CS 577: NATURAL LANGUAGE PROCESSING

Abulhair Saparov

Lecture 21: Syntax II

COMPUTATIONAL LINGUISTICS ROADMAP

- Last lecture, we started discussing syntax
  - How are words arranged to form sentences?
  - Words have parts of speech (POS).
  - We can test whether two words have the same part of speech by substituting them for each other in a sentence.
    - ‘The fast car drove by.’
    - ‘The orange car drove by.’
    - *‘The quickly car drove by.’
    - ‘The car drove by fast.’
    - ‘The car drove by quickly.’
    - *‘The car drove by orange.’

SYNTAX

- We can also substitute words with phrases:
  - ‘`The fast cat with the stripes ran to the bank`’ (grammatical)
  - ‘`The fast cat with the stripes`’ is a phrase that behaves like a noun.
    - This is a noun phrase.
- We can similarly define other kinds of phrases:
  - Verb phrase: ‘`ran to the bank`’
  - Adjective phrase: ‘`taller than a mountain`’
  - Adverbial phrase: ‘`very quickly`’
  - Prepositional phrase: ‘`to the bank`’

SYNTAX

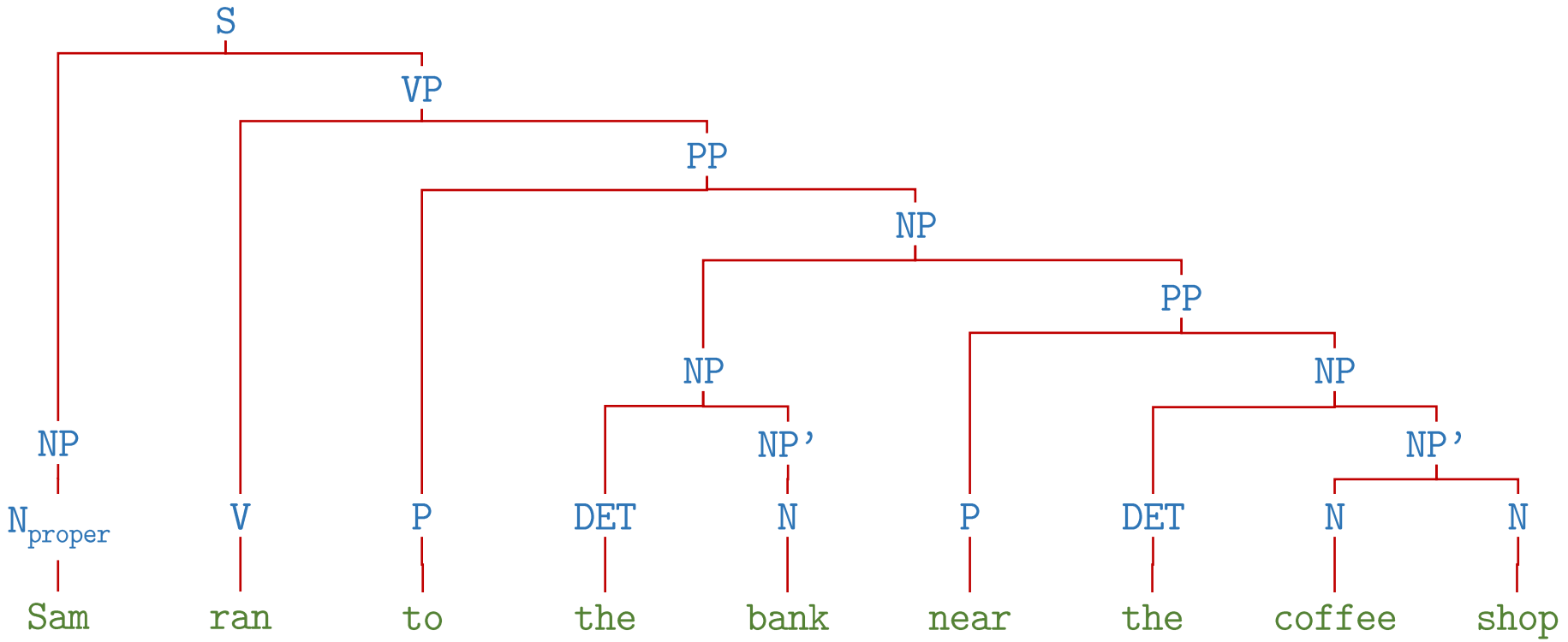
- Focusing on the phrase ‘to the bank’, each component is necessary to maintain the grammaticality and meaning of the sentence.
  - ‘Sam ran bank’ (ungrammatical)
  - ‘Sam ran to bank’ (ungrammatical)
  - ‘Sam ran the bank’ (grammatical, but different meaning)
  - All three words are needed to form the phrase ‘to the bank.’
- But looking closely at the prepositional phrase, we see that it is composed of smaller components:
  - A preposition (‘to’) and a noun phrase (‘the bank’).
  - We can conclude that one way to construct prepositional phrases is to combine any preposition with a noun phrase.

SYNTAX

- For example, we can rephrase the sentence as ‘Sam ran to Chase Bank.’
- So ‘Chase Bank’ and ‘the bank’ play the same grammatical role.
  - They are noun phrases.
- Language enables to construction of larger noun phrases:
  - E.g., ‘Sam ran to the bank near the coffee shop.’
  - Here, we made a larger noun phrase by adding the prepositional phrase “near the coffee shop.”
- So we have seen that prepositional phrases can contain subordinate noun phrases, and noun phrases can contain subordinate prepositional phrases.
  - This is an example of recursion,
  - Recursion is a key property of languages, including natural language.

