The CKY Parser

Computational Linguistics

Alexander Koller

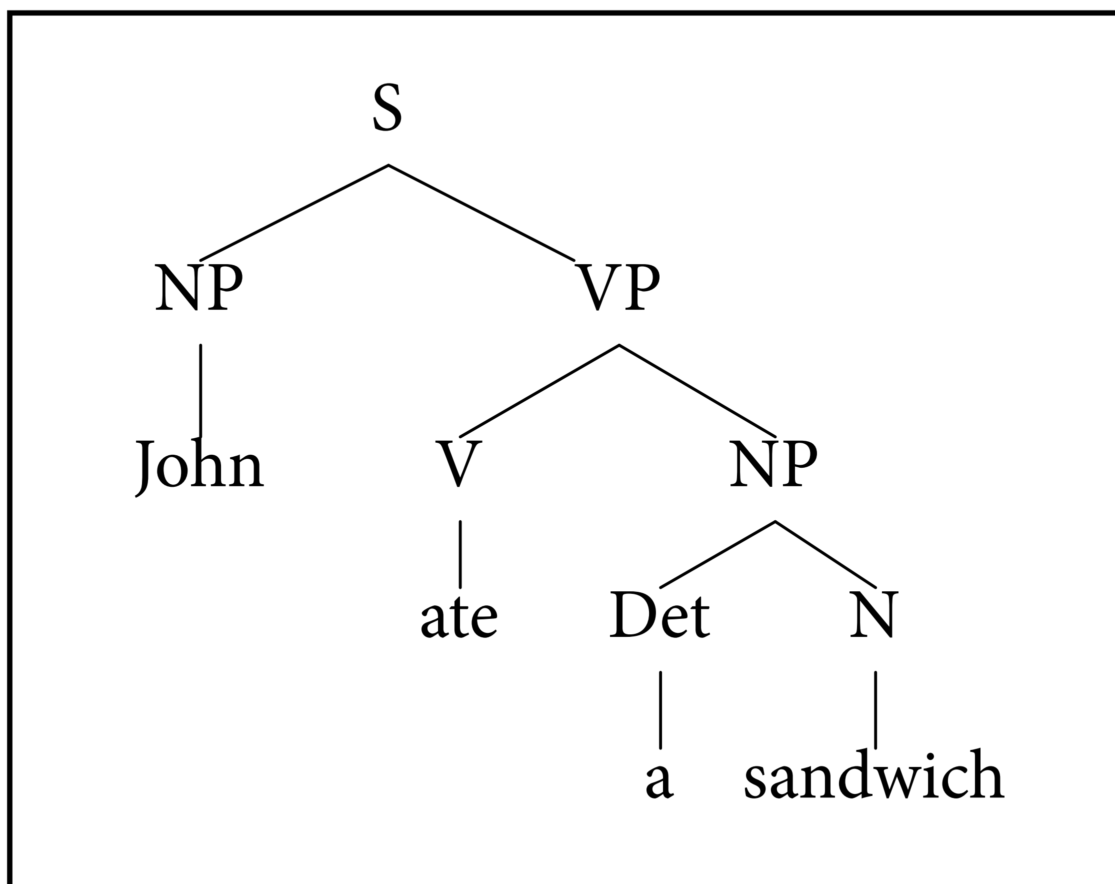
20 November 2018

Context-free grammars

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

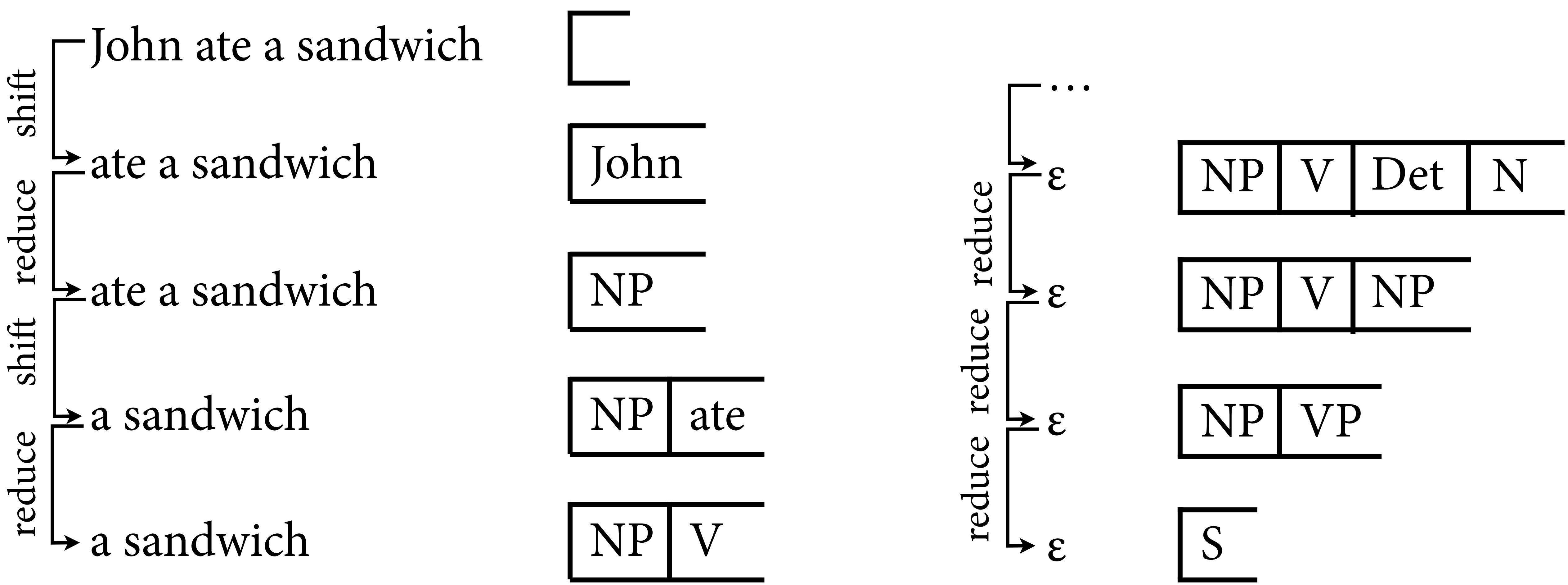


Shift-Reduce Parsing

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



Runtime of algorithms

- It is not enough to find an algorithm that is sound and complete. It should also be *efficient*.

- Runtime of an algorithm is measured:
  - as a function of input size n
  - for the worst case (= inputs of that size on which the algorithm runs longest)
  - asymptotically (= ignore constant factors)

A simple example

- Problem: test whether list of numbers is sorted.
  - given list L of ints of length n:
  - are there indices $1 \le i < j \le n$ s.t. $L_i > L_j$?
- Let's look at two algorithms for this problem.

Runtime comparison

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

Runtime

len(L)	quadratic	linear
100		
1000		
10000		
100.000		
1.000.000		

Runtime comparison

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	
1000		
10000		
100.000		
1.000.000		

Runtime comparison

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	
1000	40 ms	
10000		
100.000		
1.000.000		

Runtime comparison

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	
1000	40 ms	
10000	4.5 sec	
100.000		
1.000.000		

Runtime comparison

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	
1000	40 ms	
10000	4.5 sec	
100.000	464 sec	
1.000.000		

Runtime comparison

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

```
def linear_issorted(L):
    for i in range(len(L)-1):
        if L[i] > L[i+1]:
            return False
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	
1000	40 ms	
10000	4.5 sec	
100.000	464 sec	
1.000.000		

Runtime comparison

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

```
def linear_issorted(L):
    for i in range(len(L)-1):
        if L[i] > L[i+1]:
            return False
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	0.02 ms
1000	40 ms	
10000	4.5 sec	
100.000	464 sec	
1.000.000		

Runtime comparison

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

```
def linear_issorted(L):
    for i in range(len(L)-1):
        if L[i] > L[i+1]:
            return False
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	0.02 ms
1000	40 ms	0.1 ms
10000	4.5 sec	
100.000	464 sec	
1.000.000		

Runtime comparison

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

```
def linear_issorted(L):
    for i in range(len(L)-1):
        if L[i] > L[i+1]:
            return False
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	0.02 ms
1000	40 ms	0.1 ms
10000	4.5 sec	1.2 ms
100.000	464 sec	
1.000.000		

Runtime comparison

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

```
def linear_issorted(L):
    for i in range(len(L)-1):
        if L[i] > L[i+1]:
            return False
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	0.02 ms
1000	40 ms	0.1 ms
10000	4.5 sec	1.2 ms
100.000	464 sec	13 ms
1.000.000		

Runtime comparison

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

```
def linear_issorted(L):
    for i in range(len(L)-1):
        if L[i] > L[i+1]:
            return False
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	0.02 ms
1000	40 ms	0.1 ms
10000	4.5 sec	1.2 ms
100.000	464 sec	13 ms
1.000.000		179 ms

Runtime comparison

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

```
def linear_issorted(L):
    for i in range(len(L)-1):
        if L[i] > L[i+1]:
            return False
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	0.02 ms
1000	40 ms	0.1 ms
10000	4.5 sec	1.2 ms
100.000	464 sec	13 ms
1.000.000		179 ms

$\approx n \cdot 120$ ns

Runtime comparison

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

```
def linear_issorted(L):
    for i in range(len(L)-1):
        if L[i] > L[i+1]:
            return False
    return True
```

Runtime

len(L)	quadratic	linear
100	0.5 ms	0.02 ms
1000	40 ms	0.1 ms
10000	4.5 sec	1.2 ms
100.000	464 sec	13 ms
1.000.000		179 ms

$\approx n^2 \cdot 45$ ns

$\approx n \cdot 120$ ns

Analysis

- Important parameters:
  - input size n = len(L), i.e. length of list
  - worst case = L is sorted; every loop iterated n times
  - don't really care about time per iteration, linear is always faster if n grows large enough
- We can get a good sense of the algorithm's runtime by saying it grows *linearly* or *quadratically* with n.
  - abstraction over implementation details and hardware
  - *asymptotic* comparison of runtime classes

O Notation

- Let f, g be functions. Then we define:

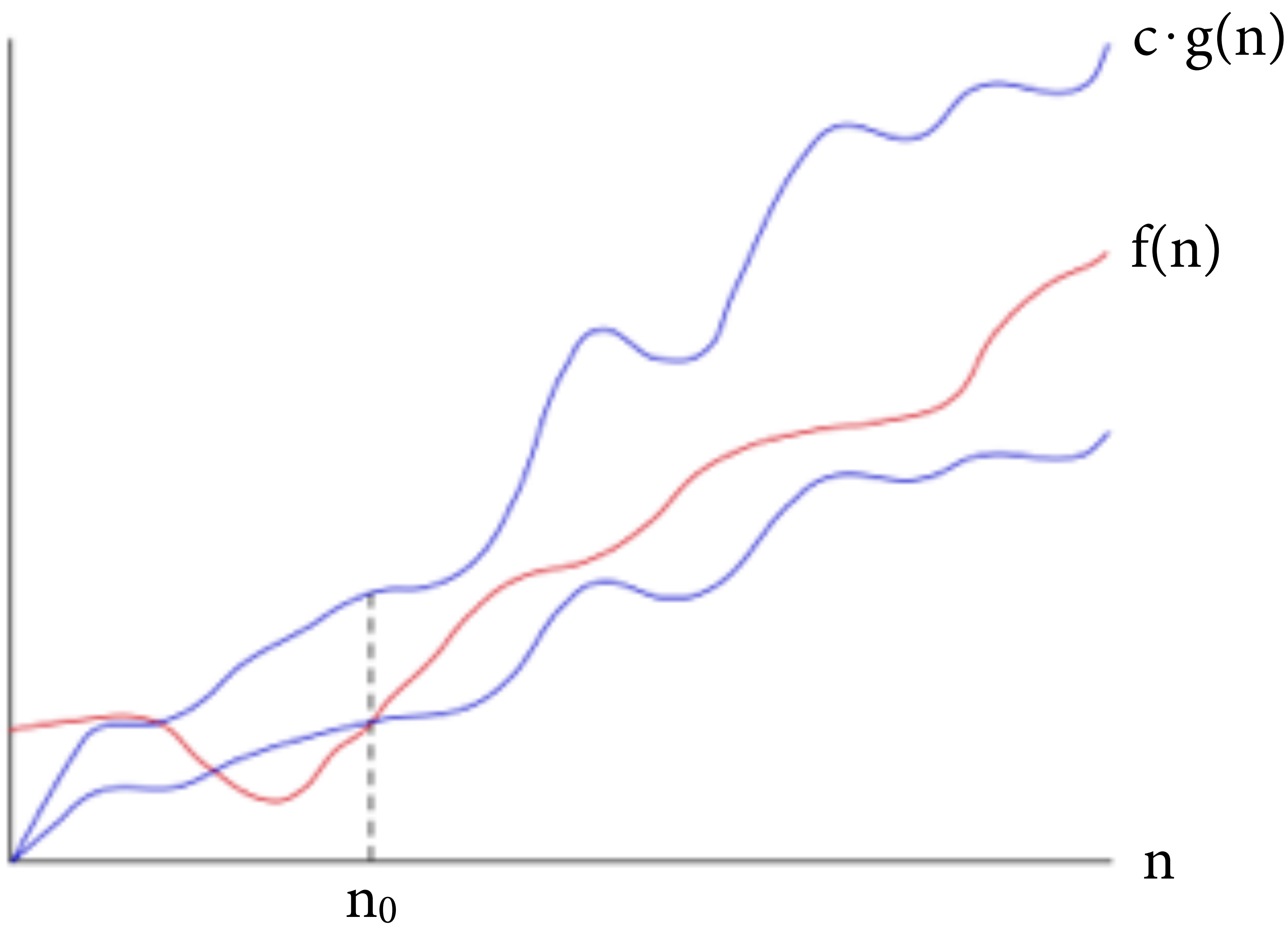
$f = O(g)$ iff exist c, n_0 s.t. $f(n) \le c \cdot g(n)$ f.a. $n \ge n_0$

- Read “f is O of g”; “=“ denotes membership in a runtime class, not equality.

- Usually take the smallest g such that $f = O(g)$.

Illustration

$f = O(g)$ iff
exist c, n_0 s.t. $f(n) \leq c \cdot g(n)$ f.a. $n \geq n_0$



Back to the example

$f = O(g)$ iff
exist c, n_0 s.t. $f(n) \leq c \cdot g(n)$ f.a. $n \geq n_0$

