LING/C SC 581:

Advanced Computational Linguistics

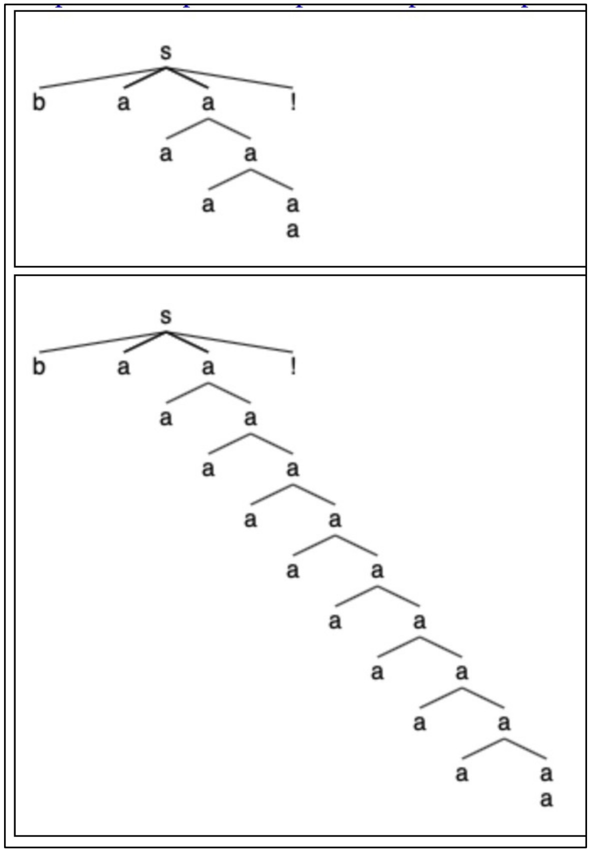
Lecture 24

Last Time

- Prolog grammar rule re-cap.
  1. is a string a member of the language (*generated by the grammar*)? Give me a parse.
  2. what are the strings (*and associated parses*) of the language?
- **Example** (*each nonterminal with a parse tree argument*):

```
s(s(b,a,A,!)) --> [b,a], a(A), [!].
a(a(a)) --> [a].
a(a(a,A)) --> [a], a(A).
```

```
1. s(Tree, [b,a,a,a,a,!], []).
2. s(Tree, String, []).
```



Language $\{a^n b^n c^n | n>0\}$ is not context-free

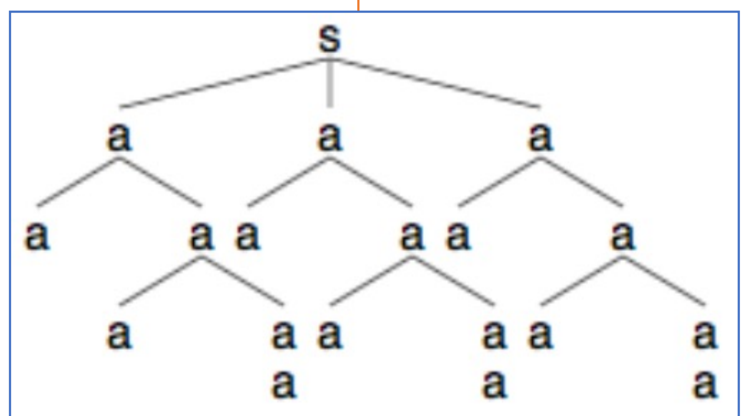
- But we can still write grammars for it:
  1. CFG (context-free grammar) + extra arguments for grammatical constraints
  2. CFG + counting, cf. Perl
  3. CSG (context-sensitive grammar) rules

Extra arguments

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

Set membership question

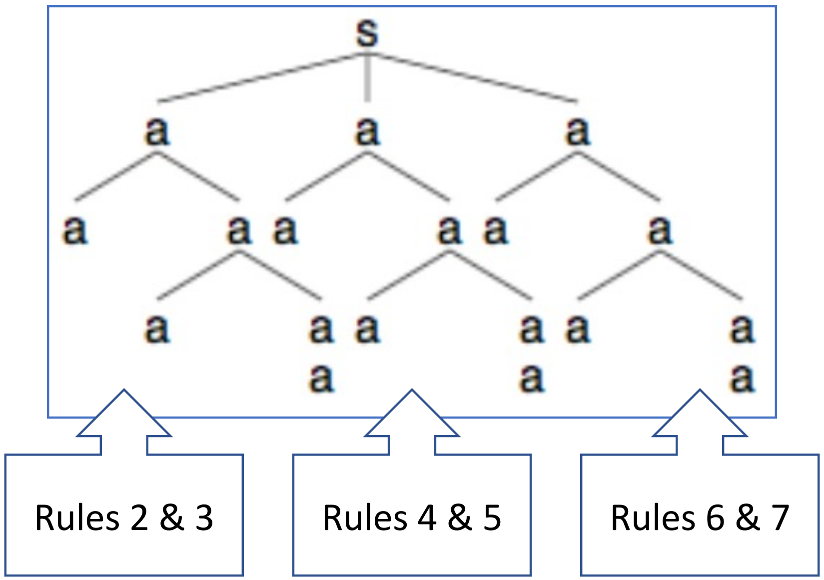
```
[?- [abc_parse].
true.

[?- 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.

?-
```