SYNTAX TREE

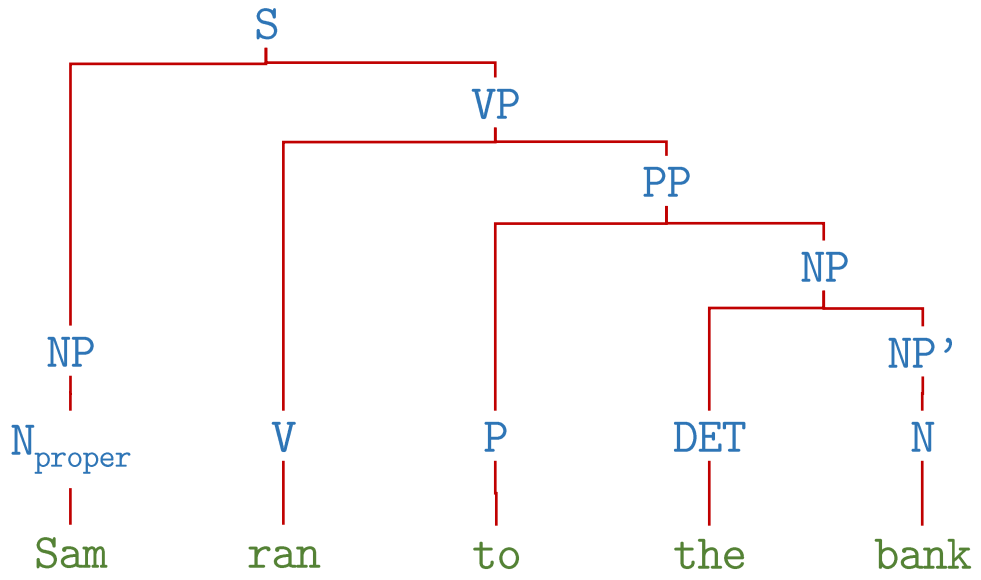
- A natural way to represent recursive structure is as a tree:



SYNTAX TREE

- Syntax trees can also be represented in bracketed notation:

```
(S
   (NP (N_proper 'Sam'))
   (VP
      (V 'ran')
      (PP
         (P 'to')
         (NP
            (DET 'the')
            (NP' (N 'bank'))
         )
      )
   )
)
```



GRAMMAR

- We have observed that noun phrases can be constructed from smaller phrases:
  - Either a proper noun (e.g., ‘Sam’),
  - A determined common noun (e.g., ‘the bank’),
  - Or a noun phrase + prepositional phrase (e.g. ‘the bank near me’),
  - (or other rules)
- Similarly, we have observed prepositional phrases can be constructed from smaller phrases.
  - And similarly for verb phrases, etc.
- These are grammatical rules.
  - Altogether, these rules form the grammar of a language.

GRAMMAR

- We can write the rules of a grammar:
- Each rule is called a production rule.
- S is the root.

```
S  -> NP VP
VP ->  V NP
VP ->  V PP
PP ->  P NP
NP -> NP PP
NP ->  N_proper
NP -> DET NP'
NP' -> N
```

```
V -> 'ran'
V -> 'sees'
V -> 'see'
P -> 'to'
P -> 'near'
N -> 'bank'
N -> 'coffee'
N -> 'shop'
```

```
N_proper -> 'Ada'
N_proper -> 'Alex'
N_proper -> 'Sam'
N_proper -> 'Zach'
DET -> 'the'
DET -> 'a'
```

GRAMMAR

- Nonterminals: S, NP, VP, PP, NP', DET, N_{proper}, N, V, P
- Terminals: 'ran', 'sees', 'see', 'to', …, 'Ada', 'Alex', …
- Preterminals: DET, N_{proper}, N, V, P (this is a subset of nonterminals)

```
S   -> NP VP
VP  ->  V NP
VP  ->  V PP
PP  ->  P NP
NP  -> NP PP
NP  ->  N_proper
NP  -> DET NP'
NP' -> N
```

```
V -> 'ran'
V -> 'sees'
V -> 'see'
P -> 'to'
P -> 'near'
N -> 'bank'
N -> 'coffee'
N -> 'shop'
```

```
N_proper -> 'Ada'
N_proper -> 'Alex'
N_proper -> 'Sam'
N_proper -> 'Zach'
DET -> 'the'
DET -> 'a'
```

GRAMMAR

- This grammar specifies a set of strings (i.e., a language).
- To sample a string from the grammar, start with the root S.
  - Recursively: For any nonterminal in the input, pick a rule with a matching left-hand side. Rewrite the nonterminal with the right-hand side.

```
S  -> NP VP
VP ->  V NP
VP ->  V PP
PP ->  P NP
NP -> NP PP
NP ->  N_proper
NP -> DET NP'
NP' -> N
```

```
V -> 'ran'
V -> 'sees'
V -> 'see'
P -> 'to'
P -> 'near'
N -> 'bank'
N -> 'coffee'
N -> 'shop'
```

```
N_proper -> 'Ada'
N_proper -> 'Alex'
N_proper -> 'Sam'
N_proper -> 'Zach'
DET -> 'the'
DET -> 'a'
```

GRAMMAR

- This grammar specifies a set of strings (i.e., a language).
- To sample a string from the grammar, start with the root S.
  - Recursively: For any nonterminal in the input, pick a rule with a matching left-hand side. Rewrite the nonterminal with the right-hand side.
- Example:
  1. S
  2. NP VP
  3. N_{proper} VP
  4. Sam VP
  5. Sam V NP
  6. Sam sees NP
  7. Sam sees N_{proper}
  8. Sam sees Ada

