Today’s Topics

• Homework 2 review
• Recap: DCG system
• New topic: Finite State Automata (FSA)
Homework 2 Review

• Question 1
  – Consider a language
    \( L_{\text{3or5}} = \{111, 11111, 111111, 111111111, 1111111111, 111111111111, \ldots \} \)
  – each member of \( L_{\text{3or5}} \) is a string containing only 1s
  – the number of 1s in each string is divisible by 3 or 5

• Give a regular grammar that generates language \( L_{\text{3or5}} \)
Homework 2 Review

- **Regular grammar:**
  - *divisible by 3 side*
  1. $s \rightarrow [1], a.$
  2. $a \rightarrow [1], b.$
  3. $b \rightarrow [1].$
  4. $b \rightarrow [1], c.$
  5. $c \rightarrow [1], d.$
  6. $d \rightarrow [1], b.$

- **Regular grammar:**
  - *divisible by 5 side*
  7. $s \rightarrow [1], \text{one}.$
  8. $\text{one} \rightarrow [1], \text{two}.$
  9. $\text{two} \rightarrow [1], \text{three}.$
  10. $\text{three} \rightarrow [1], \text{four}.$
  11. $\text{four} \rightarrow [1].$
  12. $\text{four} \rightarrow [1], \text{five}.$
  13. $\text{five} \rightarrow [1], \text{six}.$
  14. $\text{six} \rightarrow [1], \text{seven}.$
  15. $\text{seven} \rightarrow [1], \text{eight}.$
  16. $\text{eight} \rightarrow [1], \text{four}.$

*take the disjunction of the two sub-grammars, i.e. *merge them*
• **Question 2:** Language \( L = \{a^{2n}b^{n+1} \mid n \geq 1\} \) is also non-regular

• but can be generated by a regular grammar extended to allow left and right recursive rules

• Given a Prolog grammar satisfying these rules for \( L \)

• **Legit rules:**
  \[
  \begin{align*}
  X & \rightarrow aY \\
  X & \rightarrow a \\
  X & \rightarrow Ya \\
  \end{align*}
  \]
  – where \( X, Y \) are non-terminals, \( a \) is some arbitrary terminal
Homework 2 Review

- \( L = \{a^{2n}b^{n+1} \mid n \geq 1\} \)

- **DCG:**
  1. \( s \rightarrow [a], b. \)
  2. \( b \rightarrow [a], c. \)
  3. \( c \rightarrow [b], d. \)
  4. \( c \rightarrow s, [b]. \)
  5. \( d \rightarrow [b]. \)

  (center-embedded recursive tree)

  generated by rules 1 + 2 + 3 + 5
Homework 2 Review

• **Question 3 (Optional 438)**
  A right recursive regular grammar that generates a (rightmost) parse for the language \( \{a^n \mid n \geq 2\} \):
  
  - \( s(s(a,B)) \rightarrow [a], b(B) \).
  - \( b(b(a,B)) \rightarrow [a], b(B) \).
  - \( b(b(a)) \rightarrow [a] \).

• **Example**
  
  - \( s(Tree, [a,a,a], []) \).
  
  Tree = \( s(b(b(a),a),a) \)

• **Modify the right recursive grammar to produce a left recursive parse**, e.g.
  
  - \( s(Tree, [a,a,a], []) \).
  
  Tree = \( s(b(b(a),a),a) \)

• **Can you do this for any right recursive regular grammar?**
  e.g. the one for sheeptalk

• **A corresponding left recursive regular grammar will not halt in all cases when an input string is supplied**
Homework 2 Review

• Original:
  \[ s(s(a,B)) \rightarrow [a], b(B). \]
  \[ b(b(a,B)) \rightarrow [a], b(B). \]
  \[ b(b(a)) \rightarrow [a]. \]

• New:
  1. \[ s(s(B,a)) \rightarrow [a], b(B). \]
  2. \[ b(b(B,a)) \rightarrow [a], b(B). \]
  3. \[ b(b(a)) \rightarrow [a]. \]

