Context-free Grammars

BM1 Advanced Natural Language Processing

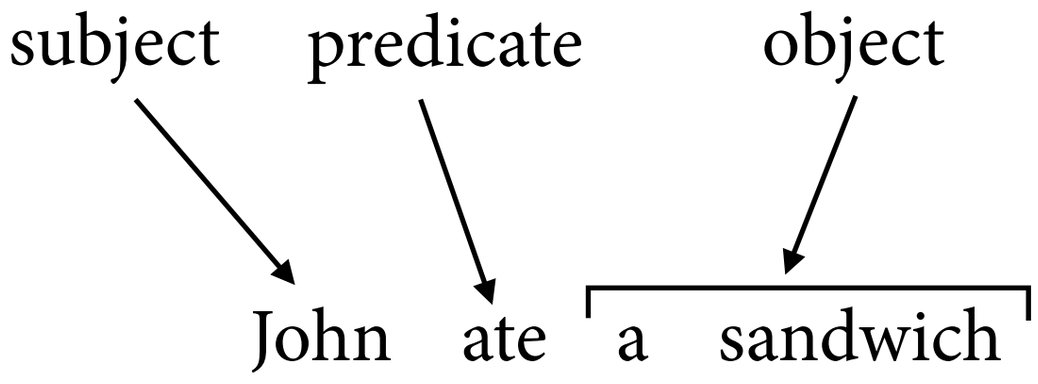
Alexander Koller

21 November 2014

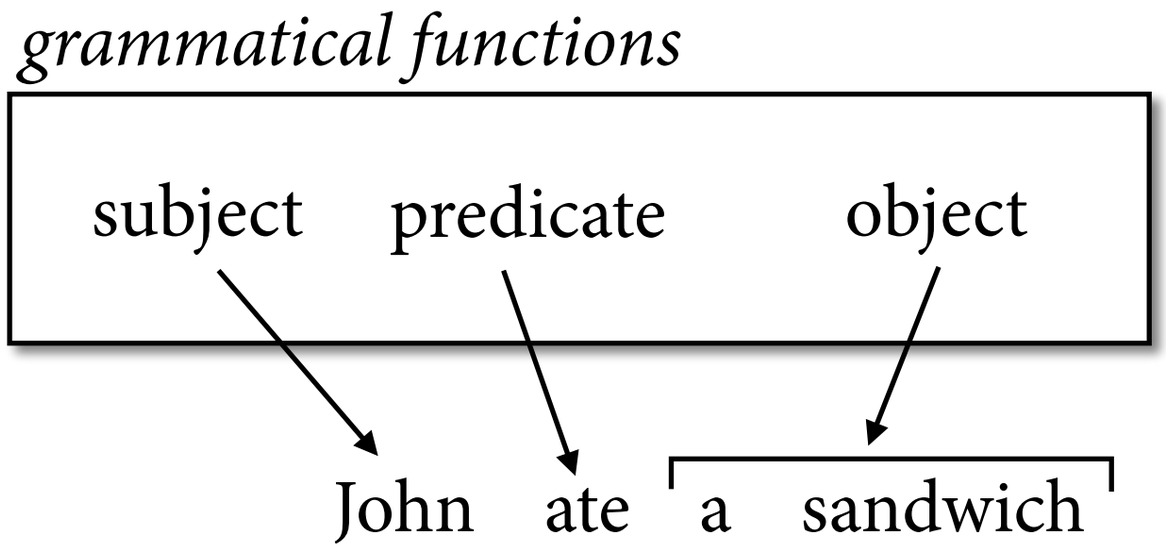
Sentences have structure

John ate a sandwich

Sentences have structure



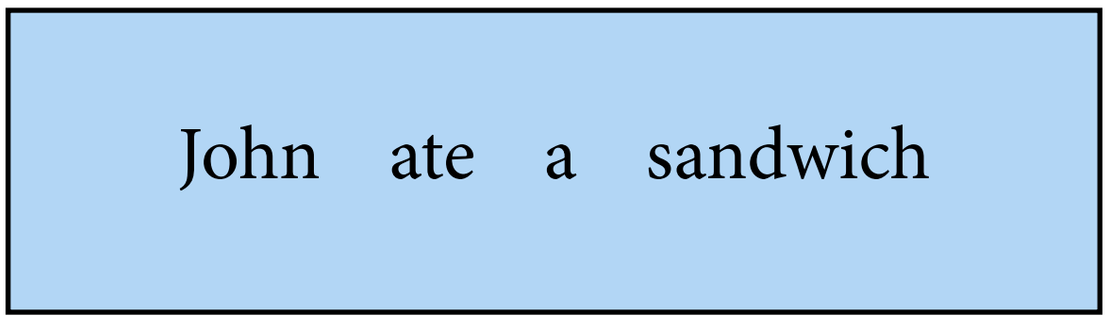
Sentences have structure



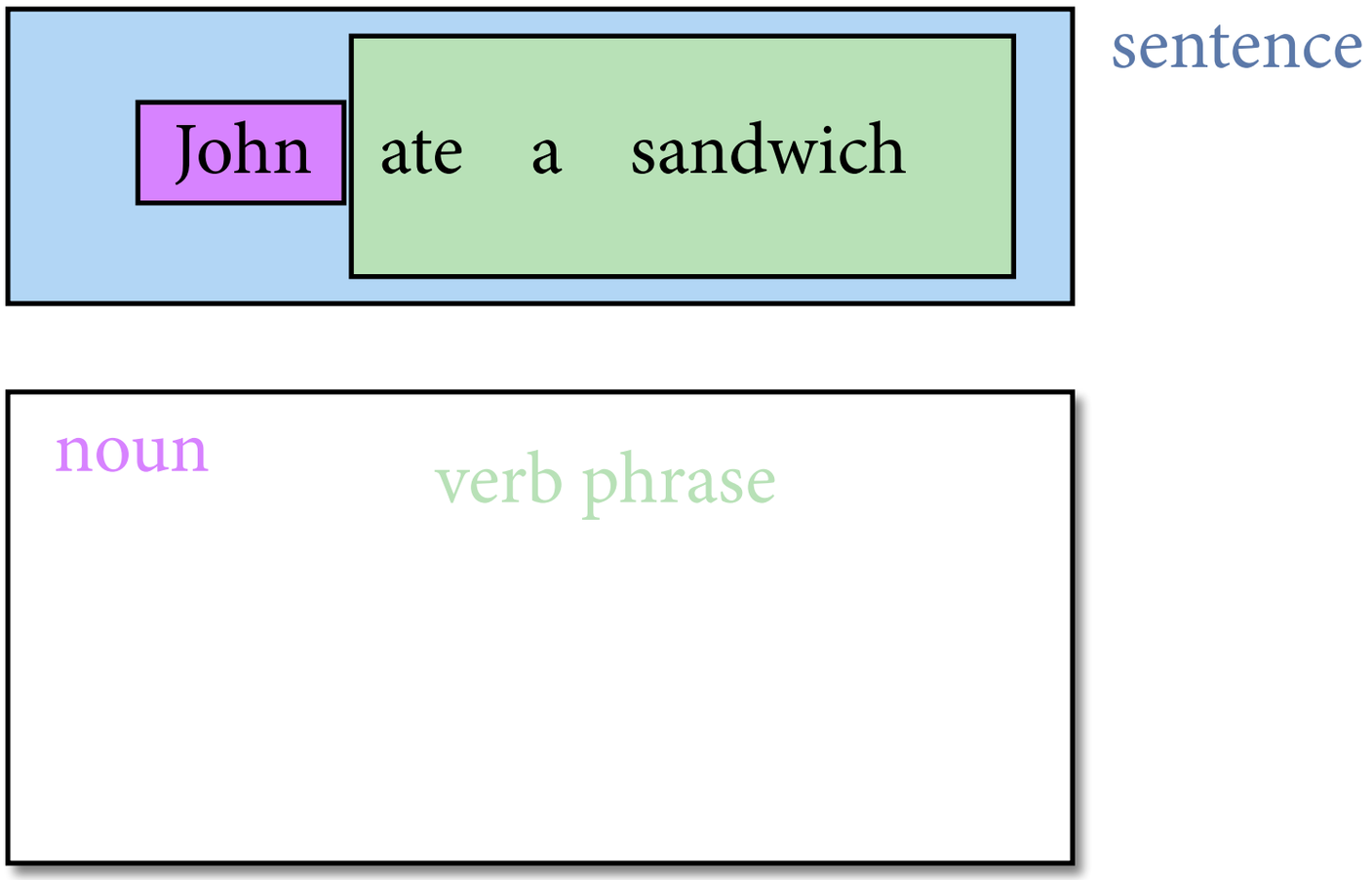
Sentences have structure

John ate a sandwich

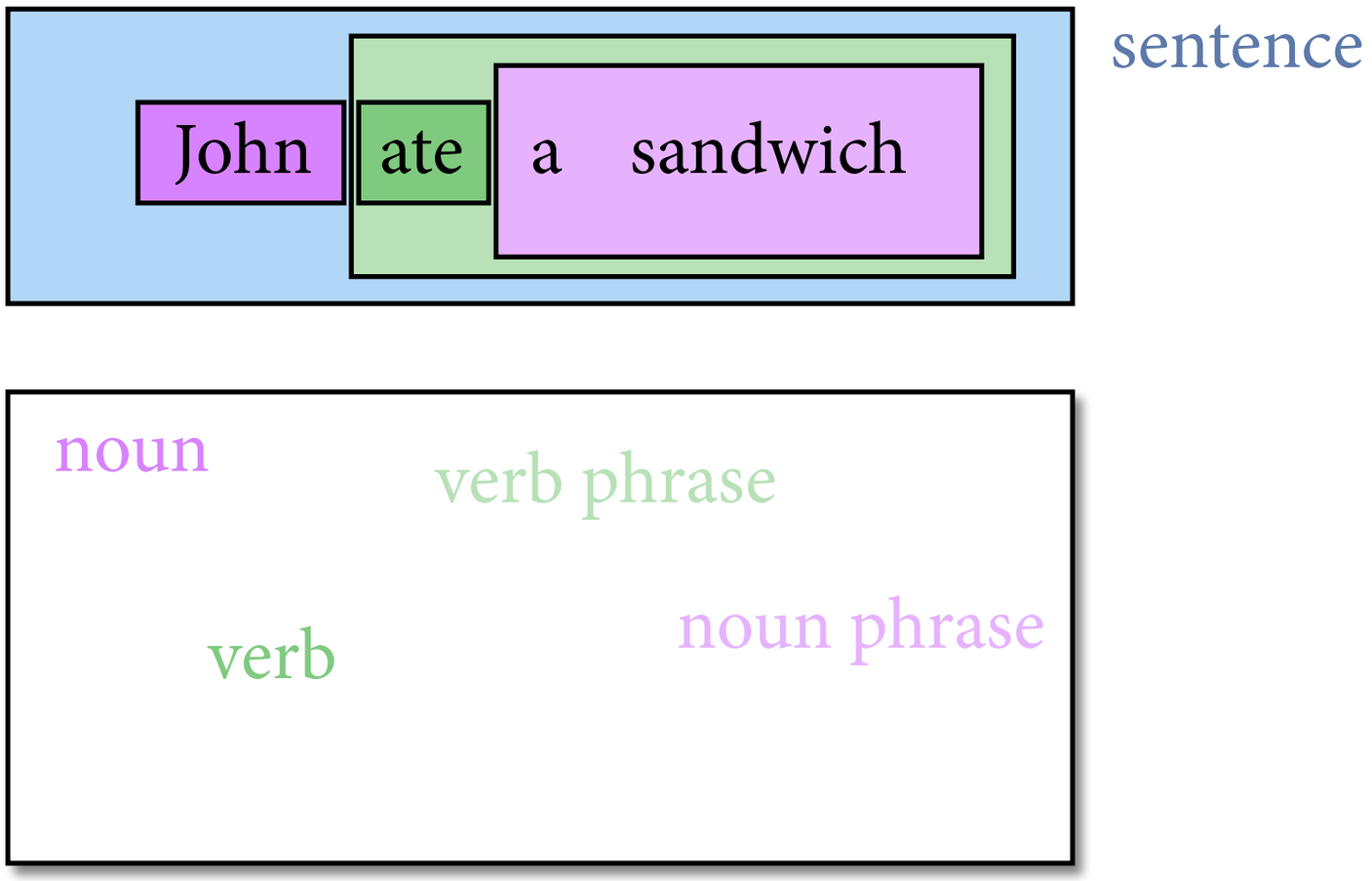