Extra arguments

- A context-free grammar (CFG) + extra argument (EA) for the context-sensitive language { $a^n b^n c^n$ | n>0}:

```
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]. % cf. b(b)
5. b(a(a,X)) --> [b], b(X).
6. c(a(a)) --> [c]. % cf. c(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

Extra arguments

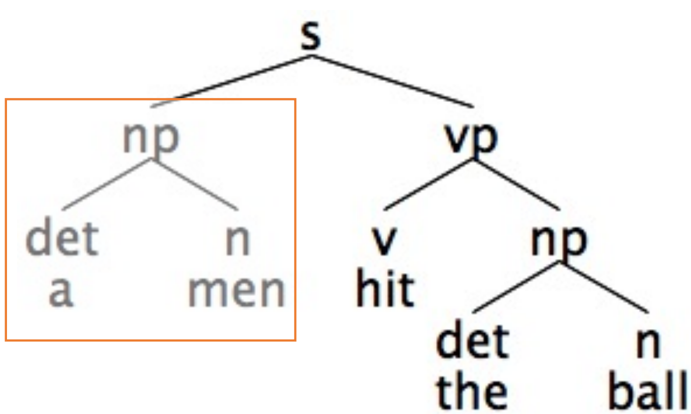
- A CFG+EA grammar for $a^n b^n c^n$ n>0:

language 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))))),
```

Extra Arguments: Agreement

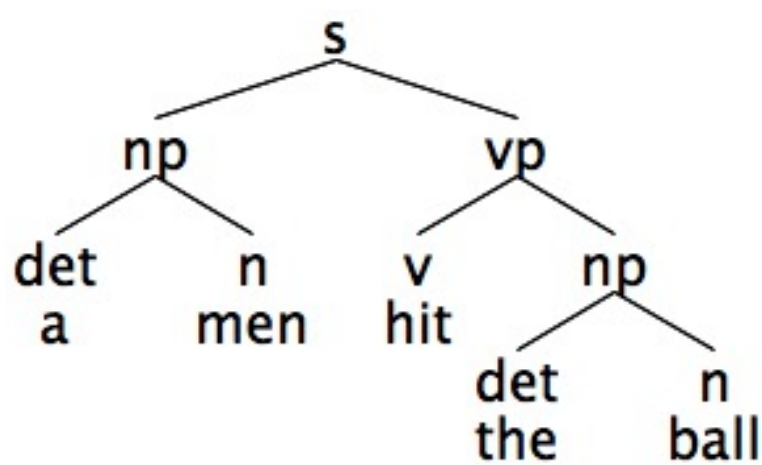
- **Idea**:
  - We can also use an extra argument to impose constraints between constituents within a DCG rule



- **Example**:
  - English determiner-noun number agreement
  - Data:
    - the man
    - the men
    - a man
    - *a men
  - Lexical Features (Number):
    - *man* valued singular (`sg`)
    - *men* valued plural (`pl`)
    - *the* valued (`sg/pl`)
    - *a* valued singular (`sg`)

Extra Arguments: Agreement

```
np(np(D,N)) --> det(D,Number), common_noun(N,Number).
det(dt(the),sg) --> [the].
det(dt(the),pl) --> [the].
det(dt(a),sg) --> [a].
common_noun(nn(ball),sg) --> [ball].
common_noun(nn(man),sg) --> [man].
common_noun(nns(men),pl) --> [men].
```



Extra Arguments: Agreement

Note:

- Use of the extra argument for agreement here is basically "syntactic sugar" and **lends no more expressive** power to the grammar rule system
- i.e. *we can enforce the agreement without the use of the extra argument at the cost of more rules*

- Instead of

```
np(np(D,N)) --> det(D,Number),
  common_noun(N,Number).
```

we could have written:

```
np(np(D,N)) --> detsg(D), common_nounsg(N).
np(np(D,N)) --> detpl(D), common_nounpl(N).
detsg(dt(a)) --> [a].
detsg(dt(the)) --> [the].
detpl(dt(the)) --> [the].
common_nounsg(nn(ball)) --> [ball].
common_nounsg(nn(man)) -->[man].
common_nounpl(nn(men)) --> [men].
```