*Taking advantage of the fact that we know we’re generating only a’s example:
In rule 1, we match a terminal a on the left side but place an a at the end for the tree*
Extra Arguments: Recap

• some uses for extra arguments on non-terminals in the grammar:
  1. to generate a parse tree
     • recording the derivation history
  2. feature agreement
  3. implement counting
Extra Arguments: Recap

Implement
Determiner-Noun
Number
Agreement

• **Data:**
  - the man/men
  - a man/*a men

• **Modified grammar:**

  \[
  \begin{align*}
  \text{np}(\text{np}(D,N)) & \rightarrow \text{det}(D,\text{Number}), \\
  & \qquad \text{common\_noun}(N,\text{Number}). \\
  \text{det}(\text{det}(\text{the}),_\_ ) & \rightarrow [\text{the}]. \\
  \text{det}(\text{det}(a),sg) & \rightarrow [a]. \\
  \text{common\_noun}(n(\text{ball}),sg) & \rightarrow [\text{ball}]. \\
  \text{common\_noun}(n(\text{man}),sg) & \rightarrow [\text{man}]. \\
  \text{common\_noun}(n(\text{men}),pl) & \rightarrow [\text{men}].
  \end{align*}
  \]
Extra Arguments: Recap

• **Class Exercise:**
  – *Implement* Case = {nom,acc} *agreement system for the grammar*
  – **Examples**: the man hit me *vs. *the man hit I

• **Modified grammar:**
  
  ```prolog
  s(Y,Z) --> np(Y,Case), vp(Z), { Case = nom }.
  np(np(Y),Case) --> pronoun(Y,Case).
  pronoun(i,nom) --> [i].
  pronoun(we,nom) --> [we].
  pronoun(me,acc) --> [me].
  np(np(D,N),_) --> det(D,Num), common_noun(N,Num).
  vp(vp(Y,Z)) --> transitive(Y), np(Z,Case), { Case = acc }.
  ```
  ```prolog
  { ... Prolog code ... }
  ```
Extra Arguments: Recap

• **Class Exercise:**
  – *Implement* Case = {nom,acc} agreement system for the grammar
  – **Examples:** the man hit me vs. *the man hit I*

• **Modified grammar:**

```
s(Y,Z) --> np(Y,nom), vp(Z).
np(np(Y),Case) --> pronoun(Y,Case).
pronoun(i,nom) --> [i].
pronoun(we,nom) --> [we].
pronoun(me,acc) --> [me].
np(np(D,N),_) --> det(D,Num), common_noun(N,Num).
vp(vp(Y,Z)) --> transitive(Y), np(Z,acc).
```
Extra Arguments: Recap

1. $s \rightarrow [a],b$.  
2. $b \rightarrow [a],b$.  
3. $b \rightarrow [b],c$.  
4. $b \rightarrow [b]$.  
5. $c \rightarrow [b],c$.  
6. $c \rightarrow [b]$.

$L = a^+ b^+$  
*a regular grammar*

$L_{ab} = \{ a^n b^n \mid n \geq 1 \}$  
*regular grammar + extra argument*

**Query:**  
?- $s(0,L,[])$. 

Extra Arguments: Recap

1. \( s(X) \rightarrow [a], b(a(X)). \)
2. \( b(X) \rightarrow [a], b(a(X)). \)
3. \( b(a(X)) \rightarrow [b], c(X). \)
4. \( b(a(0)) \rightarrow [b]. \)
5. \( c(a(X)) \rightarrow [b], c(X). \)
6. \( c(a(0)) \rightarrow [b]. \)

\[ \text{derivation tree:} \]

\[ \text{logic:} \]

\(- s(0, L, []). \)

start with 0
wrap an \( a(_\_) \) for each “a”
unwrap \( a(_\_) \) for each “b”

match \#a’s with \#b’s = counting
So far

- Equivalent formalisms

regexp → regular grammars

FSA
• **example** *(sheeptalk)*
  - baa!
  - baaa! …
• **regexp**
  - baa+!

