LING/C SC 581:

Advanced Computational Linguistics

Lecture 5

Today's Topics

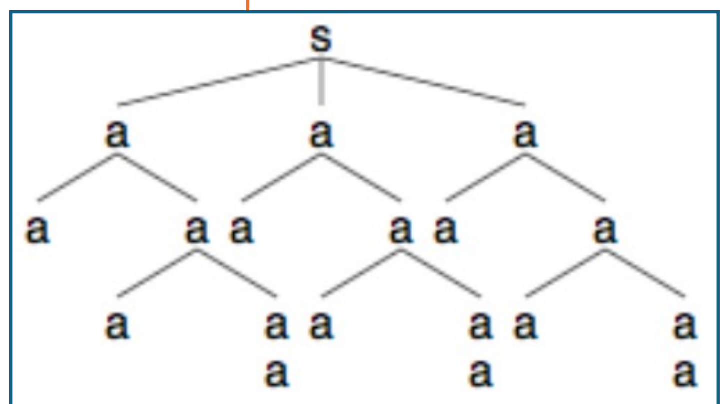
- Three ways to handle the CS language $\{a^n b^n c^n \mid n>0\}$
  1. CFG (context-free grammar) + extra arguments for grammatical constraints
  2. CFG + counting, cf. Perl
  3. CSG (context-sensitive grammar) rules
- Homework 3

Extra arguments

- `abc_parse.prolog`: a CFG+EA for $a^n b^n c^n$:

```
?- [abc_parse].
true.

?- s(Parse,[a,a,a,b,b,b,c,c,c],[]).
Parse = s(a(a, a(a, a(a))), a(a, a(a, a(a))), a(a, a(a, a(a)))) ;
false.

?- s(Parse,[a,a,a,b,b,b,c,c],[]).
false.

?- s(Parse,[a,a,a,b,b,c,c,c],[]).
false.

?- s(Parse,[a,a,b,b,b,c,c,c],[]).
false.

?- 
```

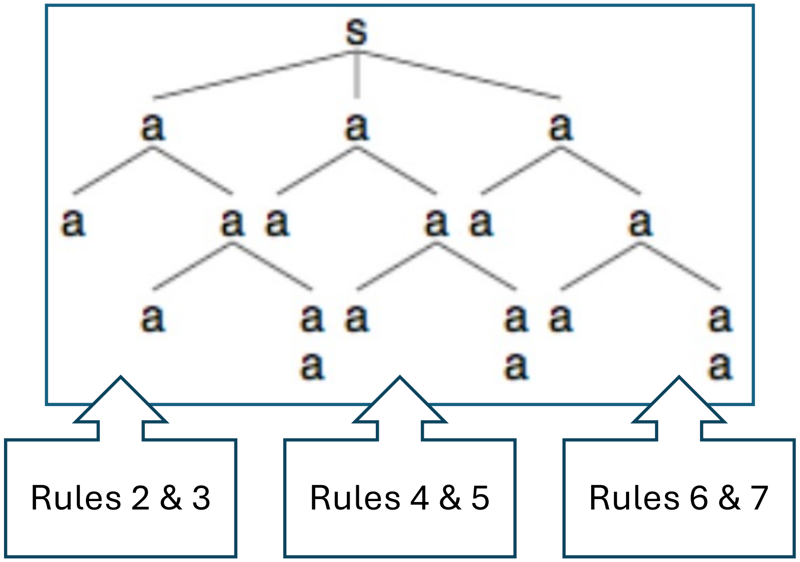
Set membership question



Extra arguments

- Consider the following context-free grammar (CFG) + an extra argument for each nonterminal` :

```
1.  s(s(A,A,A)) --> a(A), b(A), c(A).
2.  a(a(a)) --> [a].
3.  a(a(a,X)) --> [a], a(X).
4.  b(a(a)) --> [b].
5.  b(a(a,X)) --> [b], b(X).
6.  c(a(a)) --> [c].
7.  c(a(a,X)) --> [c], c(X).
```



Extra arguments

- A CFG+EA for $a^n b^n c^n$ n>0:

```
?- s(_,[a,a,b,b,c,c,c],[]).
false.

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

?- s(_,[a,a,b,b,c],[]).
false.

?- s(_,[a,a,b,b,c,c,c],[]).
false.

?- s(_,[a,a,a,b,b,b,c,c,c],[]).
true .
```

