LING/C SC 581:
Advanced Computational Linguistics

Lecture 12

Prof. Sandiway Fong

Today's Topics

- CKY Parsing algorithm
  - Dynamic Programming
  - Chomsky Normal Form (CNF)

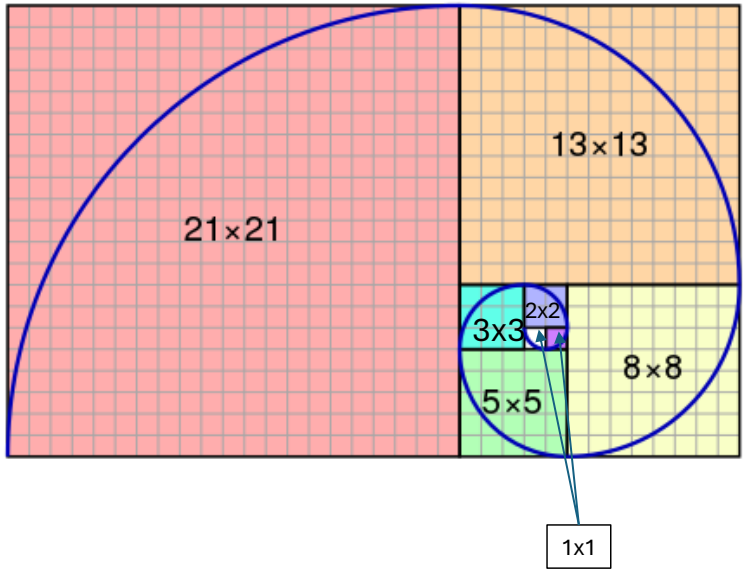
Context-Free Grammar Parsing

- Algorithm:
  - use Prolog's top-down, left-to-right depth-first search
  - *avoid left recursion problems: our grammar transformation*
- There are lots of algorithms for parsing context-free rules
  - In this course, we'll examine two
  - Today: CKY – because of the 2D table (*it's used for tabular parsing*)
  - the 2D table is computationally efficient (*memorization of partial results*)
    - particularly useful when there is structural ambiguity
    - e.g. … *with a telescope*

Memorization aka Dynamic Programming

- A general computational trick (also used in math)
- Example:
  - 0,1,1,2,3,5,8,13,21,34,55,89,…
  - Fibonacci sequence
    - f(0)=0,
    - f(1)=1,
    - f(n)=f(n-1)+f(n-2) for n>1
- Example:
  - f(5) = f(4) + f(3)
  - = f(3) + f(2) + f(3)
  - = 2 + 1 + 2

Fibonacci Spiral: adapted from wikipedia



Memorization aka Dynamic Programming

- Not a standard feature of Prolog
  - Program: `fibonacci.prolog` implements
  - Fibonacci sequence
    - f(0)=0,
    - f(1)=1,
    - f(n)=f(n-1)+f(n-2) for n>1
  - *no {...} needed here, it's not a grammar, i.e. doesn't use –-->*

```
%% f(N, M) M is Fibonacci N
f(0, 0) :- !.                    % cut off other choices
f(1, 1) :- !.
f(N, M) :- N > 1, N1 is N-1, f(N1,M1), N2 is N-2, f(N2,M2), M is M1+M2.
```

Memorization aka Dynamic Programming

- Correct but inefficient:

```
$ swipl -l fibonacci.prolog

?- f(0,N).
N = 0.

?- f(1,N).
N = 1.

?- f(5,N).
N = 5.

?- f(6,N).
N = 8.

?- f(7,N).
N = 13.

?- f(8,N).
N = 21.
```