```
def quadratic_issorted(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[j] < L[i]:
                return False
    return True
```

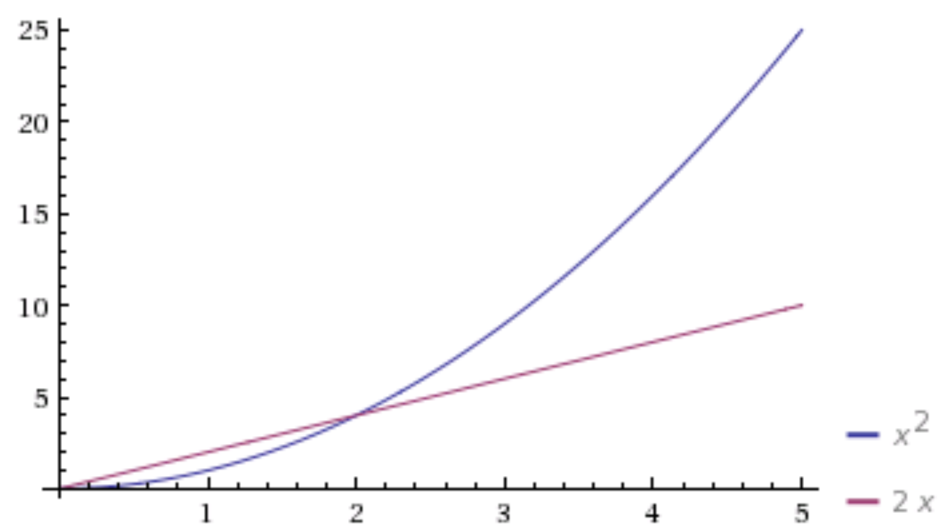
Runtime $f(n) \approx n^2 \cdot 45$ ns $= O(n^2)$
"quadratic algorithm"