Set membership question

Note: underscore (_) means don't report (*the value of the parse term argument*)

Extra arguments

- A CFG+EA grammar for $a^n b^n c^n$ $n>0$:

Set enumeration

```
?- s(Parse,Sentence,[]).
Parse = s(a(a), a(a), a(a)),
Sentence = [a, b, c] ;
Parse = s(a(a, a(a)), a(a, a(a)), a(a, a(a))),
Sentence = [a, a, b, b, c, c] ;
Parse = s(a(a, a(a, a(a))), a(a, a(a, a(a))), a(a, a(a, a(a)))),
Sentence = [a, a, a, b, b, b, c, c, c] ;
Parse = s(a(a, a(a, a(a, a(a)))), a(a, a(a, a(a, a(a)))), a(a, a(a, a(a,
a(a))))),
Sentence = [a, a, a, a, b, b, b, b, c|...] [write]
Parse = s(a(a, a(a, a(a, a(a)))), a(a, a(a, a(a, a(a)))), a(a, a(a, a(a,
a(a))))),
```

Language $\{a^n b^n c^n \mid n>0\}$

1. CFG (context-free grammar) + extra arguments for grammatical constraints
2. CFG + counting, cf. Perl regex with code inserts
3. CSG (context-sensitive grammar) rules

Another grammar for $\{a^n b^n c^n | n>0\}$

- Use Prolog's arithmetic predicates directly.
- { … } embeds Prolog code inside grammar rules

-Number **is** *+Expr* *[ISO]*

True when *Number* is the value to which *Expr* evaluates. Typically, is/2 should be used with unbound left operand. If equality is to be tested, =:=/2 should be used. For example:

`?- 1 is sin(pi/2).` Fails! sin(pi/2) evaluates to the float 1.0, which does not unify with the integer 1.

`?- 1 =:= sin(pi/2).` Succeeds as expected.

```
?- X is 7*8.
X = 56.

?- Y = 2, X is 3+Y.
Y = 2,
X = 5.

?- X is 3+Y.
ERROR: is/2: Arguments are not sufficiently instantiated
?-
```

These are not nonterminal or terminal symbols. Used in grammar rules, we must enclose these statements within curly braces.
Recall (?{… Perl code …})

Another Grammar for $\{a^n b^n c^n | n>0\}$

- Explicit computation of the number of a's using arithmetic.
- { ... } embeds Prolog code inside grammar rules

```
s --> a(N), b(N), c(N).
a(1) --> [a].
a(N) --> [a], a(M), {N is M+1}.
b(1) --> [b].
b(N) --> [b], b(M), {N is M+1}.
c(1) --> [c].
c(N) --> [c], c(M), {N is M+1}.
```

value for M must be known for N to be computed

Illegal to write:

```
a(N) --> [a], {N is M+1}, a(M).
```

Another Grammar for $\{a^n b^n c^n | n>0\}$

```
[trace]  ?- s([a,a,b,b,c,c],[]).
   Call: (7) s([a, a, b, b, c, c], []) ?
   Call: (8) a(_G446, [a, a, b, b, c, c], _G448) ?
   Call: (9) a(_G446, [a, b, b, c, c], _G448) ?
   Call: (10) a(_G446, [b, b, c, c], _G448) ?
   Fail: (10) a(_G446, [b, b, c, c], _G448) ?
   Redo: (9) a(_G446, [a, b, b, c, c], _G448) ?
   Exit: (9) a(1, [a, b, b, c, c], [b, b, c, c]) ?
^  Call: (9) _G452 is 1+1 ?
^  Exit: (9) 2 is 1+1 ?
   Call: (9) _G452=[b, b, c, c] ?
   Exit: (9) [b, b, c, c]=[b, b, c, c] ?
   Exit: (8) a(2, [a, a, b, b, c, c], [b, b, c, c]) ?
```