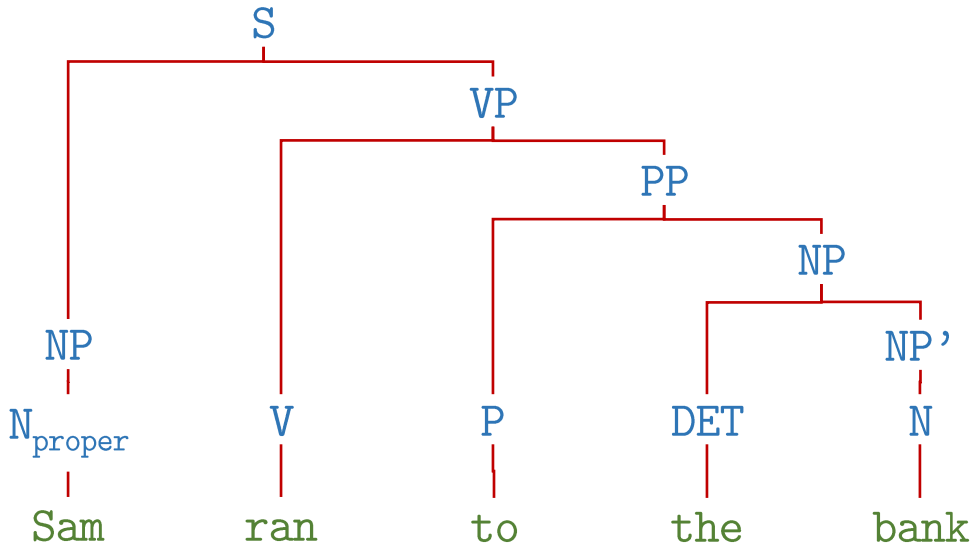
GRAMMAR

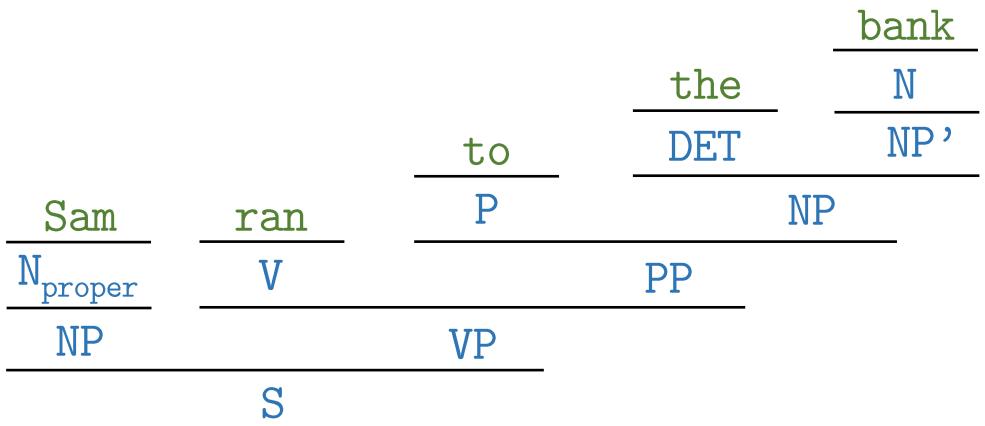
- This grammar specifies a set of strings (i.e., a language).
- To sample a string from the grammar, start with the root S.
  - Recursively: For any nonterminal in the input, pick a rule with a matching left-hand side. Rewrite the nonterminal with the right-hand side.
- The language generated by a grammar is the set of all strings s that can be generated with the above procedure.
- In this example grammar, the generated language is infinite:

```
L = {'Sam sees Ada', 'Sam see Ada', 'Alex sees the coffee', 'Zach ran the shop',
     'Sam ran to the bank', 'Sam ran to the bank near the shop',
     'Sam ran to the bank near the shop near Ada', …}
```

GRAMMAR

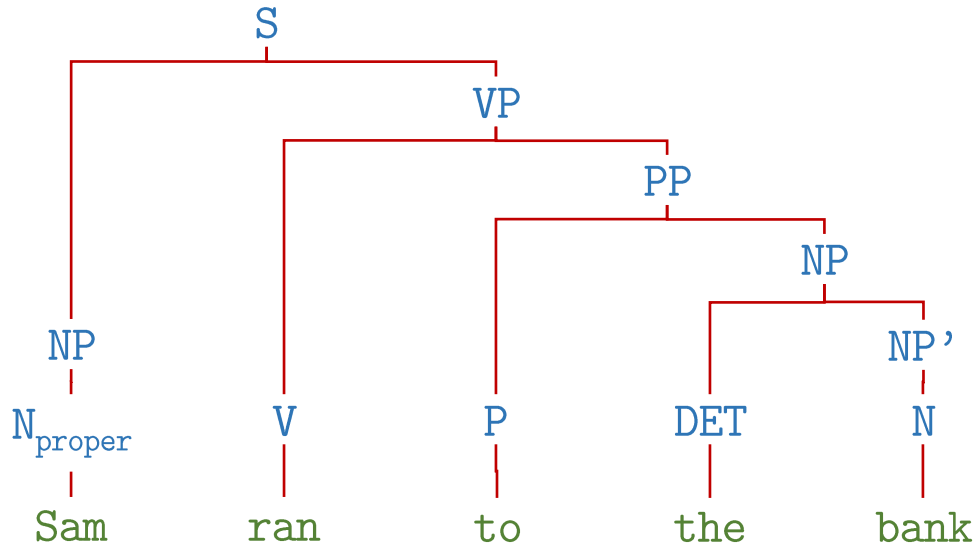
- Another way to think about grammars is as a deductive process.
- Each production rule can be interpreted as a deduction rule.
- E.g., from ‘the’, we can deduce DET from the rule DET -> ‘the’.
- From NP and VP, we can deduce S from the rule S -> NP VP.

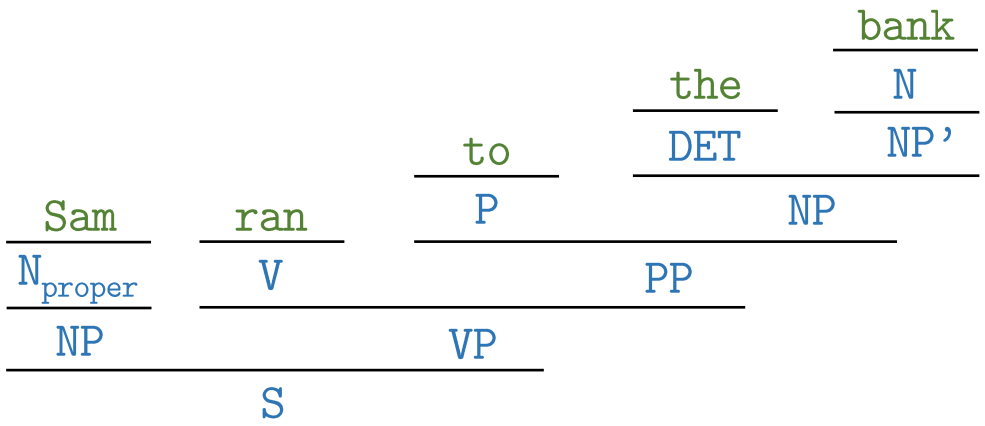




GRAMMAR

- The syntax tree is effectively a proof tree:
  - We have "proven" S from 'Sam ran to the bank'.
- Thus, syntax trees are also interchangeably called derivation trees.
- The task of parsing is equivalent to deriving/proving S from an input string.





GRAMMAR

- It is easier to parse some grammars than others.
- Consider the language containing strings with only 1's.
  - It's easy to recognize strings in this language:
  - Just check if the input string contains only 1's.
- Can we write this as a grammar?
- Note the two grammars below generate the same set of strings.
  - Thus, they are called weakly equivalent.

```
S -> '1' S
S -> '1'
```

or

```
S -> S '1'
S -> '1'
```

CONTEXT-FREE GRAMMARS

- Context-free grammars are grammars where each production rule has form:

  `A -> α`

  where `A` is a nonterminal,

  and `α` is any sequence containing terminal or nonterminal symbols.

- A context-free language is a language generated by a context-free grammar.
  - E.g., the language of strings containing '(' and ')' where the parentheses are balanced.

