Context-free Grammars

Computational Linguistics

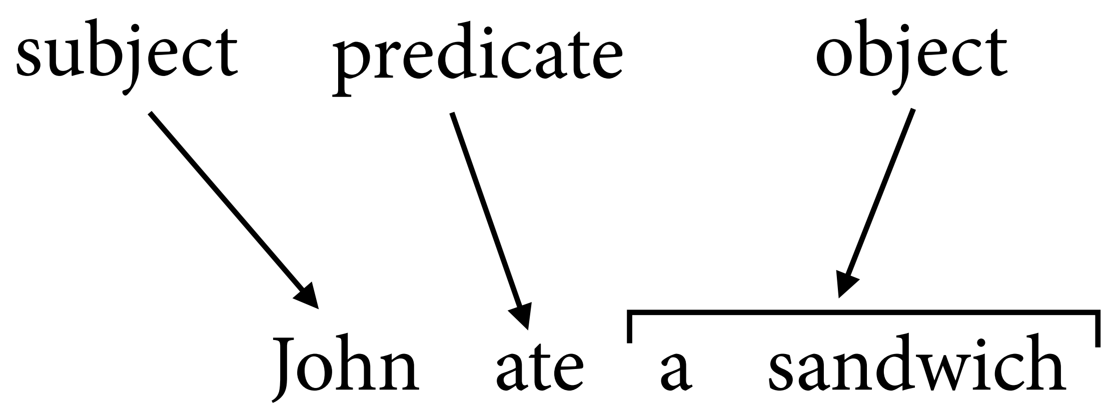
Alexander Koller

16 November 2018

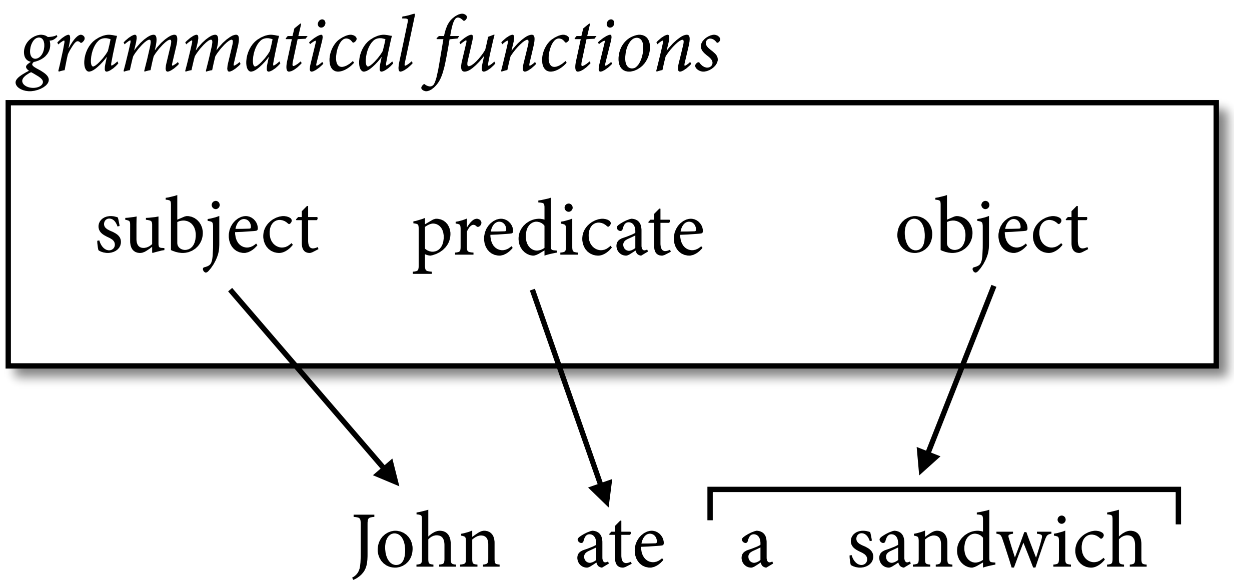
Sentences have structure

John ate a sandwich

Sentences have structure



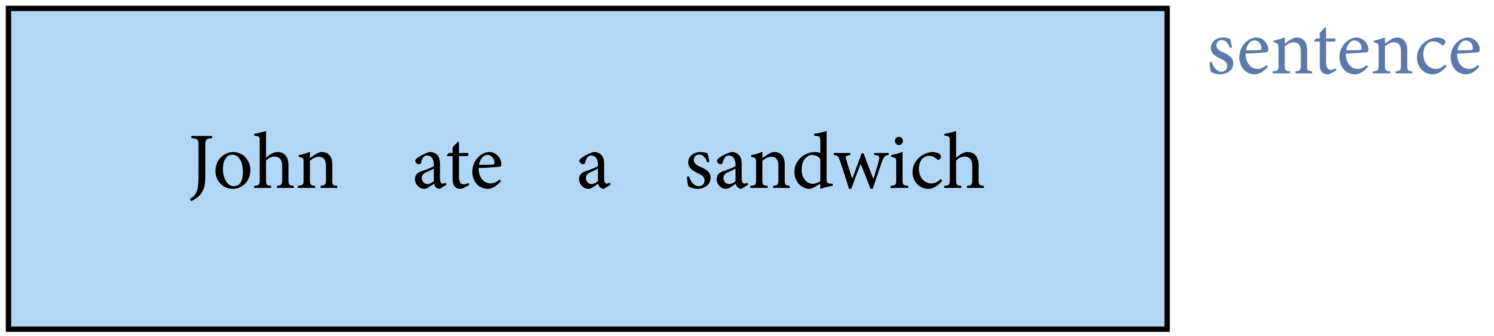
Sentences have structure