Sentences have structure



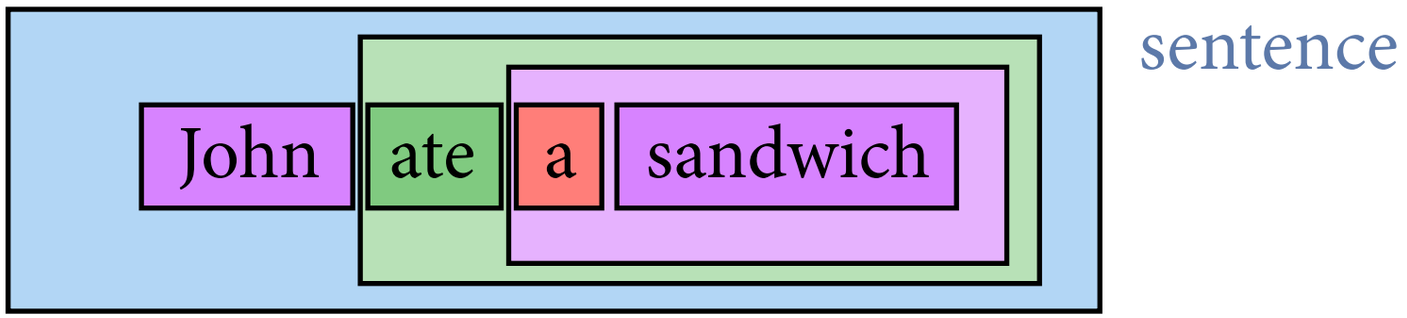
Sentences have structure

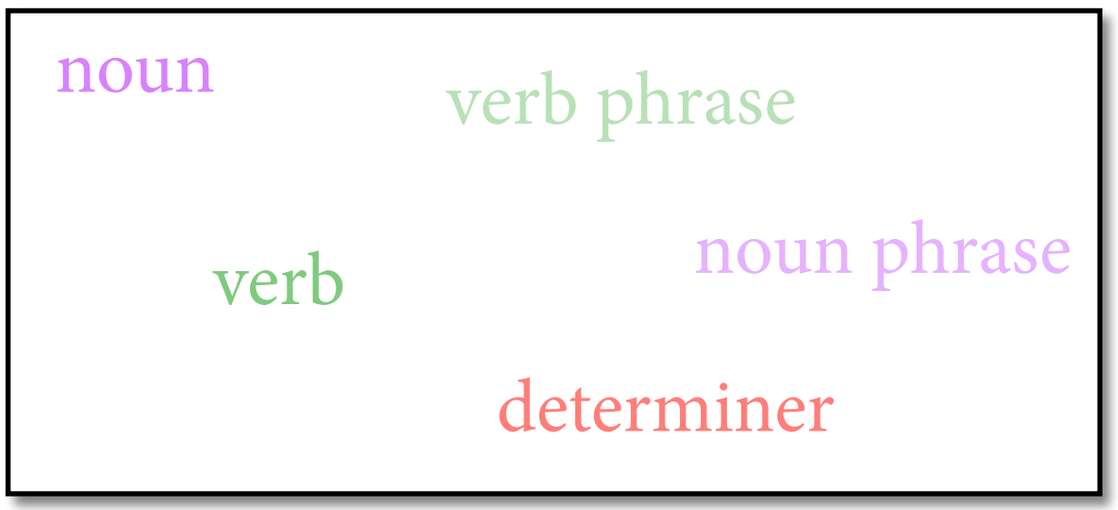


Sentences have structure



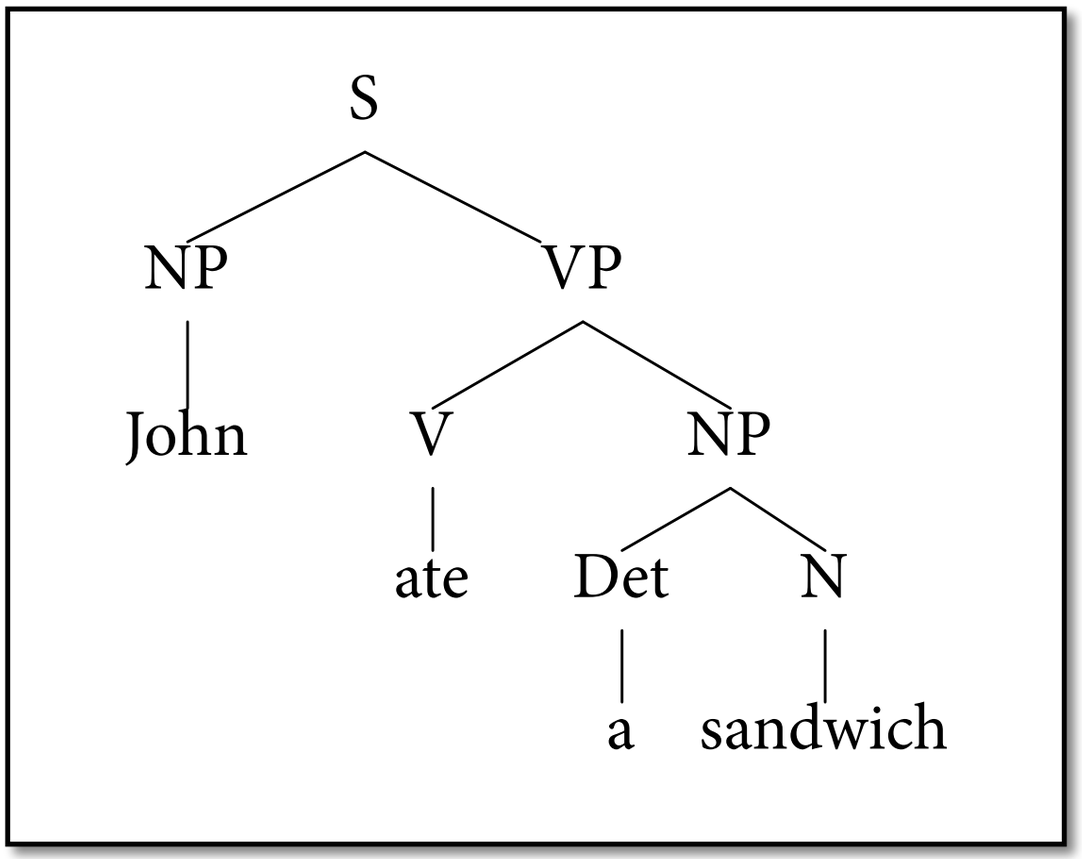
Sentences have structure





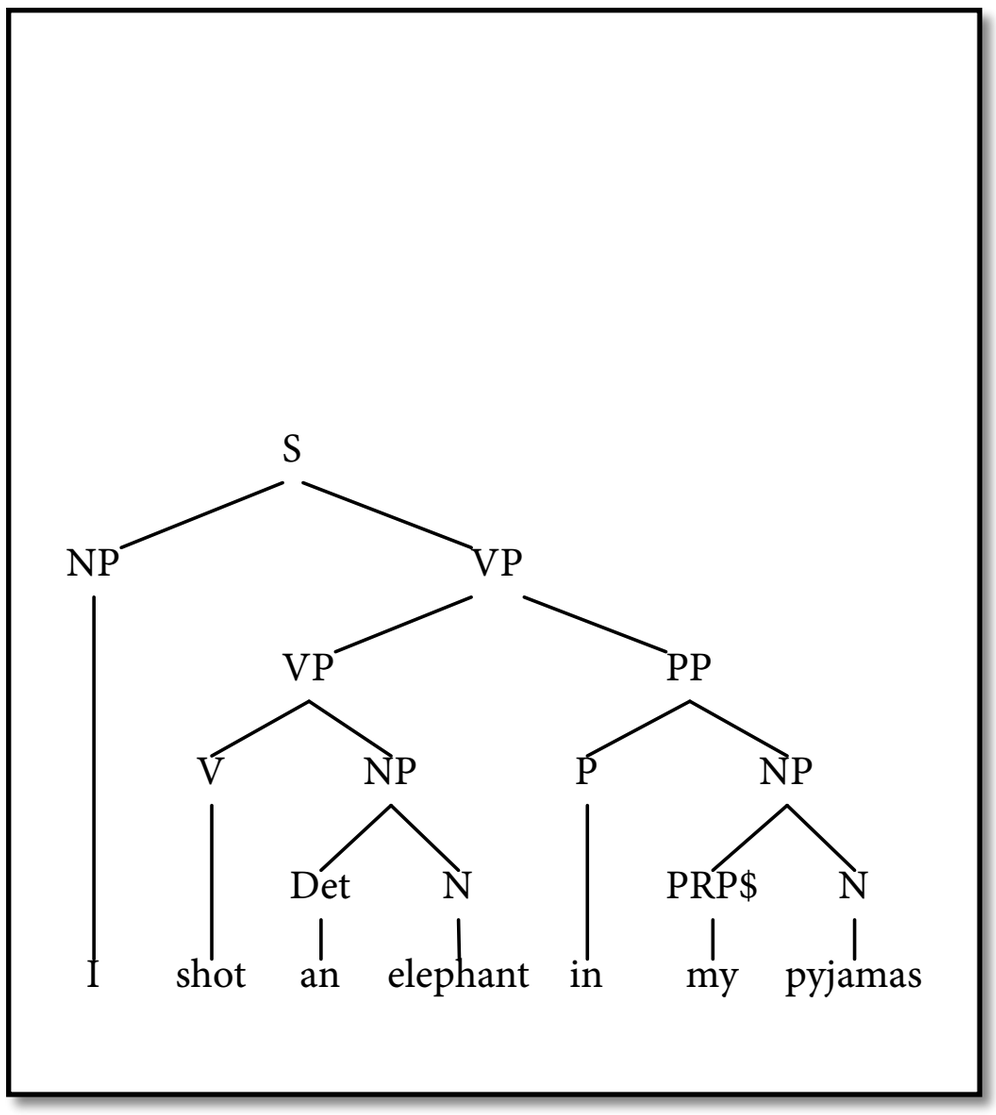
Sentences have structure

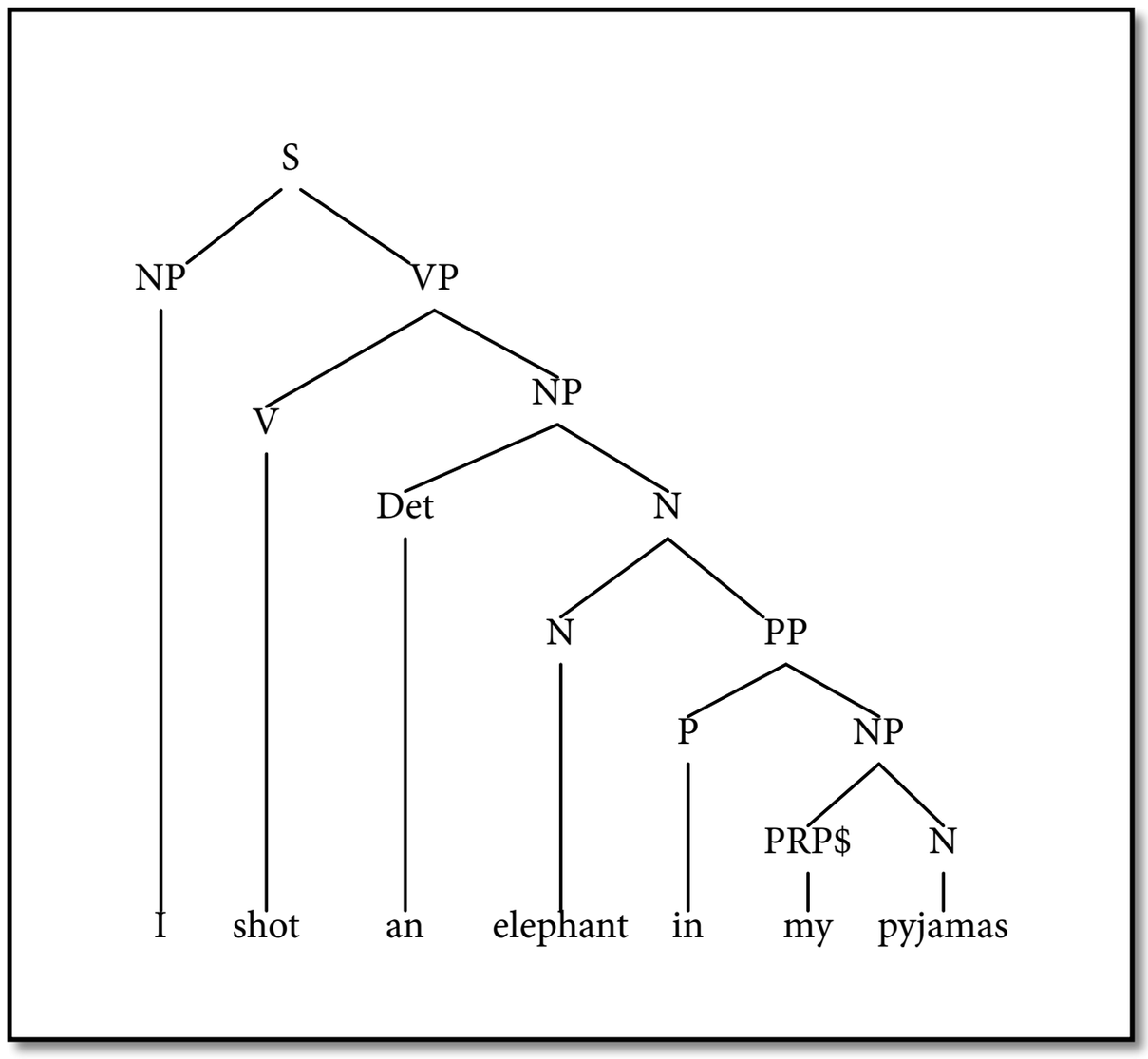
Record it conveniently in *phrase structure tree.*



Ambiguity

Special challenge: sentences can have many possible structures.