```
def linear_issorted(L):
    for i in range(len(L)-1):
        if L[i] > L[i+1]:
            return False
    return True
```

Runtime $f(n) \approx n \cdot 120$ ns $= O(n)$
"linear algorithm"

Hierarchy of runtime classes

- For all c, c', we have $c \cdot n \leq c' \cdot n^2$ after a certain point:



- For large n, low-rank polynomials are faster:
  - O(n) linear < $O(n^2)$ quadratic
    (even for n + 5, 100 · n - 27 etc.)
  - $O(n^2)$ quadratic < $O(n^3)$ cubic
  - etc.

Analyzing Shift-Reduce

S → B S	B → b	S → c
T → C T	C → b	T → c

b b b c

Analyzing Shift-Reduce

$S \to B\ S$	$B \to b$	$S \to c$
$T \to C\ T$	$C \to b$	$T \to c$

b b b c

$\to^*$ C C C T $\to^*$ T ✗

Analyzing Shift-Reduce

$S \rightarrow B\ S$	$B \rightarrow b$	$S \rightarrow c$
$T \rightarrow C\ T$	$C \rightarrow b$	$T \rightarrow c$

$b \quad b \quad b \quad c$

$\rightarrow^* \quad C \quad C \quad C \quad T \quad \rightarrow^* \quad T$ ✗

$\rightarrow^* \quad C \quad C \quad B \quad T$ ✗

Analyzing Shift-Reduce

S → B S	B → b	S → c
T → C T	C → b	T → c

	b	b	b	c		
→*	C	C	C	T	→* T ✗	
→*	C	C	B	T ✗		
→*	C	B	C	T	→* C B T ✗	

Analyzing Shift-Reduce