    `L = {'()', '()()', '(())', '((()(()))())()', …}`

CONTEXT-FREE GRAMMARS

- Context-free grammars are grammars where each production rule has form:

  $A \rightarrow \alpha$

  where A is a nonterminal,

  and α is any sequence containing terminal or nonterminal symbols.

- A context-free language is a language generated by a context-free grammar.
  - E.g., the language of strings containing '(' and ')' where the parentheses are balanced.

```
S -> '(' S ')'
S -> S S
S -> '()'
```

CONTEXT-FREE GRAMMARS

- Another example CFG is the fragment of English from the previous lecture:

```
S  -> NP VP
VP ->  V NP
VP ->  V PP
PP ->  P NP
NP -> NP PP
NP ->  N_proper
NP -> DET NP'
NP' -> N
```

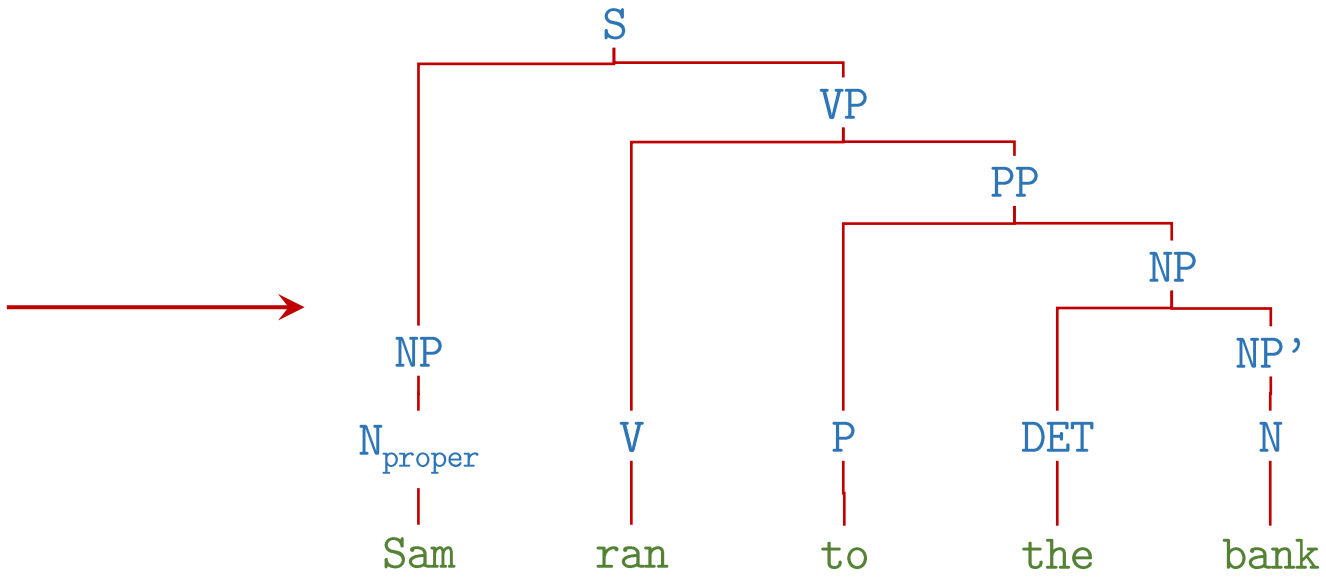
```
V -> 'ran'
V -> 'sees'
V -> 'see'
P -> 'to'
P -> 'near'
N -> 'bank'
N -> 'coffee'
N -> 'shop'
```

```
N_proper -> 'Ada'
N_proper -> 'Alex'
N_proper -> 'Sam'
N_proper -> 'Zach'
DET -> 'the'
DET -> 'a'
```

PARSING CFGS

- (Syntactic) parsing is the task of converting a sentence into a syntax tree, given a grammar.

‘Sam ran to the bank’



CHOMSKY NORMAL FORM

- Some CFGs are written in a simpler form called Chomsky normal form (Chomsky 1959).
- A CFG is in Chomsky normal form if all of its production rules are of the form:

```
A -> B C
A -> 't'
```

  where `A`, `B`, and `C` are nonterminals and `t` is a terminal.

- Some parsing algorithms are simpler to describe when applied to grammars in Chomsky normal form.
- Any CFG can be converted into an equivalent CFG in Chomsky normal form.

CONVERTING TO CHOMSKY NORMAL FORM

- Take the balanced-parentheses grammar from earlier:

```
S -> '(' S ')'
S -> S S
S -> '()'
```

- The first step is to change any production rule with mixed terminal-nonterminal symbols in the right-hand side into an equivalent rule that contains only nonterminals.

```
S -> P_L S P_R        P_L -> '('
S -> S S              P_R -> ')'
S -> '()'
```

CONVERTING TO CHOMSKY NORMAL FORM

- Next, repeat the following:
  - For any rule with more than two symbols in the right-hand side,
    - E.g., $A \rightarrow B_1\ B_2\ \ldots\ B_n$
  - Create a new nonterminal that replaces the last n – 1 symbols,
    - E.g., $A \rightarrow B_1\ C$
  - And create a new production rule for this replacement.
    - E.g., $C \rightarrow B_2\ \ldots\ B_n$

$S \rightarrow P_L\ S\ P_R$
$S \rightarrow S\ S$
$S \rightarrow$ '()'

$P_L \rightarrow$ '('
$P_R \rightarrow$ ')'

CONVERTING TO CHOMSKY NORMAL FORM

- Next, repeat the following:
  - For any rule with more than two symbols in the right-hand side,
    - E.g., $A \rightarrow B_1\ B_2\ \ldots\ B_n$
  - Create a new nonterminal that replaces the last n – 1 symbols,
    - E.g., $A \rightarrow B_1\ C$
  - And create a new production rule for this replacement.
    - E.g., $C \rightarrow B_2\ \ldots\ B_n$

$S \rightarrow P_L\ S_2$
$S \rightarrow S\ S$
$S \rightarrow$ '()'

$P_L \rightarrow$ '('
$P_R \rightarrow$ ')'
$S_2 \rightarrow S\ P_R$

CONVERTING TO CHOMSKY NORMAL FORM

- If there are any "unit" rules `A -> B`, for nonterminals `A` and `B`,
  - Convert any rule of the form `B -> ...` into `A -> ...`
  - And we can remove the rule `A -> B` as well.
- This type of rule doesn't appear in the example grammar below.
- But may appear in CFGs more generally.

$$S \rightarrow P_L\ S_2$$
$$S \rightarrow S\ S$$
$$S \rightarrow \text{`()'}$$

$$P_L \rightarrow \text{`('}$$
$$P_R \rightarrow \text{`)'}$$
$$S_2 \rightarrow S\ P_R$$

PARSING CFGS

- The CKY (Cocke-Kasami-Younger; or CYK) algorithm is a simple application of dynamic programming to parsing CFGs in Chomsky normal form.
- The algorithm was described by Kasami (1965) and Younger (1967), and rediscovered later by Cocke and Schwartz (1970).