Parsing the a's

Another Grammar for $\{a^n b^n c^n | n>0\}$

- Computing the b's

```
Call: (8) b(2, [b, b, c, c], _G451) ?
Call: (9) b(_G449, [b, c, c], _G451) ?
Call: (10) b(_G449, [c, c], _G451) ?
Fail: (10) b(_G449, [c, c], _G451) ?
Redo: (9) b(_G449, [b, c, c], _G451) ?
Exit: (9) b(1, [b, c, c], [c, c]) ?
Call: (9) 2 is 1+1 ?
Exit: (9) 2 is 1+1 ?
Call: (9) _G455=[c, c] ?
Exit: (9) [c, c]=[c, c] ?
Exit: (8) b(2, [b, b, c, c], [c, c]) ?
```

Another Grammar for $\{a^n b^n c^n | n>0\}$

- Computing the c's

```
Call: (8) c(2, [c, c], []) ?
Call: (9) c(_G452, [c], _G454) ?
Call: (10) c(_G452, [], _G454) ?
Fail: (10) c(_G452, [], _G454) ?
Redo: (9) c(_G452, [c], _G454) ?
Exit: (9) c(1, [c], []) ?
Call: (9) 2 is 1+1 ?
Exit: (9) 2 is 1+1 ?
Call: (9) []=[] ?
Exit: (9) []=[] ?
Exit: (8) c(2, [c, c], []) ?
Exit: (7) s([a, a, b, b, c, c], []) ?
```

Another grammar for $\{a^n b^n c^n \mid n>0\}$

- Grammar is “correct” – *it answers the set membership question*, but it's not so efficient...
  - consider string [a,a,b,b,b,b,b,b,b,c,c]

```
1. s --> a(X), b(X), c(X).
2. a(1) --> [a].
3. a(N) --> [a], a(M), {N is M+1}.
4. b(1) --> [b].
5. b(N) --> [b], b(M), {N is M+1}.
6. c(1) --> [c].
7. c(N) --> [c], c(M), {N is M+1}.
```

counts upwards

could change to count down from value computed by a(N).

Another grammar for $\{a^n b^n c^n \mid n>0\}$

- Comparison:

```
?- time(s([a,a,b,b,b,b,b,b,b,c,c],[])).
% 38 inferences, 0.000 CPU in 0.000 seconds (69% CPU,
603175 Lips)
false.

?- [abc_count2].
true.
?- time(s([a,a,b,b,b,b,b,b,b,c,c],[])).
% 9 inferences, 0.000 CPU in 0.000 seconds (71% CPU,
145161 Lips)
false.
```

down counter
for b's and c's

Predicate time/1

arity 1
(*takes one argument*)

- Documentation:
  - https://www.swi-prolog.org/pldoc/man?predicate=time/1

Availability: `:- use_module(library(statistics)).` *(can be autoloaded)*

time(*:Goal*) *[nondet]*

Execute *Goal*, reporting statistics to the user. If *Goal* succeeds non-deterministically, retrying reports the statistics for providing the next answer.

Statistics are retrieved using thread_statistics/3 on the calling thread. Note that not all systems support thread-specific CPU time. Notable, this is lacking on MacOS X.

See also

- statistics/2 for obtaining statistics in your program and understanding the reported values.
- call_time/2, call_time/3 to obtain the timing in a dict.

bug

Inference statistics are often a few off.

Language $\{a^n b^n c^n | n>0\}$

1. CFG (context-free grammar) + extra arguments for grammatical constraints
2. CFG + counting, cf. Perl
3. CSG (context-sensitive grammar) rules

A context-sensitive grammar for $\{a^n b^n c^n \mid n>0\}$

Non-contracting grammar definition

- A CSG (type-1) has rules of the form LHS → RHS
  - such that both LHS and RHS can be arbitrary strings of terminals and non-terminals, and
  - |RHS| ≥ |LHS| (*without this restriction*: type-0)
    - **Notation**:
      - |*symbols*| = # symbols
    - **exception**: permit S → ε, in Prolog s `-->  [].`, *assuming S doesn't appear on the RHS of any rule*

A context-sensitive grammar for $\{a^n b^n c^n | n>0\}$

Context-sensitive definition

- Consider a context-free rule of the form N → *γ*
  - N a single nonterminal
  - *γ* a nonempty string of terminals and nonterminals
- Then a CSG rule has the form *α*N*β* → *αγβ*
  - *α, β* are strings of terminals and nonterminals (possibly empty)
  - **exception**: S → ε (*assuming S doesn't appear on the RHS of any rule)*