Memorization aka Dynamic Programming

```
?- spy(f).
% Spy point on f/2
true.

[debug]  ?- f(5, N).
   Call: (10) f(5, _34086) ? leap
   Call: (11) f(4, _35386) ? leap
   Call: (12) f(3, _36288) ? leap
   Call: (13) f(2, _37190) ? leap
   Call: (14) f(1, _38092) ? leap
   Exit: (14) f(1, 1) ? leap
   Call: (14) f(0, _39886) ? leap
   Exit: (14) f(0, 0) ? leap
   Exit: (13) f(2, 1) ? leap
   Call: (13) f(1, _42578) ? leap
   Exit: (13) f(1, 1) ? leap
   Exit: (12) f(3, 2) ? leap
   Call: (12) f(2, _45270) ? leap
   Call: (13) f(1, _46172) ? leap
   Exit: (13) f(1, 1) ? leap
   Call: (13) f(0, _47966) ? leap
   Exit: (13) f(0, 0) ? leap
   Exit: (12) f(2, 1) ? leap
   Exit: (11) f(4, 3) ? leap
   Call: (11) f(3, _51556) ? leap
   Call: (12) f(2, _52458) ? leap
   Call: (13) f(1, _53360) ? leap
   Exit: (13) f(1, 1) ? leap
   Call: (13) f(0, _55154) ? leap
   Exit: (13) f(0, 0) ? leap
   Exit: (12) f(2, 1) ? leap
   Call: (12) f(1, _57846) ? leap
   Exit: (12) f(1, 1) ? leap
   Exit: (11) f(3, 2) ? leap
   Exit: (10) f(5, 5) ? leap
N = 5 ;
   Fail: (12) f(1, _57846) ? leap
   Fail: (13) f(0, _55154) ? leap
   Fail: (13) f(1, _53360) ? leap
   Fail: (12) f(2, _52458) ? leap
   Fail: (11) f(3, _51556) ? leap
   Fail: (13) f(0, _47966) ? leap
   Fail: (13) f(1, _46172) ? leap
   Fail: (14) f(0, _39886) ? leap
   Fail: (14) f(1, _38092) ? leap
   Fail: (13) f(2, _37190) ? leap
   Fail: (12) f(3, _36288) ? leap
   Fail: (11) f(4, _35386) ? leap
   Fail: (10) f(5, _34086) ? leap
false.
```

type l for leap

Computes f(2) = 1 three times!

Memorization aka Dynamic Programming

- Using `time/1` to measure inferences:

```
?- time(f(10,N)).
% 352 inferences, 0.000 CPU in 0.000 seconds (83% CPU, 5587302 Lips)
N = 55.

?- time(f(20,N)).
% 43,779 inferences, 0.008 CPU in 0.010 seconds (81% CPU, 5229216 Lips)
N = 6765.

?- time(f(30,N)).
% 5,385,071 inferences, 0.389 CPU in 0.466 seconds (84% CPU, 13836505 Lips)
N = 832040.

?- time(f(40,N)).
% 662,320,559 inferences, 45.644 CPU in 54.645 seconds (84% CPU, 14510661 Lips)
N = 102334155.
```

Memorization aka Dynamic Programming

- How can we improve its efficiency?
  - *Computes f(2) = 1 three times!*
- Memorize result in a table first time it's computed
  - and use the answer when asked again
- Class Exercise:
  - let's compute using a table!

Memorization aka Dynamic Programming

Naïve algorithm

```
?- time(f(10,N)).
% 352 inferences, 0.000 CPU in 0.000 seconds (83% CPU, 5587302 Lips)
N = 55.

?- time(f(20,N)).
% 43,779 inferences, 0.008 CPU in 0.010 seconds (81% CPU, 5229216 Lips)
N = 6765.

?- time(f(30,N)).
% 5,385,071 inferences, 0.389 CPU in 0.466 seconds (84% CPU, 13836505
Lips)
N = 832040.

?- time(f(40,N)).
% 662,320,559 inferences, 45.644 CPU in 54.645 seconds (84% CPU,
14510661 Lips)
N = 102334155.
```

Using a table

```
?- time(f(10,N)).
% 108 inferences, 0.000 CPU in 0.000 seconds (88% CPU, 3085714 Lips)
N = 55.
?- ^D
?- time(f(20,N)).
% 228 inferences, 0.000 CPU in 0.000 seconds (87% CPU, 3864407 Lips)
N = 6765.
?- ^D
?- time(f(30,N)).
% 348 inferences, 0.000 CPU in 0.000 seconds (85% CPU, 4350000 Lips)
N = 832040.
?- ^D

?- time(f(40,N)).
% 468 inferences, 0.000 CPU in 0.000 seconds (88% CPU, 4500000 Lips)
N = 102334155.
```