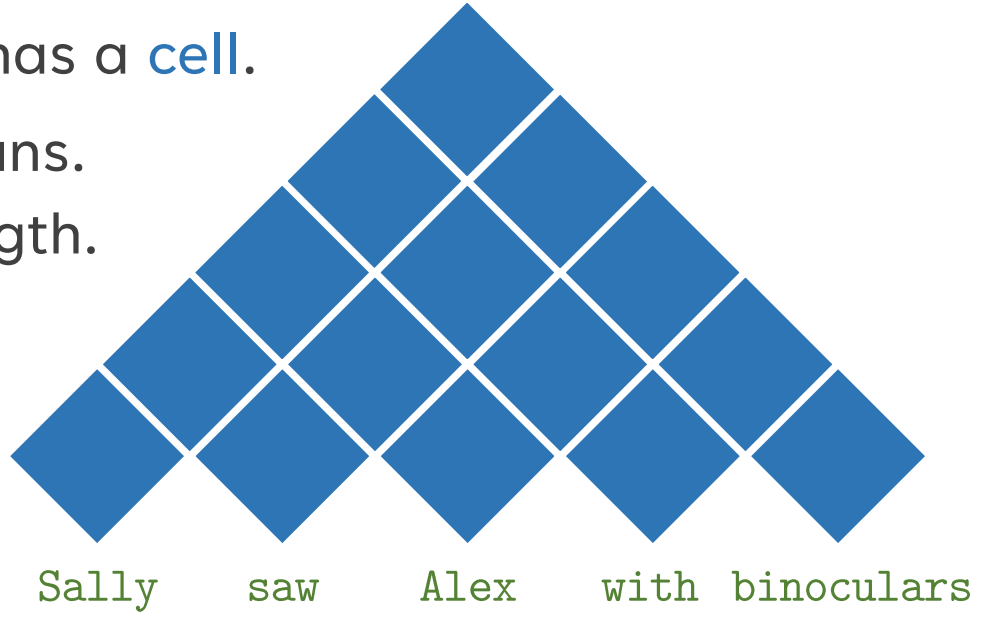
PARSING CFGS

- The CKY (Cocke-Kasami-Younger; or CYK) algorithm is a simple application of dynamic programming to parsing CFGs in Chomsky normal form.
- In dynamic programming, we solve a problem by breaking it down into simpler subproblems.
  - The solutions to the subproblems makes it easier to solve the full problem.
- In parsing CFGs,
  - Suppose we are given a sentence '`Sally saw Alex with binoculars.`'
  - And suppose we know that '`Sally`' is a noun phrase (`NP`) and '`saw Alex with binoculars`' is a verb phrase (`VP`).
  - Our grammar contains the rule `S -> NP VP`.
  - We can conclude that the full sentence can be parsed as `S`.

CKY PARSING

- The CKY (Cocke-Kasami-Younger; or CYK) algorithm is a simple application of dynamic programming to parsing CFGs in Chomsky normal form.
- We use a data structure called a chart.
  - For each span `(i,j)` the chart has a cell.
- Start parsing with the smallest spans.
  - Progressively increase span length.

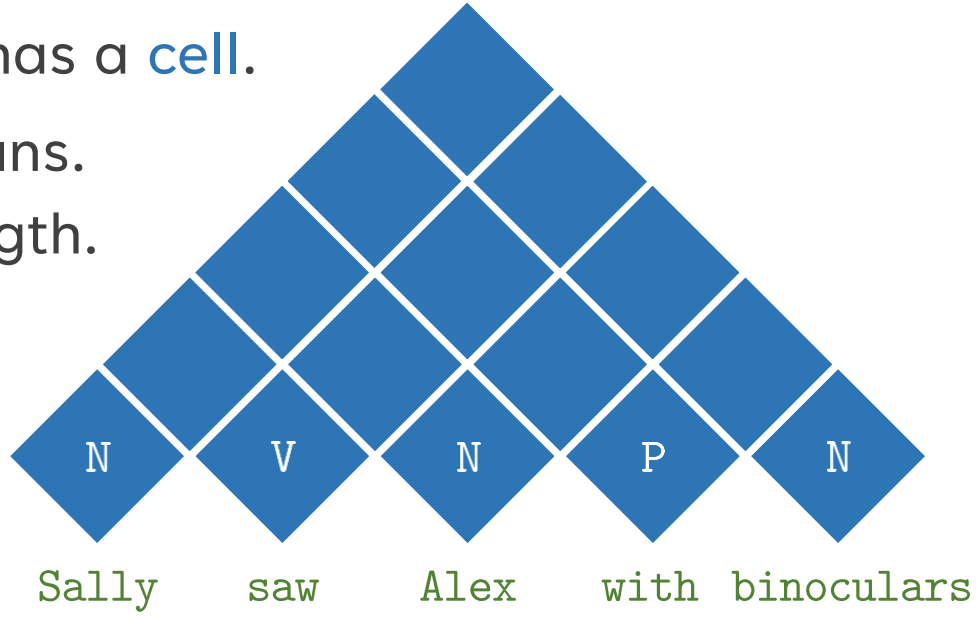
```
S  -> NP VP       V -> 'saw'
VP ->  V NP       P -> 'with'
VP -> VP PP       N -> 'binoculars'
PP ->  P NP       N -> 'Sally'
NP -> NP PP       N -> 'Alex'
NP ->  N
```



CKY PARSING

- The CKY (Cocke-Kasami-Younger; or CYK) algorithm is a simple application of dynamic programming to parsing CFGs in Chomsky normal form.
- We use a data structure called a chart.
  - For each span `(i,j)` the chart has a cell.
- Start parsing with the smallest spans.
  - Progressively increase span length.