A context-sensitive grammar for {$a^n b^n c^n$ | n>0}

- SWI Prolog permits some quirky extensions to the DCG rules:
  - General format: LHS --> RHS.
  - LHS must begin with a nonterminal.
  - Cannot have a CS rule like `[a], a --> [a].` *still sufficient to encode a lot.*
  - Rest of LHS could be anything…
- Examples:

```
• s --> a, b.
• a, b --> [c].
• a --> [a].
• a --> [a], a.
• b --> [b].
```

```
• s --> a, b.
• a, b --> [c].
• a, [a], b --> [d].
• a --> [a].
• a --> a, [a].
• b --> [b].
```

```
• s --> a, b.
• a --> [a].
• b --> [b].
• [a], b --> [c].
```

ERROR: No permission to define dcg_nonterminal `[a]'

A context-sensitive grammar for $\{a^n b^n c^n | n>0\}$

- This is *almost* a normal Prolog DCG (`abc_cs.prolog`):
  - (*but rules 5 & 6 have more than only a single non-terminal on the LHS, ∴ not context-free*):

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

- *satisfies noncontracting constraint*
- ***Note**: rules 5 and 6 are responsible for shuffling the c's to the end*

A context-sensitive grammar for $\{a^n b^n c^n | n>0\}$

- Case: n = 1

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

- Rule 1 suffices.

A context-sensitive grammar for $\{a^n b^n c^n | n>0\}$

- Case: n = 2

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

Note: list notation
- [a,b,c] is short for [a],[b],[c]
- [b,c] is short for [b], [c]
  etc.

- Sentential forms:
  - (expanding items in red)

1. s
2. [a], a, [b,c] (rule 2)
3. [a], [a,b], c, [b,c] (rule 3)
4. [a,a,b], c, [b],[c] (list notation)
5. [a,a,b], [b], c, [c] (rule 5)
6. [a,a,b], [b], [c,c] (rule 6)
7. [a,a,b,b,c,c] (list notation)

A context-sensitive grammar for $\{a^n b^n c^n | n>0\}$

- Case: n = 3

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

Note: list notation

- [a,b,c] is short for [a],[b],[c]
- [b,c] is short for [b], [c]

etc.

1.	s	
2.	[a], a, [b,c]	(rule 2)
3.	[a], [a,b], c, [b,c]	(rule 3)
3.	[a], [a],a,[b],c, [b,c]	(rule 4)
4.	[a,a], [a,b],c, [b], c,[b,c]	(rule 3)
5.	[a,a], [a,b],[b],c, c,[b,c]	(rule 5)
6.	[a,a,a,b,b],c, [b],c, [c]	(rule 5)
7.	[a,a,a,b,b], [b],c, c, [c]	(rule 5)
8.	[a,a,a,b,b], [b],c, [c,c]	(rule 6)
9.	[a,a,a,b,b], [b],[c,c],[c]	(rule 6)
10.	[a,a,a,b,b,b,c,c,c]	

A context-sensitive grammar for $\{a^n b^n c^n | n>0\}$

```
?- listing([s,a,c]).
1.  s([a, b, c|A], A).
2.  s([a|A], C) :- a(A, B), B=[b, c|C].
3.  a([a, b|A], B) :- c(A, B).
4.  a([a|A], D) :- a(A, B), B=[b|C], c(C, D).
5.  c(A, C) :- A=[b|B], c(B, D), C=[b|D].
6.  c([c, c|A], [c|A]).
```

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

Difference lists

`s(List1, List2)`

`?-s([a,b,c],[])`

List1	List2		Difference
[1,2,3]	[3]	=>	[1,2]
[1,2,3,4,5]	[5]	=>	[1,2,3,4]
[1,2,3,4,5]	[4,5]	=>	[1,2,3]

A context-sensitive grammar for $\{a^n b^n c^n | n>0\}$

- `c,[c] --> [c,c].` ⟹ `c([c, c|A], [c|A]).`
- Grammar rule says:
  - *nonterminal c gets expanded into terminal c when the nonterminal c is followed by a terminal c*
  - cf. context-free counterpart `c --> [c].`
- Prolog code says:
  - nonterminal c expands into terminal c
  - **Input**: `[c,c, …]` (green is right context, but not part of
  - **Output**: `[c, …]` nonterminal c expansion)

