• Grammar rules are translated when the program is loaded into Prolog rules.
• Solves the mystery why we have to type two arguments with the nonterminal at the command prompt
• Recall list notation:
  – \([1|[2,3,4]]\) = \([1,2,3,4]\)

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. \(c \rightarrow [b].\)

1. \(s([a|A], B) :- b(A, B).\)
2. \(b([a|A], B) :- b(A, B).\)
3. \(b([b|A], B) :- c(A, B).\)
4. \(b([b|A], A).\)
5. \(c([b|A], B) :- c(A, B).\)
6. \(c([b|A], A).\)
Regex, FSA and a Regular Grammar

Textbook Exercise: find a RE for

4. the set of all strings from the alphabet $a, b$ such that each $a$ is immediately preceded by and immediately followed by a $b$;

Examples (* denotes string not in the language):

*ab  *ba
bab
λ (empty string)
bb
*baba
babab
Regex, FSA and a Regular Grammar

• Draw a FSA and convert it to a RE:

Ungraded exercise:
Could use the technique shown in a previous class: eliminate one state at a time
4. The set of all strings from the alphabet $a, b$ such that each $a$ is immediately preceded by and immediately followed by a $b$.

- Regular Grammar in Prolog.
- Let’s begin with something like:
  - $s \rightarrow [b], b$. (grammar generates $bb^+$)
  - $b \rightarrow [b]$.
  - $b \rightarrow [b], b$.
  - Add rule:
    - $b \rightarrow [a], b$. (grammar generates $b+(a \mid b)^*b$)

- Classroom Exercise:
  - How would you fix this grammar so it can handle the specified language?
Regex, FSA and a Regular Grammar

4. the set of all strings from the alphabet $a, b$ such that each $a$ is immediately preceded by and immediately followed by a $b$;

- Regular Grammar in Prolog notation:
  - $s \rightarrow []$. ($S = \text{"start state"}$)
  - $s \rightarrow [b], b$. ($b = \text{"seen a b"}$)
  - $s \rightarrow [b], s$.
  - $b \rightarrow [a], c$. ($c = \text{"expect a b"}$)
  - $c \rightarrow [b]$.
  - $c \rightarrow [b], b$.
  - $c \rightarrow [b], c$. 
Regex, FSA and a Regular Grammar

- Compare the FSA with our Regular Grammar (RG)
  - s --> []. (s = "start state")
  - s --> [b], b. (b = "seen a b")
  - s --> [b], s.
  - b --> [a], c. (c = "expect a b")
  - c --> [b].
  - c --> [b], b.
  - c --> [b], c.

There is a straightforward correspondence between right recursive RGs and FSA
Regex, FSA and a Regular Grammar

• Informally, we can convert RG to a FSA
  – by treating
  – non-terminals as states
  – and introducing (new) states for rules of the form x --> [a].

[Powerpoint animation] in order of the RG rules

1. s --> [].
2. s --> [b], b.
3. s --> [b], s.
4. b --> [a], c.
5. c --> [b].
6. c --> [b], b.
7. c --> [b], c.
4. the set of all strings from the alphabet $a, b$ such that each $a$ is immediately preceded by and immediately followed by a $b$;

- Test cases (* denotes string not in the language):
  - *ab *ba $[a,b] [b,a]$  
  - bab $[b,a,b]$  
  - $\lambda$ (empty string) $[]$  
  - bb $[b,b]$  
  - *baba $[b,a,b,a]$  
  - babab $[b,a,b,a,b]$
Regex, FSA and a Regular Grammar

• Test cases
• (* denotes string not in the language):
  – *ab *ba \([a,b] [b,a]\)
  – bab \([b,a,b]\)
  – \(\lambda\) (empty string) \([]\)
  – bb \([b,b]\)
  – *baba \([b,a,b,a]\)
  – babab \([b,a,b,a,b]\)

• Output:

  ?- s([a,b],[□]).
  false.

  ?- s([b,a],[□]).
  false.

  ?- s([b,a,b],[□]).
  true ;
  false.