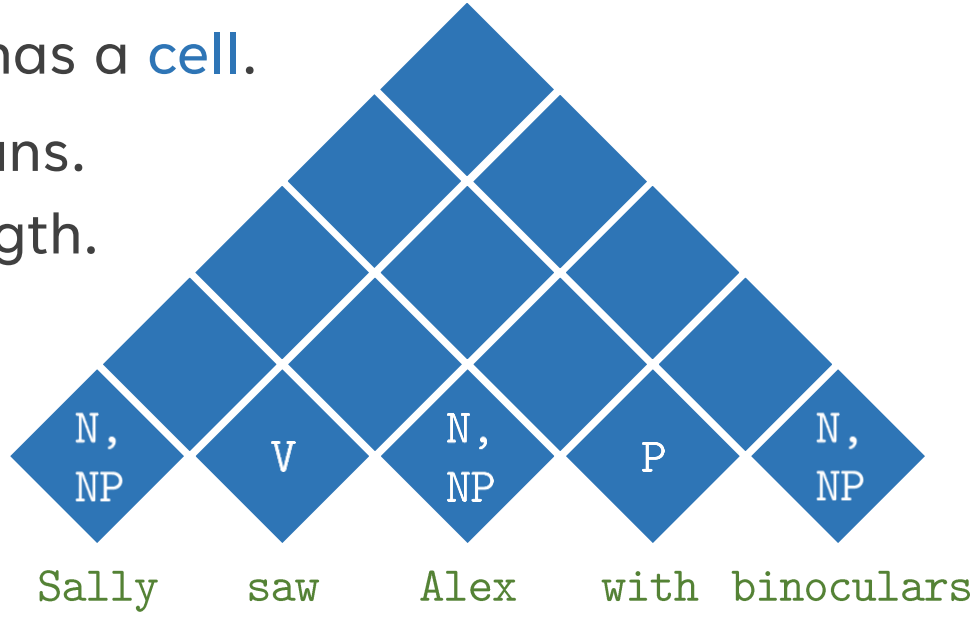
```
S  -> NP VP       V -> 'saw'
VP ->  V NP       P -> 'with'
VP -> VP PP       N -> 'binoculars'
PP ->  P NP       N -> 'Sally'
NP -> NP PP       N -> 'Alex'
NP ->  N
```



CKY PARSING

- The CKY (Cocke-Kasami-Younger; or CYK) algorithm is a simple application of dynamic programming to parsing CFGs in Chomsky normal form.
- We use a data structure called a chart.
  - For each span `(i,j)` the chart has a cell.
- Start parsing with the smallest spans.
  - Progressively increase span length.

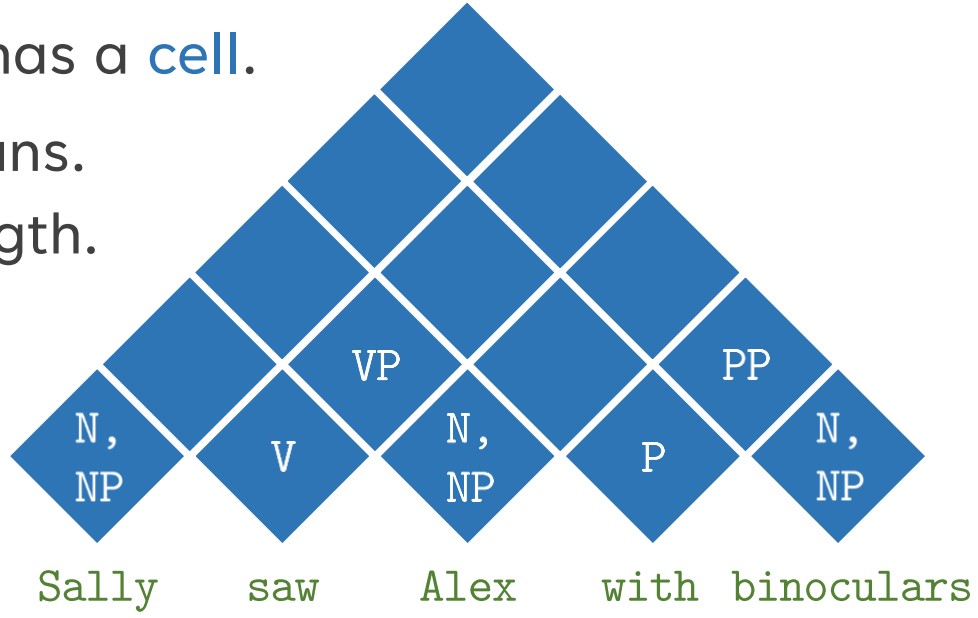
```
S  -> NP VP       V -> 'saw'
VP ->  V NP       P -> 'with'
VP -> VP PP       N -> 'binoculars'
PP ->  P NP       N -> 'Sally'
NP -> NP PP       N -> 'Alex'
NP ->  N
```



CKY PARSING

- The CKY (Cocke-Kasami-Younger; or CYK) algorithm is a simple application of dynamic programming to parsing CFGs in Chomsky normal form.
- We use a data structure called a chart.
  - For each span (i,j) the chart has a cell.
- Start parsing with the smallest spans.
  - Progressively increase span length.

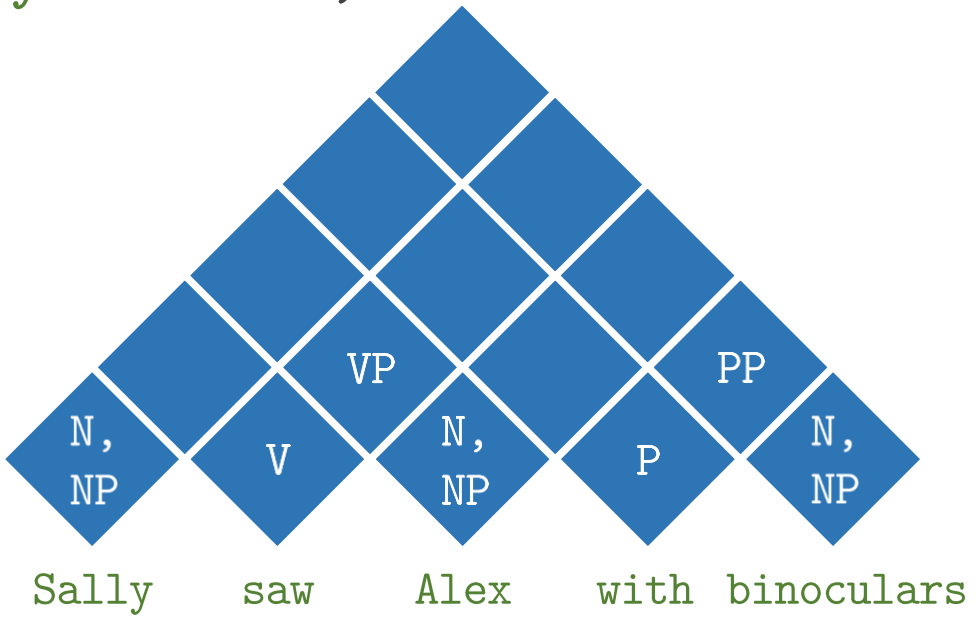
```
S  -> NP VP       V -> 'saw'
VP ->  V NP       P -> 'with'
VP -> VP PP       N -> 'binoculars'
PP ->  P NP       N -> 'Sally'
NP -> NP PP       N -> 'Alex'
NP ->  N
```