S → B S	B → b	S → c
T → C T	C → b	T → c

b b b c

→* C C C T →* T ✗

→* C C B T ✗

→* C B C T →* C B T ✗

→* C B B T ✗

Analyzing Shift-Reduce

S → B S	B → b	S → c
T → C T	C → b	T → c

	b	b	b	c	
→*	C	C	C	T	→* T ✗
→*	C	C	B	T ✗	
→*	C	B	C	T	→* C B T ✗
→*	C	B	B	T ✗	
…					
→*	B	B	B	S	

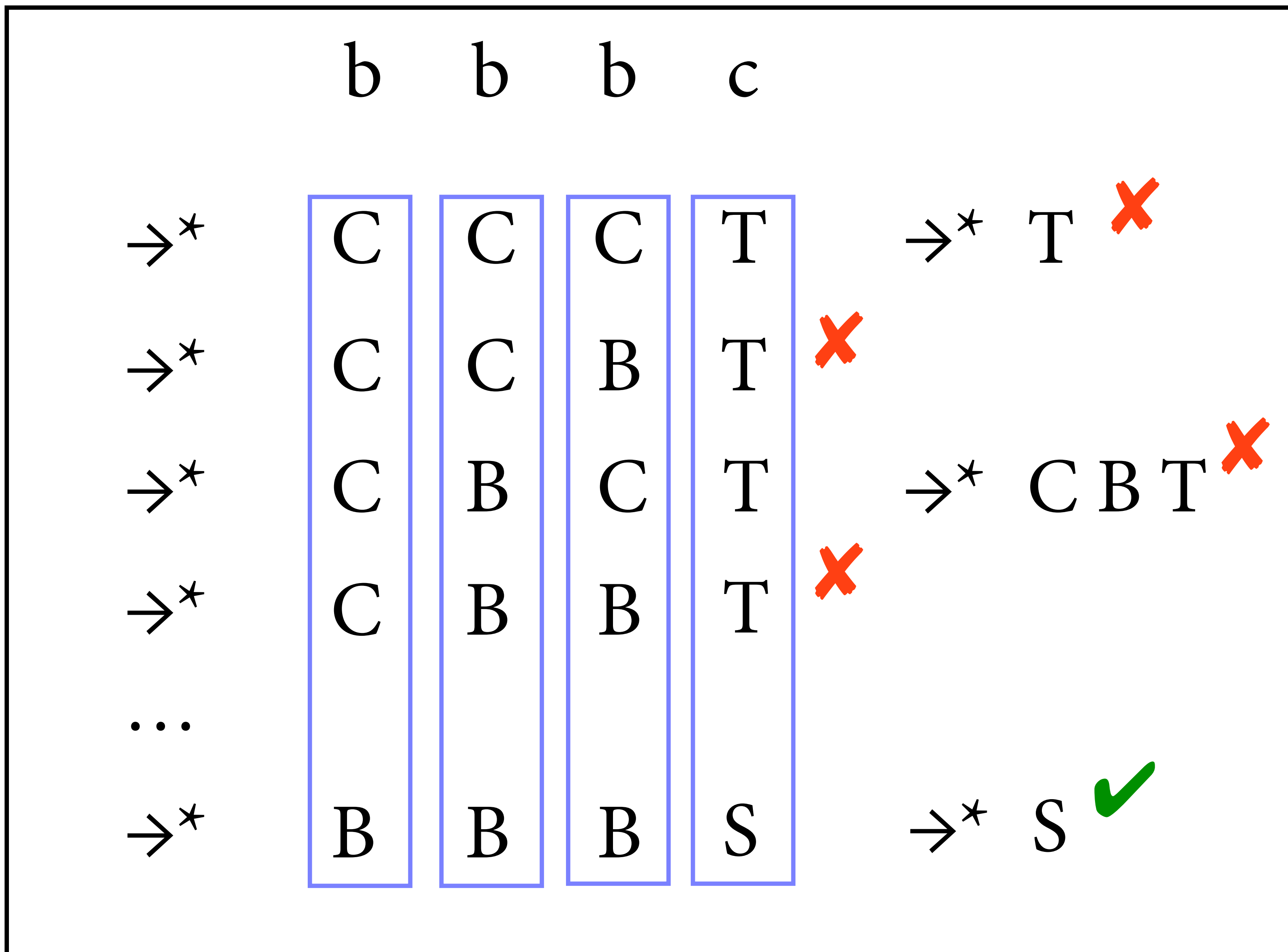
Analyzing Shift-Reduce

S → B S	B → b	S → c
T → C T	C → b	T → c

	b	b	b	c		
→*	C	C	C	T	→* T ✗	
→*	C	C	B	T ✗		
→*	C	B	C	T	→* C B T ✗	
→*	C	B	B	T ✗		
…						
→*	B	B	B	S	→* S ✔	

Analyzing Shift-Reduce

S → B S   B → b   S → c
T → C T   C → b   T → c



Analyzing Shift-Reduce

$S \rightarrow B\ S$ $\quad B \rightarrow b$

$T \rightarrow C\ T$ $\quad C \rightarrow b$ $\quad T \rightarrow c$

b b b c

Analyzing Shift-Reduce

S → B S	B → b	
T → C T	C → b	T → c

b b b c

→* C C C T →* T ✘

Analyzing Shift-Reduce

S → B S	B → b	
T → C T	C → b	T → c

b b b c

→* C C C T →* T ✗

→* C C B T ✗

Analyzing Shift-Reduce

S → B S	B → b	
T → C T	C → b	T → c

	b	b	b	c	
→*	C	C	C	T	→* T ✗
→*	C	C	B	T ✗	
→*	C	B	C	T	→* C B T ✗

Analyzing Shift-Reduce

S → B S	B → b	
T → C T	C → b	T → c

	b	b	b	c		
→*	C	C	C	T	→* T ✗	
