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(List,[]).

  • $X$ is a Prolog variable
  • asks the question for what values of variable $List$ is $s(List,[])$ true?
  • ; is disjunction (look for alternative answers)

why? Prolog matches rules in the order in which they’re written

writes out list in full
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:

```prolog
?- 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] ;
```
Set Enumeration using Prolog

- Similarly, if we swap rules 6 and 7

- 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. \( c \rightarrow [b], c \).
  7. \( c \rightarrow [b], b \).

- Prolog enumeration:
  ```prolog
  ?- s(L, []).
  ?- 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] ;
  L = [b, a, b, b, b, b, b, b] ;
  L = [b, a, 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*)
  • $\text{id}(\text{Goal})$ tries to prove Goal using iterative deepening

• Example:

```prolog
?- 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, a, b, b, b] ;
L = [b, b, a, b, b] ;
L = [b, b, b, a, b] ;
L = [b, 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 (*nonterminal on left*)

• Question
  • What is the language of this grammar?

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

• Sentential forms:
  • s
  • a!
  • baa!
  • baa!
  • baa!

Underscoring used here to indicate a nonterminal
Left Recursion and Set Enumeration

• Example:
  1. $s \rightarrow a, [!]$.
  2. $a \rightarrow ba, [a]$.
  3. $a \rightarrow a, [a]$.
  4. $ba \rightarrow b, [a]$.
  5. $b \rightarrow [b]$.

• 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:
  s --> a, [!].
  a --> ba, [a].
  a --> a, [a].
  ba --> b, [a].
  b --> [b].

• In fact...

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

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

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

ERROR: Out of local stack

• Why?
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.

    ```
    ?- 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, !]
    ```
Left Recursion and Set Enumeration

• left recursive regular grammar:
  1. s → a, [!] .
  2. a → ba, [a] .
  3. a → a, [a] .
  4. ba → b, [a] .
  5. b → [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 re-ordered** 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?**

[Powerpoint animation]

descends infinitely using rule #2
So far ...

- We've talked about:
  - 1. Declarative (logical) reading of grammar rules
  - 2. Prolog query: `s(String,[]).`
    - Case 1. *String* is known: Is *String* ∈ L(G)?
    - Case 2. *String* is unknown: enumerate L(G)
  - 3. Different search strategies
    - Prolog's (left-to-right) depth-first search
    - Iterative deepening
Beyond Regular Languages

- Beyond regular languages
  - $a^nb^n = \{ab, aabb, aaabbb, aaaabbbb, \ldots \} \ n \geq 1$
  - is not a regular language

- That means no FSA, RE or RG can be built for this set

1. We only have a finite number of states to play with ...
2. We’re only allowed simple free iteration (looping)
3. Pumping Lemma proof
Beyond Regular Languages

• Language
  • \( a^n b^n \) = \{ab, aabb, aaabbb, aaaabbbb, \ldots \} \( n \geq 1 \)

• A regular grammar extended to allow both left and right recursive rules can accept/generate it:
  1. \( a \rightarrow [a], \ b. \)
  2. \( b \rightarrow [b]. \)
  3. \( b \rightarrow a, [b]. \)

• Example:

  
  ```prolog
  \?- a([b,b,a,a],[\[\]).
  false.
  
  \?- a([a,b],[\[). true ;
  false.
  
  \?- a([a,a,b],[\[). false.
  
  \?- a([a,a,b,b],[\[). true ;
  false.
  
  \?- a([a,a,b,b,b],[\[). false.
  
  \?- a([a,b,a,b],[\[). false.
  
  ?- a([a,b],[\[]. false.
  
  ?- a(L,[\[). Set membership
  L = [a, b] ;
  L = [a, a, b, b] ;
  L = [a, a, a, b, b] ;
  L = [a, a, a, a, b, b, b] ;
  ```