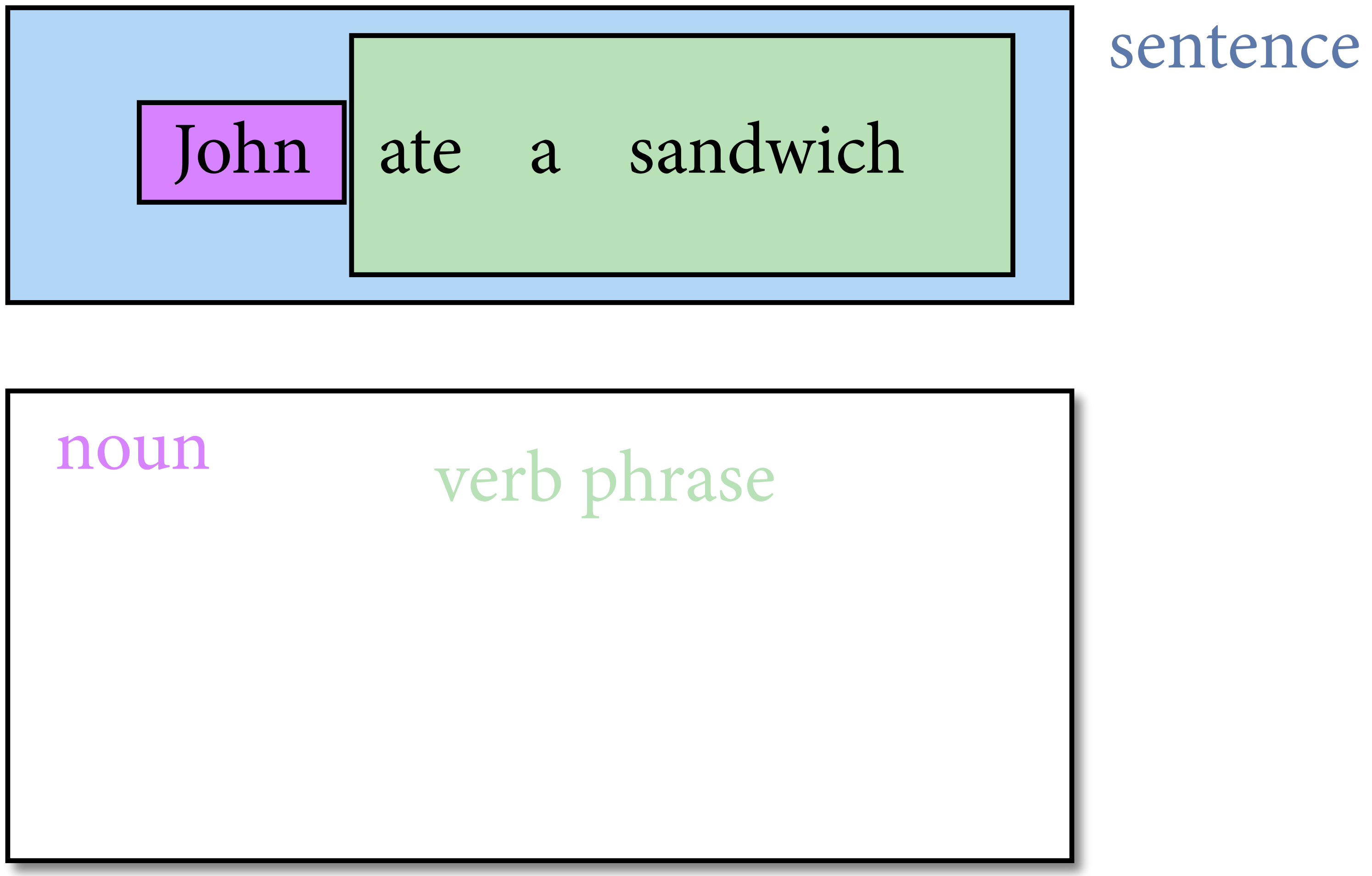
Sentences have structure

John ate a sandwich

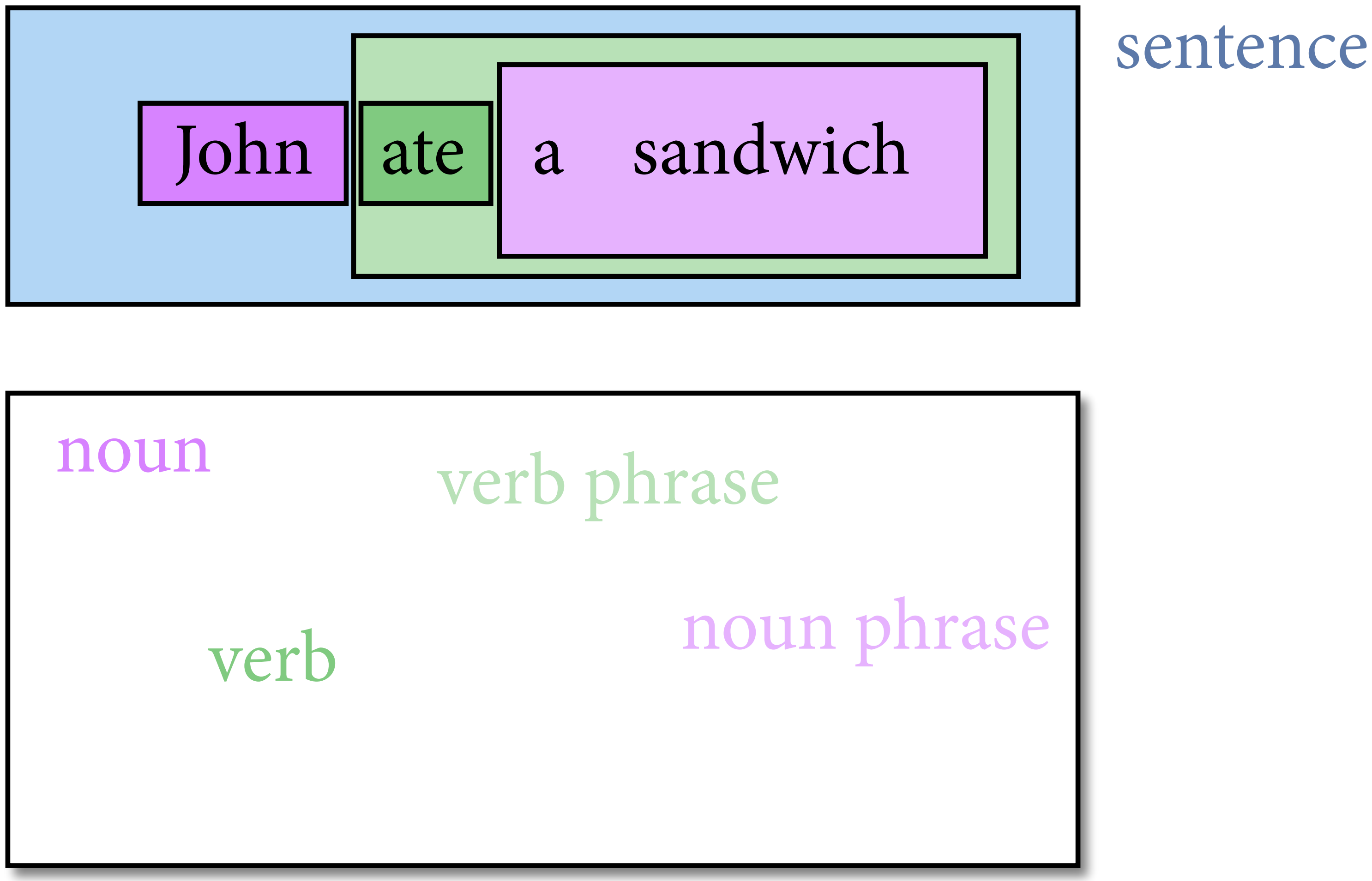
Sentences have structure



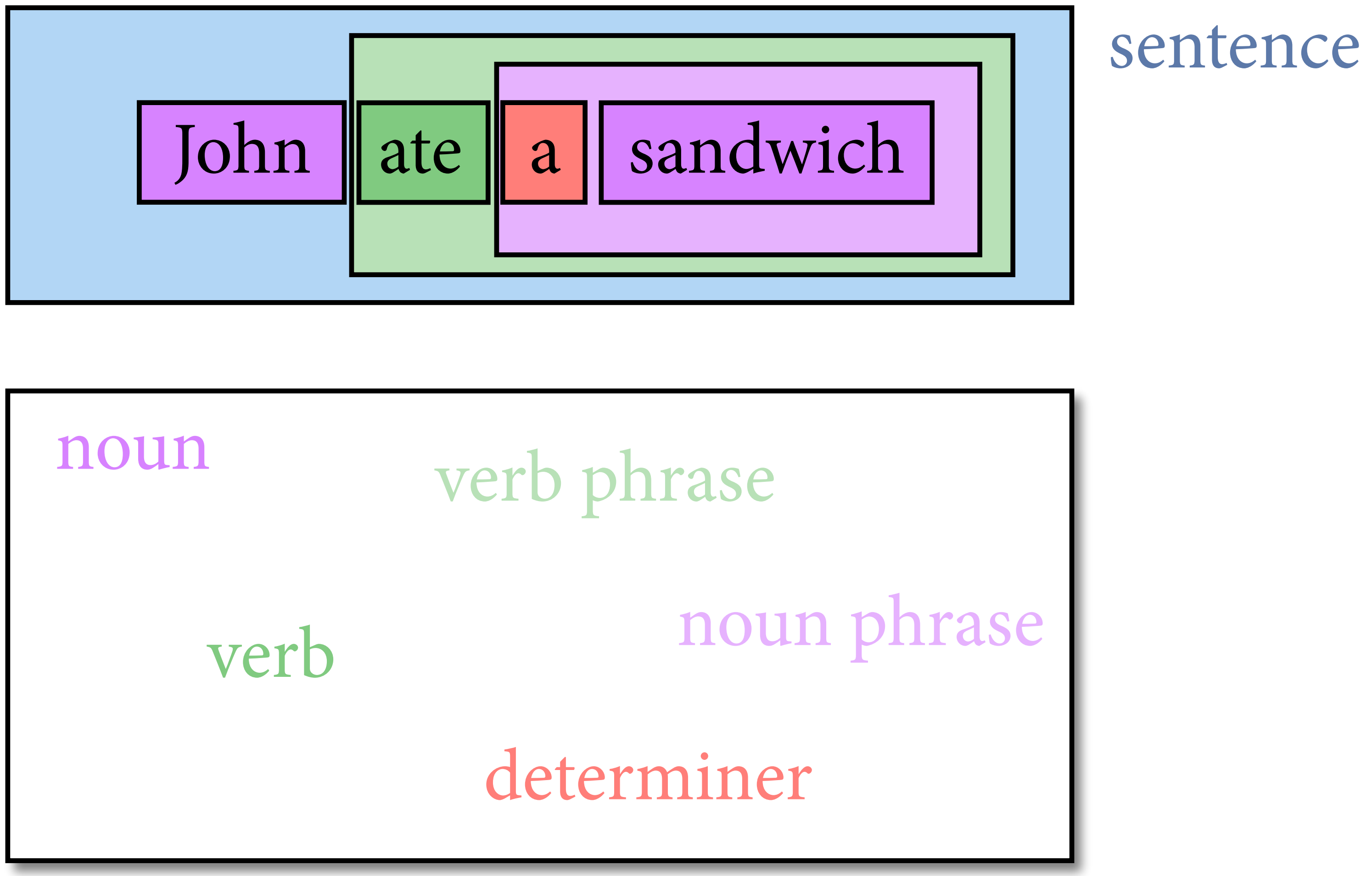
Sentences have structure



Sentences have structure

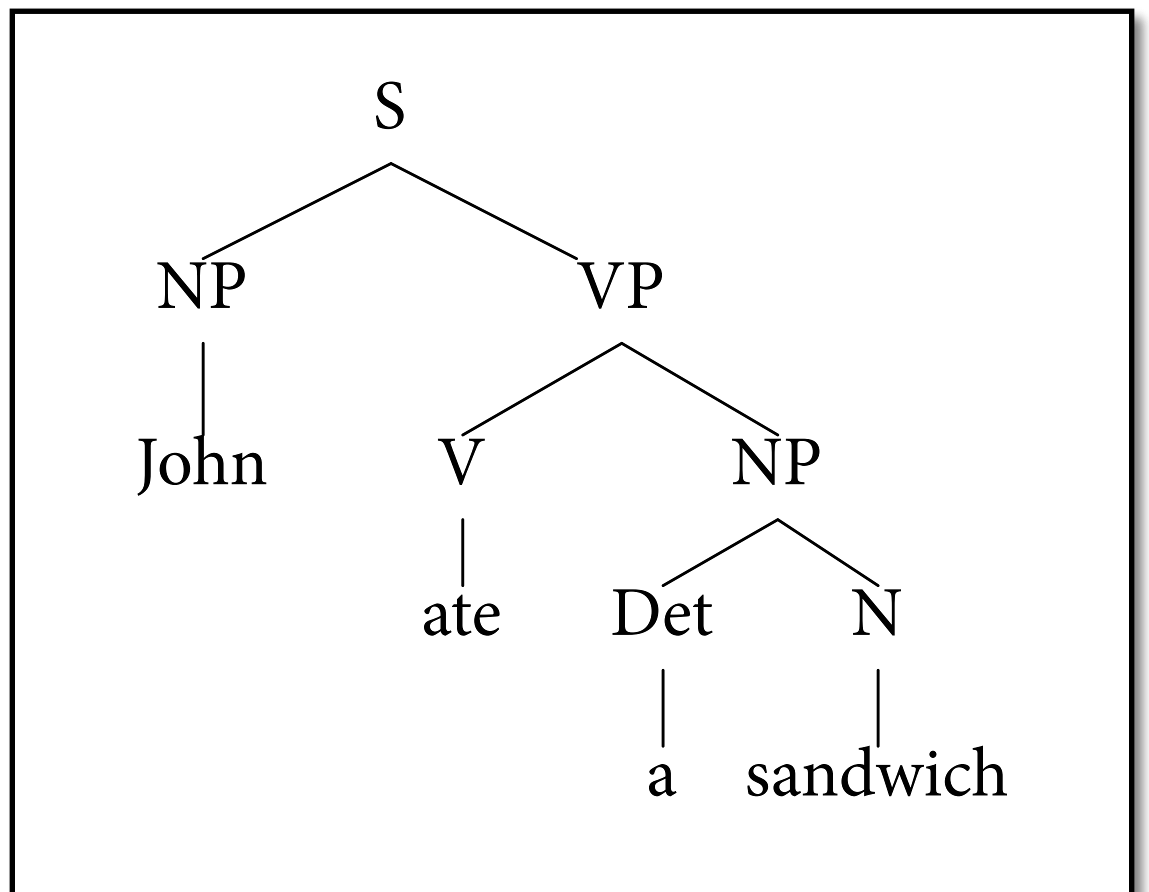


Sentences have structure