→*	C	C	B	T ✗		
→*	C	B	C	T	→* C B T ✗	
→*	C	B	B	T ✗		

Analyzing Shift-Reduce

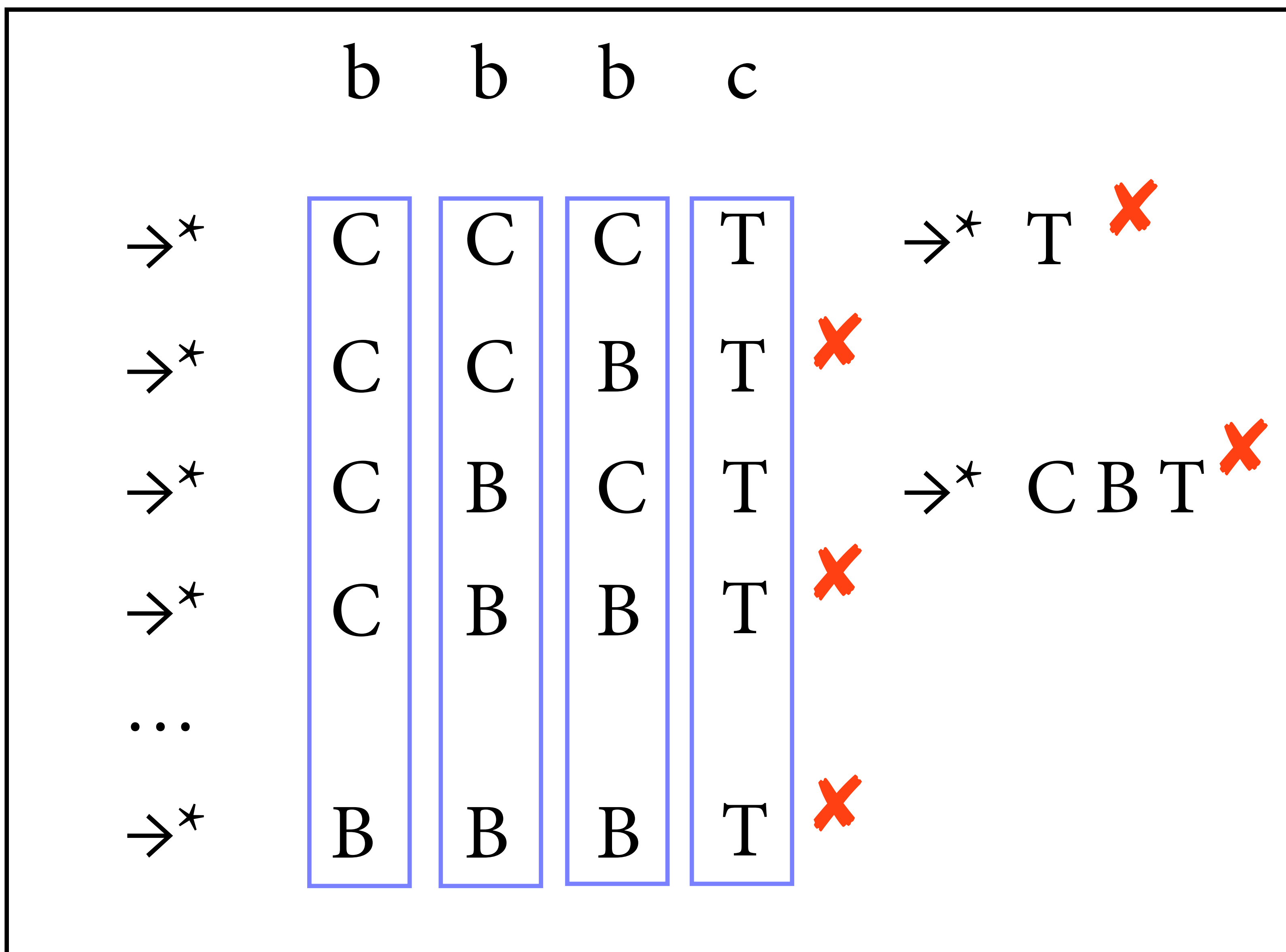
$S \rightarrow B\ S$	$B \rightarrow b$	
$T \rightarrow C\ T$	$C \rightarrow b$	$T \rightarrow c$

	b	b	b	c		
$\rightarrow^*$	C	C	C	T	$\rightarrow^*$ T ✗	
$\rightarrow^*$	C	C	B	T ✗		
$\rightarrow^*$	C	B	C	T	$\rightarrow^*$ C B T ✗	
$\rightarrow^*$	C	B	B	T ✗		
…						
$\rightarrow^*$	B	B	B	T ✗		

Analyzing Shift-Reduce

S → B S   B → b

T → C T   C → b   T → c



Analyzing Shift-Reduce

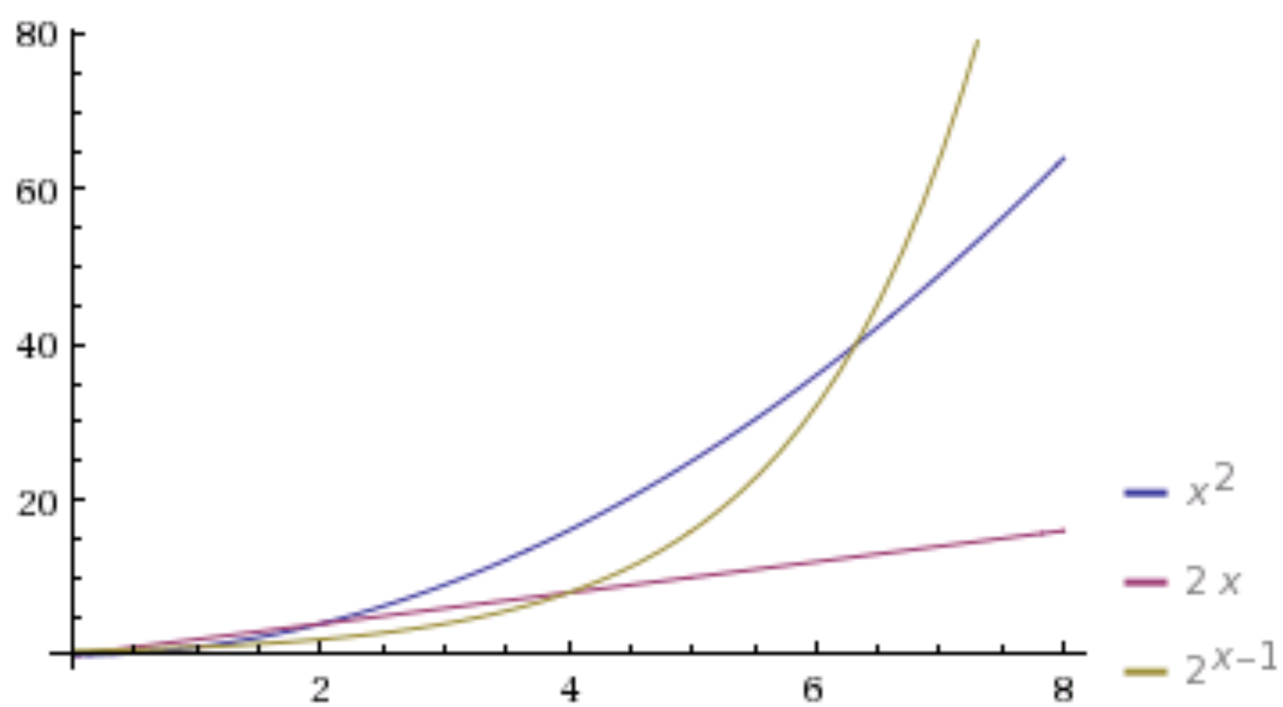
- If string has length n and grammar has k nonterminals, then there are $O(k^n)$ ways of assigning strings of nonterminals to words.

- These can all be explored, especially when the string is *not* in the language.

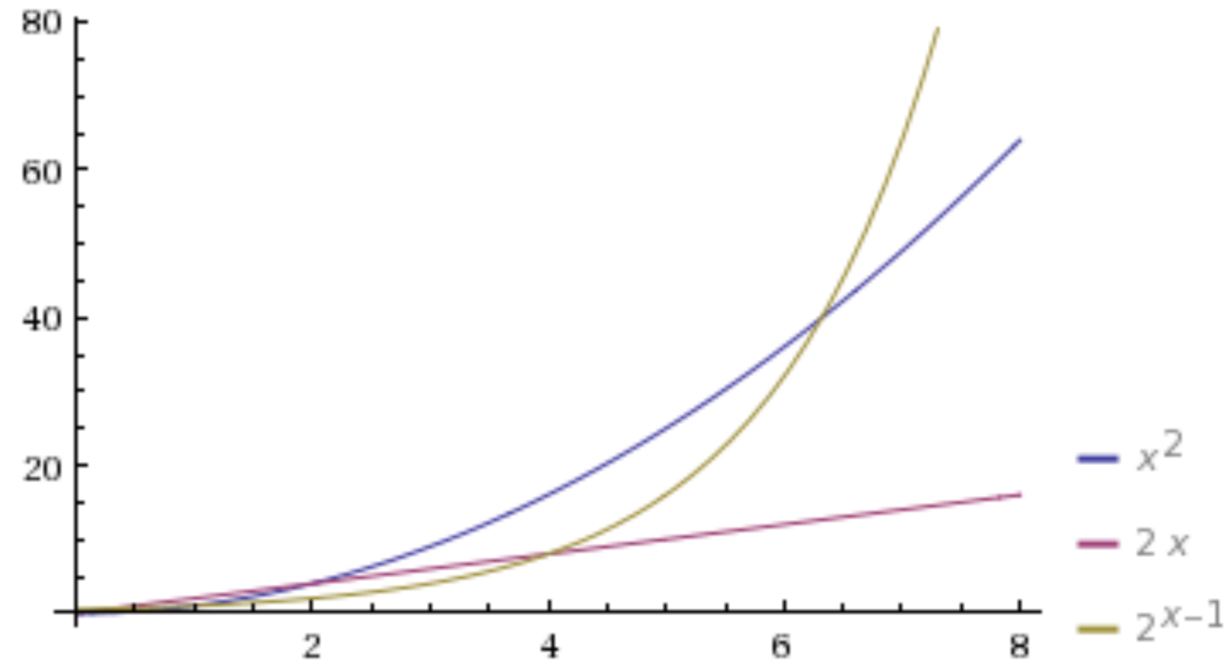
Exponential runtime

- Worst case runtime of shift-reduce: roughly k^n computation steps.

- Exponential functions grow faster than every polynomial: if $k > 1$, then there is no m such that $k^n = O(n^m)$.



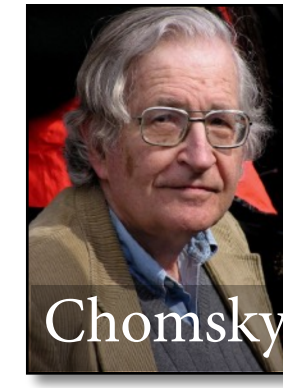
Polynomial vs. exponential