  Set enumeration
Beyond Regular Languages

• Language
  • $a^n b^n = \{ab, aabb, aaabbb, aaaabbbb, \ldots \}$ $n \geq 1$

• A regular grammar extended to allow both left and right recursive rules can accept/generate it:
  1. $a \rightarrow [a], b.$
  2. $b \rightarrow [b].$
  3. $b \rightarrow a, [b].$

• Intuition:
  • grammar implements the stacking of partial trees balanced for a’s and b’s:
Beyond Regular Languages

• Language
  • $a^n b^n = \{ab, aabb, aaabbb, aaaabbbb, \ldots \} \ n \geq 1$
  
• A regular grammar extended to allow both left and right recursive rules can accept/generate it:
  1. $a \rightarrow [a], b.$
  2. $b \rightarrow [b].$
  3. $b \rightarrow a, [b].$

• A type-2 or context-free grammar (CFG) has no restrictions on what can go on the RHS of a grammar rule

• Note:
  • CFGs still have a single nonterminal limit for the LHS of a rule

• Example:
  1. $s \rightarrow [a], [b].$
  2. $s \rightarrow [a], s, [b].$
Extra Argument: Parse Tree

- **Recovering a parse tree**
  - when want Prolog to return more than just *true/false* answers
  - in case of *true*, we can compute a syntax tree representation of the parse
  - by adding an extra argument to nonterminals
  - applies to all grammar rules (not just regular grammars)

Example
- *sheeptalk* again
- **DCG (non-regular, context-free):**
  
  \[
  \begin{align*}
  s & \rightarrow [b], [a], a, [!]. \\
  a & \rightarrow [a]. \text{ (base case)} \\
  a & \rightarrow [a], a. \text{ (recursive case)}
  \end{align*}
  \]
Extra Argument: Parse Tree

- **Tree:**
  \[
  s(\textcolor{blue}{b}, \textcolor{green}{a}, \textcolor{black}{a(\textcolor{black}{a}, \textcolor{black}{a(a)})}, \textcolor{red}{!})
  \]

- **Prolog term data structure:**
  - hierarchical
  - allows sequencing of arguments
  - functor(\textcolor{blue}{arg}_1,..,\textcolor{black}{arg}_n)
  - each \textcolor{blue}{arg}_i could be another term or simple atom
Extra Arguments: Parse Tree

- **DCG**
  - $s \rightarrow [b], [a], a, [!]$. (base case)
  - $a \rightarrow [a]$. (right recursive case)
  - $a \rightarrow [a], a$. (right recursive case)

- **base case**
  - $a \rightarrow [a]$.
  - $a(subtree) \rightarrow [a]$.
  - $a(a(a)) \rightarrow [a]$.

- **recursive case**
  - $a \rightarrow [a], a$.
  - $a(subtree) \rightarrow [a], a(subtree)$.
  - $a(a(a,A)) \rightarrow [a], a(A)$.

**Idea:** for each nonterminal, add an argument to store its subtree
Extra Arguments: Parse Tree

• Prolog grammar
  • $s \rightarrow [b], [a], a, !.$
  • $a \rightarrow [a].$ (base case)
  • $a \rightarrow [a], a.$ (right recursive case)

  • base and recursive cases
    - $a(a(a)) \rightarrow [a].$
    - $a(a(a, A)) \rightarrow [a], a(A).$

  • start symbol case
    - $s \rightarrow [b], [a], a, !.$
    - $s(tree) \rightarrow [b], [a], a(subtree), !.$
    - $s(s(b, a, A, !)) \rightarrow [b], [a], a(A), !.$
Extra Arguments: Parse Tree

• Prolog grammar
  • s --> [b], [a], a, ![].
  • a --> [a]. (base case)
  • a --> [a], a. (right recursive case)

• Equivalent Prolog grammar computing a parse
  - s(s(b,a,A,!)) --> [b], [a], a(A), ![].
  - a(a(a)) --> [a].
  - a(a(a,A)) --> [a], a(A).