CKY Algorithm

CKY = Cocke-Kasami-Younger, sometimes **CYK**

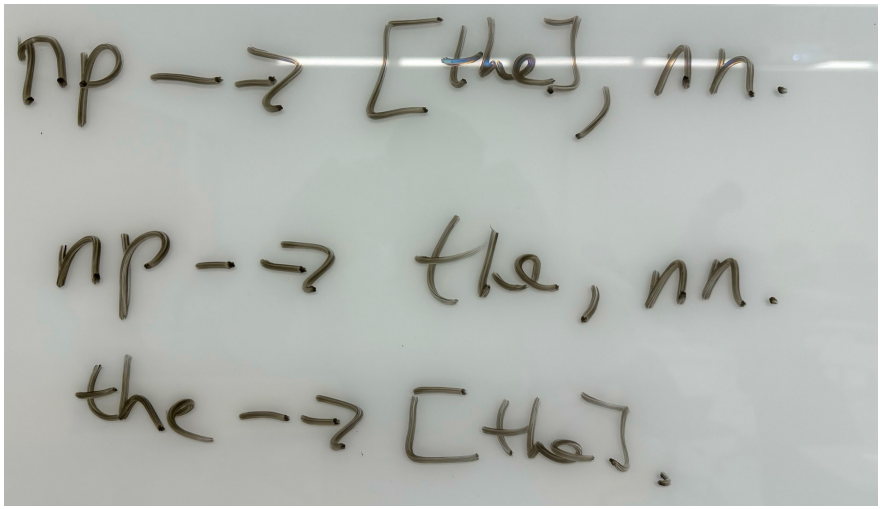
- A **tabular method** (*dynamic programming*) for parsing Context-Free Grammars (CFG) (JM textbook, 13.4)
- Works with **CFG**s expressed in **Chomsky Normal Form** (**CNF**) format (JM textbook, 12.5):
  1. `x --> y, z.`
  2. `x --> [w].`
  3. `s --> [].` (**s** start symbol) (optional rule, not for non-start symbols)
  - Note 1:
    - all CFGs can be expressed in this format, although it destroys linguistic structure (*which can be rebuilt*).
  - Note 2:
    - cf. regular grammars: rule 1 vs. `x --> [w], y.` or `x --> y, [w].`
  - Note 3:
    - other normal forms are also possible:
    - e.g. **Greibach Normal Form** (GNF): `x --> [a], y..z`
    - non-left recursive!

Chomsky Normal Form (CNF)

- **Conversion steps:**
  1. new start symbol s_0 `--> s.`
     (`s` original start symbol) to cope with possible empty derivation.
     s_0 is the only nonterminal allowed to have form s_0 `--> s`
  2. Make new nonterminal for terminals on the RHS:
     `x --> … [w]` … (|RHS| > 1)
     becomes `x -->` … x_w … and x_w `--> [w].` (x_w a new nonterminal)
  3. Binarize the RHS:
     `x -->` y_1`,` …`,`y_{n-1}`,`y_n`.` (|RHS| > 2)
     becomes a cascading sequence of binary rules:
     `x -->` y_1`,`x_1`.` x_1 `-->` y_2`,`x_2`.` … x_{n-1} `-->` y_{n-1}`,`y_n`.`
  4. Delete epsilon rules:
     for each instance of `x --> [].` and `y -->` … `x` …
     add a copy of y-rule without `x`, i.e`.` `y -->` … … then delete `x --> [].`
  5. Remove singleton rules:
     remove `x --> y.` (exception: `x` cannot be s_0) Replace `x` by `y` in other rules.

Chomsky Normal Form (CNF)