This sentence is example of *attachment ambiguity.*

Grammars

- A *grammar* is a finite device for describing large (possibly infinite) set of strings.
  - strings = NL expressions of various types
  - grammar captures linguistic knowledge about syntactic structure

- There are many different grammar formalisms that are being used in NLP.

- In this course we focus on *context-free grammars*.

Context-free grammars

- Context-free grammar (cfg) G is 4-tuple (N,T,S,P):
  - N and T are disjoint finite sets of symbols:
    T = *terminal* symbols; N = *nonterminal* symbols.
  - S ∈ N is the *start symbol.*
  - P is a finite set of *production rules* of the form A → w,
    where A is nonterminal and w is a string from $(N \cup T)^*$.

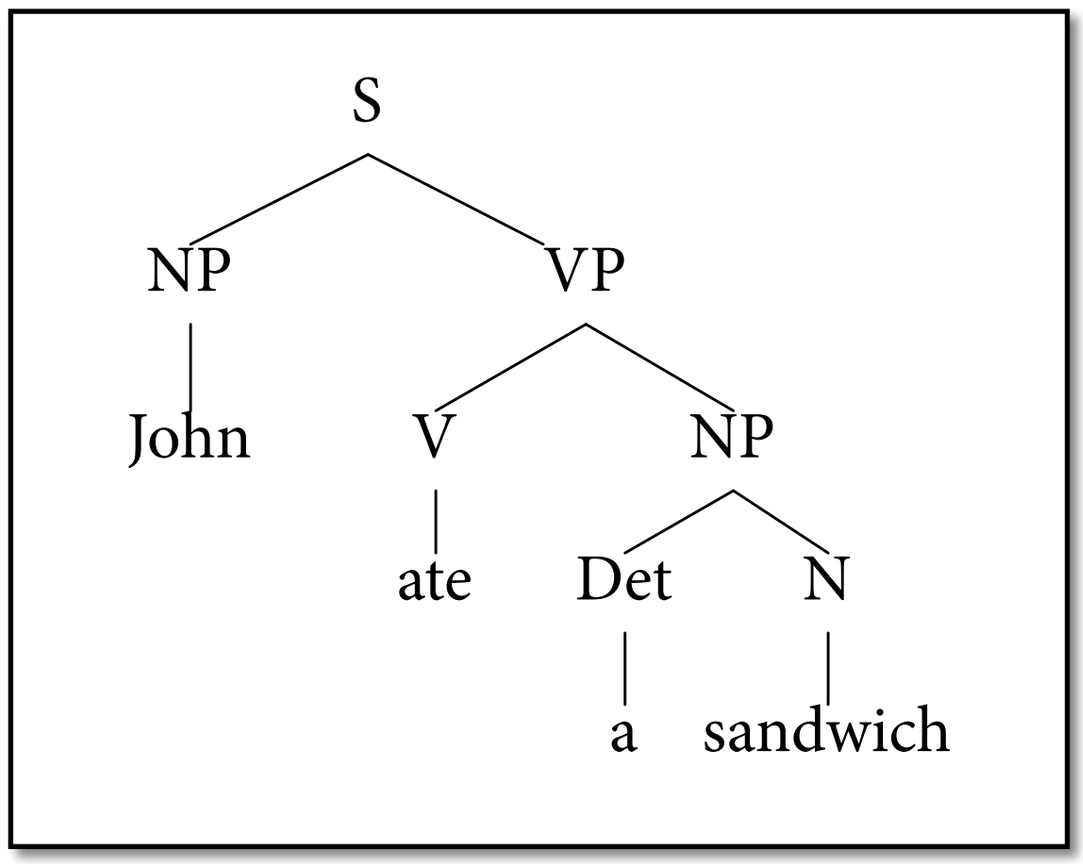
- Why "context-free"?
  - Left-hand side of production is a single nonterminal A.
  - Rule can't look at context in which A appears.
  - *Context-sensitive* grammars can do that.

Example

T = {John, ate, sandwich, a}
N = {S, NP, VP, V, N, Det}; start symbol: S

Production rules:

S → NP VP
NP → Det N
VP → V NP
V → ate
NP → John
Det → a
N → sandwich



Some important concepts

- *One-step derivation* relation $\Rightarrow$:

  $w_1 \, A \, w_2 \Rightarrow w_1 \, w \, w_2$ iff $A \rightarrow w$ is in P

  (w_1, w_2, w are strings from $(N \cup T)^*$)

- *Derivation* relation $\Rightarrow^*$ is reflexive, transitive closure:

  $w \Rightarrow^* w_n$ if $w \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n$ (for some $n \geq 0$)

- *Language* $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$

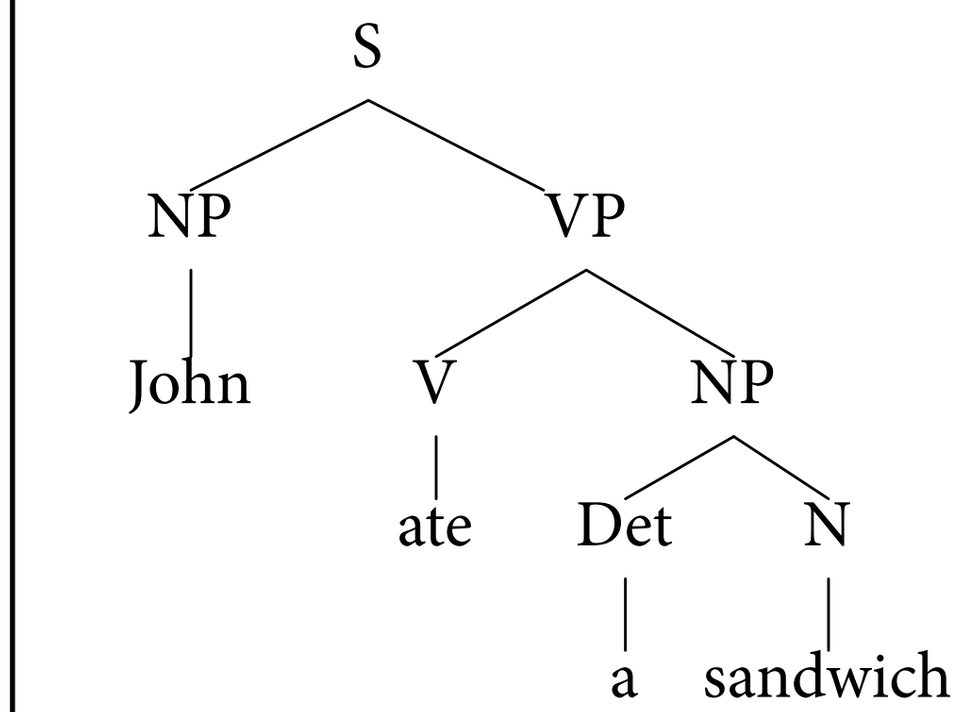
Derivations and parse trees

Parse tree provides readable, high-level view of derivation.

derivation

S ⇒ NP VP ⇒ John VP
⇒ John V NP ⇒ John ate NP
⇒ John ate Det N
⇒ John ate a N
⇒ John ate a sandwich

parse tree



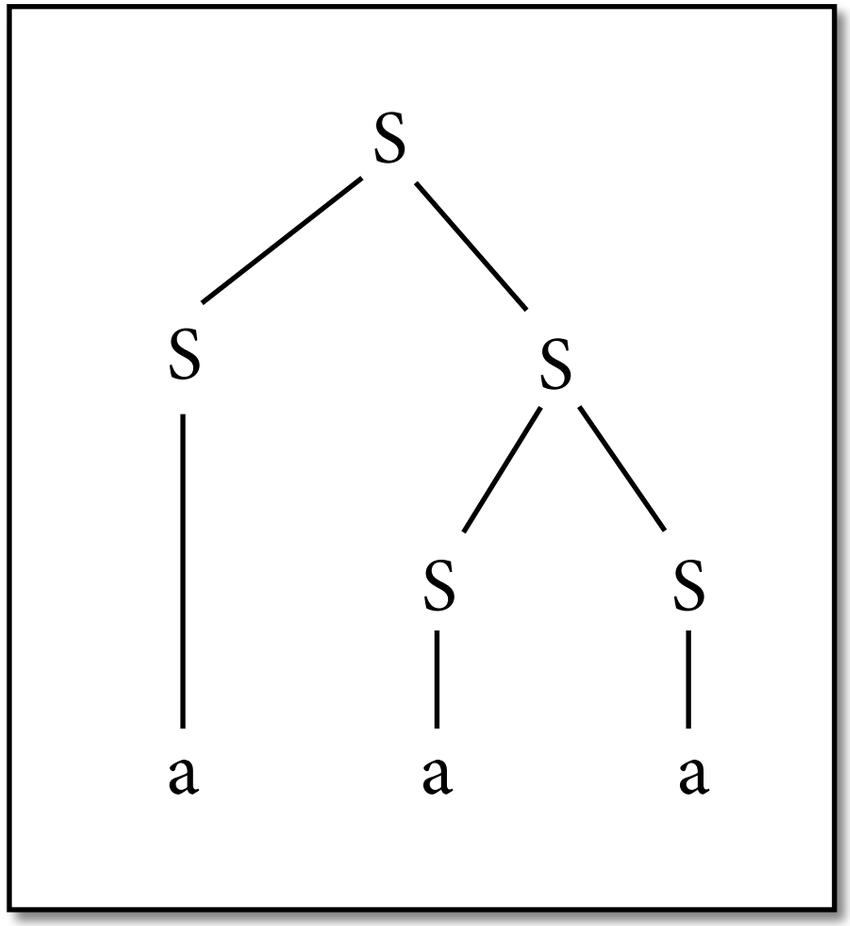
Recognition and parsing

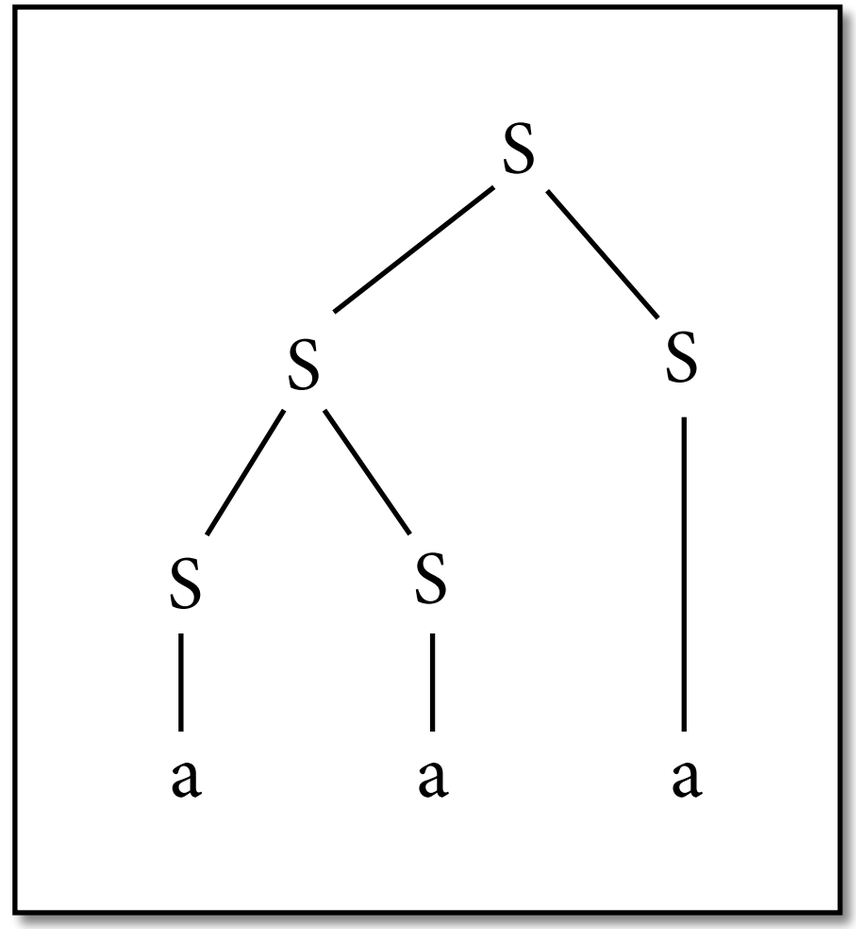
- Let G be a cfg and w be a string.
- *Word problem:* is $w \in L(G)$?
  - Algorithms that solve it are called *recognizers*.
- *Parsing problem:* enumerate all parse trees of w.
  - Algorithms that solve it are called *parsers*.
- Every parser also solves the word problem.

Big languages

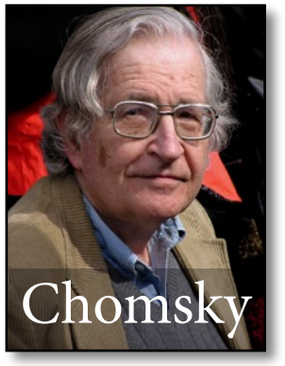
Number of parse trees can grow exponentially in string length.