  ?- s([],[]).
  true.

  ?- s([b,b],[□]).
  true.

  ?- s([b,a,b,a,b],[□]).
  true ;
  false.
Regex, FSA and a Regular Grammar

DCG to Prolog conversion:
note the function of the two list arguments

1  s  -->  [ ] .
2  s  -->  [b] ,  b .
3  s  -->  [b] ,  s .
4  b  -->  [a] ,  c .
5  c  -->  [b] .
6  c  -->  [b] ,  b .
7  c  -->  [b] ,  c .

?- [g16].
true.
?- listing([s,b,c]).
s(A, A).
s([b|A], B) :-
  b(A, B).
s([b|A], B) :-
  s(A, B).
b([a|A], B) :-
  c(A, B).
c([b|A], A).
c([b|A], B) :-
  b(A, B).
c([b|A], B) :-
  c(A, B).
true.
Set Enumeration using Prolog

- Regular Grammar
  1. \( s \rightarrow [] \).
  2. \( s \rightarrow [b], b \).
  3. \( s \rightarrow [b], s \).
  4. \( b \rightarrow [a], c \).
  5. \( c \rightarrow [b] \).
  6. \( c \rightarrow [b], b \).
  7. \( c \rightarrow [b], c \).

 Normally, we ask the set membership question when posing a Prolog query:
  - e.g.
    
    \[?- s([a, b], []).\]
  - no

- Prolog enumeration:
  
  \[?- s(X, []).\]
  - \( X \) is a Prolog variable
  - asks the question for what values of \( X \) is \( s(X, []) \) true?
  - ; is disjunction (look for alternative answers)

\textbf{why?} Prolog matches rules in the order in which they’re written
Set Enumeration using Prolog

Let’s swap rules 2 and 3

• Regular Grammar
  1. s --> [].
  2. s --> [b], s.
  3. s --> [b], b.
  4. b --> [a], c.
  5. c --> [b].
  6. c --> [b], b.
  7. c --> [b], c.

• Prolog enumeration:
  ?- s(X, []).
  ?- s(L, []).
  L = [] ;
  L = [b] ;
  L = [b, b] ;
  L = [b, b, b] ;
  L = [b, b, b, b] ;
  L = [b, b, b, b, b] ;
  L = [b, b, b, b, b, b] ;
  L = [b, b, b, b, b, b, b] ;
  L = [b, b, b, b, b, b, b, b] ;
  L = [b, b, b, b, b, b, b, b, b] ;
Set Enumeration using Prolog

• Similarly, if we swap rules 6 and 7
• Regular Grammar
  1. s --> [].
  2. s --> [b], b.
  3. s --> [b], s.
  4. b --> [a], c.
  5. c --> [b].
  6. c --> [b], c.
  7. c --> [b], b.

  Prolog enumeration:
  ?- s(X,[]).

  ?- s(L,[]).
  L = [] ;
  L = [b, a, b] ;
  L = [b, a, b, b] ;
  L = [b, a, b, b, b] ;
  L = [b, a, b, b, b, b] ;
  L = [b, a, b, b, b, b, b, b] ;
  L = [b, a, b, b, b, b, b, b, b, b, b] ;

Prolog's left-to-right depth-first search strategy cannot handle this. Need a breadth-first strategy or (equivalently) an iterative deepening strategy.
Set Enumeration using Prolog

- **Iterative deepening:**
  - run Prolog depth-first strategy repeatedly allowing maximum depth of expansion at 1, 2, 3, 4, 5, and so on...
  - inefficient but simple
  - can build a meta-interpreter for this (*beyond the scope of this course*)
  - id_meta.pl
  - `id(Goal)` tries to prove Goal using iterative deepening

- **Example:**

```
?- id(s(L,[])).
L = [] ;
L = [b] ;
L = [b, a, b] ;
L = [b, b] ;
L = [b, a, b, b] ;
L = [b, b, a, b] ;
L = [b, b, b] ;
L = [b, a, b, a, b] ;
L = [b, b, a, b, b] ;
L = [b, b, a, a, b] ;
L = [b, b, b, b] .
```
Left Recursion and Set Enumeration

• Example:
  1. s --> a, [!].
  2. a --> ba, [a].
  3. a --> a, [a].
  4. ba --> b, [a].
  5. b --> [b].