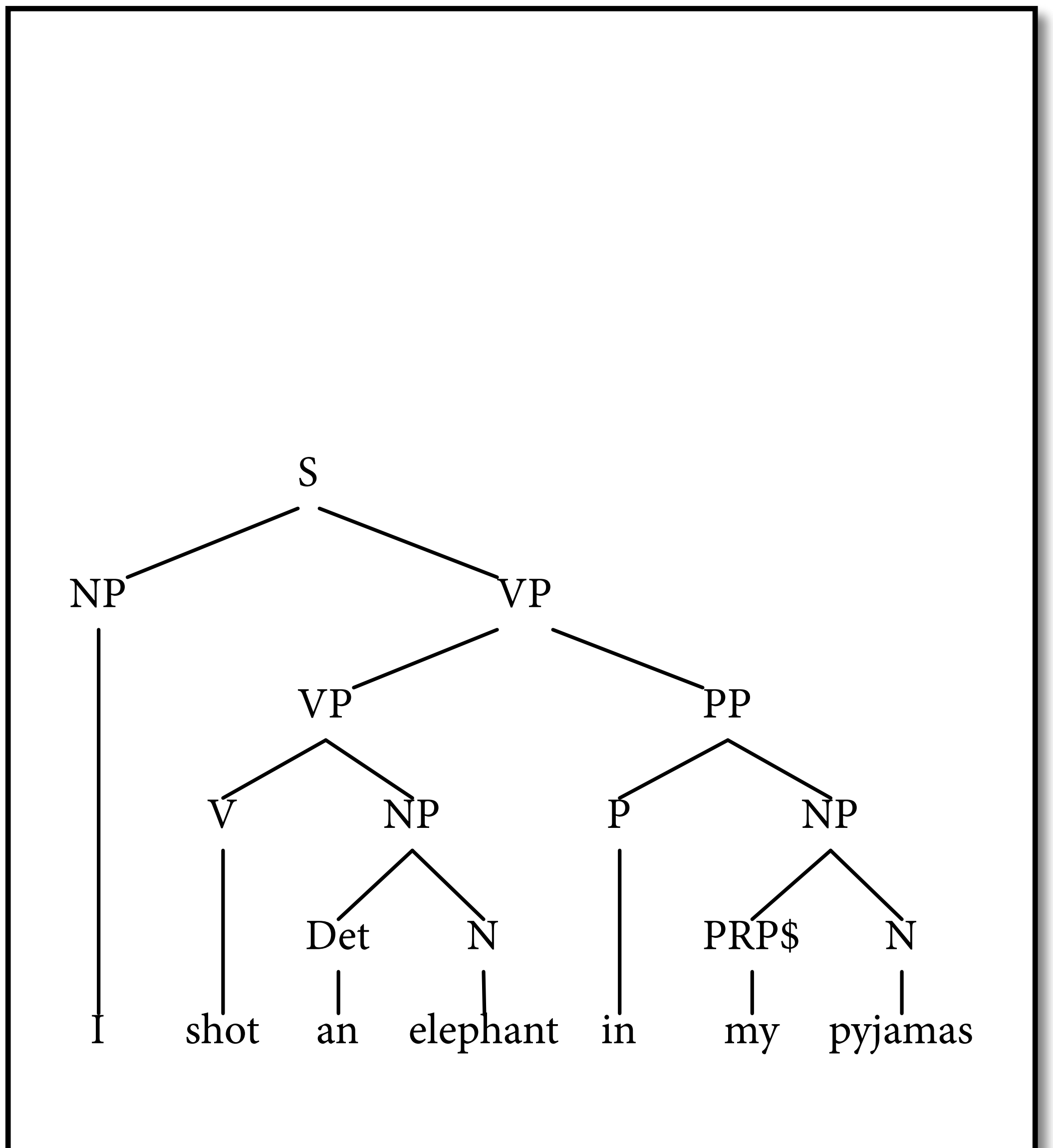
Sentences have structure

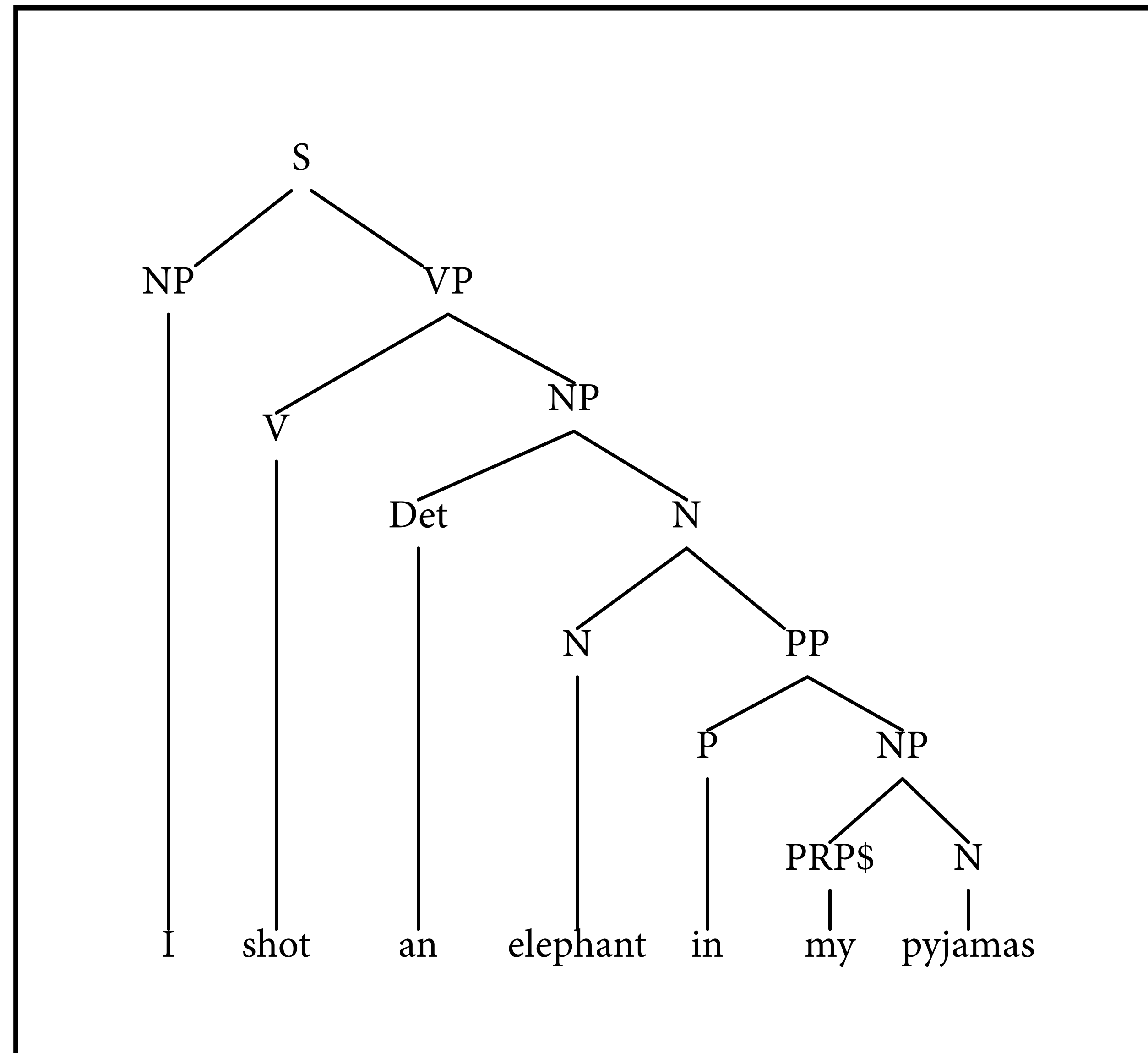
Record it conveniently in *phrase structure tree.*



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.
  - $S \in N$ is the *start symbol.*
  - P is a finite set of *production rules* of the form $A \rightarrow w$,