$S \rightarrow S\ S$ $\qquad$ $S \rightarrow a$





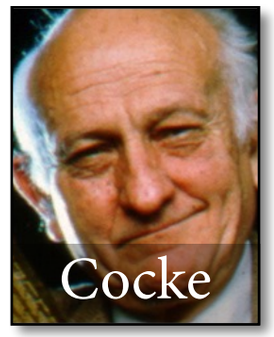
Chomsky Normal Form

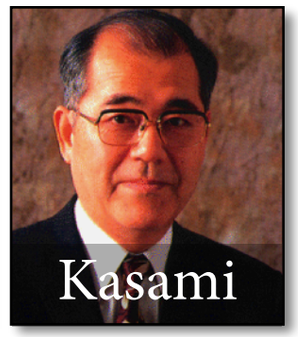


- A cfg is *in Chomsky normal form (CNF)* if each of its production rules has one of these two forms:
  - A → B C: right-hand side is exactly two nonterminals
  - A → c: right-hand side is exactly one terminal

- For every cfg G, there is a weakly equivalent cfg G' which is in CNF.
  - that is, L(G) = L(G')

The CKY Algorithm

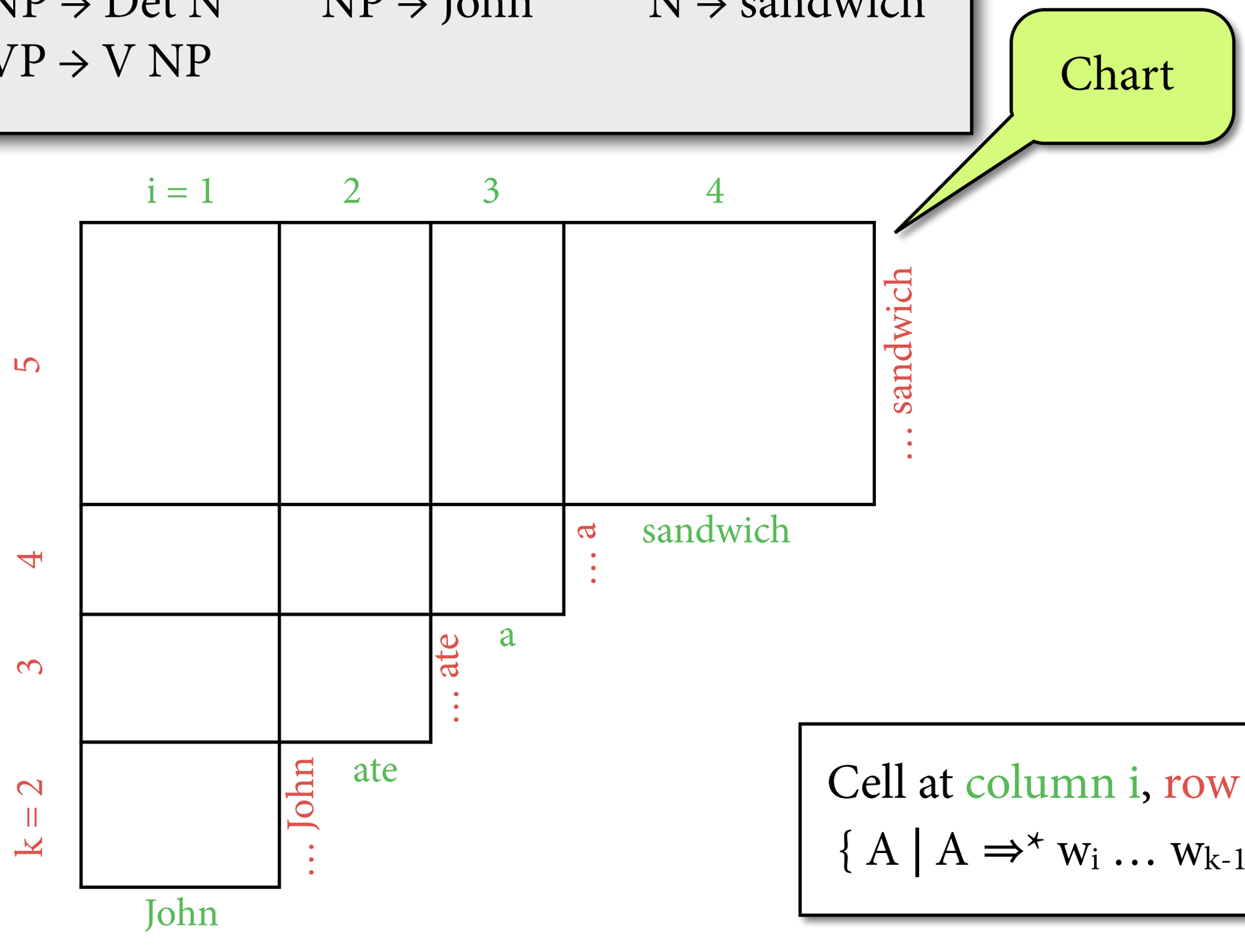




- Simplest and most-used chart parser for cfgs in CNF.
- Developed independently in the 1960s by John Cocke, Daniel Younger, and Tadao Kasami.
  - sometimes also called CYK algorithm
- Bottom-up algorithm for discovering statements of the form "$A \Rightarrow^* w_i \ldots w_{k-1}$?"

The CKY Recognizer

S → NP VP	V → ate	Det → a
NP → Det N	NP → John	N → sandwich
VP → V NP		



Cell at column i, row k:

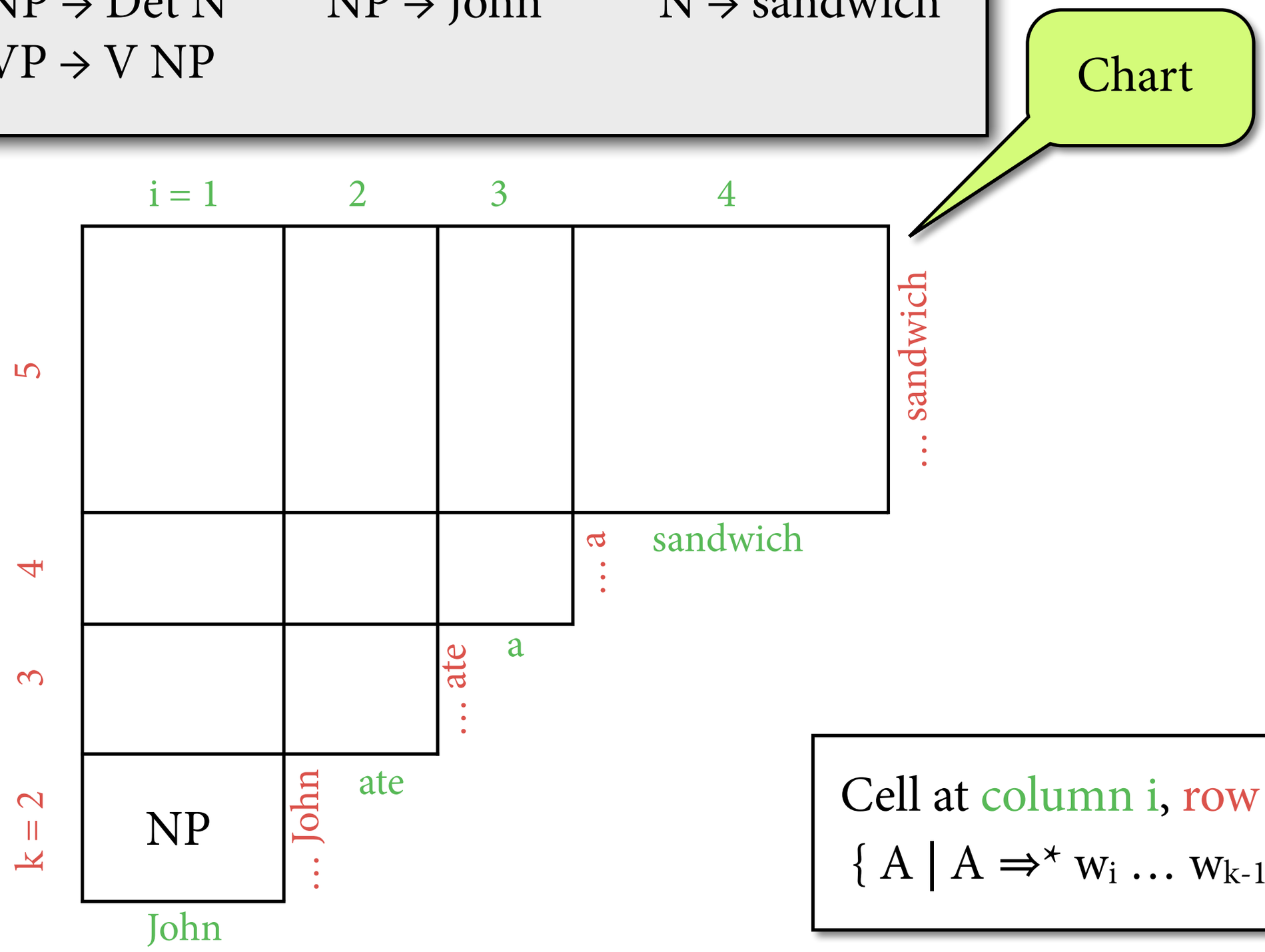
$\{ A \mid A \Rightarrow^* w_i \ldots w_{k-1} \}$

The CKY Recognizer

S → NP VP
NP → Det N
VP → V NP

V → ate
NP → John

Det → a
N → sandwich



Cell at column i, row k:

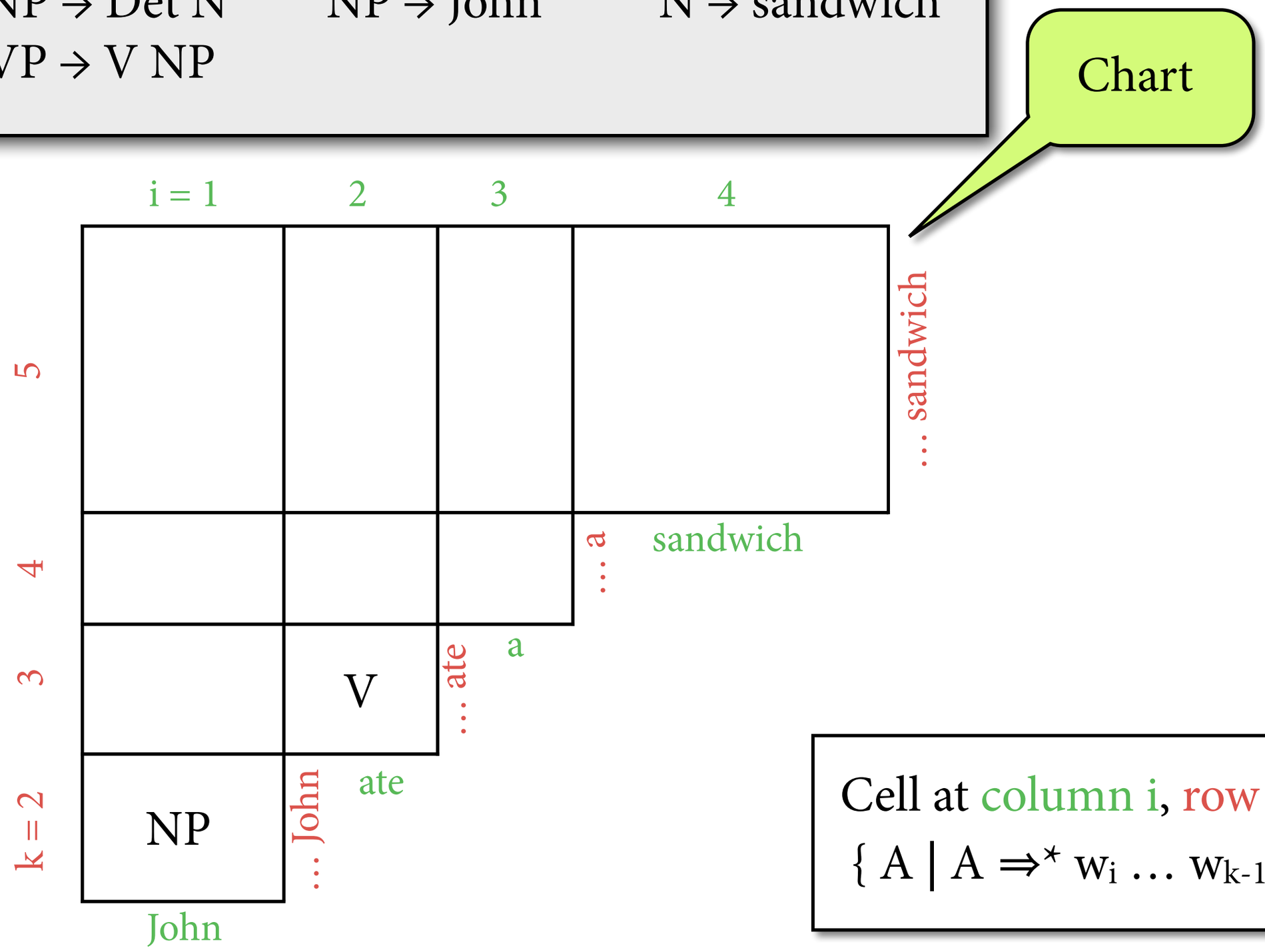
$\{ A \mid A \Rightarrow^* w_i \ldots w_{k-1} \}$