• Grammar is:
  – a regular grammar
  – left recursive

• Question
  – What is the language of this grammar?

• Answer: *Sheeptalk*
  • ba..a! (# a's > 1)

• Sentential forms:
  • s
  • a!
  • baa!
  • baa!
Left Recursion and Set Enumeration

• Example:
  1. s --> a, [!].
  2. a --> ba, [a].
  3. a --> a, [a].
  4. ba --> b, [a].
  5. b --> [b].

• Prolog query:
  ?- s([b,a,a,!],[]).
  true

  ?- s([b,a,a,a,!],[]).
  true

  • But it doesn’t halt when faced with a string not in the language
  ?- s([b,a,!],[ ]).
  ERROR: Out of local stack
Left Recursion and Set Enumeration

- Example:
  \[ \begin{align*}
  s & \rightarrow a, [!] . \\
  a & \rightarrow ba, [a] . \\
  a & \rightarrow a, [a] . \\
  ba & \rightarrow b, [a] . \\
  b & \rightarrow [b] .
  \end{align*} \]

- In fact...

\[ \begin{align*}
? - s([b,a,a,!] , []) . \\
true ; \\
ERROR: Out of local stack \\
? - s([b,a,a,a,!] , []) . \\
true ; \\
ERROR: Out of local stack \\
? - s([b,a,!] , []) . \\
ERROR: Out of local stack \\
? - \end{align*} \]
Left Recursion and Set Enumeration

- Example:
  1. s → a, [!].
  2. a → ba, [a].
  3. a → a, [a].
  4. ba → b, [a].
  5. b → [b].

?- s([b,a,!,[]],[[]]).
ERROR: Out of local stack
Left Recursion and Set Enumeration

• left recursive regular grammar:
  1. $s \rightarrow a, [!]$.
  2. $a \rightarrow ba, [a]$.
  3. $a \rightarrow a, [a]$.
  4. $ba \rightarrow b, [a]$.
  5. $b \rightarrow [b]$.

• Behavior
  – halts when presented with a string that is in the language
  – doesn’t halt when faced with a string not in the language
  – unable to decide the language membership question

• Surprisingly, the query:
  ?- $s(L, [])$.
  enumerates the strings in the language just fine.

```prolog
?- s(L, []). L = [b, a, a, !] ; L = [b, a, a, a, !] ; L = [b, a, a, a, a, !] ; L = [b, a, a, a, a, a, !] ; L = [b, a, a, a, a, a, a, !] ; L = [b, a, a, a, a, a, a, a, !] ; L = [b, a, a, a, a, a, a, a, a, !] ; L = [b, a, a, a, a, a, a, a, a, a] w L = [b, a, a, a, a, a, a, a, a, a, a, !] ;
```

Left Recursion and Set Enumeration

- left recursive regular grammar:
  1. $s \rightarrow a, [!]$.
  2. $a \rightarrow ba, [a]$.
  3. $a \rightarrow a, [a]$.
  4. $ba \rightarrow b, [a]$.
  5. $b \rightarrow [b]$.

- Behavior
  - halts when presented with a string that is in the language
  - doesn’t halt when faced with a string not in the language

- derivation tree for $s(L, [])$.
  - $L = [b, a, a, !]$

- [Powerpoint animation]
Left Recursion and Set Enumeration

• However, this **slightly reordered** left recursive regular grammar:

1. \( s \rightarrow a, [!] \).
2. \( a \rightarrow a, [a] \).
3. \( a \rightarrow ba, [a] \).
4. \( ba \rightarrow b, [a] \).
5. \( b \rightarrow [b] \).

• *(rules 2 and 3 swapped)* won’t halt when enumerating

• **Why?**