    where A is nonterminal and w is a string from $(N \cup T)^*$.

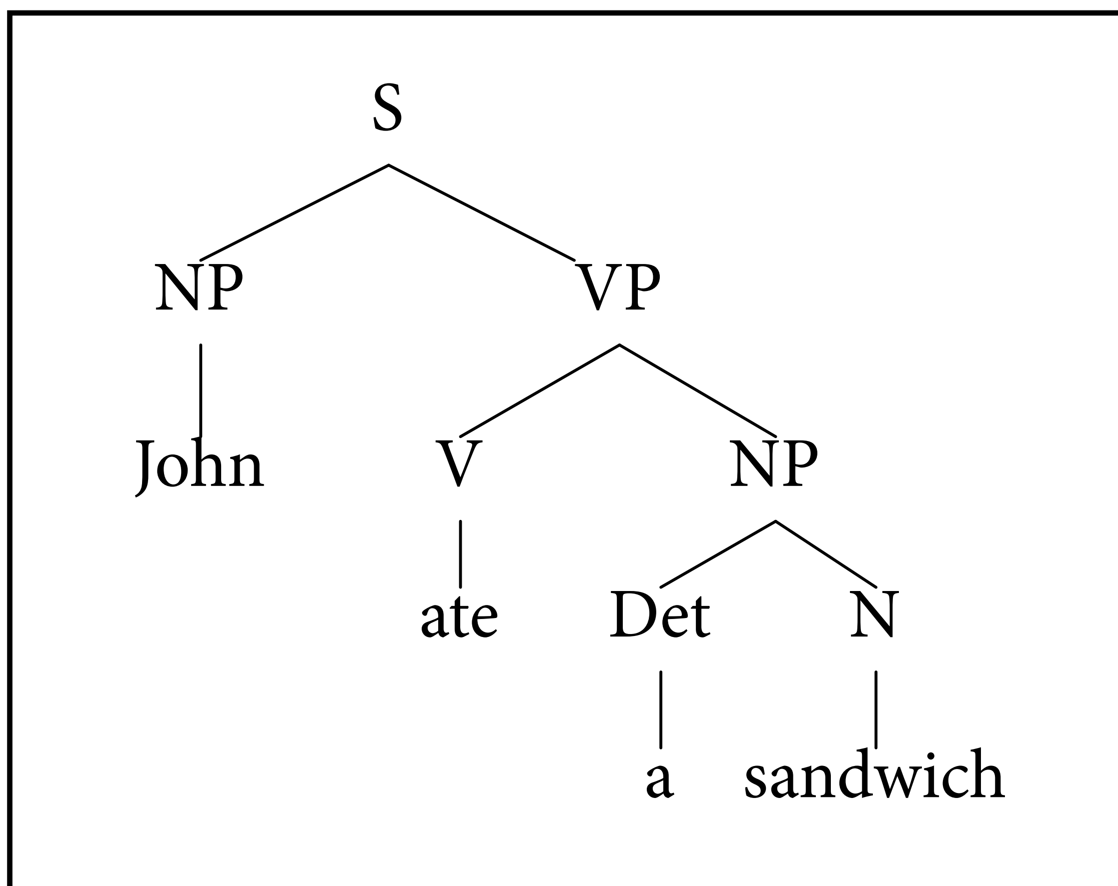
- Why "context-free"?
  - Left-hand side of production is a single nonterminal A.
  - Rule can't look at context in which A appears.
  - *Context-sensitive* grammars can do that.