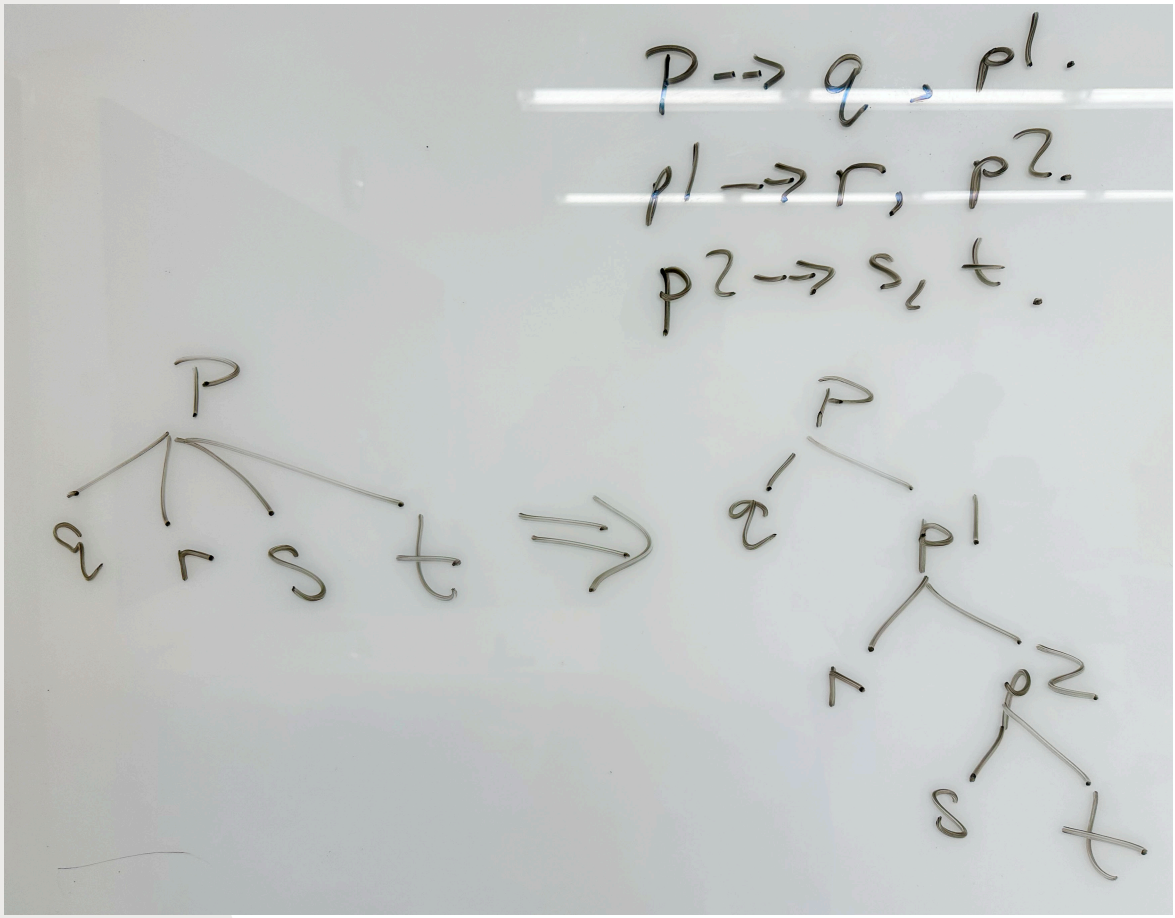
2. Replace terminals on the RHS, e.g. `[the]`.



Chomsky Normal Form (CNF)

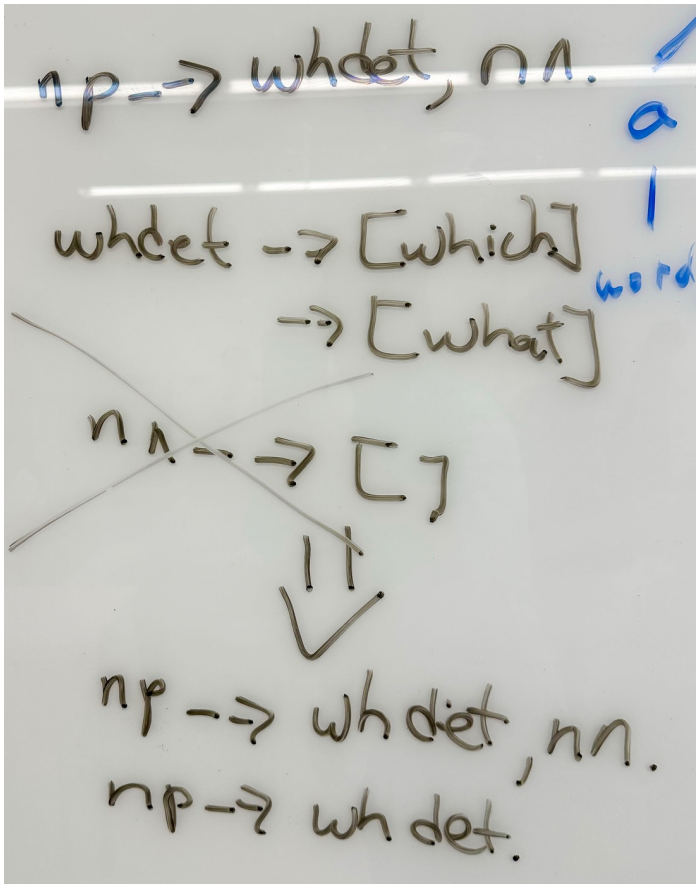
3. Binarization of *n*-ary rules (*n* > 2), e.g.