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

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

- Use Prolog's arithmetic predicates.
- { … } 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}.
```

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

- Grammar is "correct" but 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

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

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
  - $|RHS| \geq |LHS|$ (otherwise type-0)
    - **Notation**: |*symbols*| = # symbols
    - (**exception**: S → ε, S not in RHS)

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

Context-sensitive definition

- Consider a context-free rule of the form $N \rightarrow \gamma$
  - N a single nonterminal
  - γ a nonempty string of terminals and nonterminals
- Then a CSG rule has the form $\alpha N \beta \rightarrow \alpha \gamma \beta$
  - α, β are strings of terminals and nonterminals (possibly empty)
  - (**exception**: $S \rightarrow \varepsilon$, S not in RHS)

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 rule like, e.g. [a], a --> [a].
  - 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 $\{a^n b^n c^n \mid n>0\}$

- 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:

			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]

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

Dotted Rules

- Dot (●) indicates where we are in a grammar rule
- Examples:
  - S -> ● NP VP [the, man, saw, the, dog]
  - S -> NP ● VP [saw, the, dog]
  - S -> NP VP ● []

  - VP -> ● V NP [saw, the, dog]
  - VP -> V ● NP [the, dog]
  - VP -> V NP ● []

  - NP -> ● DT NN [the, man, saw, the, dog]
  - NP -> DT ● NN [man, saw, the, dog]
  - NP -> DT NN ● [saw, the, dog]

Dotted Rules

- Dot (●) can also indicate where we are in the grammar
- S -> ● NP VP ; NP -> ● DT NN [the, man, saw, the, dog]
- S -> ● NP VP ; NP -> DT ● NN [man, saw, the, dog]
- S -> NP ● VP ; NP -> DT NN ● [saw, the, dog]

- S -> NP ● VP ; VP -> ● V NP [saw, the, dog]
- S -> NP ● VP ; VP -> V ● NP [the, dog]
- S -> NP ● VP ; VP -> V ● NP ; NP -> ● DT NN [the, dog]
- S -> NP ● VP ; VP -> V ● NP ; NP -> DT ● NN [dog]
- S -> NP VP ● ; VP -> V NP ● ; NP -> DT NN ● []

Used in various parsing algorithms:

- Earley Algorithm (in textbook)
- LR Algorithm (in this course)

Dotted rule: n = 2 [a,a,b,b,c,c]

- Derivation:
  - S -> abc; S -> aAbc; A -> abC; A -> aAbC; Cb -> bC; Cc -> cc

1.	S -> ●aAbc	[a,a,b,b,c,c]
2.	S -> a●Abc; A -> ●abC	[a,b,b,c,c]
3.	S -> a●Abc; A -> a●AbC ; A -> ab●C; Cb -> ●bC	[b,c,c]
4.	S -> a●Abc; A -> a●AbC ; A -> ab●C; Cb -> b●C ; Cc -> ●cc	[c,c]
5.	S -> a●Abc; A -> a●AbC ; A -> ab●C; Cb -> b●C ; Cc -> cc●	[c]
6.	S -> a●Abc; A -> a●AbC ; A -> ab●C; Cb -> bC●	[b,c]
7.	S -> a●Abc; A -> a●AbC ; A -> abC●	[b,c]
8.	S -> a●Abc; A -> aA●bC	[b,c]
9.	S -> a●Abc; A -> aAb●C ; Cc -> ●cc	[c,c]
10.	S -> a●Abc; A -> aAb●C ; Cb -> b●C ; Cc -> cc●	[c]
11.	S -> a●Abc; A -> aAb●C ; Cb -> bC●	[b,c]
12.	S -> a●Abc; A -> aAbC●	[b,c]
13.	S -> aA●bc	[b,c]
14.	S -> aAbc●	[]

- Grammar:

```
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].
```

Trace: n = 2 [a,a,b,b,c,c]

s spans [a,a,b,b,c,c] leaving [] afterwards

dot (●) indicates our current position

1.	**Call:** (10) s([a, a, b, b, c, c], []) ?	rule 2: s → ●[a],a,[b,c]
		rule 2: s → [a] ● a,[b,c]
2.	**Call:** (11) a([a, b, b, c, c], _10600) ?	rule 3: a → ● [a,b], c
		rule 3: a → [a,b] ● c
3.	**Call:** (12) c([b, c, c], _10644) ?	rule 5: c,[b] → ● [b], c
		rule 5: c,[b] → [b] ● c
4.	**Call:** (13) c([c, c], _10782) ?	rule 6: c,[c] → ● [c,c]
5.	**Exit:** (13) c([c, c], [c])	rule 6: c,[c] → [c,c] ●
6.	**Exit:** (12) c([b, c, c], [b, c])	rule 5: c,[b] → [b], c ●
7.	**Exit:** (11) a([a, b, b, c, c], [b, c])	rule 3: a → [a,b], c ●
		rule 2: s → [a],a ●[b,c]
8.	**Exit:** (10) s([a, a, b, b, c, c], [])	rule 2: s → [a],a [b,c] ●

- Grammar:

```
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].
```

Dotted rule: n = 3 [a,a,a,b,b,b,b,c,c]

- Derivation:
  - S -> abc; S -> aAbc; A -> abC; A -> aAbC; Cb -> bC; Cc -> cc

1.	S -> ●aAbc	[a,a,a,b,b,b,c,c,c]
2.	S -> a●Abc; A -> ●aAbC	[a,a,b,b,b,c,c,c]
3.	S -> a●Abc; A -> a●AbC; A -> ●abC	[a,b,b,b,c,c,c]
4.	S -> a●Abc; A -> a●AbC ; A -> ab●C; Cb -> ●bC	[b,b,c,c,c]
5.	S -> a●Abc; A -> a●AbC ; A -> ab●C; Cb -> b●C ; Cb -> ●bC	[b,c,c,c]
6.	S -> a●Abc; A -> a●AbC ; A -> ab●C; Cb -> b●C ; Cb -> b●C ; Cc -> ●cc	[c,c,c]
7.	S -> a●Abc; A -> a●AbC ; A -> ab●C; Cb -> b●C ; Cb -> b●C ; Cc -> cc●	[c,c]
8.	S -> a●Abc; A -> a●AbC ; A -> ab●C; Cb -> b●C ; Cb -> bC●	[b,c,c]
9.	S -> a●Abc; A -> a●AbC ; A -> ab●C; Cb -> bC●	[b,b,c,c]
10.	S -> a●Abc; A -> a●AbC ; A -> abC●	[b,b,c,c]
11.	S -> a●Abc; A -> aA●bC	[b,b,c,c]
12.	S -> a●Abc; A -> aAb●C ; Cb -> ●bC	[b,c,c]
13.	S -> a●Abc; A -> aAb●C ; Cb -> b●C ; Cc -> ●cc	[c,c]
14.	S -> a●Abc; A -> aAb●C ; Cb -> b●C ; Cc -> cc●	[c]
15.	S -> a●Abc; A -> aAb●C ; Cb -> bC●	[b,c]
16.	S -> a●Abc; A -> aAbC●	[b,c]
17.	S -> aA●bc	[b,c]
18.	S -> aAbc●	[]

- Grammar:

```
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].
```

Trace: n = 3   [a,a,a,b,b,b,b,c,c]

1.	**Call:** (10) s([a, a, a, b, b, b, c, c, c], []) ?	rule 2: s → ●[a],a,[b,c]
2.	**Call:** (11) a([a, a, b, b, b, c, c, c], _13226) ?	rule 4: a → ●[a],a,[b],c
3.	**Call:** (12) a([a, b, b, b, c, c, c], _13270) ?	rule 3: a → ● [a,b], c
4.	**Call:** (13) c([b, b, c, c, c], _13314) ?	rule 5: c,[b] → ● [b], c
5.	**Call:** (14) c([b, c, c, c], _13452) ?	rule 5: c,[b] → ● [b], c
6.	**Call:** (15) c([c, c, c], _13590) ?	rule 6: c,[c] → ● [c,c]
7.	**Exit:** (15) c([c, c, c], [c, c])	rule 6: c,[c] → [c,c] ●
8.	**Exit:** (14) c([b, c, c, c], [b, c, c])	rule 5: c,[b] → [b], c ●
9.	**Exit:** (13) c([b, b, c, c, c], [b, b, c, c])	rule 5: c,[b] → [b], c ●
10.	**Exit:** (12) a([a, b, b, b, c, c, c], [b, b, c, c])	rule 3: a → [a,b], c ● ; rule 4: a → [a],a ●[b], c
11.	**Call:** (12) c([b, c, c], _14236) ?	rule 5: c,[b] → ● [b], c
12.	**Call:** (13) c([c, c], _14374) ?	rule 6: c,[c] → ● [c,c]
13.	**Exit:** (13) c([c, c], [c])	rule 6: c,[c] → [c,c] ●
14.	**Exit:** (12) c([b, c, c], [b, c])	rule 5: c,[b] → [b], c ●
15.	**Exit:** (11) a([a, a, b, b, b, c, c, c], [b, c])	rule 4: a → [a],a [b], c ● ; rule 2: s → [a],a ●[b,c]
16.	**Exit:** (10) s([a, a, a, b, b, b, c, c, c], []) ?	rule 2: s → [a],a [b,c] ●

- Grammar:

```
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].
```