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

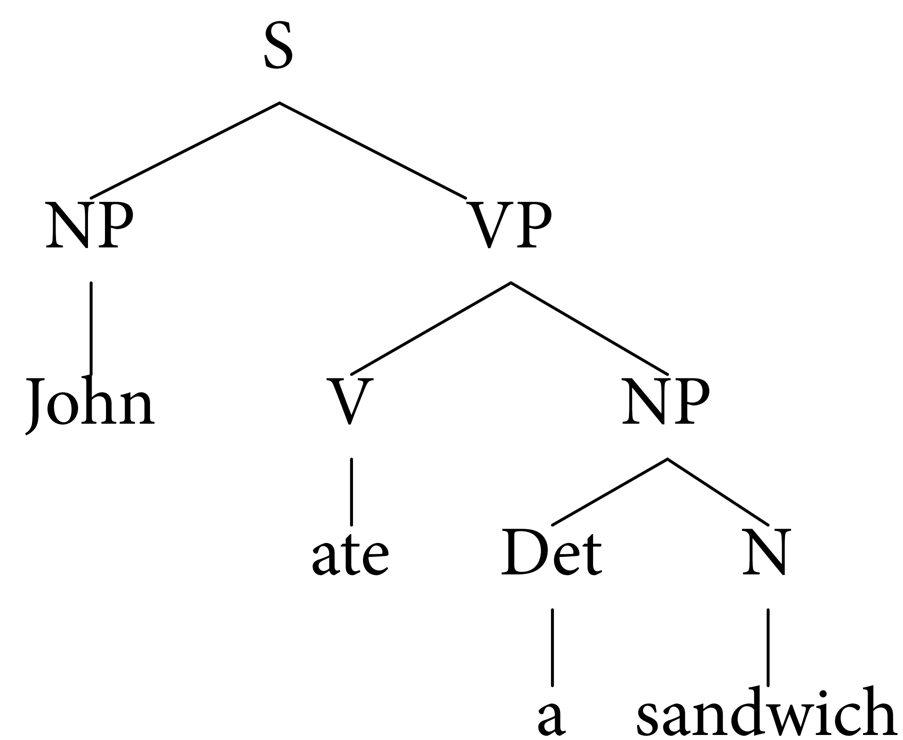
Derivations and parse trees

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

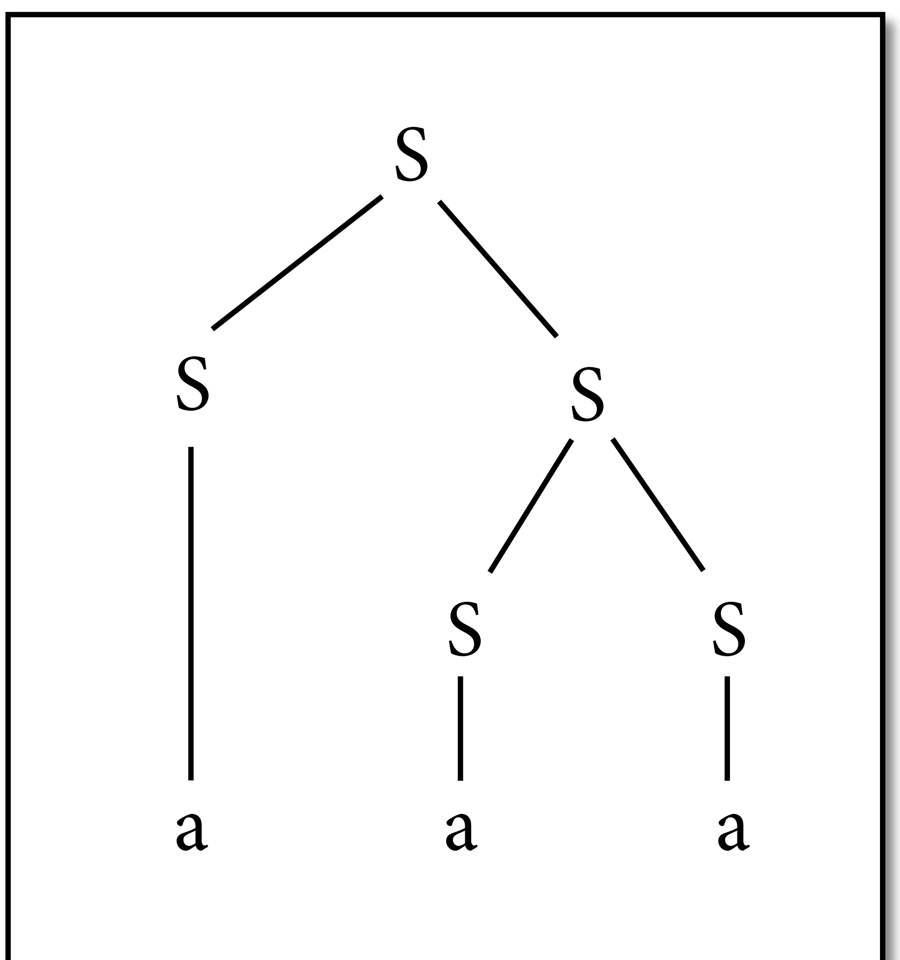
parse tree

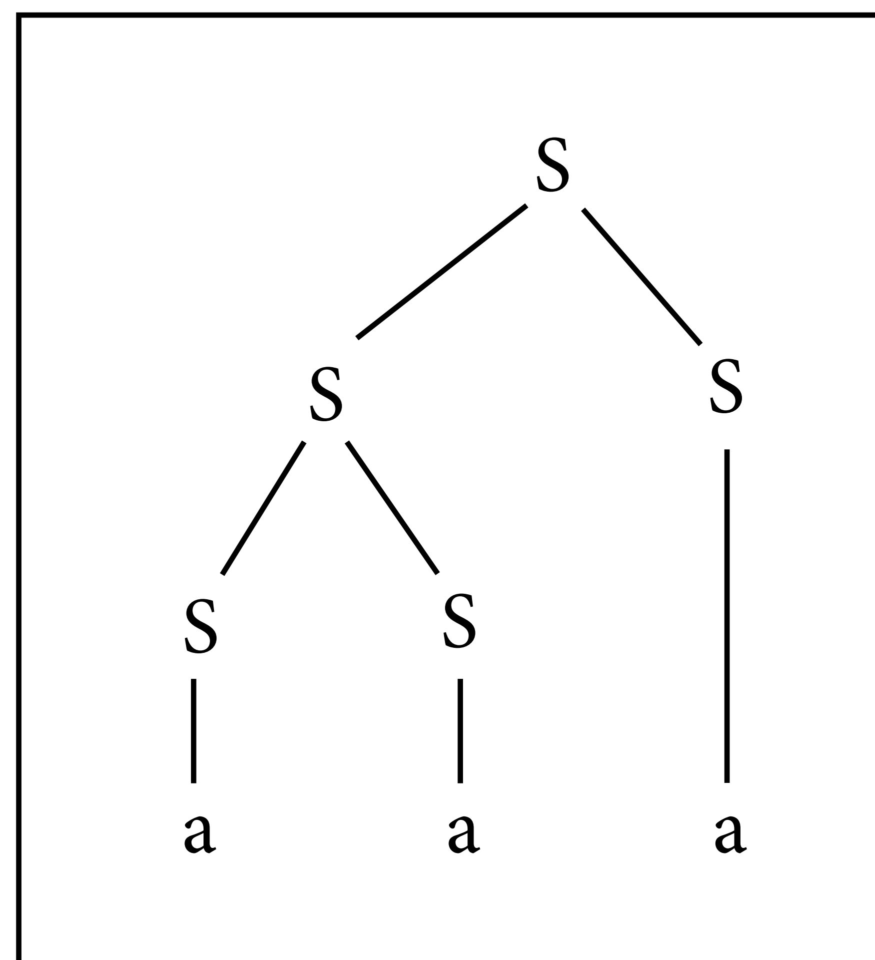


Big languages

Number of parse trees can grow exponentially in string length.

S → S S   S → a





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.

Parsing algorithms

- How can we solve the word and parsing problem so systematically that we can implement it?
- One simple approach: shift-reduce algorithm (here: only for the word problem).
- Next time: Analyze efficiency of SR and replace it with faster algorithm: CKY.

Shift-Reduce Parsing

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

Production rules:

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

John ate a sandwich

Shift-Reduce Parsing

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

Production rules:

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

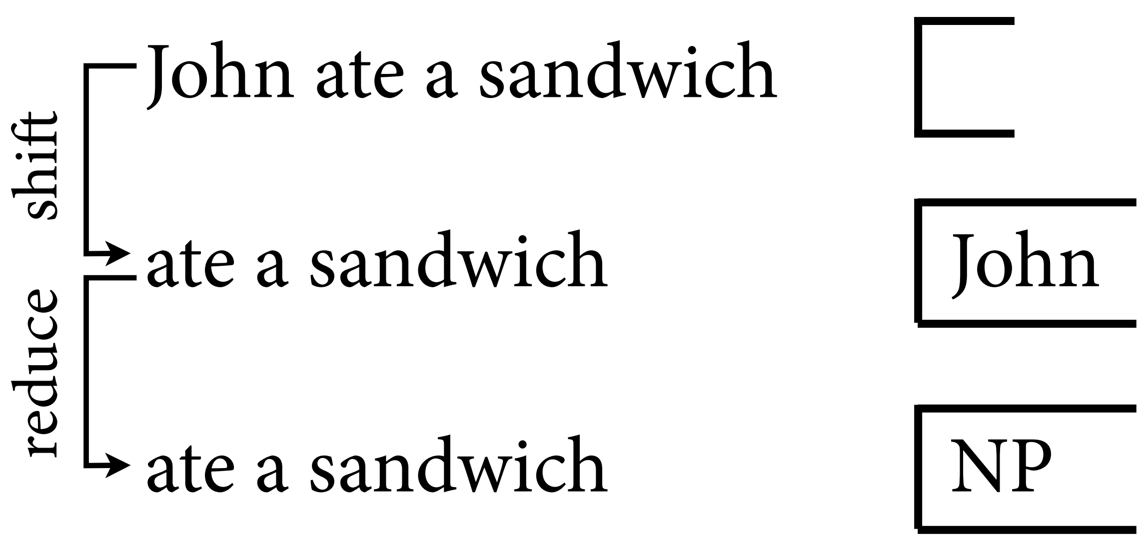
	Input	Stack
	John ate a sandwich	
shift	ate a sandwich	John