**basic idea:** “just follow the arrows”

**state transition:**
in state *s*,
if current input symbol is *b*
go to state *w*

**accepting computation:**
in a final state,
if there is no more input, accept string

points to start state

end state marked in red
FSA: Construction

- step-by-step
- regexp
  - baa+!
  - baaa*!
FSA: Construction

- step-by-step
- **regexp**
  - `baaa*!`
  - `b`
  - from state `s`,
  - see a ‘b’,
  - move to state `w`
FSA: Construction

- step-by-step
- regexp
  - baaa*!
  - ba

- from state w,
- see an ‘a’,
- move to state x
FSA: Construction

- step-by-step
- regexp
  - baaa*!
  - baa
  - from state \(x\),
  - see an ‘a’,
  - move to state \(y\)
FSA: Construction

- step-by-step
- regexp
  - baaa*!
  - baaa*
  - baa
  - baaa
  - baaaa...
  - from y,
  - see an ‘a’,
  - move to ?

but machine must have a finite number of states!
Regular Expressions: Example

- step-by-step
- regexp
  - `baaa*!`
  - `baa*`
  - `baa`
  - `baaa`
  - `baaaa...`
  - **from state y,**
  - **see an ‘a’,**
  - “**loop”, i.e. move back to, state y**
Regular Expressions: Example

- step-by-step
- regexp
  - baaa*!
  - baaa*!
  - from state y,
  - see an ‘!’,
  - move to final state z (indicated in red)

Note:
machine cannot finish (i.e. reach the end of the input string) in states s, w, x or y
Finite State Automata (FSA)

• construction
  – the step-by-step FSA construction method we just used
  – works for any regular expression

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

  – *an important step in showing that FSA and regexps are formally equivalent*
regexp operators and FSA

• **basic wildcards**
  - . and *
    - . any single character
    - e.g. p.t
    - * zero or more characters

[Diagram showing a one loop for each character over the whole alphabet]
regexp operators and FSA

- **basic wildcards**
  - +
    - one or more of the preceding character
    - e.g. a+
  - [ ]
    - range of characters
    - e.g. [aeiou]
regexp operators and FSA

• **basic wildcards**
  – ?
    • zero or one of the preceding character
    • e.g. a?

• **Non-determinism**
  – any FSA with an empty transition is non-deterministic
  – **see example**: could be in state $x$ or $y$ simultaneous
  – any FSA with an empty transition can be rewritten without the empty transition

$\lambda$ denotes the empty transition
regexp operators and FSA

- backreferences:
  - by state splitting
  - e.g. ([aeiou])\1
Finite State Automata (FSA)

- more formally
  - \((Q,s,f,\Sigma,\delta)\)
    1. set of states \(Q\): \(\{s,w,x,y,z\}\) 4 states
    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
       * \(\delta(b,s)=w\)
       * \(\delta(a,w)=x\)
       * \(\delta(a,x)=y\)
       * \(\delta(a,y)=y\)
       * \(\delta(!,y)=z\)

\[\begin{array}{c}
\text{s} \quad b \quad \text{w} \quad a \quad \text{x}
\end{array}\]

\[\begin{array}{c}
\text{must be a finite set}
\end{array}\]

\[\begin{array}{c}
\text{y} \quad ! \quad \text{z}
\end{array}\]
FSA

- Finite State Automata (FSA) have a limited amount of expressive power

- Let’s look at a modification to FSA and its effect on its power
String Transitions

– so far...

• all machines have had just a single character label on the arc

• so if we allow strings to label arcs
  – do they endow the FSA with any more power?

• Answer: **No**
  – because we can always convert a machine with string-transitions into one without
Finite State Automata (FSA)

- equivalent

5 state machine

3 state machine using string transitions