A context-sensitive grammar for $\{a^n b^n c^n | n>0\}$

- `c, [b] --> [b], c.` ⇒ `c(A, C) :- A=[b|B], c(B, D), C=[b|D].`
- Grammar rule says:
  - *flip order of nonterminal c and terminal b*
- Prolog code says:
  - A, B, C and D are lists

			Example:	**Example:**
Input:	`[b,…c…, …]`	(A)	`[b,c,c]`	`[b,b,c,c,c]`
Call c:	`[…c…, …]`	(B)	`[c,c]`	`[b,c,c,c]`
Exit c:	`[ …]`	(D)	`[c]`	`[b,c,c]`
Output:	`[b,…]`	(C)	`[b,c]`	`[b,b,c,c]`

list A with
sub-lists at B and D

↑	b	↑	… c …	↑	*… rest of string …*
A		B		D	

Homework 3

- Consider the DCG counting grammar described in class for $\{a^n b^n c^n \mid n>0\}$, i.e. the one with arithmetic rules such as:
  - `a(1) --> [a].`
  - `a(N) --> [a], a(M), {N is M+1}.`
- Question 1:
  - This grammar counts "up", i.e. the more *a*'s seen, the higher the count.
  - Suggested in class that a *more efficient* grammar (measured using time/1) could be built to reject strings not in the grammar by counting down the b's and c's.
  - Give this grammar.
  - Show your grammar working and compared to `abc_count.prolog` for inputs:
    - `[a,a,b,b,b,b,b,b,b,b,c,c]` (8 *b*'s)
    - `[a,a,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,b,c,c]` (16 *b*'s)

Homework 3

a grammar rule could contain true/false expressions like `{X >= 0}`

- https://www.swi-prolog.org/pldoc/man?section=arith

4.27.2 General purpose arithmetic

The general arithmetic predicates are optionally compiled (see set_prolog_flag/2 and the **-O** command line option). Compiled arithmetic reduces global stack requirements and improves performance. Unfortunately compiled arithmetic cannot be traced, which is why it is optional.

+Expr1 > +Expr2 *[ISO]*

True if expression *Expr1* evaluates to a larger number than *Expr2*.

+Expr1 < +Expr2 *[ISO]*

True if expression *Expr1* evaluates to a smaller number than *Expr2*.

+Expr1 =< +Expr2 *[ISO]*

True if expression *Expr1* evaluates to a smaller or equal number to *Expr2*.

+Expr1 >= +Expr2 *[ISO]*

True if expression *Expr1* evaluates to a larger or equal number to *Expr2*.

Homework 3

a grammar rule could contain true/false expressions like `{X is Y + Z}`
Note: `{X is X + 1}` is always false, cf. Python `X = X + 1, X += 1`

```
?- X = 1, X is X+1.
false.
```

-Number is +Expr *[ISO]*

True when *Number* is the value to which *Expr* evaluates. Typically, is/2 should be used with unbound left operand. If equality is to be tested, =:=/2 should be used. For example:

`?- 1 is sin(pi/2).`	Fails! sin(pi/2) evaluates to the float 1.0, which does not unify with the integer 1.
`?- 1 =:= sin(pi/2).`	Succeeds as expected.

Homework 3

- Question 2:
  - Give a counting DCG for $\{a^n b^{2n} c^{n+1} \mid n>0\}$
  - It should accept inputs such as:
    - abbcc
    - aabbbbccc
    - aaabbbbbbcccc
  - but reject inputs such as:
    - aabbcc
    - aabbbccc
    - aaabbbbbbccc
  - Show your grammar working

Homework 3

- Submit to sandiway@arizona.edu
- SUBJECT: 581 Homework 3 *YOUR NAME*
- One PDF file (for grading)
  - include your grammar code and SWI-Prolog screenshots in your answer
- Attachments (if I need to run your code):
  - source code for the two grammars
- Deadline:
  - midnight Monday
  - we will review the homework on Tuesday