- We often distinguish between *polynomial* and *exponential* runtime. Rule of thumb: exponential = too slow for practical use.

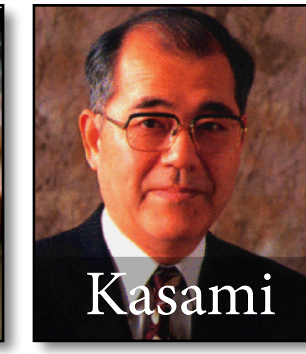
- Is there a polynomial algorithm for the word problem?

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

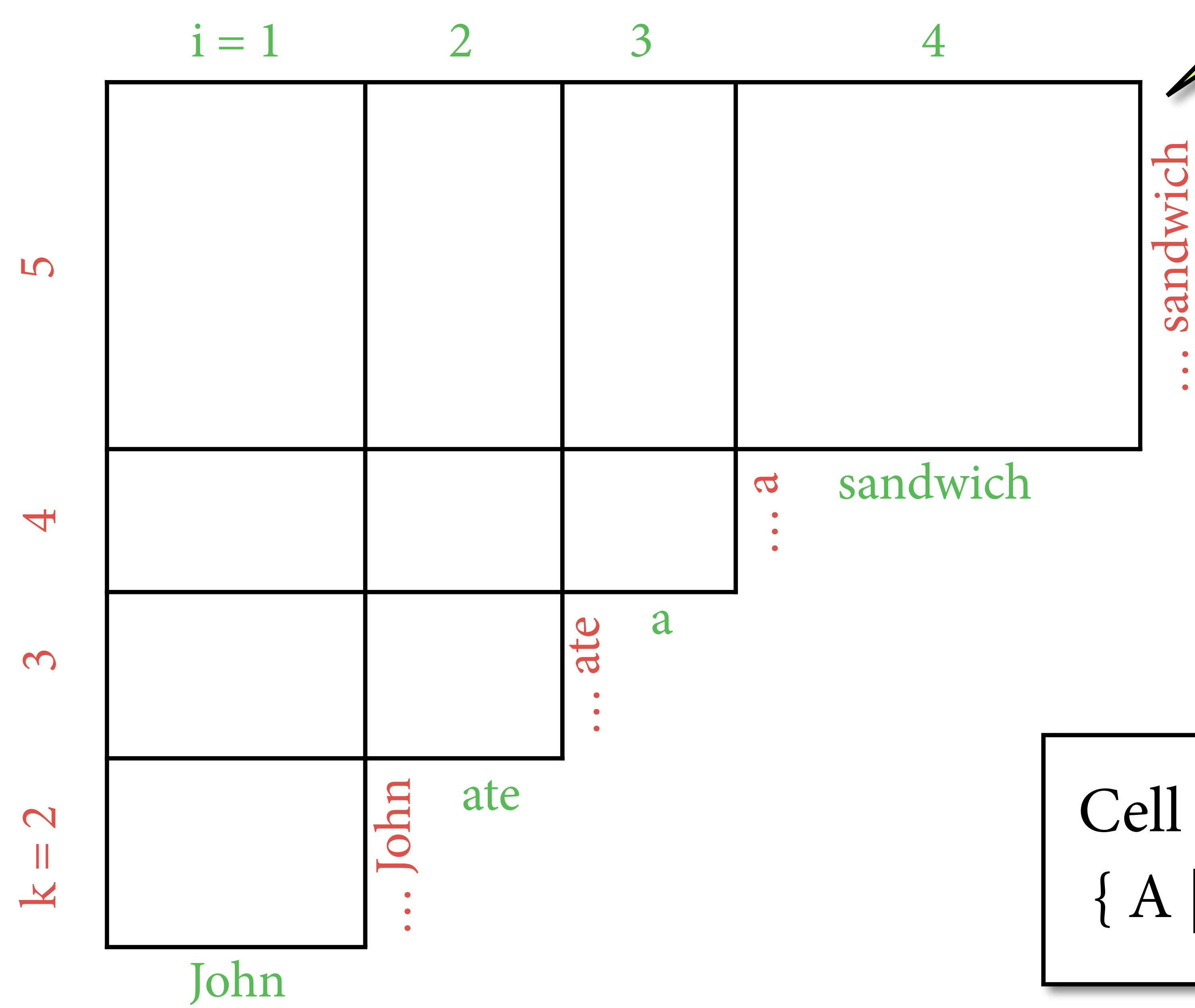


- Simplest and most-used chart parser for cfgs in CNF.
- Developed independently in the 1960s by John Cocke, Daniel Younger, and Tadao Kasami.
  - sometimes also called CYK algorithm
- Bottom-up algorithm for discovering statements of the form "$A \Rightarrow^* w_i \ldots w_{k-1}$?"

The CKY Recognizer

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

Chart



Cell at column i, row k:

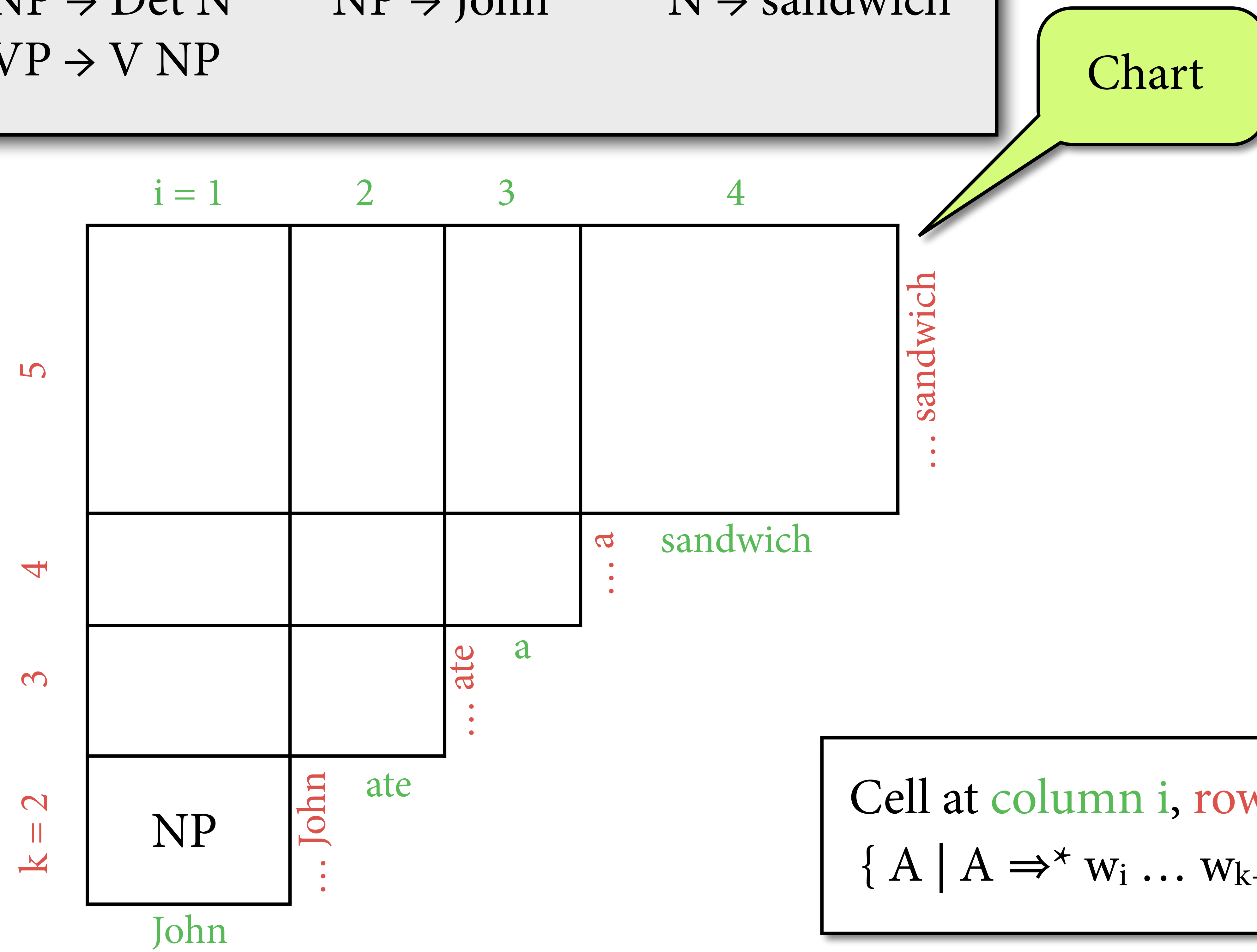
$\{ A \mid A \Rightarrow^* w_i \ldots w_{k-1} \}$

The CKY Recognizer

S → NP VP
NP → Det N
VP → V NP

V → ate
NP → John

Det → a
N → sandwich

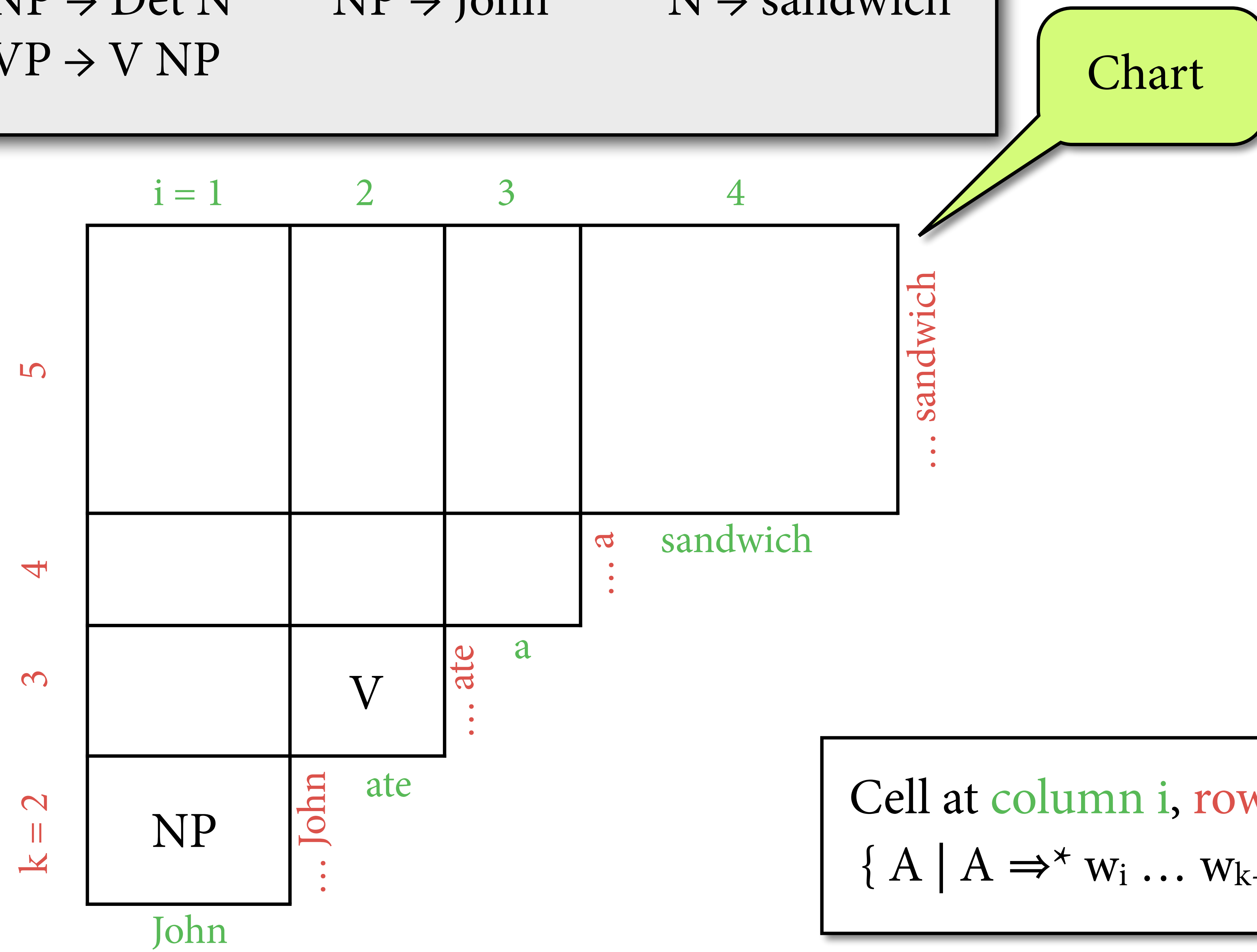


Cell at column i, row k:

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

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:

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

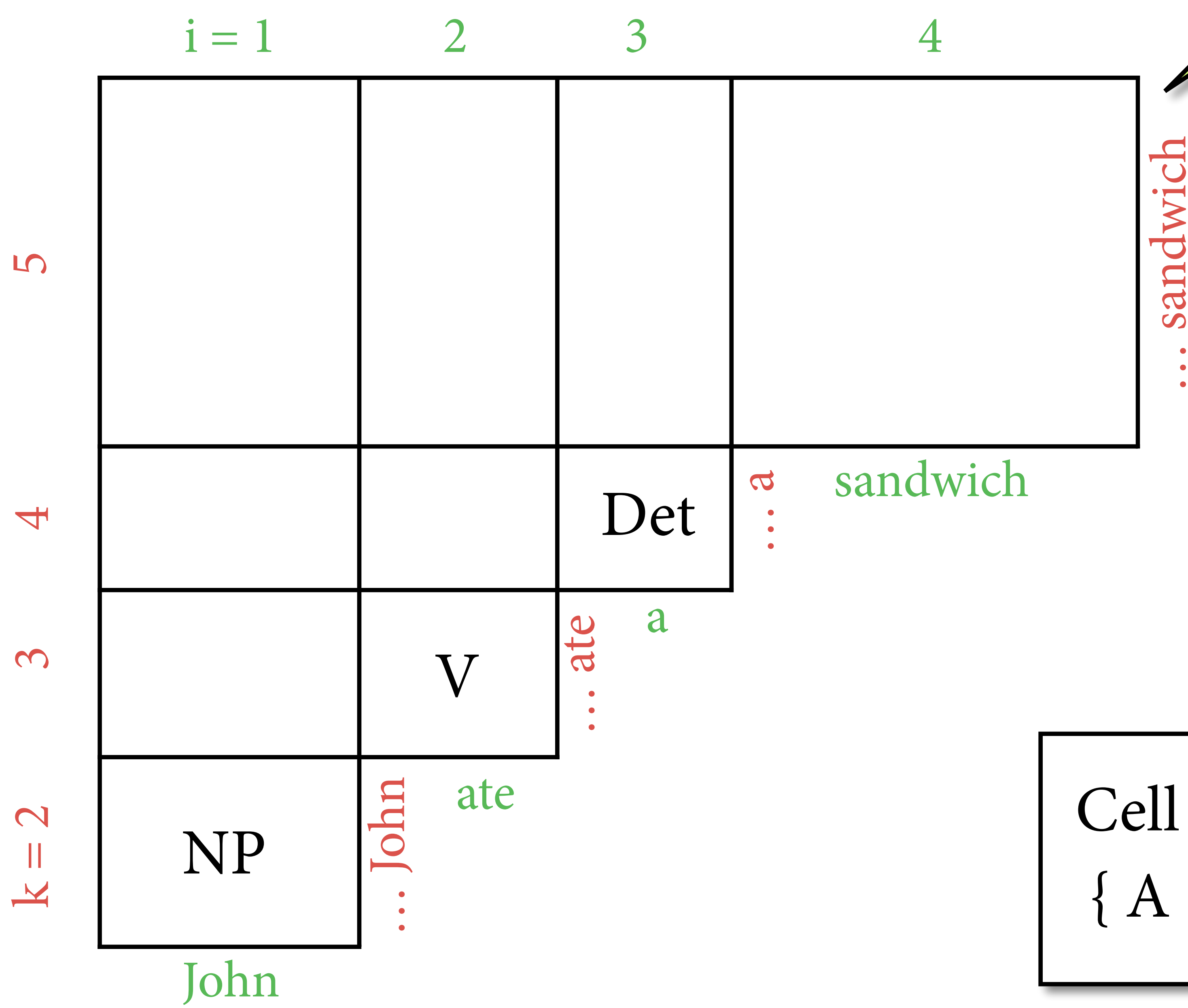