The CKY Recognizer

S → NP VP
NP → Det N
VP → V NP

V → ate
NP → John

Det → a
N → sandwich

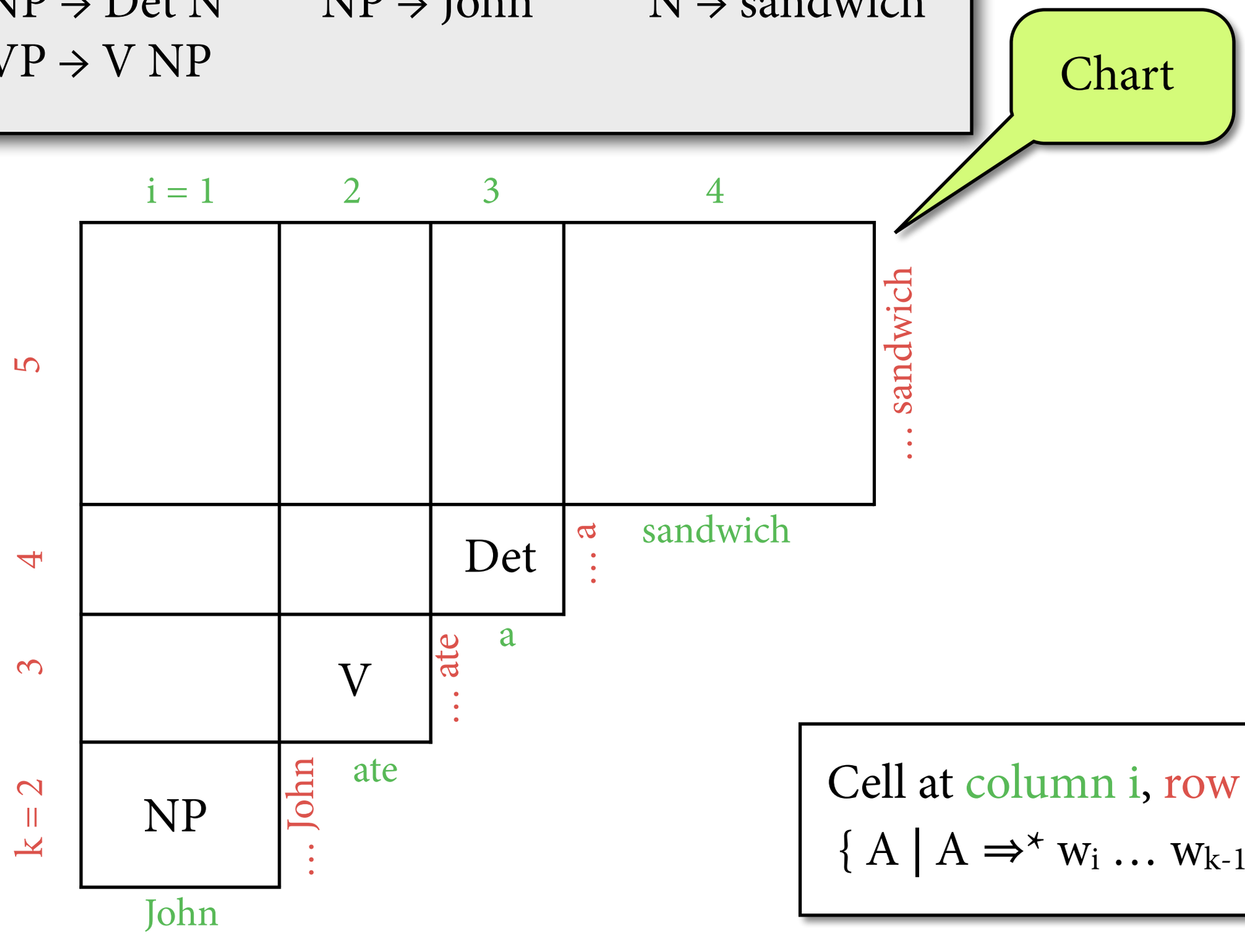


Cell at column i, row k:

$\{ A \mid A \Rightarrow^* w_i \ldots w_{k-1} \}$

The CKY Recognizer

S → NP VP	V → ate	Det → a
NP → Det N	NP → John	N → sandwich
VP → V NP		

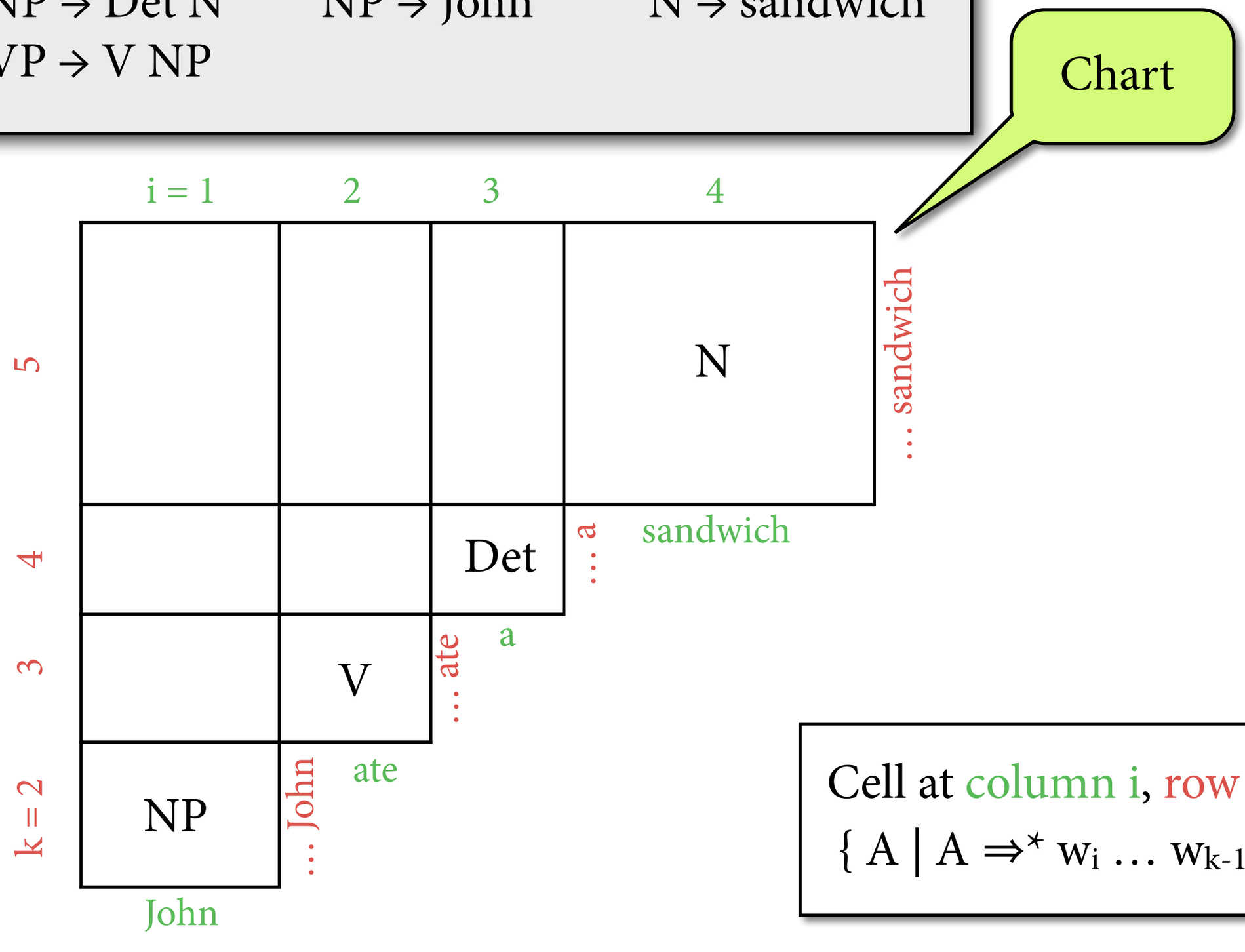


Cell at column i, row k:

$\{ A \mid A \Rightarrow^* w_i \ldots w_{k-1} \}$

The CKY Recognizer

S → NP VP	V → ate	Det → a
NP → Det N	NP → John	N → sandwich
VP → V NP		



Cell at column i, row k:

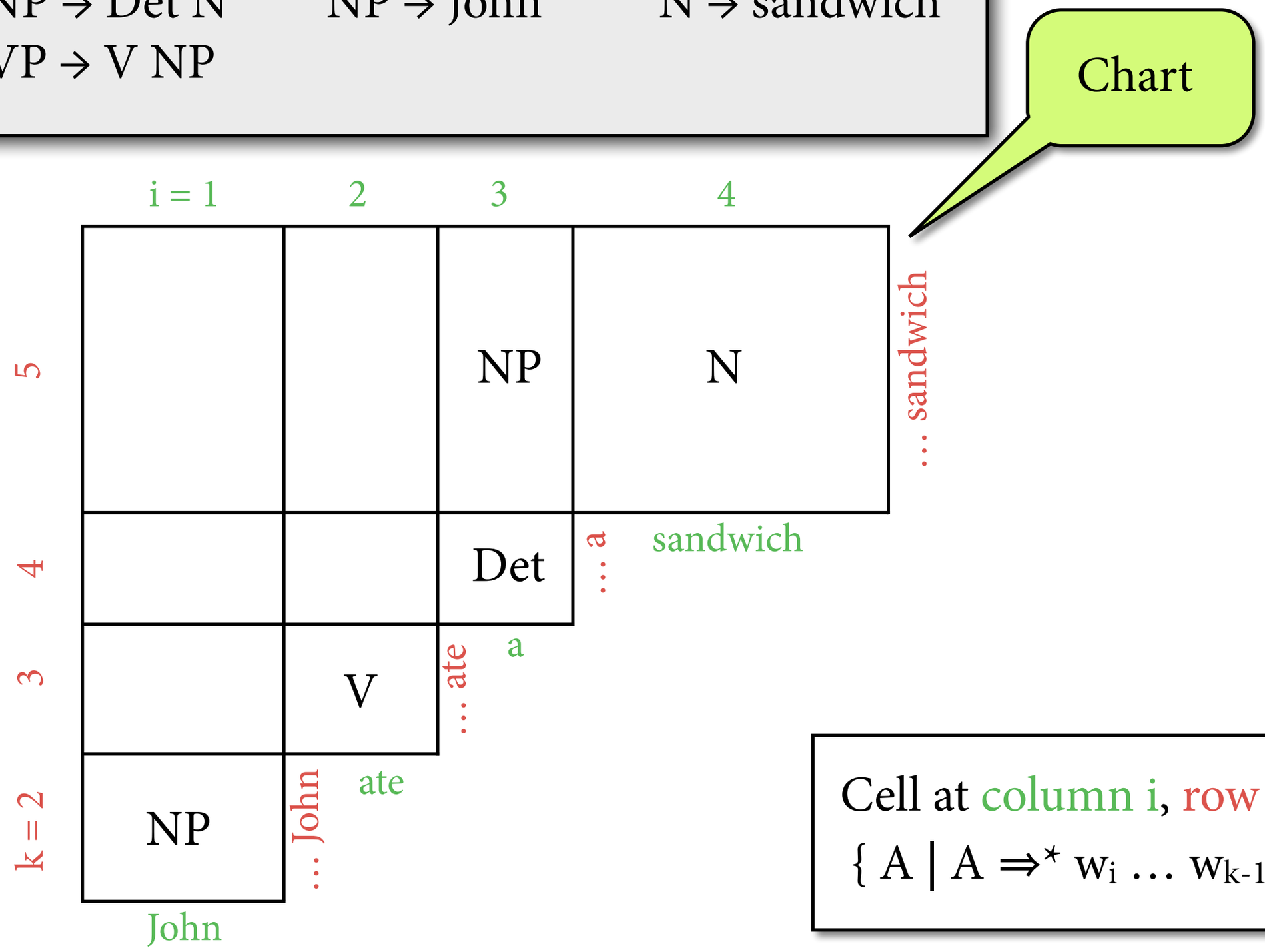
$\{ A \mid A \Rightarrow^* w_i \ldots w_{k-1} \}$

The CKY Recognizer

S → NP VP
NP → Det N
VP → V NP

V → ate
NP → John

Det → a
N → sandwich



Cell at column i, row k:

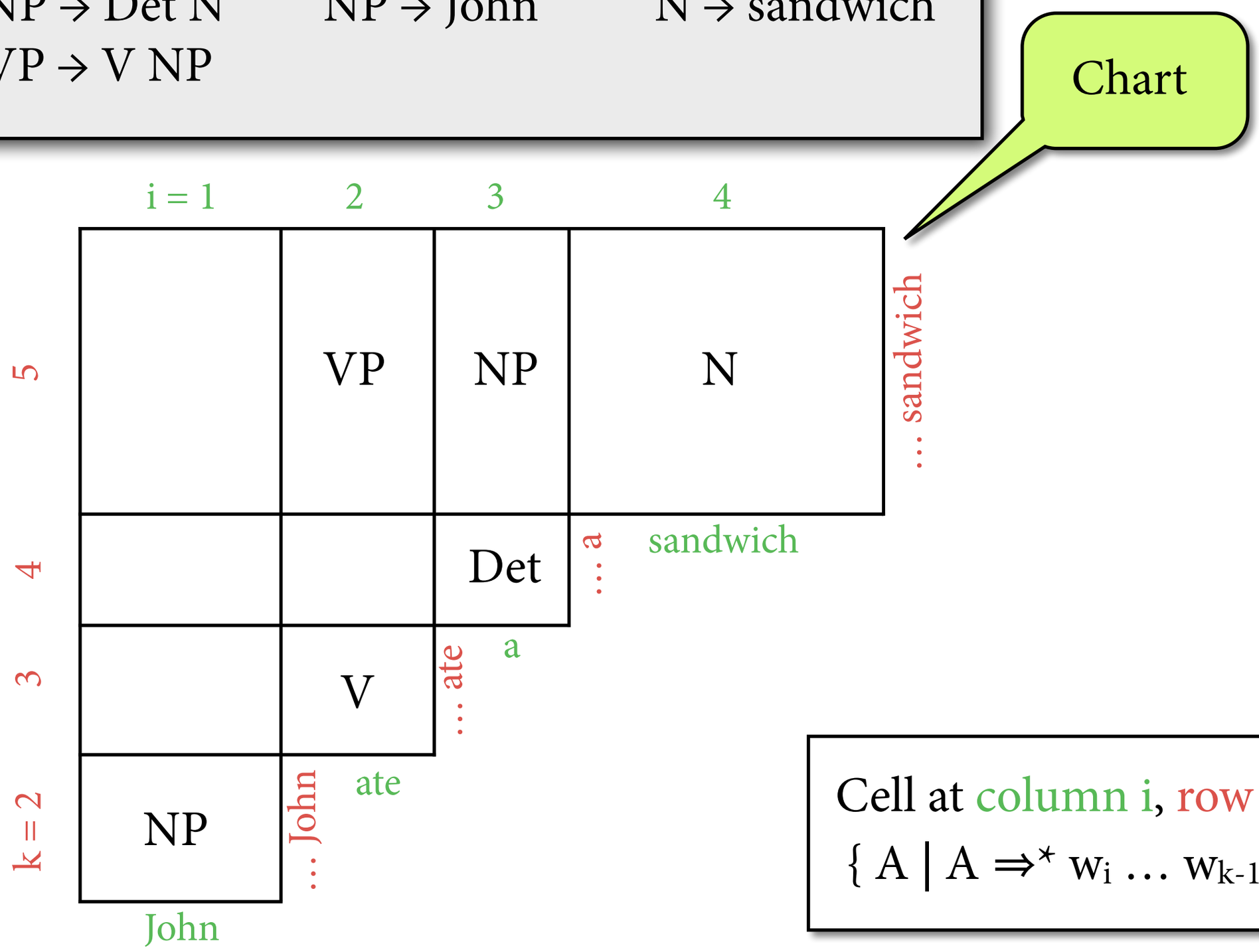
$\{ A \mid A \Rightarrow^* w_i \ldots w_{k-1} \}$

The CKY Recognizer

S → NP VP
NP → Det N
VP → V NP

V → ate
NP → John

Det → a
N → sandwich



Cell at column i, row k:

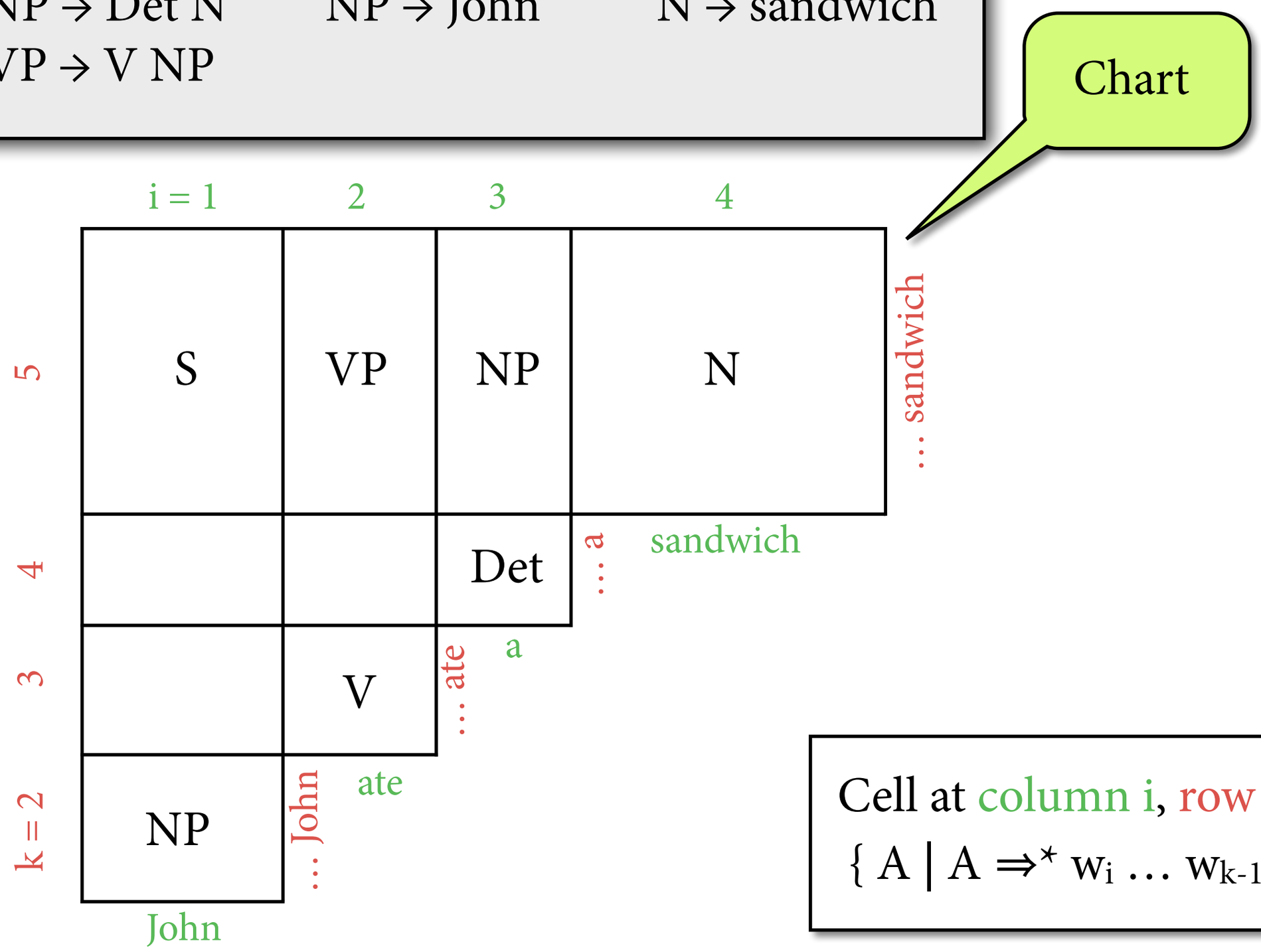
$\{ A \mid A \Rightarrow^* w_i \ldots w_{k-1} \}$

The CKY Recognizer

S → NP VP
NP → Det N
VP → V NP

V → ate
NP → John

Det → a
N → sandwich



Cell at column i, row k:

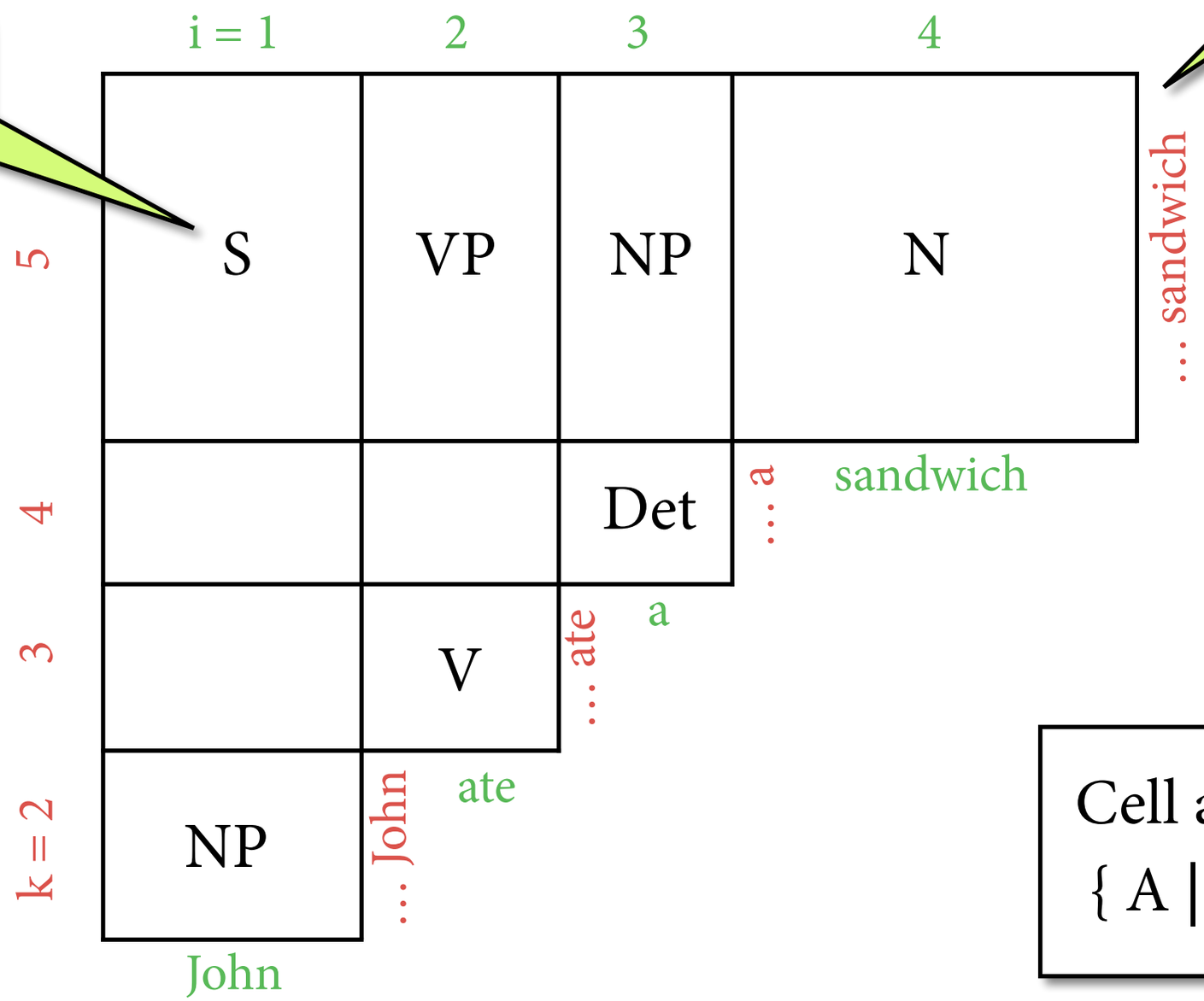
$\{ A \mid A \Rightarrow^* w_i \ldots w_{k-1} \}$

