LING/C SC/PSYC 438/538

Lecture 25

Sandiway Fong

Today's Topics

- Homework 13 Review (*not yet graded*)
- Beyond regular grammars …
- How to explicitly compute a parse tree
  - by adding an extra argument to for each grammar rule
- Left recursion and infinite loops

Homework 13 Review

Question 1: using the grammar file `bab.prolog`, describe what happens when you run the Prolog query `s(List, []).` typing in `;` repeatedly

```
?- s(List,[]).
List = [] ;
List = [b, a, b] ;
List = [b, a, b, a, b] ;
List = [b, a, b, a, b, a, b] ;
List = [b, a, b, a, b, a, b, a, b] ;
List = [b, a, b, a, b, a, b, a, b|...]
```

b(ab)+|ε
only a sub-language!

Homework 13 Review

```
1 s --> [].¶
2 s --> [b], seen_b.
3 s --> [b], s.
4 ¶
5 seen_b --> [a], seen_a.
6 ¶
7 seen_a --> [b].¶
8 seen_a --> [b], seen_b.¶
9 seen_a --> [b], seen_a.¶
```

never get to this rule on line 9!

- nonterminal seen_a has 3 rules (*lines* 7-9):
  - *what happens if we flip lines 8 and 9*?

Homework 13 Review

Question 3: Rearrange the order of the rules so that the query `s(List, [])`., followed by `;` can generate strings `[]`, `[b,a,b]`, `[b,a,b,b]`, `[b,a,b,b,b]` *etc.*

bab+|ε
is another sub-language!

```
?- [bab2].
true.

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

```
1 s --> [].¶
2 s --> [b], seen_b.
3 s --> [b], s.
4 ¶
5 seen_b --> [a], seen_a.
6 ¶
7 seen_a --> [b].¶
8 seen_a --> [b], seen_a.¶
9 seen_a --> [b], seen_b.
```

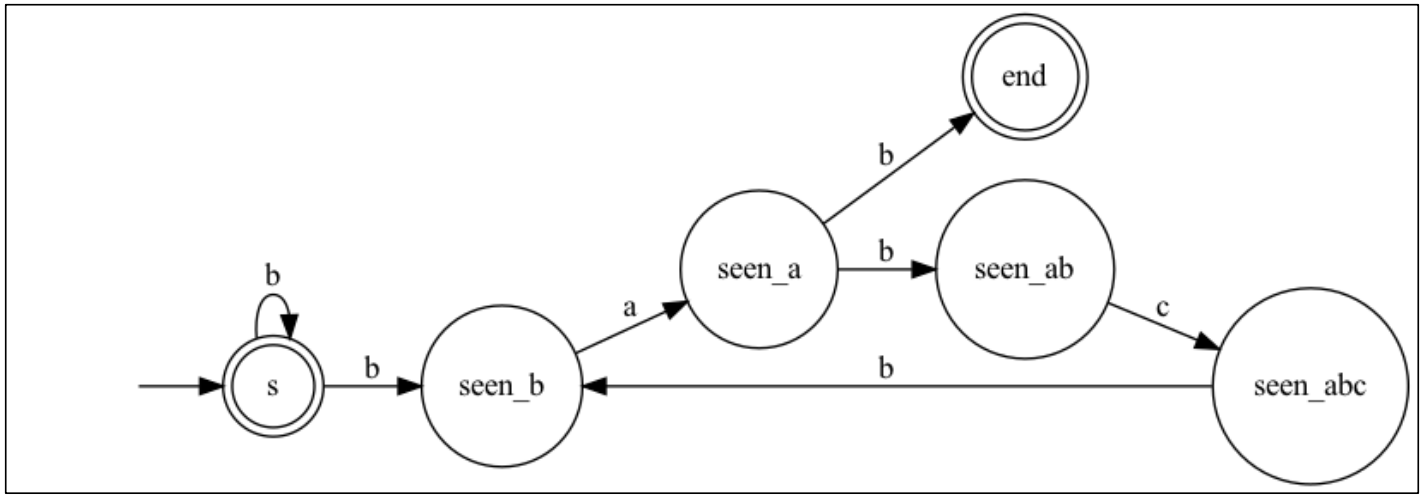
Homework 13 Review

- Question 4: enumerate the language `b(abcb)*ab|ε`:
  1. `[]`
  2. `bab`
  3. `babcbab`
  4. `babcbabcbab`
  5. `babcbabcbabcbab` *etc…*
  - i.e.

- Grammar:

```
1. s --> [].
2. s --> [b], seen_b.
3. s --> [b], s.
4. seen_b --> [a], seen_a.
5. seen_a --> [b].
6. seen_a --> [b], seen_ab.
7. seen_ab --> [c], seen_abc.
8. seen_abc --> [b], seen_b.
```

Homework 13 Review




```
1. s --> [].
2. s --> [b], seen_b.
3. s --> [b], s.
4. seen_b --> [a], seen_a.
5. seen_a --> [b].
6. seen_a --> [b], seen_ab.
7. seen_ab --> [c], seen_abc.
8. seen_abc --> [b], seen_b.
```

Breadth-first Search: enumeration is possible

```
?- [bab].
true.