Shift-Reduce Parsing

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

Production rules:

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

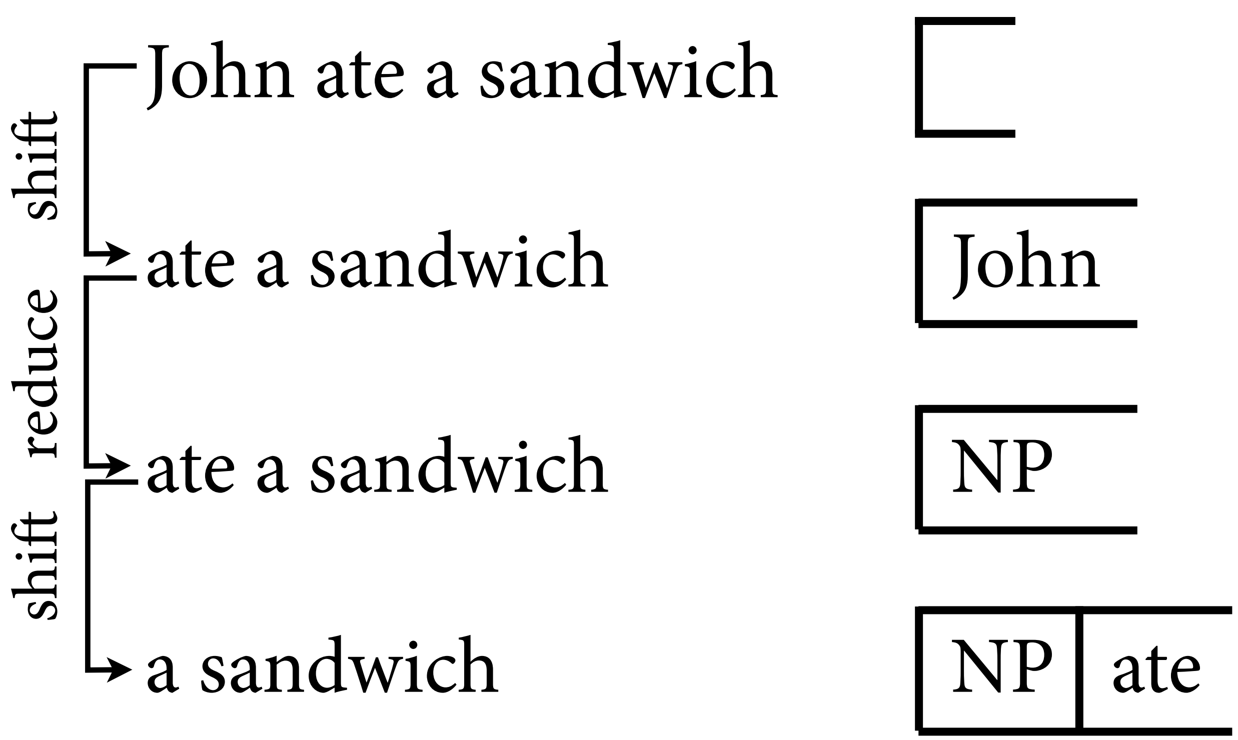


Shift-Reduce Parsing

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

Production rules:

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

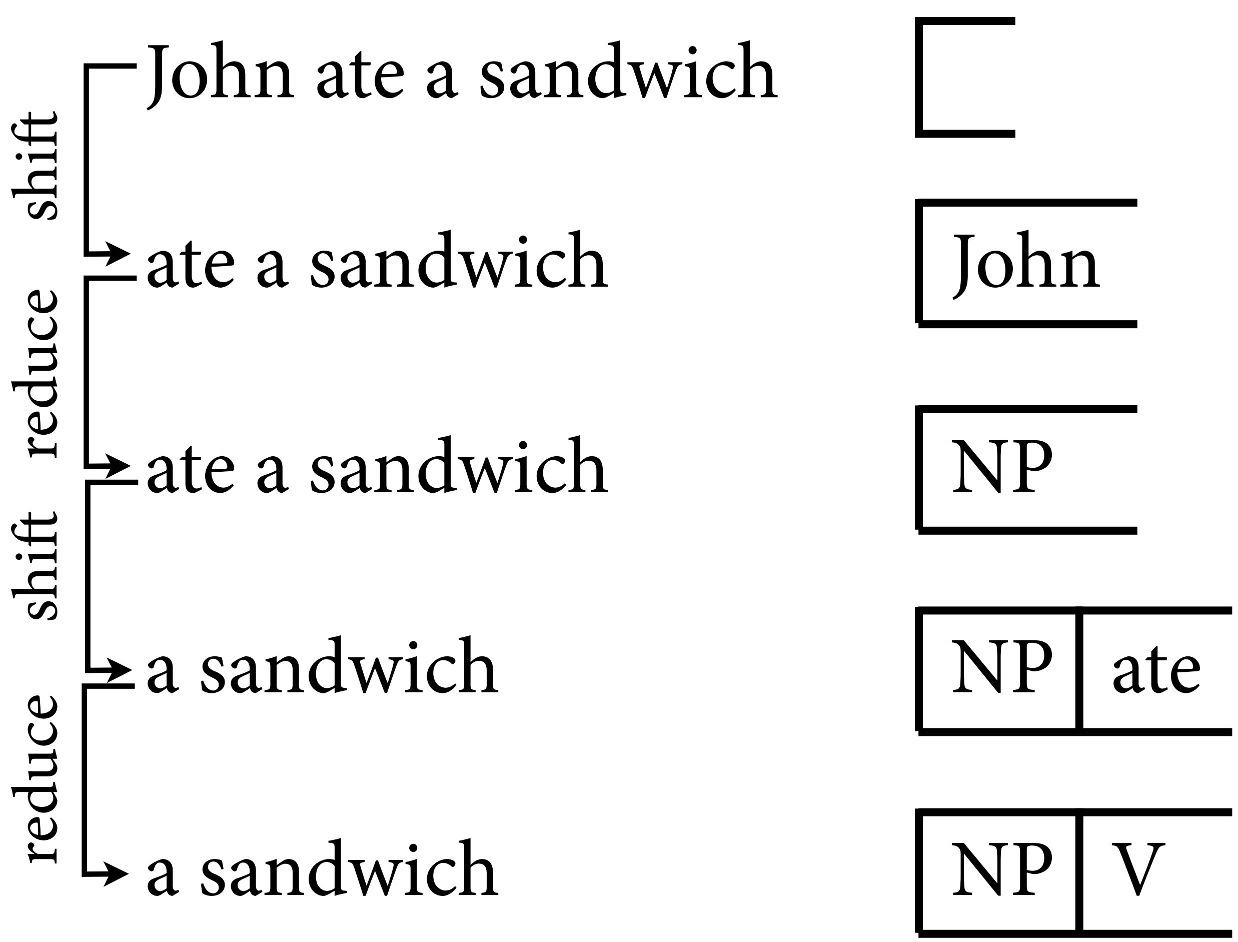


Shift-Reduce Parsing

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

Production rules:

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

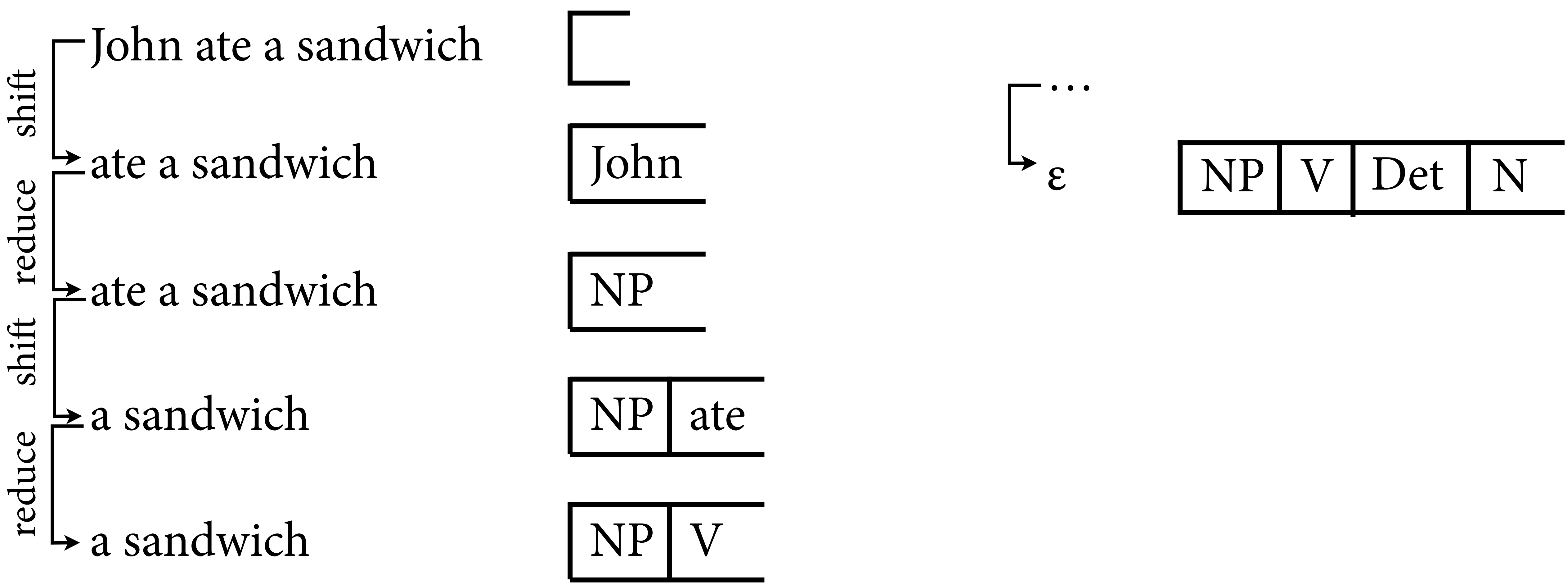


Shift-Reduce Parsing

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

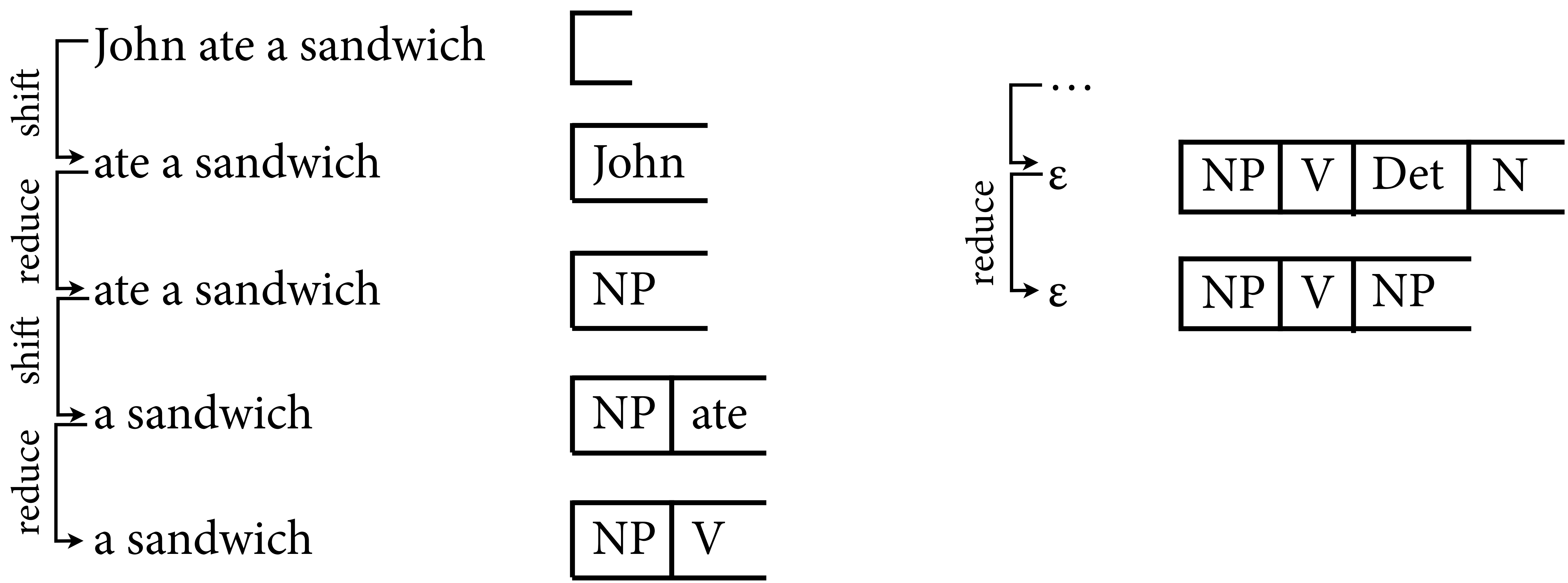


Shift-Reduce Parsing

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

Production rules:

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

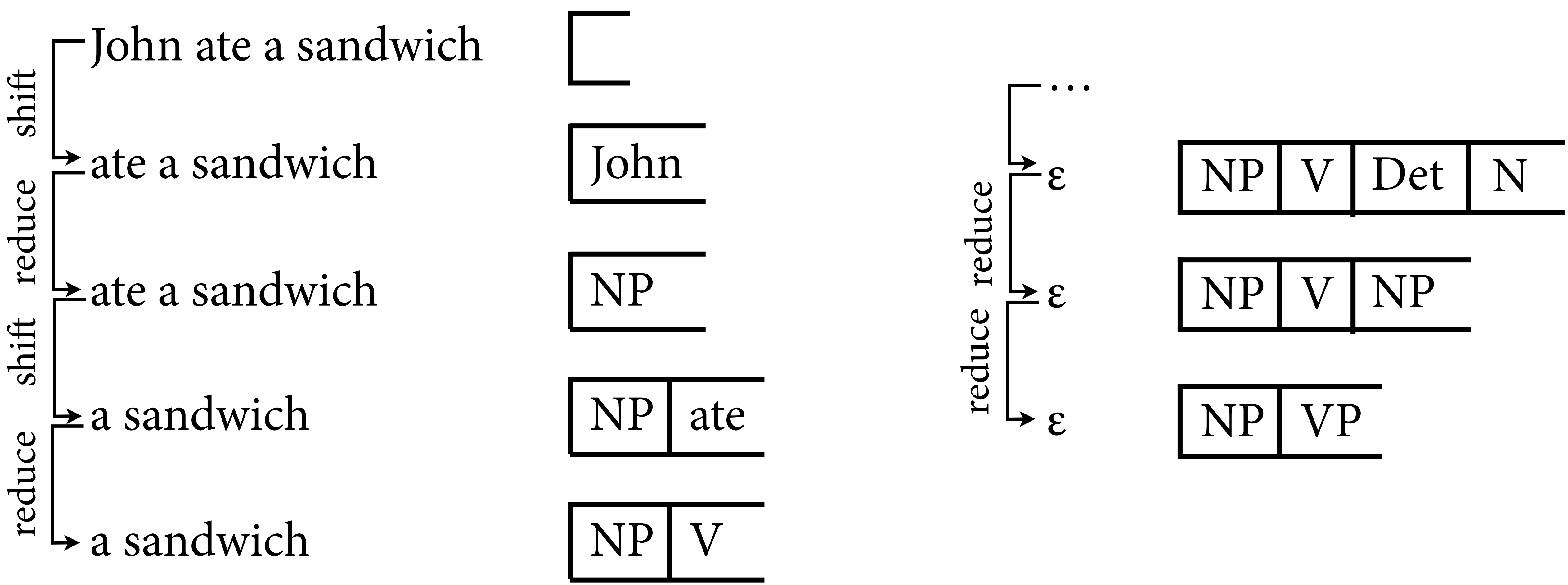


Shift-Reduce Parsing

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

Production rules:

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

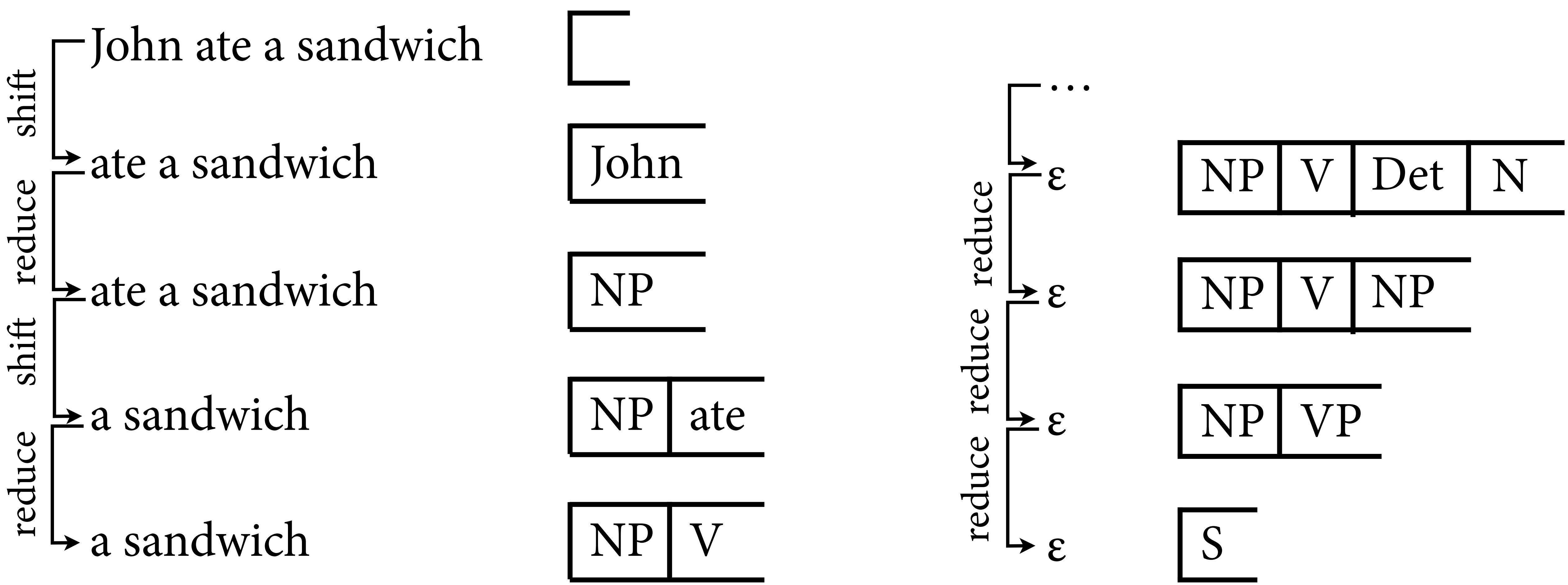


Shift-Reduce Parsing

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

Production rules:

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



Shift-Reduce Parsing

- Read input string step by step. In each step, we have
  - the remaining input words we have not shifted yet
  - a *stack* of terminal and nonterminal symbols
- In each step, apply a rule:
  - Shift: moves the next input word to the top of the stack
  - Reduce: applies a production rule to replace top of stack with the nonterminal on the left-hand side
- Sentence is in language of cfg iff we can read the whole string and stack contains only start symbol.

Shift-Reduce Parsing

- Shift rule:
  $(s, a \cdot w) \rightarrow (s \cdot a, w)$
- Reduce rule:
  $(s \cdot w', w) \rightarrow (s \cdot A, w)$ if $A \rightarrow w'$ in P
- Start: (ε, w)
- Apply rules *nondeterministically:*
  Claim $w \in L(G)$ if there *exists* some sequence of steps that derive (S, ε) from (ε, w).

Nondeterminism

- Claim that string is in language of cfg iff (S, ε) can be derived by *any* sequence of shift and reduce steps.

- This is very important because there are many stack-string pairs where multiple rules can be applied:
  - shift-reduce conflict
  - reduce-reduce conflict

- In practice, we need to try all sequences out.
  - Compilers for programming languages avoid this by careful language design: no ambiguity in grammar.

Parsing Schemata

- Parsing algorithm derives claims about the string. Record such claims in *parse items*.

- At each step, apply a *parsing rule* to infer new parse items from earlier ones.

- If there is a way to derive a *goal item* from the *start item(s)* for a given input string, then claim that this string is in the language.

Schema for shift-reduce

- Items are of the form (s,w') where w' is a suffix of the input string w, and s is the stack.
  - Claim of this item: Underlying cfg allows the derivation $s\ w' \Rightarrow^* w$

- Start item: (ε, w); goal item: (S, ε)

- Parsing rules:

$$\frac{(s,\ a \cdot w')}{(s \cdot a,\ w')}\ \text{(shift)} \qquad\qquad \frac{(s \cdot s',\ w') \quad A \rightarrow s' \text{ in } P}{(s \cdot A,\ w')}\ \text{(reduce)}$$

Implementing schemas

- Can generally implement parser for given schema in the following way:
  - maintain an *agenda*: queue of items that we have discovered, but not yet attempted to combine with other items
  - maintain a *chart* of all seen items for the sentence

```
initialize chart and agenda with all start items

while agenda not empty:
  item = dequeue(agenda)
  for each combination c of item with other item in the chart:
    if c not in chart:
      add c to chart
      enqueue c in agenda

if chart contains a goal item, claim w ∈ L(G)
```

rules of parsing schema used here

Implementing schemas

- Can generally implement parser for given schema in the following way:
  - maintain an *agenda*: queue of items that we have discovered, but not yet attempted to combine with other items
  - maintain a *chart* of all seen items for the sentence

```
initialize chart and agenda with all start items

while agenda not empty:
  item = dequeue(agenda)
  for each combination c of item with other item in the chart:
    if c not in chart:
      add c to chart
      enqueue c in agenda

if chart contains a goal item, claim w ∈ L(G)
```

rules of parsing schema used here

essential to do this efficiently

Correctness of shift-reduce

- Why should we believe that the SR parser always makes correct claims about the word problem?

- To convince ourselves, we need to prove:
  - *soundness:* SR recognizer only claims $w \in L(G)$ if this is true;
  - *completeness:* if $w \in L(G)$ is true, then SR recognizer claims it is.

Soundness

- Show: If SR recognizer claims $w \in L(G)$, then it is true.

- Prove by induction over derivation length k that all items that are being derived are true.
  - k = 0: Item is start item (ε, w). This is trivially true.
  - k → k+1: Any derivation of k+1 steps ends in a last step.
    - *Shift:* (ε, w) →* (s, a w') → (s a, w').
      By induction hypothesis, (s, a w') is true, i.e. s a w' ⇒* w.
      Thus, (s a, w') is obviously true as well.
    - *Reduce:* (ε, w) →* (s s', w') → (s A, w').
      By induction hypothesis, (s s', w') is true, i.e. s s' w' ⇒* w.
      Thus we have s A w' ⇒ s s' w' ⇒* w, i.e. (s A, w') is true.

Completeness

- Show: If $w \in L(G)$, then SR recognizer claims it is true.

- Prove by induction over length of CFG derivation that if $A \Rightarrow^* w_i \dots w_k$, then $(\varepsilon, w_i \dots w_k) \underset{SR}{\rightarrow}^* (A, \varepsilon)$.
  - length = 1: one shift + one reduce does it
  - length k → k+1: $A \Rightarrow B\ C \Rightarrow^* \underbrace{w_i \dots w_{j-1}}_{B}\ \underbrace{w_j \dots w_k}_{C}$

    Then by induction hypothesis, can derive
    $(\varepsilon, w_i \dots w_k) \underset{SR}{\rightarrow}^* (B, w_j \dots w_k) \underset{SR}{\rightarrow}^* (BC, \varepsilon) \underset{R}{\rightarrow} (A, \varepsilon)$

Conclusion

- Context-free grammars: most popular grammar formalism in NLP.
- Parsing algorithms.
  - today, shift-reduce
  - next time, CKY
- Outlook:
  - combine CFG parsing with statistics
  - more expressive grammar formalisms