- `p --> q,r,s,t.`



Chomsky Normal Form (CNF)

4. Delete epsilon rules, e.g. *what/which man* vs. *what/which*



Chomsky Normal Form (CNF)

5. Singleton rules:
   - e.g. *cats are nice*

Chomsky Normal Form (CNF)

- Example:

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

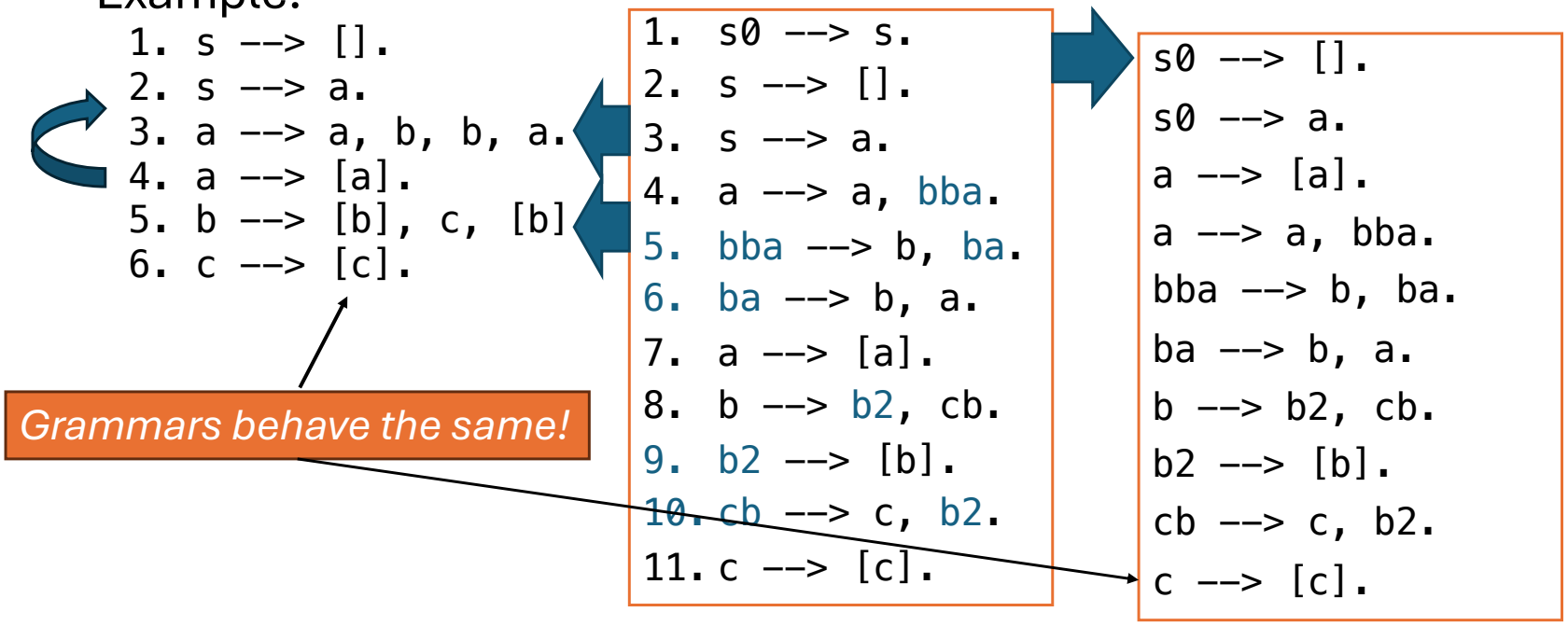
1. []
2. [a]
3. [a, b, c, b, b, c, b, a]
4. [a, b, c, b, b, c, b, a, b, c, b, b, c, b, a]
5. [a, b, c, b, b, c, b, a, b, c, b, b, c, b, a, b, c, b, b, c, b, a]
6. …

- *Let's convert this grammar to CNF*

Chomsky Normal Form (CNF)