?- [id_meta].
true.

?- id(s(List,[])).
List = [] ;
List = [b] ;
List = [b, a, b] ;
List = [b, b] ;
List = [b, a, b, b] ;
List = [b, b, a, b] ;
List = [b, b, b] ;
List = [b, a, b, a, b] ;
List = [b, a, b, b, b] ;
List = [b, b, a, b, b] ;
List = [b, b, b, a, b] ;
List = [b, b, b, b, a] ;
List = [b, b, b, b, b] ;
List = [b, a, b, a, b, a, b] ;
List = [b, a, b, a, b, b, b] ;
List = [b, a, b, b, a, b, b] ;
List = [b, a, b, b, b, a, b] ;
List = [b, a, b, b, b, b, a] ;
List = [b, a, b, b, b, b, b] ;
List = [b, b, a, b, b, b, b] ;
List = [b, b, a, b, a, b, b] ;
List = [b, b, a, b, b, a, b] ;
List = [b, b, a, b, b, b, b] ;
List = [b, b, b, a, b, a, b] ;
List = [b, b, b, a, b, b, b] ;
List = [b, b, b, b, a, b, b] ;
List = [b, b, b, b, b, a, b] ;
List = [b, b, b, b, b, b]
```

Iterative Deepening

- Suppose you have only the depth-first search strategy, e.g. Prolog, but you want to do breadth-first search.
- Let level n = the depth of the search. Then:
  - Do complete depth-first search to level 1
  - Do complete depth-first search to level 2
  - and so …
- This is the same as breadth-first search (*but less efficient*)

Breadth-first Search: enumeration is possible

- How many a's in a string of length N?
  - Suppose we have 3 a's in a string
  - Every a needs a b on either side, so the minimum length of the string has to be 7.
- **Formula**:
  - let #a be the number of a's.
  - minlen = #a * 3 – (#a – 1), for #a ≥ 1
  - minlen = #a * 2 + 1, for #a ≥ 1
  - Note that minlen must always be odd
  - Length is even: must be all b's or any odd string (of length one less) in the language + one additional b.

Beyond Regular Languages

- Beyond regular languages
  - $a^n b^n$ = {ab, aabb, aaabbb, aaaabbbb, ... } n≥1
  - is not a regular language
- That means no FSA, RE or RG can be built for this language

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, ... } n≥1
- A regular grammar extended to allow both left and right recursive rules can accept/generate it:
- File `anbn.prolog`

```
1. a --> [a], b.
2. b --> [b].
3. b --> a, [b].
```

- Example:

```
?- 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.
```

Set membership

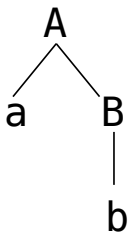
```
?- a(L,[]).
L = [a, b] ;
L = [a, a, b, b] ;
L = [a, a, a, b, b, b] ;
L = [a, a, a, a, b, b, b, b] ;
L = [a, a, a, a, a, b, b, b, b, b]
```

Set enumeration

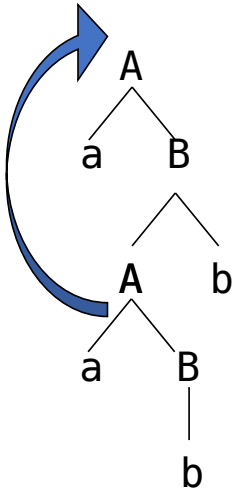
Beyond Regular Languages

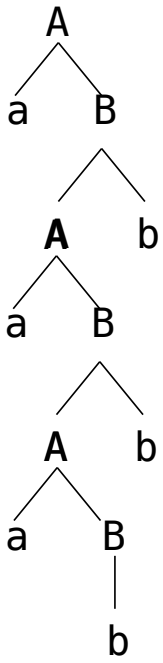
- Intuition:
  - grammar implements the stacking of partial trees balanced for a's and b's:

```
1. a --> [a], b.
2. b --> [b].
3. b --> a, [b].
```



A, B: nonterminals
a, b: terminals



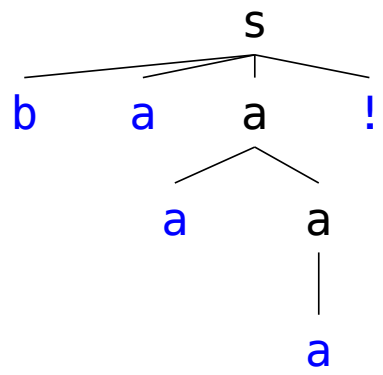


- A type-2 or context-free grammar (CFG) has no restrictions on what can go on the RHS of a grammar rule
- **Note**: *CFGs still have a single nonterminal limit for the LHS of a rule.*
- Example:

```
1. s --> [a], [b].
2. s --> [a], s, [b].
```

Parse Trees using Terms

- **Tree:**

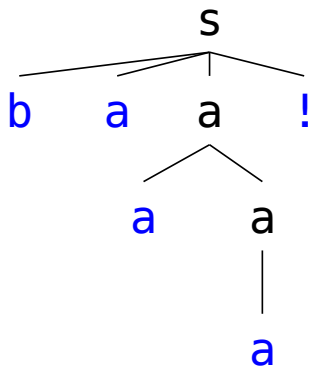


`s(b,a,a(a,a(a)),!)`

- Prolog term data structure:
  - `functor(arg`$_1$`,..,arg`$_n$`)`
  - each `arg`$_i$ could be another term or simple atom (symbol).
  - encodes hierarchy
  - allows a linear sequence of arguments

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 *incrementally* compute a Prolog term representation of the parse
  - by adding an extra argument to each nonterminal
  - applies to all rules (*not just regular grammars*)

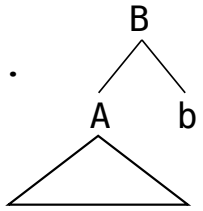


Extra Argument: Parse Tree

- **DCG**
  - `a --> [a], b.` (start symbol a)
  - `b --> [b].` (base case)
  - `b --> a, [b].` (left recursive case)

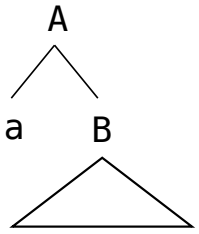
- **base case**
  - `b --> [b].`
  - `b(`*`subtree`*`) --> [b].`
  - `b(b(b)) --> [b].`

- **recursive case**
  - `b --> a, [b].`
  - `b(`*`subtree`*`) --> a(`*`subtree'`*`), [b].`
  - `b(b(A,b)) --> a(A), [b].`



- **Idea**:
  - for each nonterminal, add an argument to store its subtree

- **start symbol case**
  - `a --> [a], b.`
  - `a(`*`tree`*`) --> [a], b(`*`subtree'`*`).`
  - `a(a(a,B)) --> [a], b(B).`



Extra Argument: Parse Tree

- **Prolog grammar:**
  - anbn.prolog
  - `a --> [a], b.`
  - `b --> [b].`
  - `b --> a, [b].`
- **Equivalent Prolog grammar computing a parse**
  - anbn2.prolog
  - `a(a(a,B)) --> [a], b(B).`
  - `b(b(b)) --> [b].`
  - `b(b(A,b)) --> a(A), [b].`

Every nonterminal *x* has an extra argument *Term*: *x*(*Term*)

Extra Argument: Parse Tree

- anbn2.prolog
  1. `a(a(a,B)) --> [a], b(B).`
  2. `b(b(b)) --> [b].`
  3. `b(b(A,b)) --> a(A), [b].`
- To run the code, we also have to add the extra argument (for our parse) to the query.

```
?- [anbn2].
true.

?- a(Tree, [a,b], []).
Tree = a(a, b(b)) ;
false.

?- a(Tree, [a,a,b], []).
false.

?- a(Tree, [a,a,b,b], []).
Tree = a(a, b(a(a, b(b)), b)) ;
false.
```

Extra Arguments

- Extra arguments are powerful
  - they allow us to impose (grammatical) constraints and change the expressive power of the system
    - if used as read-able memory
    - but for computing a parse tree only: no.
- **Example**:
  - $a^n b^n c^n$ n>0 is not a context-free language (type-2)
  - *i.e. you cannot write rules of the form* X --> RHS *to generate this language*
  - in fact, it's context-sensitive (type-1)
  - but can use context-free rules plus an extra argument to constrain #a,b,c's.

Left Recursion and Set Enumeration

- Example (`lrrg.prolog`):

```
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*!
  - $ba^n!$ (# a's = n > 1)
- Sentential forms:
  - s
  - a!
  - baa!
  - baa!
  - baa!

