all α and β**
  - αβ = 0p, because p ≤ n
  - β = 0j, 0 < j ≤ p





Using the Pumping Lemma is like an Argument with the Devil

I think L is regular.

I don't.

I choose n.

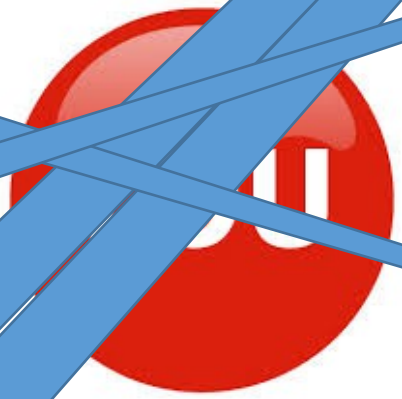
Fine, let □ = …

I get to choose any α and β such that σ = αβγ

Fine, I choose j such that □' = αβjγ ∉ L

Hence, L is not regular!

I win.



Examples

- L = {aibj|i < j}
- L = {ap|p is prime}
- L = {aibj | i = j mod 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