- Example:

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



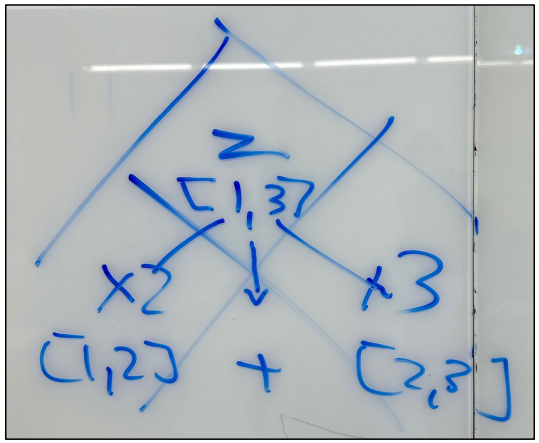
CKY Algorithm

- Grammar transformed into:
  ```
  1. x --> y, z.
  2. x --> [w].
  ```
- Given a sentence $w_1,..,w_n$, represent the possible parsings of this string by a table, where each cell may hold a nonterminal covering a substring.
- **Example**: $^0\,w_1\,^1\,w_2\,^2\,w_3\,^3$ ($^0..^3$ are position markers)

w_1	w_2	w_3
[0,1]	[0,2]	[0,3]
	[1,2]	[1,3]
		[2,3]

CKY Algorithm

- Chomsky Normal Form (CNF)
  - binary branching
- 2D table



w_1	w_2	w_3
[0,1] **x1**	[0,2] **y**	[0,3] **s**
	[1,2] **x2**	[1,3] **z**
		[2,3] **x3**

```
s --> y, x3.
s --> x1, z.
s --> y, z.
y --> x1, x2.
z --> x2, x3.
x1 --> [w1].
x2 --> [w2].
x3 --> [w3].
```

CKY Algorithm

- Bottom-up:

w_1	w_2	w_3
[0,1] **x1**	[0,2]	[0,3]
	[1,2] **x2**	[1,3]
		[2,3] **x3**

```
x1 --> [w1].
x2 --> [w2].
x3 --> [w3].
```

CKY Algorithm

- Bottom-up: $^{0}\,w_1\,^{1}\,w_2\,^{2}\,w_3\,^{3}$

w_1	w_2	w_3
[0,1] **x1**	[0,2] **y**	[0,3]
	[1,2] **x2**	[1,3] **z**
		[2,3] **x3**

```
y --> x1, x2.
z --> x2, x3.
```

CKY Algorithm

- Bottom-up:

w_1	w_2	w_3
[0,1] **x1**	[0,2] **y**	[0,3]
	[1,2] **x2**	[1,3] **z**
		[2,3] **x3**

```
s --> y, z.
```

s is not derivable because **[0,2]y** overlaps with **[1,3]z**

CKY Algorithm

- Bottom-up:

w_1	w_2	w_3
[0,1] **x1**	[0,2] **y**	[0,3] **s**
	[1,2] **x2**	[1,3] **z**
		[2,3] **x3**

```
s --> x1, z.
```

[0,3]s is derivable because **[0,1]x1** concats with **[1,3]z**

CKY Algorithm

- Bottom-up:

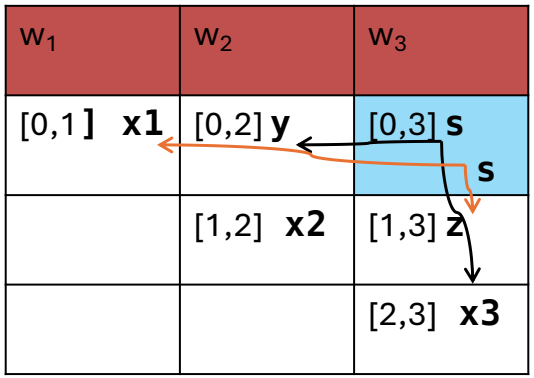
w_1	w_2	w_3
[0,1] **x1**	[0,2] **y**	[0,3] **s**
	[1,2] **x2**	[1,3] **z**
		[2,3] **x3**

```
s --> y, x3.
```

[0,3]s is derivable because **[0,2]y** concats with **[2,3]x3**

CKY Algorithm

- Backpointers indicate parse:



CKY Algorithm


```
function CKY-PARSE(words, grammar) returns table

  for j ← from 1 to LENGTH(words) do
      table[j − 1, j] ← {A | A → words[j] ∈ grammar}
      for i ← from j − 2 downto 0 do
          for k ← i + 1 to j − 1 do
              table[i,j] ← table[i,j] ∪
                           {A | A → BC ∈ grammar,
                                B ∈ table[i,k],
                                C ∈ table[k, j]}
```

Figure 13.10 The CKY algorithm.