The CKY Recognizer

S → NP VP	V → ate	Det → a
NP → Det N	NP → John	N → sandwich
VP → V NP		

Chart

$S \Rightarrow^* w$

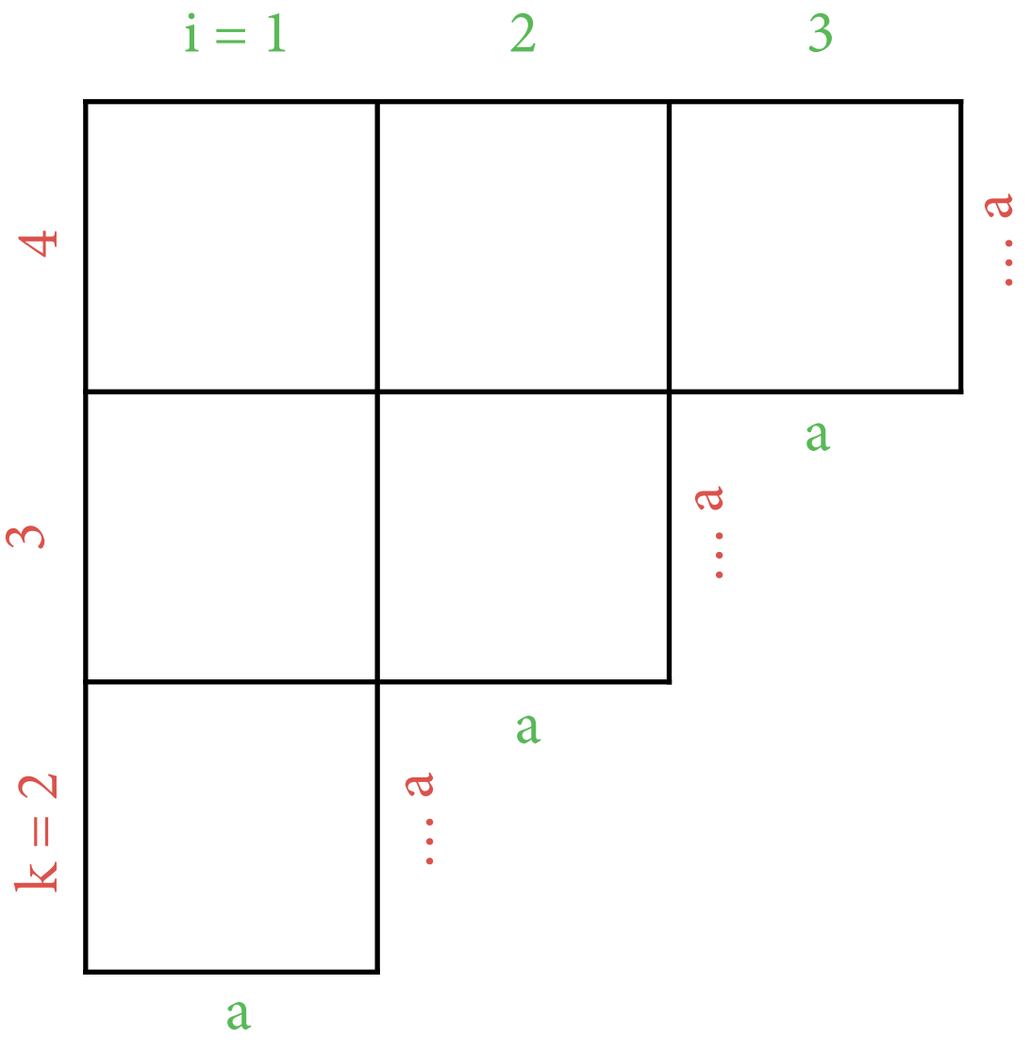


Cell at column i, row k:

$\{ A \mid A \Rightarrow^* w_i \ldots w_{k-1} \}$

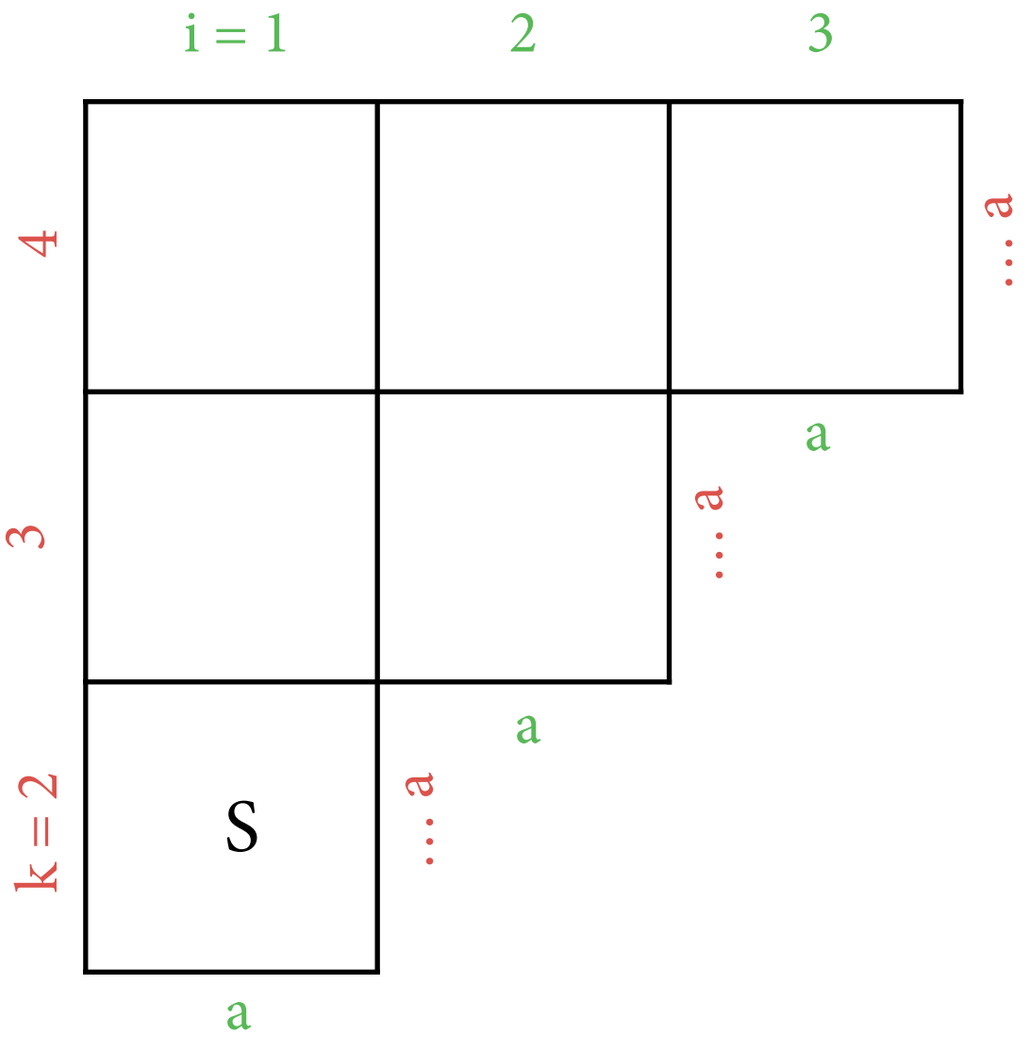
The CKY Recognizer

$S \rightarrow S\ S \qquad S \rightarrow a$



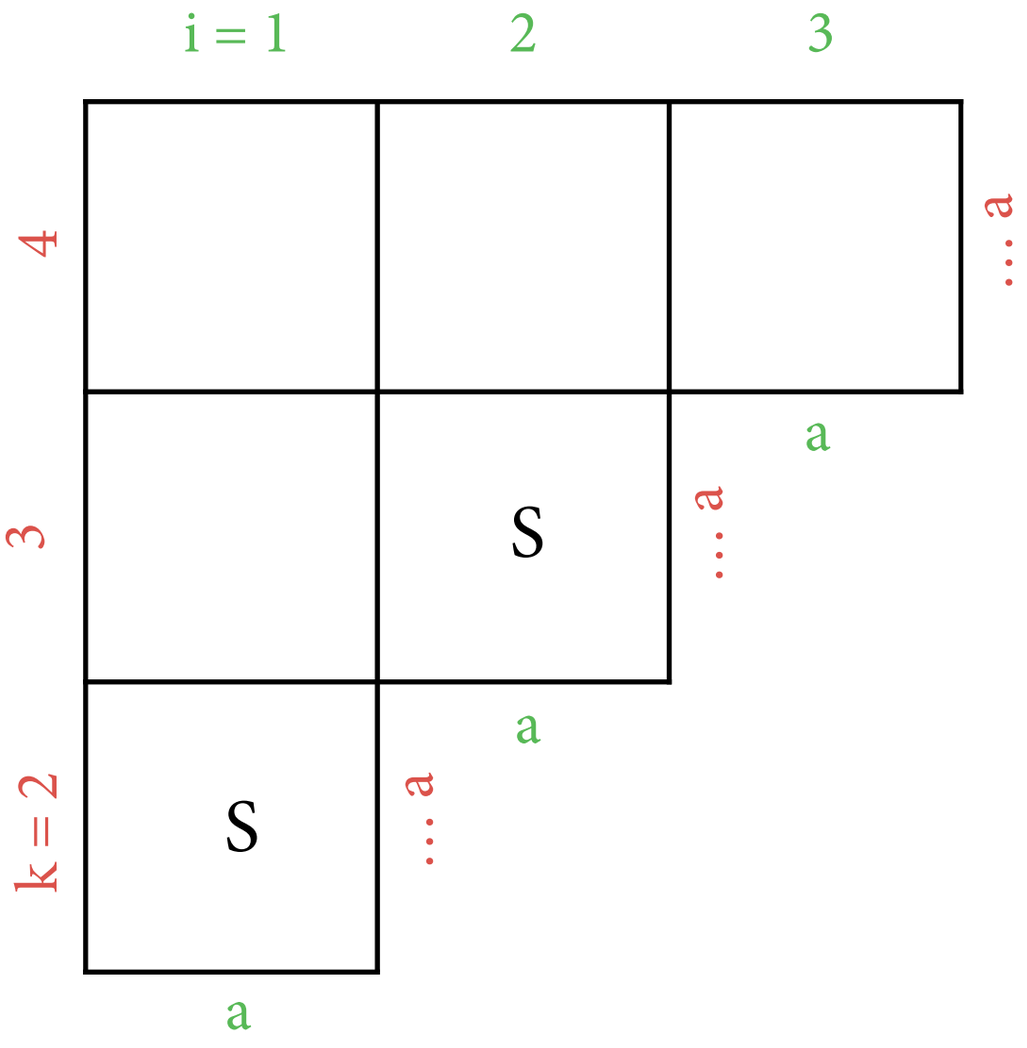
The CKY Recognizer

$S \rightarrow S\ S \qquad S \rightarrow a$



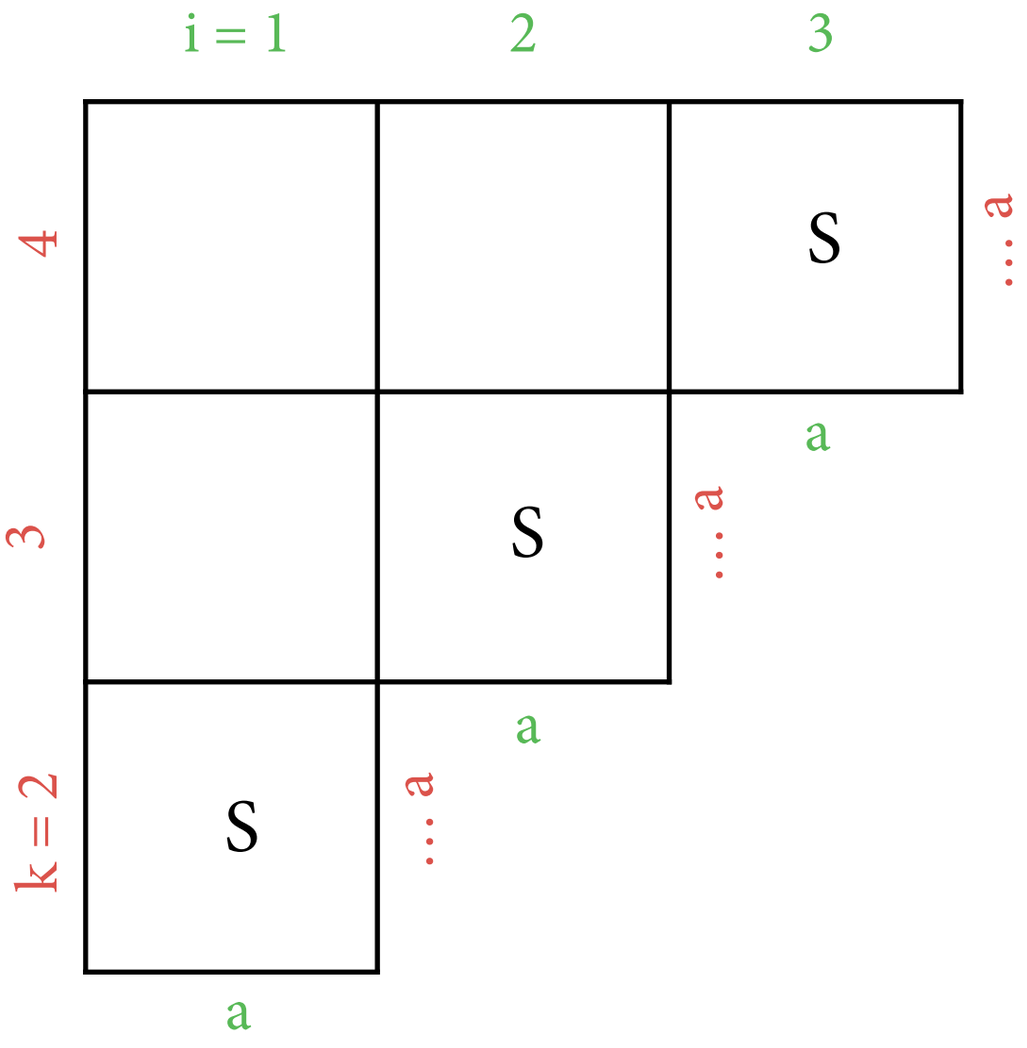
The CKY Recognizer

$S \rightarrow S\ S \qquad S \rightarrow a$



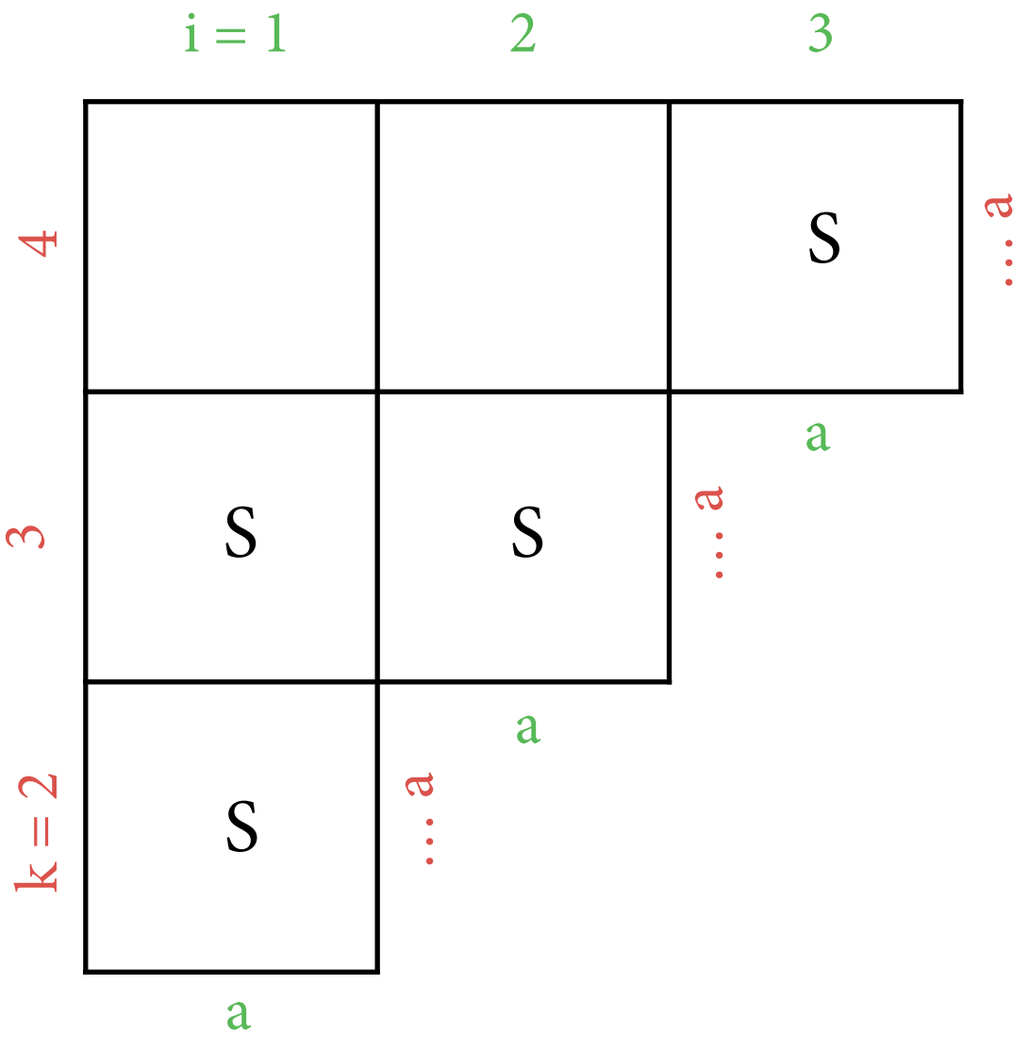
The CKY Recognizer

$S \rightarrow S\ S \qquad S \rightarrow a$



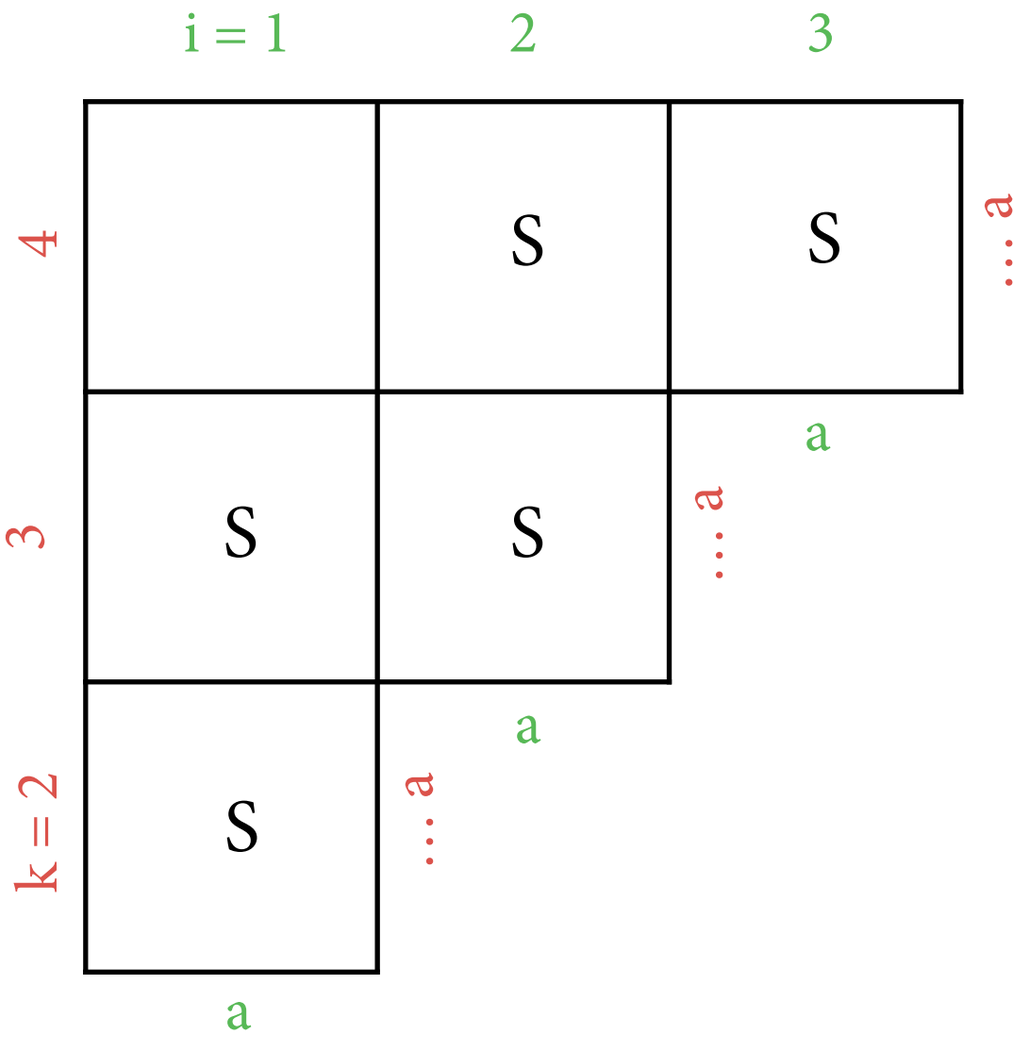
The CKY Recognizer

$S \rightarrow S\ S \qquad S \rightarrow a$



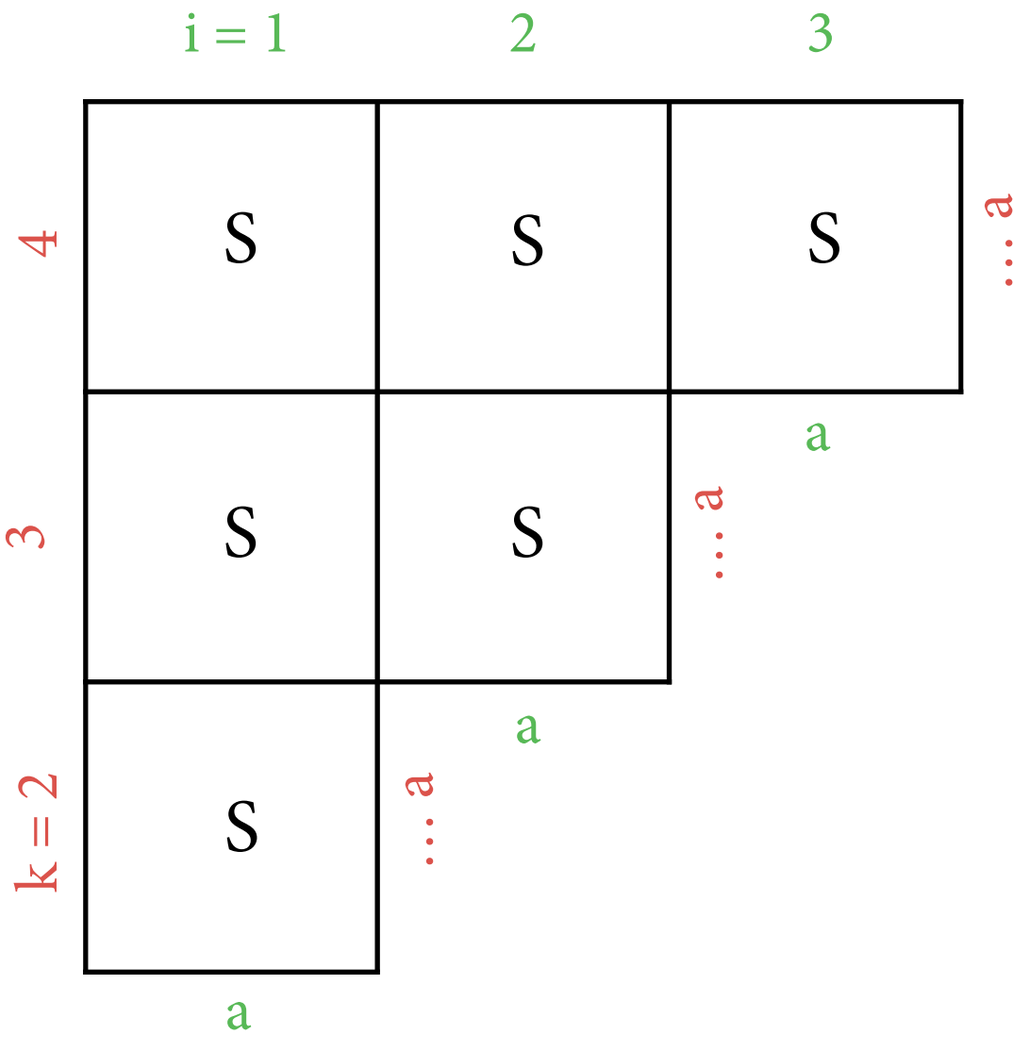
The CKY Recognizer

$S \rightarrow S\ S \qquad S \rightarrow a$



The CKY Recognizer

$S \rightarrow S\ S \qquad S \rightarrow a$



CKY recognizer: pseudocode

Data structure: Ch(i,k) contains $\{A \mid A \Rightarrow^* w_i \ldots w_{k-1}\}$
(initially all empty).

```
for each i from 1 to n:
    for each production rule A → wi:
        add A to Ch(i, i+1)

for each width b from 2 to n:
    for each start position i from 1 to n-b+1:
        for each left width k from 1 to b-1:
            for each B ∈ Ch(i, i+k) and C ∈ Ch(i+k,i+b):
                for each production rule A → B C:
                    add A to Ch(i,i+b)
```

claim that $w \in L(G)$ iff $S \in Ch(1,n+1)$

Complexity

- *Time* complexity of CKY recognizer is $O(n^3)$, although number of parse trees grows exponentially.

- *Space* complexity of CKY recognizer is $O(n^2)$ (one cell for each substring).

- Efficiency depends crucially on CNF.
  Naive generalization of CKY to rules $A \rightarrow B_1 \ldots B_r$ raises time complexity to $O(n^{r+1})$.

Correctness

- *Soundness:* CKY *only* derives true statements.
  - If CKY puts A into Ch(i,k), then there is rule $A \to BC$ and some j with $B \in Ch(i,j)$ and $C \in Ch(j,k)$.
  - Induction hypothesis: for shorter spans, have $B \Rightarrow^* w_i \ldots w_{j-1}$.

    Thus $A \Rightarrow B\ C \Rightarrow^* w_i \ldots w_{j-1}\ C \Rightarrow^* w_i \ldots w_{k-1}$

- Completeness: CKY derives *all* true statements.
  - Each derivation $A \Rightarrow^* w_i \ldots w_{k-1}$ starts with a first step;

    say $A \Rightarrow B\ C \Rightarrow^* w_i \ldots w_{j-1}\ C \Rightarrow^* w_i \ldots w_{k-1}$
  - Important: ensure that all nonterminals for shorter spans are known before filling Ch(i,k).

Recognizer to Parser

- Parser: need to construct parse trees from chart.
- Do this by memorizing how each $A \in Ch(i,k)$ can be constructed from smaller parts.
  - built from $B \in Ch(i,j)$ and $C \in Ch(j,k)$ using $A \rightarrow B\ C$: store (B,C,j) in *backpointer* for A in Ch(i,k).
  - analogous to backpointers in HMMs
- Once chart has been filled, enumerate trees recursively by following backpointers, starting at $S \in Ch(1,n+1)$.

Conclusion

- Context-free grammars: most popular grammar formalism in NLP.
  - there are also other, more expressive grammar formalisms
- CKY: most popular parser for cfgs.
  - there are also other, more complicated algorithms
- Next week: put parsing and statistics together.