The CKY Recognizer

S → NP VP
NP → Det N
VP → V NP

V → ate
NP → John

Det → a
N → sandwich

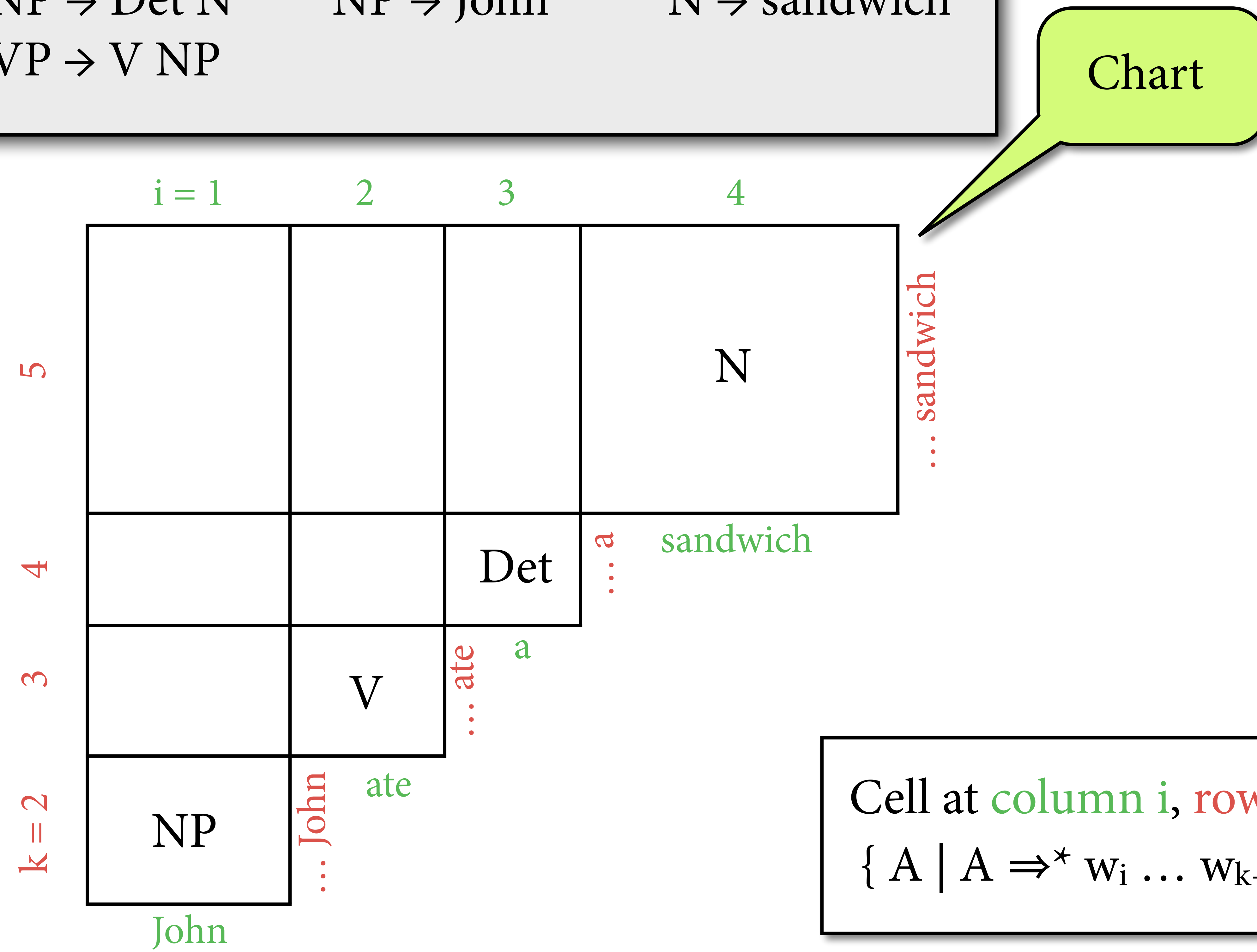
Chart



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		

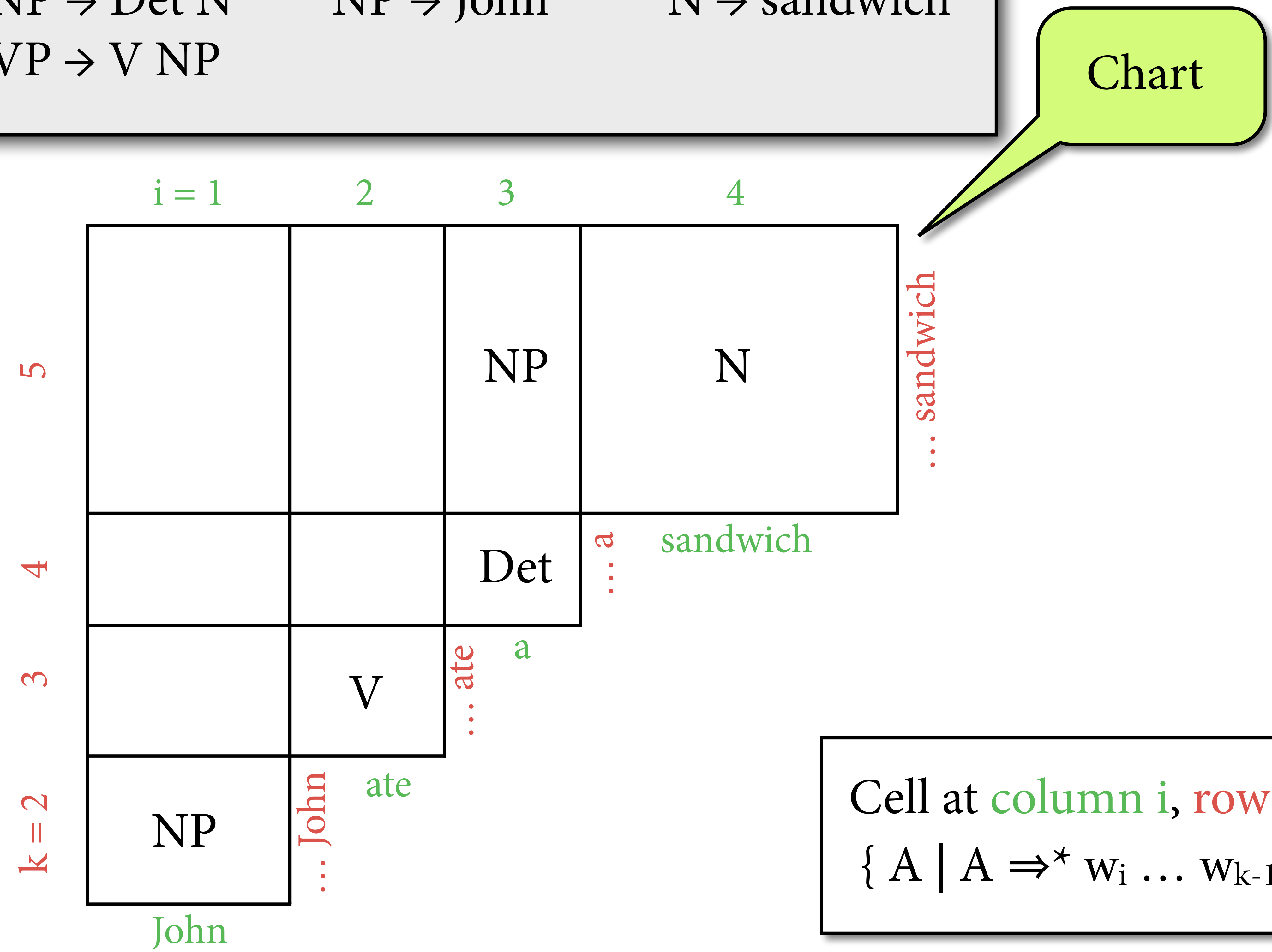


Cell at column i, row k:

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

The CKY Recognizer

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

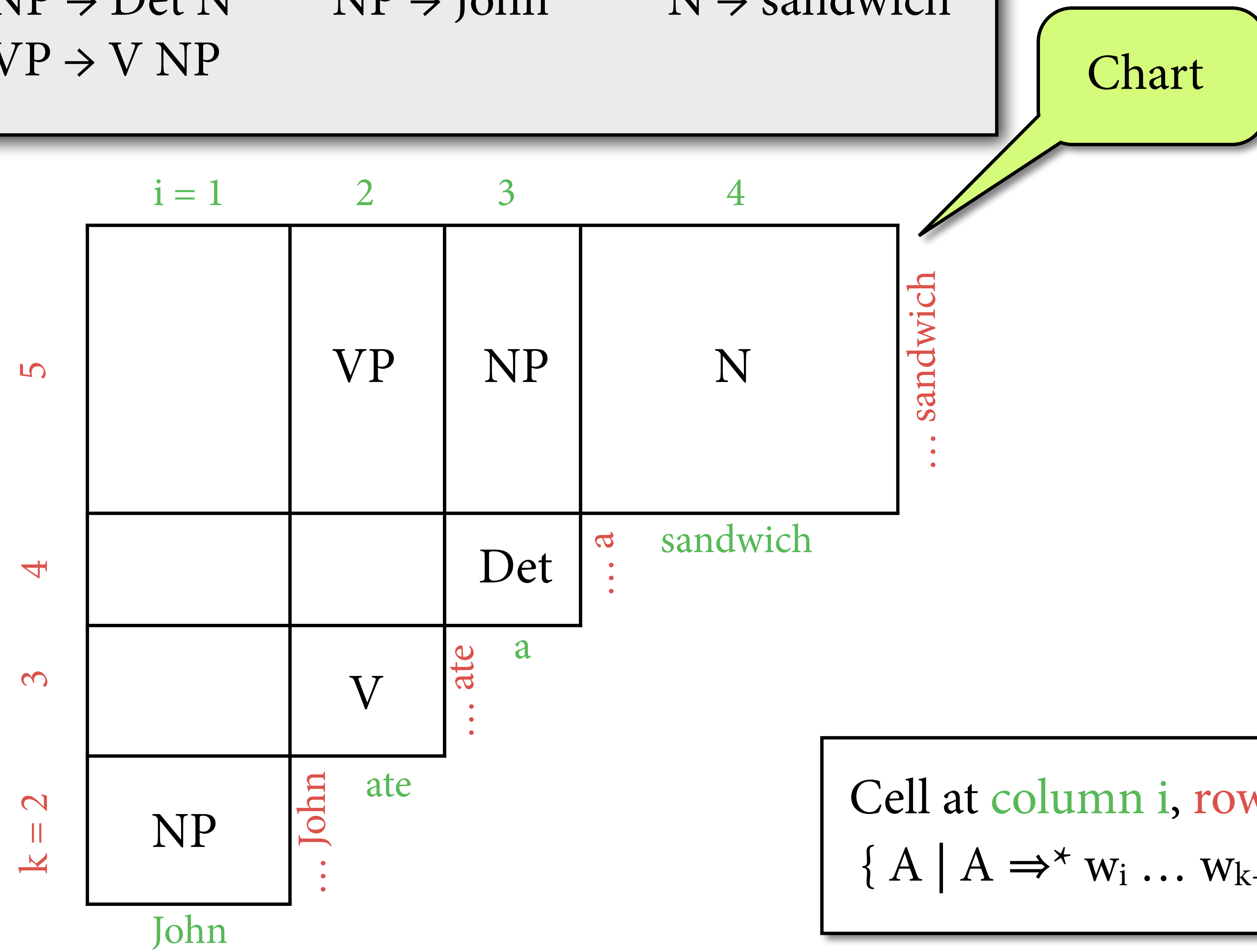


Cell at column i, row k:

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

The CKY Recognizer

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



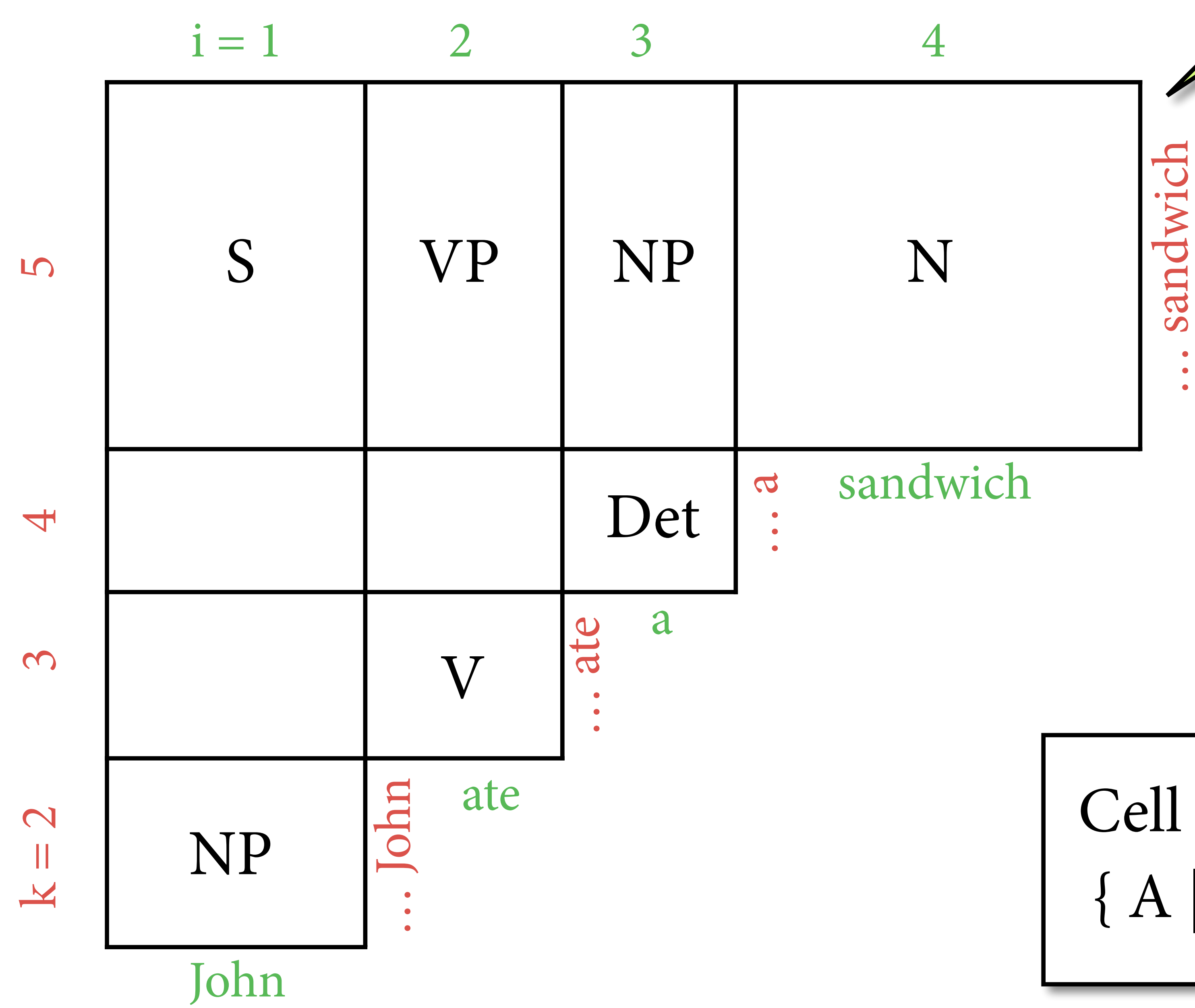
Cell at column i, row k:

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

The CKY Recognizer

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

Chart

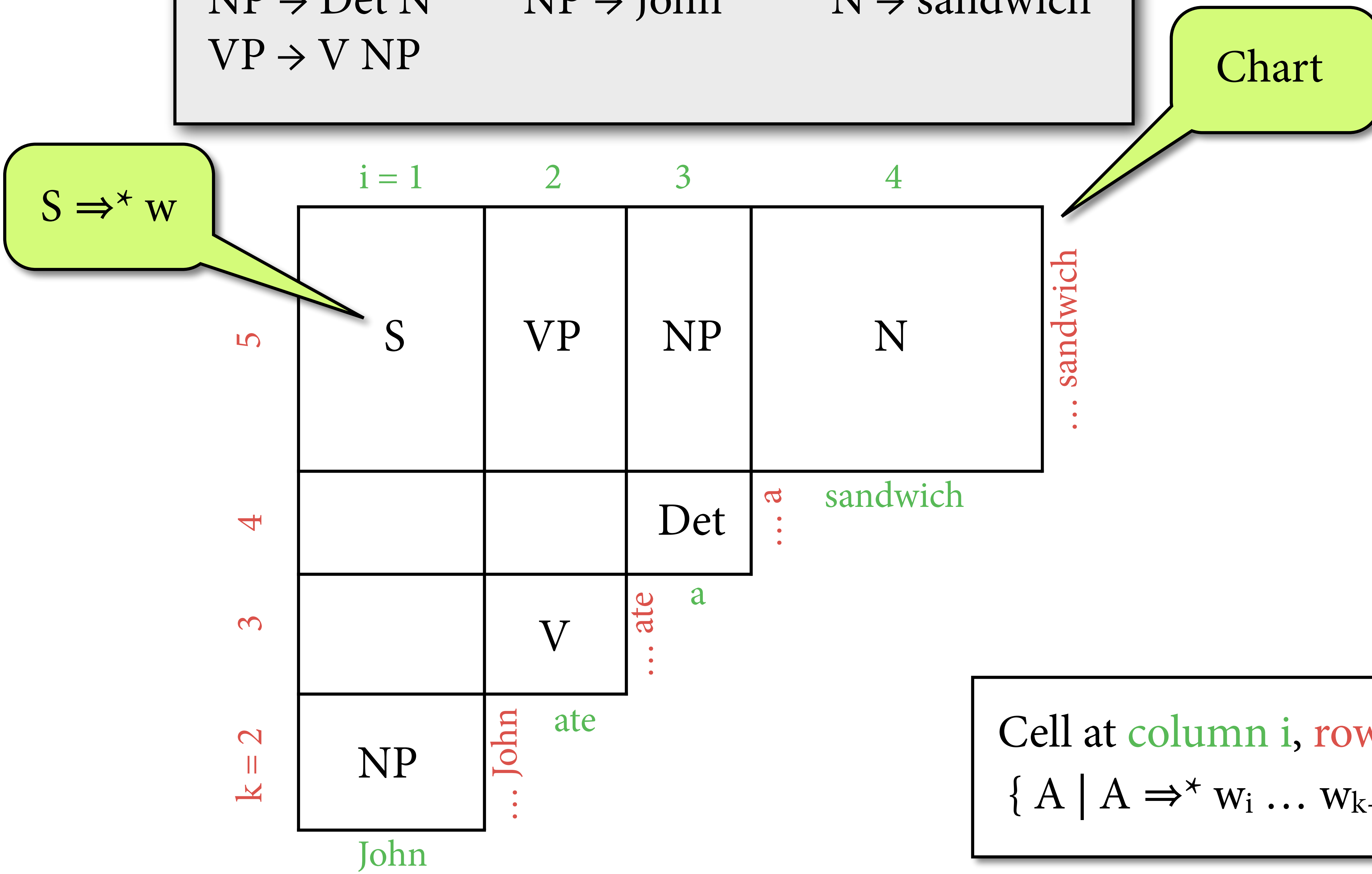


Cell at column i, row k:

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

The CKY Recognizer

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

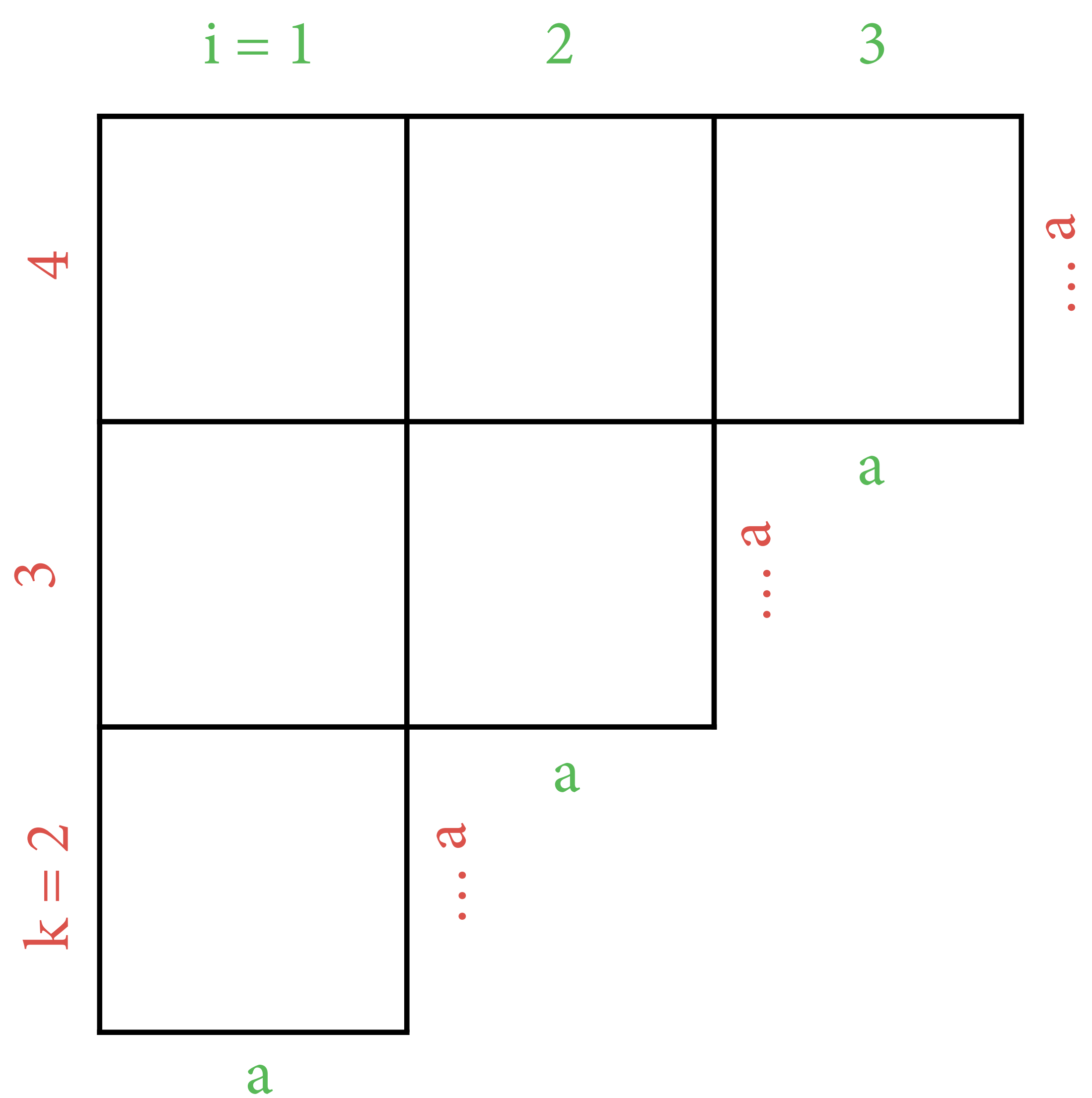


Cell at column i, row k:

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