CKY Algorithm and Prolog Grammar Rules

- CNF:
  - we can make conversion *transparent*
  - because we can decide what to produce as a parse tree using an extra argument, cf. *left recursive parse transformation*
- Table representation:
  - (if using Prolog) must be careful about variable bindings:
  - solution: make fresh copies of variables
- Agreement (*feature propagation*):
  - must save all arguments into table
  - and relink up properly, e.g.
  - `np(np(DT,NN)) --> dt(DT,Num), nn(NN,Num).`

nltk book: chapter 8

- Link:
  - https://www.nltk.org/book/ch08.html
- Steps:

```
$ python
Python 3.9.12 (main, Jun  1 2022, 06:34:44)
>>> import nltk
>>> cnf = open('cnf.txt').read()
>>> cnf
"s -> y x3\ns -> x1 z\ns -> y z\ny -> x1 x2\nz -> x2 x3\nx1 -> 'w1'\nx2 -> 'w2'\nx3 -> 'w3'\n"
>>> cfg = nltk.CFG.fromstring(cnf)
>>> cfg
<Grammar with 8 productions>
>>> p = nltk.ChartParser(cfg)
```

cnf.txt on website

```
1 s -> y x3
2 s -> x1 z
3 s -> y z
4 y -> x1 x2
5 z -> x2 x3
6 x1 -> 'w1'
7 x2 -> 'w2'
8 x3 -> 'w3'
```

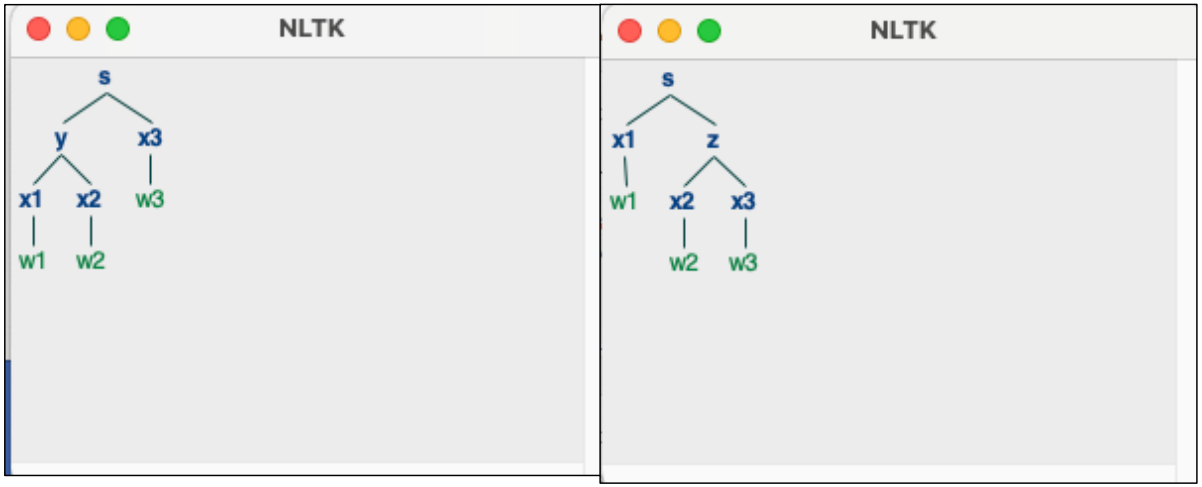
cnf is a string,
cfg is a grammar

nltk book: chapter 8

- To parse, supply a pre-tokenized sentence (a list):

```
>>> for tree in p.parse(['w1','w2','w3']):
...     tree.draw()
...
>>>
```