CKY PARSING

- For each cell (i,j), we have to consider all pairs of cells (i,k),(k,j) for all k such that $i \leq k \leq j$.
- For example, for the cell with 'Sally saw Alex,'

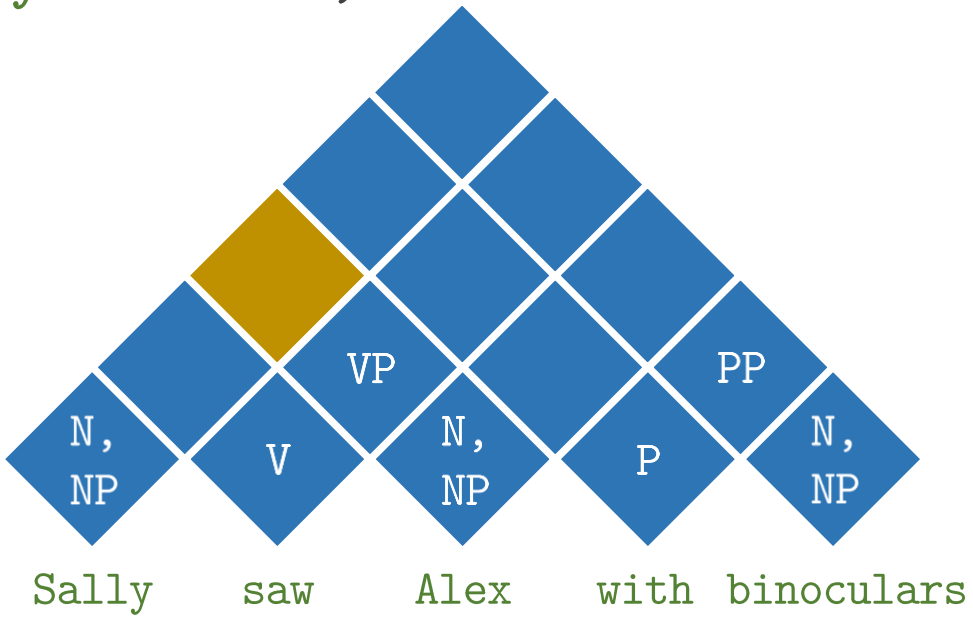
```
S  -> NP VP      V -> 'saw'
VP ->  V NP      P -> 'with'
VP -> VP PP      N -> 'binoculars'
PP ->  P NP      N -> 'Sally'
NP -> NP PP      N -> 'Alex'
NP ->  N
```



CKY PARSING

- For each cell (i,j), we have to consider all pairs of cells (i,k),(k,j) for all k such that $i \leq k \leq j$.
- For example, for the cell with 'Sally saw Alex,'
- We must consider combining both:

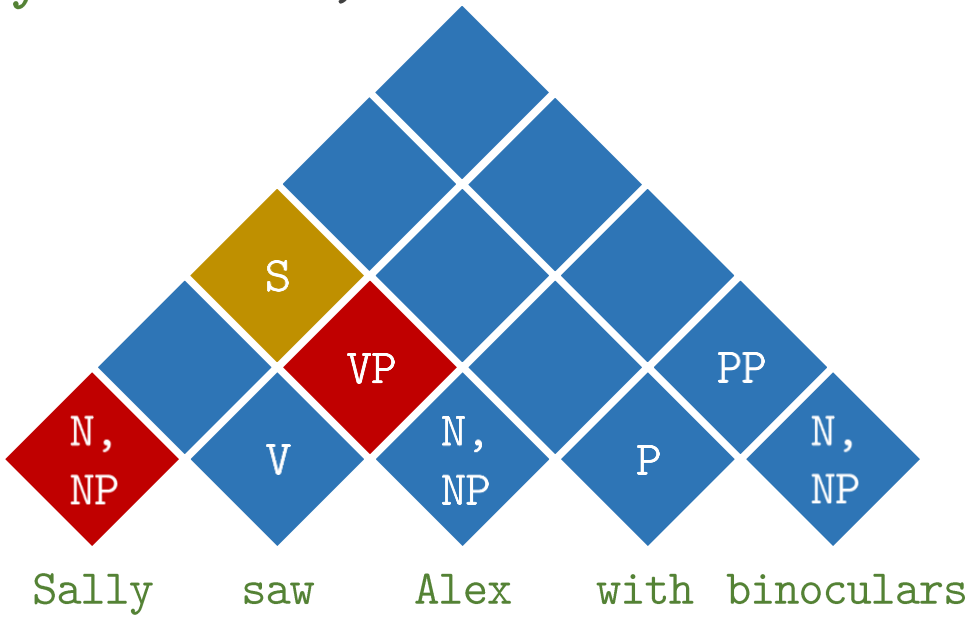
```
S  -> NP VP      V -> 'saw'
VP ->  V NP      P -> 'with'
VP -> VP PP      N -> 'binoculars'
PP ->  P NP      N -> 'Sally'
NP -> NP PP      N -> 'Alex'
NP ->  N
```



CKY PARSING

- For each cell `(i,j)`, we have to consider all pairs of cells `(i,k),(k,j)` for all `k` such that `i` ≤ `k` ≤ `j`.
- For example, for the cell with '`Sally saw Alex`,'
- We must consider combining both:
  - '`Sally`' and '`saw Alex`'