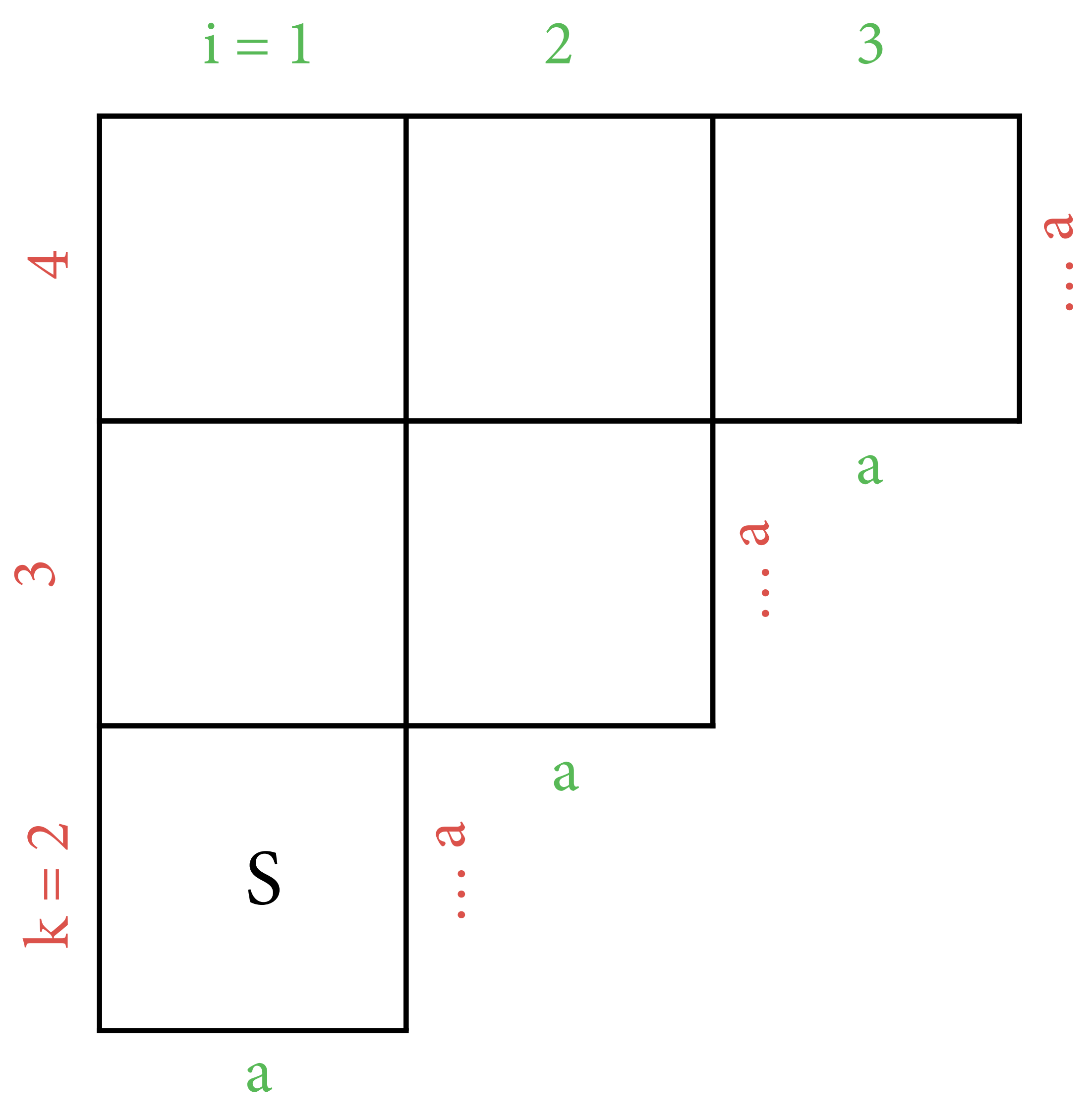
The CKY Recognizer

S → S S   S → a



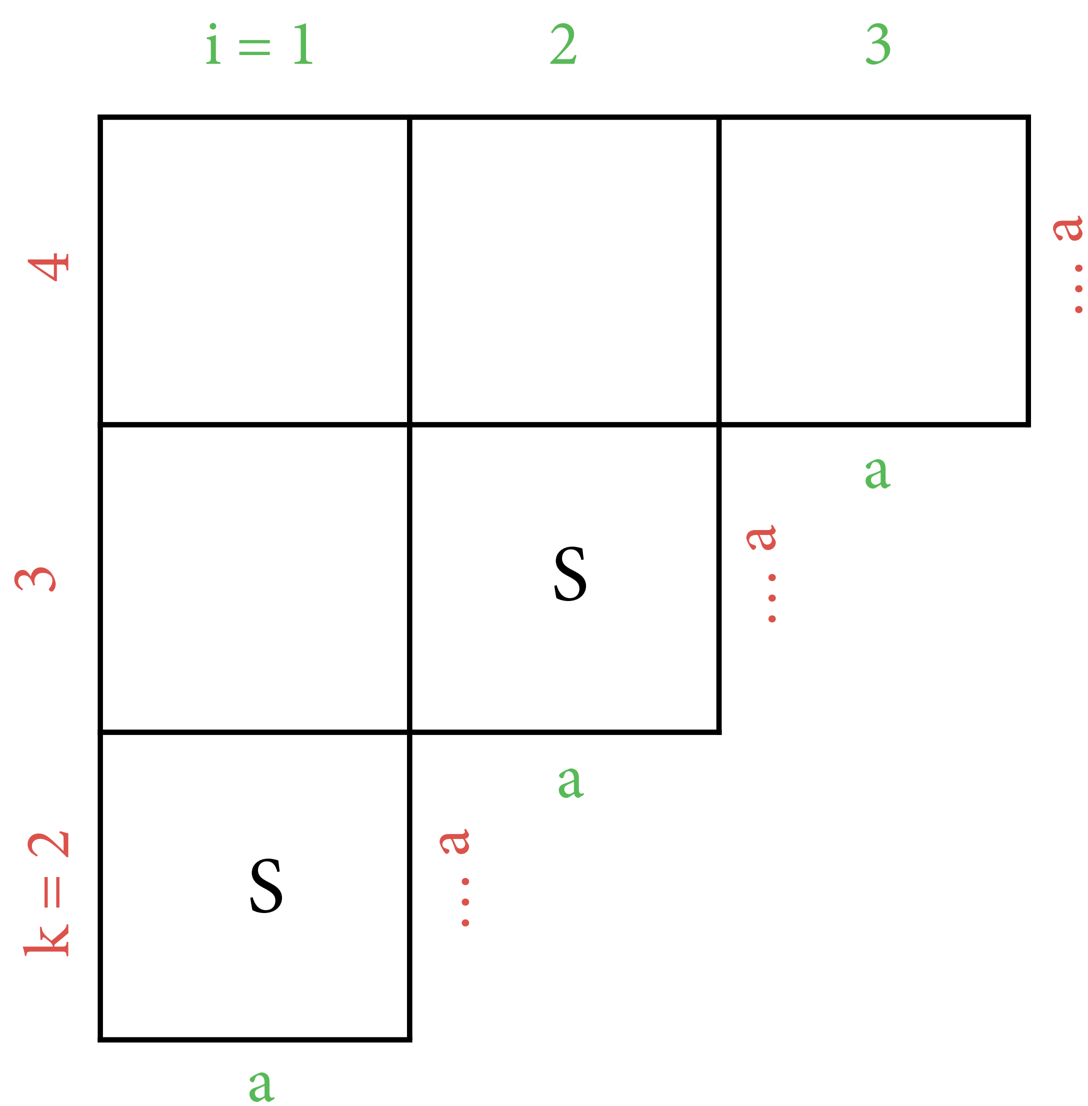
The CKY Recognizer

S → S S S → a



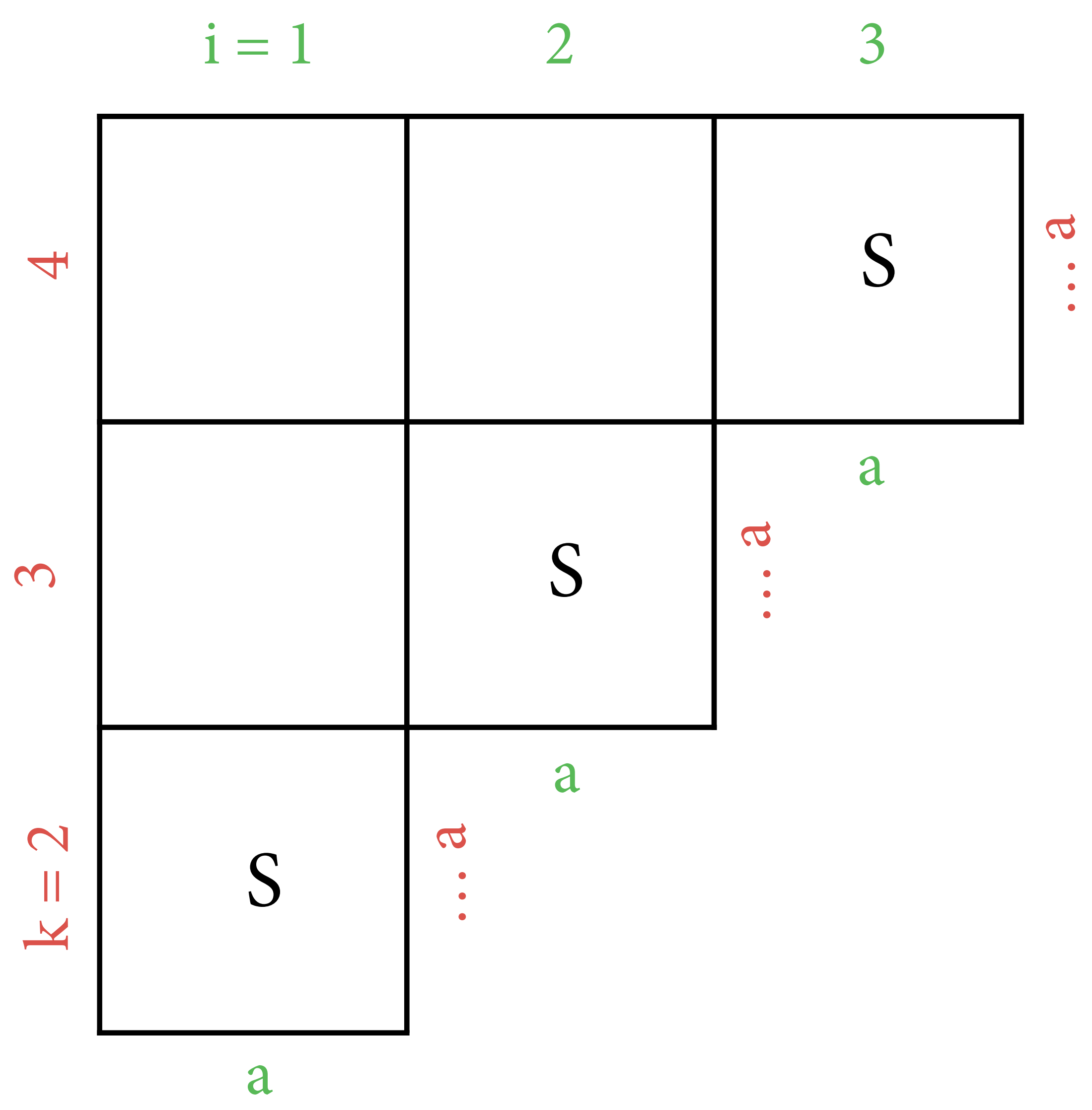
The CKY Recognizer

S → S S   S → a



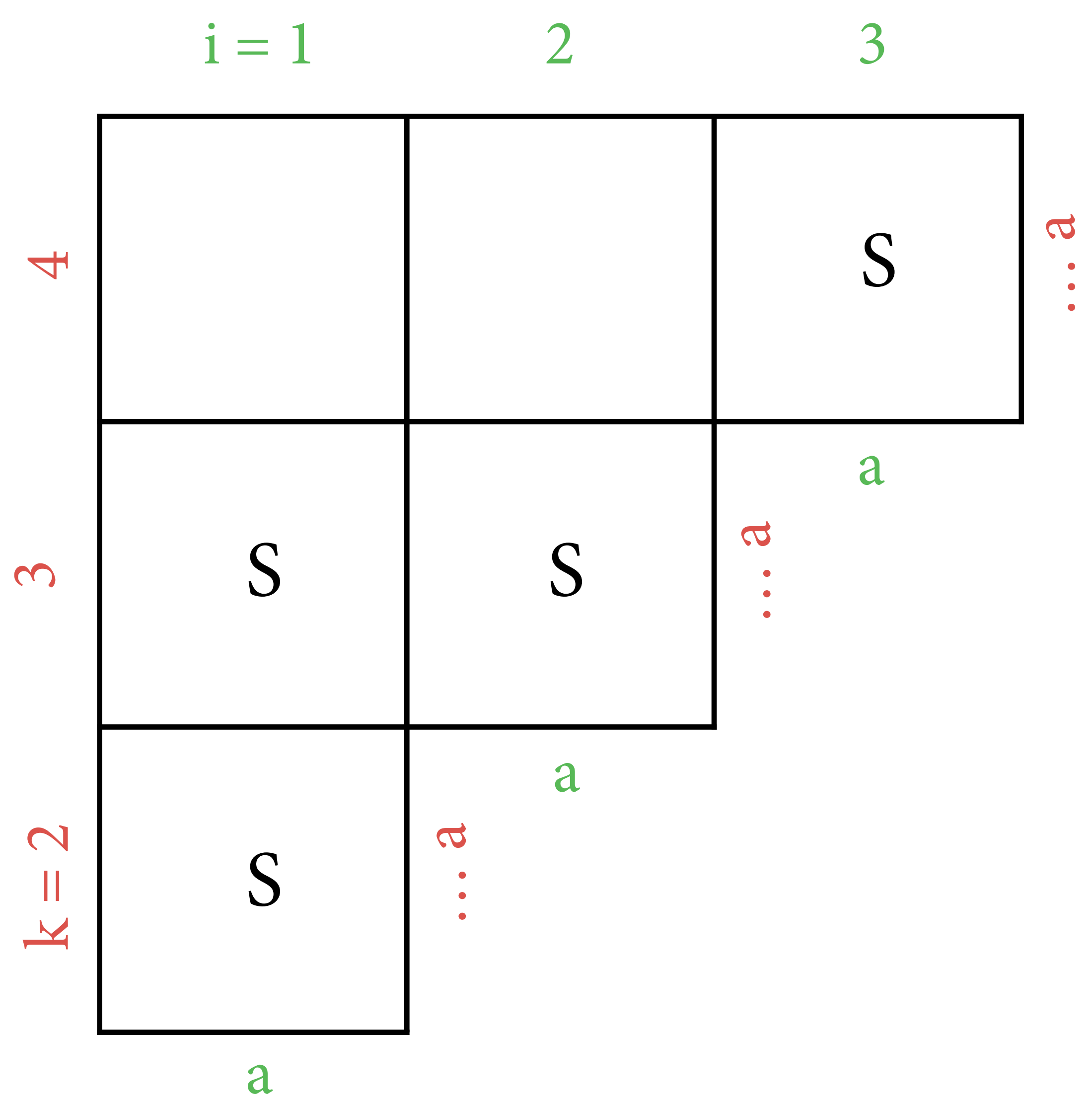
The CKY Recognizer

S → S S S → a



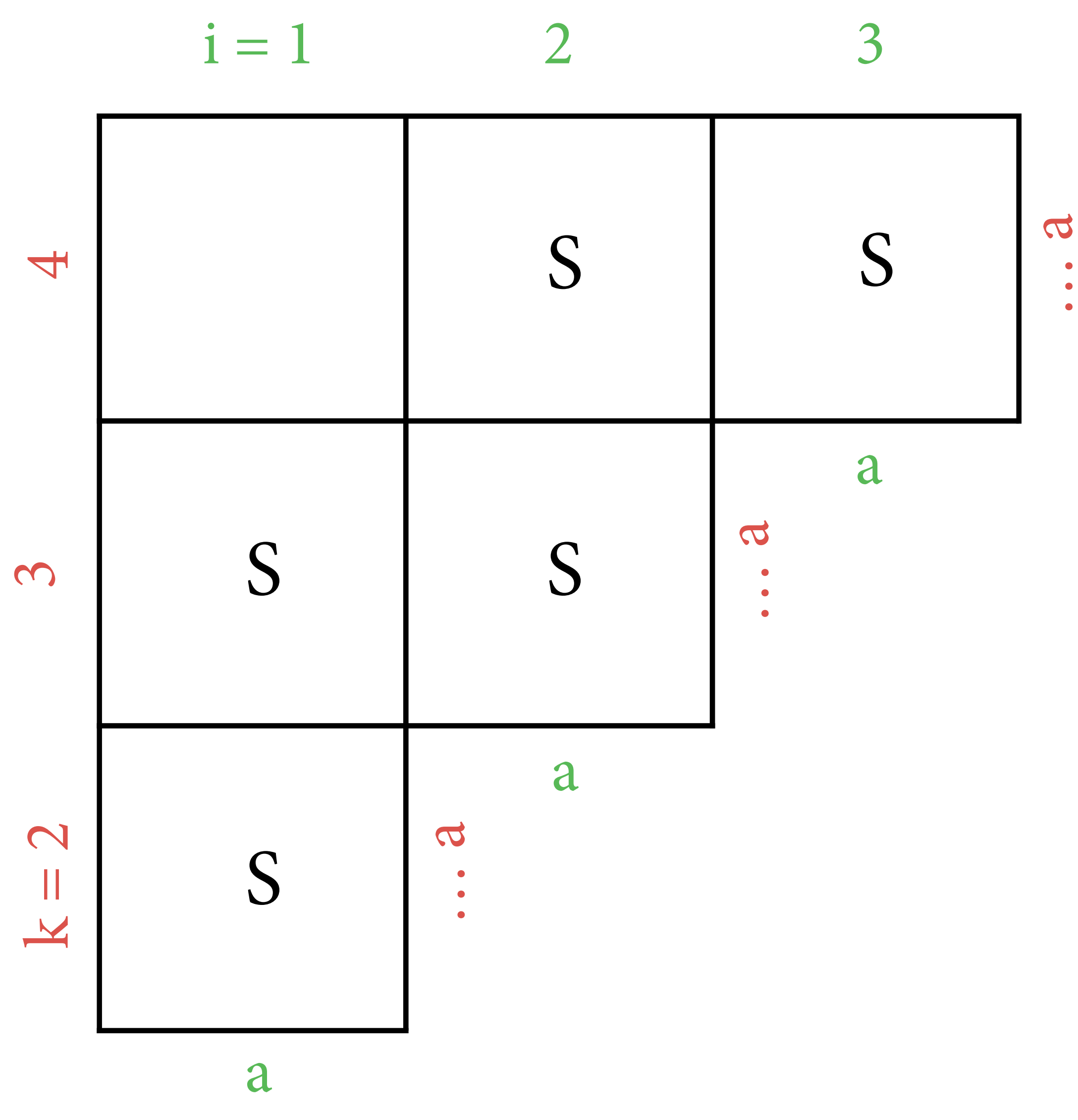
The CKY Recognizer

S → S S   S → a



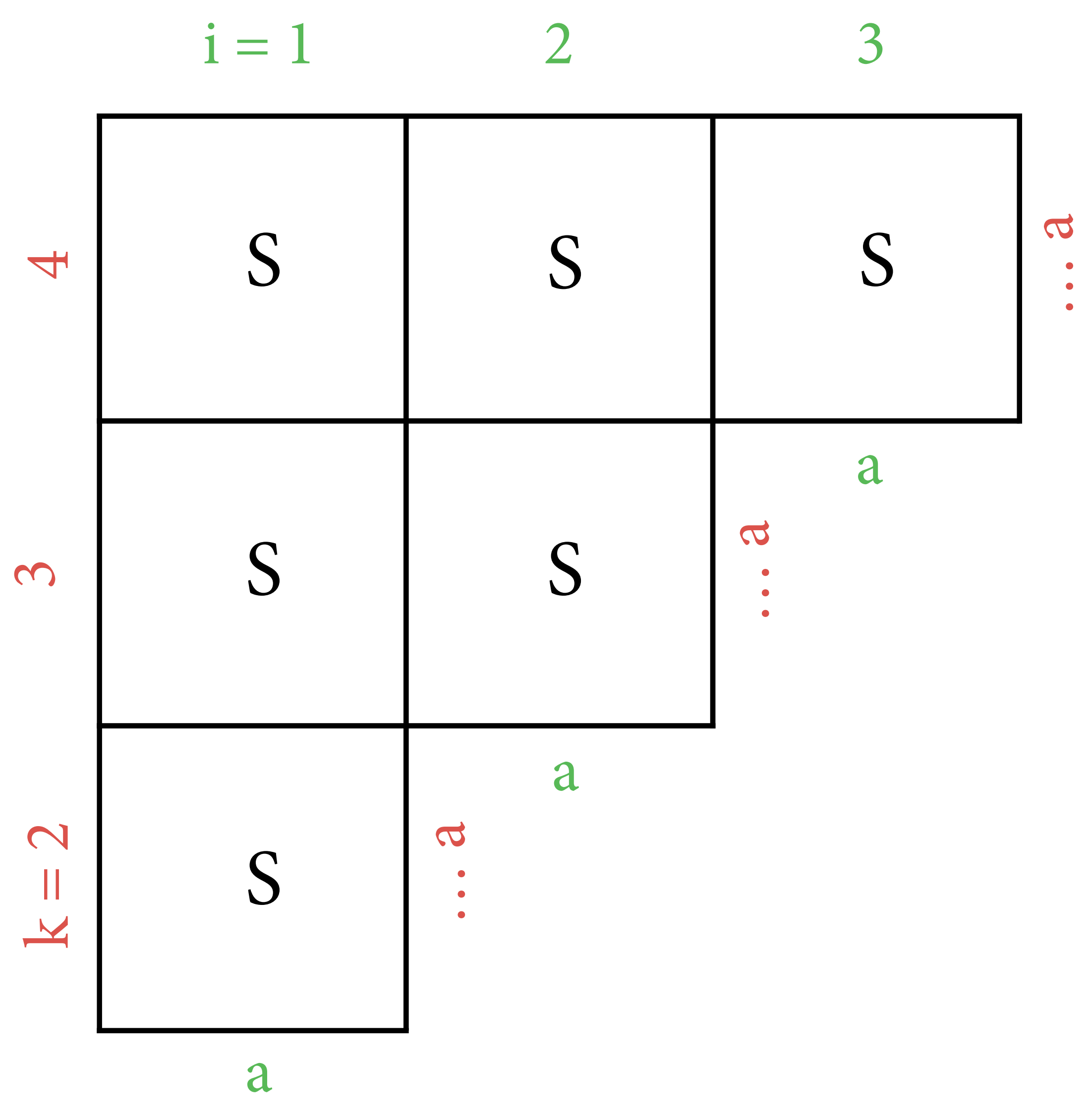
The CKY Recognizer

S → S S   S → a



The CKY Recognizer

S → S S   S → a



CKY recognizer: pseudocode

Data structure: Ch(i,k) eventually 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 ∈ L(G) iff S ∈ 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 $A \rightarrow BC$ 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 ∈ Ch(i,k) can be constructed from smaller parts.
    - built from B ∈ Ch(i,j) and C ∈ Ch(j,k) using A → 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 ∈ 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.
  - very simple polynomial algorithm, works well in practice
  - there are also other, more complicated algorithms
- Next time: put parsing and statistics together.