<TAB> to indent, and need a blank line as well



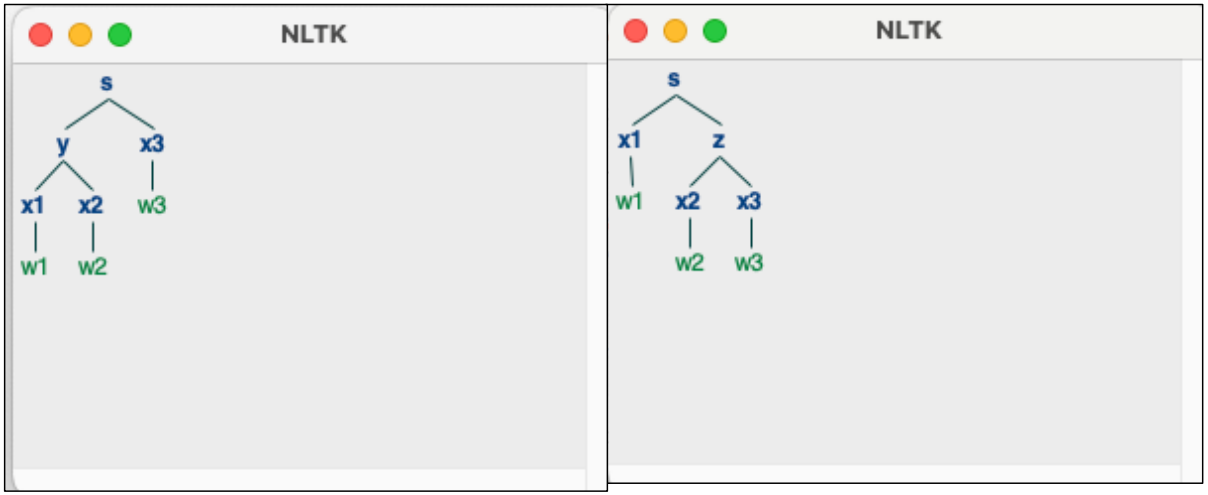
nltk book: chapter 8

- To perform parsing, call:
  - `p.parse(list)` p is a parser
- But, first define p, the parser you want to use.
- Different parsing strategies available, e.g.:

```
1. p = nltk.ChartParser(cfg)
2. p = nltk.ShiftReduceParser(cfg)        doesn't do backtracking
3. p = nltk.RecursiveDescentParser(cfg)   doesn't do left recursion
```

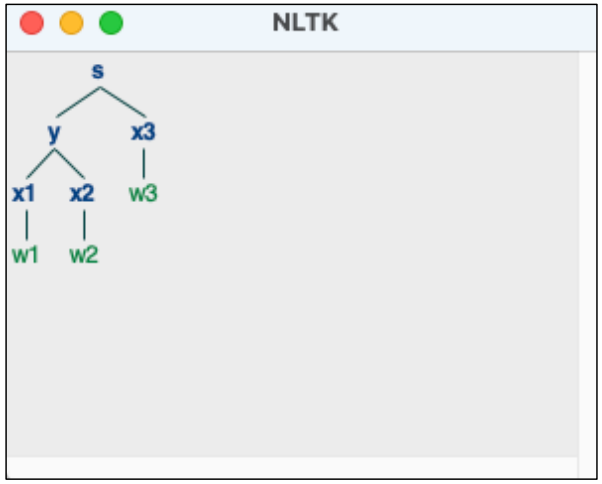
nltk book: chapter 8

- Same sentence, different parser:
- `>>> p = nltk.RecursiveDescentParser(cfg)`
- `>>> for tree in p.parse(['w1','w2','w3']):`
- `...     tree.draw()`
- `...`
- `>>>`



nltk book: chapter 8

- Same sentence, different parser:
- `>>> p = nltk.ShiftReduceParser(cfg)`
- `>>> for tree in p.parse(['w1','w2','w3']):`
- `...     tree.draw()`
- `...`
- `>>>`



nltk book: chapter 8

- Notes on grammar format:
  - plain text file
  - 1st line defines the start-symbol (S)
  - -> is the rewrite symbol (cf. Prolog `-->`)
  - space between symbols (cf. Prolog `,`)
  - lexical items are quoted, e.g. 'I' (cf. Prolog `[word]`)
  - *you cannot combine grammatical categories with lexical items*
    - PP -> 'of' NP (disallowed, cf. `pp --> [of], np.`)
  - *you are not permitted multi-word lexical items*
    - not ok: NP -> 'New York'
    - ok: NP -> 'New_York' (cf. Prolog `[new,york]`)