Last Time

- **FSA**
  - gave a formal definition
    - \((Q,s,f,\Sigma,\delta)\)
  - background reading
    - Chapter 2: *Regular Expressions and Finite State Automata*
  - many practical applications
    - *can be encoded and run efficiently on a computer*
    - implement regular expressions
    - compress large dictionaries
    - build morphological analyzers (suffixation)
      - *see chapter 3 of textbook*
    - speech recognizers
      - (Hidden) Markov models = FSA + probabilities
FSA: Recap

- **formally**
  - \((Q, s, f, \Sigma, \delta)\)
  1. set of states \((Q)\): \{s, w, x, y, z\}
     4 states *must be a finite set*
  2. start state \((s)\): \(s\)
  3. end state(s) \((f)\): \(z\)
  4. alphabet \((\Sigma)\): \{a, b, !\}
  5. transition function \(\delta\):
     signature: character \(\times\) state \(\rightarrow\) state
     - \(\delta(b, s) = w\)
     - \(\delta(a, w) = x\)
     - \(\delta(a, x) = y\)
     - \(\delta(a, y) = y\)
     - \(\delta(!, y) = z\)

\(\lambda\) (or \(\varepsilon\)) denotes the empty transition
regexps and FSA: recap

anything we can encode with a regular expression, we can build a FSA for it

• **basic operators, e.g.**
  - +
    - one or more of the preceding character
    - e.g. a+

• **backreferences:**
  - *by state splitting*
  - e.g. ([aeiou])\1
Today’s Lecture

• how to implement FSA
  – in Prolog
  – two ways: one based directly on the formal definition...
Finite State Automata (FSA)

- formally
  - \((Q, s, f, \Sigma, \delta)\)
  1. \(Q\): \{s, x, y\}
  2. \(s\): s
  3. \(f\): y
  4. \(\Sigma\): \{a, b\}
  5. transition function \(\delta\):
     - signature: character × state → state
     - \(\delta(a, s) = x\)
     - \(\delta(a, x) = x\)
     - \(\delta(b, x) = y\)
     - \(\delta(b, y) = y\)
Finite State Automata (FSA)

- directly implement the formal definition
  - define a predicate `fsa/2`
  - takes two arguments
  - `S` = a start state
  - `L` = string (as a list) we’re interested in testing

Prolog code (for any FSA)
1. `fsa(S,L) :-`
   
   `L = [C|M], transition(S,C,T), fsa(T,M).`

2. `fsa(E,[]):- end_state(E).`

:– to be read as “true if”
Finite State Automata (FSA)

• **Prolog code (for any FSA)**
  - fsa(S,L) :-
    L = [C|M],
    transition(S,C,T),
    fsa(T,M).
  - fsa(E,[]) :- end_state(E).

• **Facts (FSA-particular)**
  - end_state(y).
  - transition(s,a,x).
  - transition(x,a,x).
  - transition(x,b,y).
  - transition(y,b,y).

*transition function* $\delta$:
  - $\delta(a,s)=x$
  - $\delta(a,x)=x$
  - $\delta(b,x)=y$
  - $\delta(b,y)=y$
Finite State Automata (FSA)

- computation tree

```
?- fsa(s, [a,a,b]).
?- transition(s,a,T).       T=x
?- fsa(x, [a,b]).
    ?- transition(x,a,T').   T'=x
?- fsa(x, [b]).
    ?- transition(x,b,T'').  T''=y
?- fsa(y, []).
?- end_state(y).           Yes
```

```
fsa(S,L) :-
    L = [C|M],
    transition(S,C,T),
    fsa(T,M).
fsa(y, []) :- end_state(E).
```
Finite State Automata (FSA)

- **deterministic FSA (DFSA)**
  - no ambiguity about where to go at any given state

- **non-deterministic FSA (NDFSA)**
  - no restriction on ambiguity (surprisingly, no increase in formal power)

- **textbook**
  - D-RECOGNIZE (FIGURE 2.13)
  - ND-RECOGNIZE (FIGURE 2.21)

---

```prolog
fsa(S,L) :-
    L = [C|M],
    transition(S,C,T),
    fsa(T,M).
fsa(y,[[]]) :- end_state(E).
```

---

```prolog
function D-RECOGNIZE(tape, machine) returns accept or reject
    index ← Beginning of tape
    current_state ← Initial state of machine
    loop
        if End of input has been reached then
            if current_state is an accept state then
                return accept
            else
                return reject
        elseif transition_table[current-state, tape[index]] ≠ nil then
            return reject
        else
            current-state ← transition_table[current-state, tape[index]]
            index ← index + 1
        end
    end
```

```prolog
function ND-RECOGNIZE(tape, machine) returns accept or reject
    agenda ← [[Initial state of machine, beginning of tape]]
    current-search-state ← NEXT(agenda)
    loop
        if ACCEPT-STATE(current-search-state) returns True then
            return accept
        else
            agenda ← agenda \ (Current-State(current-search-state))
            if agenda is empty then
                return reject
            end
    end
```

---

```prolog
STATE(x) returns a set of search
    current-search-state is in
    the current search state is looking at
    from transition table as follows:
    transition(S,T, index)
    index <- index + 1
end
```

---

```prolog
function ACCEPT-State(search state) returns true or false
    corresponded : the mode search state is in
    the point on the tape search state is looking at
    if [index] is the end of the tape and current-node is an accept state of machine
    then
        return true
    else
        return false
    end
```
Finite State Automata (FSA)

- **Prolog Advantages**
  - no change in code for NDFSA
  - Prolog computation rule takes care of choice point management

- **example**
  - one change
  - “a” instead of “b” from x to y
  - non-deterministic
  - what regular language does this machine accept?
Finite State Automata (FSA)

- another possible Prolog encoding strategy
  - define one predicate for each state
    - taking one argument (the input string)
    - consume input character
    - call next state with remaining input string
  - query
    - $?- s(\text{L}).$

  call start state s
Finite State Automata (FSA)

- **state s:** *(start state)*
  - $s([a|L]) :- x(L)$.
    match input string beginning with $a$
    and call state $x$ with remainder of input

- **state x:**
  - $x([a|L]) :- x(L)$.
  - $x([b|L]) :- y(L)$.

- **state y:** *(end state)*
  - $y([])$.
  - $y([b|L]) :- y(L)$.
Finite State Automata (FSA)

example:

1. ?- s([a,a,b]).
2. ?- x([a,b]).
3. ?- x([b]).
4. ?- y([]). Yes

\[
\begin{align*}
s([a|L]) & : - x(L). \\
x([a|L]) & : - x(L). \\
x([b|L]) & : - y(L). \\
y([]) & . \\
y([b|L]) & : - y(L). \\
\end{align*}
\]
Finite State Automata (FSA)

• Note:
  – non-deterministic properties of Prolog’s computation rule still applies here
Finite State Automata (FSA)

example

1. \( ?- s([a,b,a]) \).
2. \( ?- x([b,a]) \).
3. \( ?- y([a]) \). No

\[
\begin{align*}
s([a|L]) & : \leftarrow x(L). \\
x([a|L]) & : \leftarrow x(L). \\
x([b|L]) & : \leftarrow x(L). \\
y([b|L]) & : \leftarrow y(L). \\
y([b|L]) & : \leftarrow y(L). \\
\end{align*}
\]
Class Project

• Implement the following FSA in Prolog
  – using the one predicate per state model

![FSA Diagram]

*make sure your program can generate*