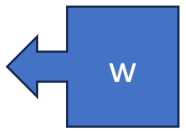
Underscores used here to indicate a nonterminal

Left Recursion and Set Enumeration

```
?- [lrrg].
true.

?- s(String,[]).
String = [b, a, a, !] ;
String = [b, a, a, a, !] ;
String = [b, a, a, a, a, !] ;
String = [b, a, a, a, a, a, !] ;
String = [b, a, a, a, a, a, a, !] ;
String = [b, a, a, a, a, a, a, a, !] ;
String = [b, a, a, a, a, a, a, a, a|...] ;
String = [b, a, a, a, a, a, a, a, a|...] [write]
String = [b, a, a, a, a, a, a, a, a, a, !] .
```

lrrg.prolog works for enumeration!



Left Recursion and Set Enumeration

- It also partially works for set membership:

```
?- s([b,a,a,!],[]).
true ;
ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 0.9Gb, global: 84.2Mb, trail: 0Kb
ERROR:   Stack depth: 11,029,028, last-call: 0%, Choice points: 4
ERROR:   Probable infinite recursion (cycle):
ERROR:     [11,029,026] user:a([length:4], _22083396)
ERROR:     [11,029,025] user:a([length:4], _22083422)
   Exception: (11,029,026) a([b, a, a, !], _22083324) ? abort
% Execution Aborted
?-
```

Left Recursion and Set Enumeration

- What happens if we present a string not in Sheeptalk!?

lrrg.prolog doesn't work as a decider: response should be yes/no.

```
?- s([b,a,a,b,a,a,!],[]).
ERROR: Stack limit (1.0Gb) exceeded
ERROR:   Stack sizes: local: 0.9Gb, global: 84.1Mb, trail: 0Kb
ERROR:   Stack depth: 11,028,563, last-call: 0%, Choice points: 4
ERROR:   Probable infinite recursion (cycle):
ERROR:     [11,028,561] user:a([length:7], _22059520)
ERROR:     [11,028,560] user:a([length:7], _22059546)
   Exception: (11,028,561) a([b, a, a, b, a, a, !], _22059448) ? abort
% Execution Aborted
```

Left Recursion and Set Enumeration

- left recursive regular grammar `lrrg.prolog`:

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

- Summary of Behavior:
  1. enumerates the language
  2. halts when presented with a string that is in the language
  3. doesn't halt when faced with a string not in the language
     - *unable to decide the language membership question*

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].
```

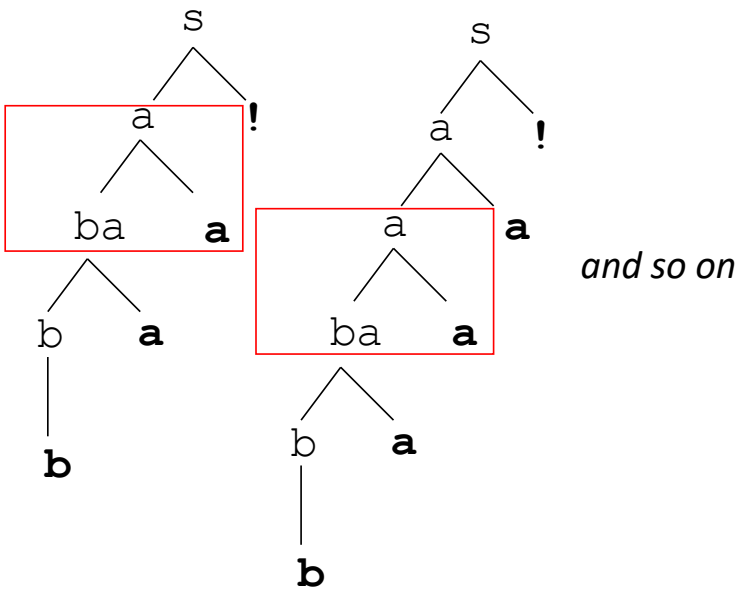
Choice point

- 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,[]).`

[Powerpoint animation]

L = [b,a,a,!]



and so on

Left Recursion and Set Enumeration

- However, this **slightly re-ordered** left recursive regular grammar:

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

- (*rules 2 and 3 swapped*) won't halt when enumerating

- **Why**?

[Powerpoint animation]