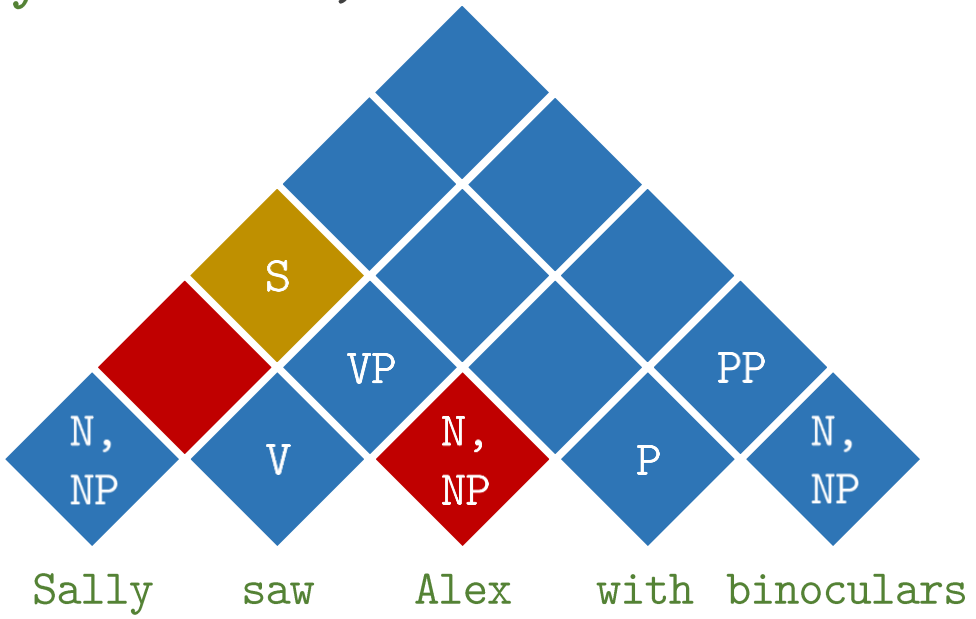
```
S  -> NP VP      V -> 'saw'
VP ->  V NP      P -> 'with'
VP -> VP PP      N -> 'binoculars'
PP ->  P NP      N -> 'Sally'
NP -> NP PP      N -> 'Alex'
NP ->  N
```



CKY PARSING

- For each cell `(i,j)`, we have to consider all pairs of cells `(i,k),(k,j)` for all `k` such that `i` ≤ `k` ≤ `j`.
- For example, for the cell with ‘`Sally saw Alex`,’
- We must consider combining both:
  - ‘`Sally`’ and ‘`saw Alex`’
  - ‘`Sally saw`’ and ‘`Alex`’

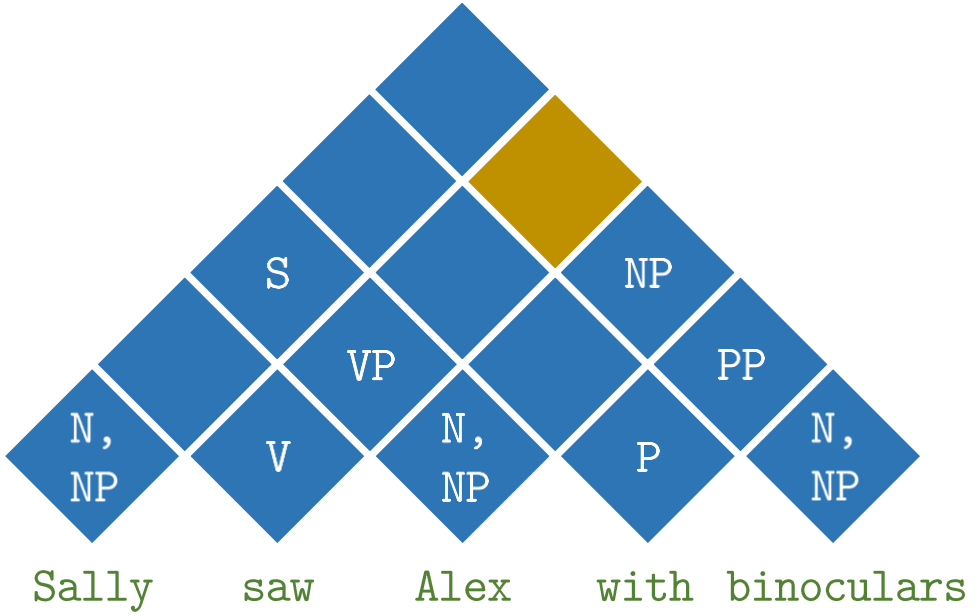
```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```



CKY PARSING

- Similarly for the span ‘saw Alex with binoculars’ we must consider:

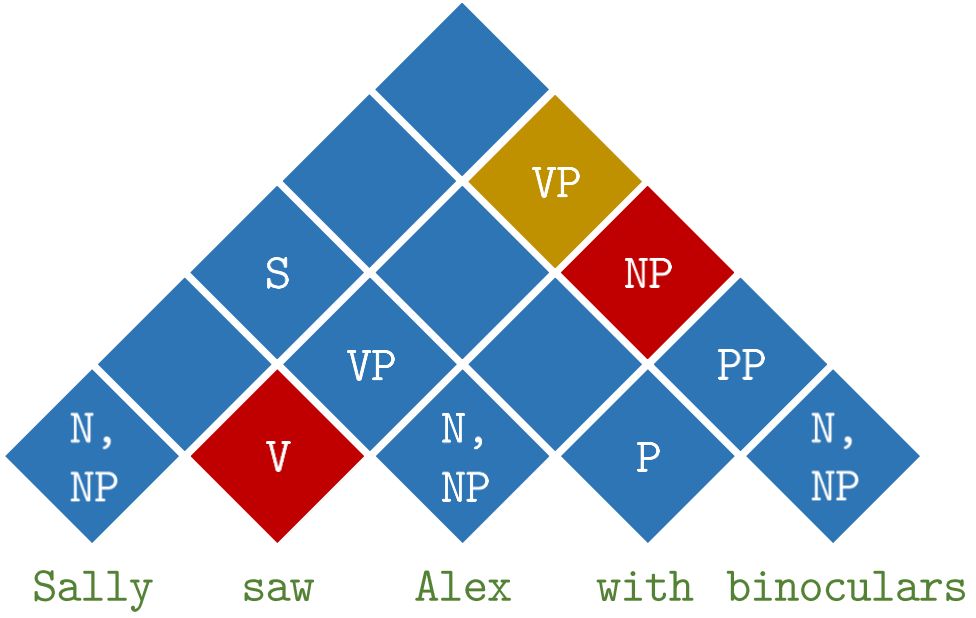
```
S  -> NP VP        V -> ‘saw’
VP ->  V NP        P -> ‘with’
VP -> VP PP        N -> ‘binoculars’
PP ->  P NP        N -> ‘Sally’
NP -> NP PP        N -> ‘Alex’
NP ->  N
```



CKY PARSING

- Similarly for the span ‘saw Alex with binoculars’ we must consider:
  - ‘saw’ and ‘Alex with binoculars’

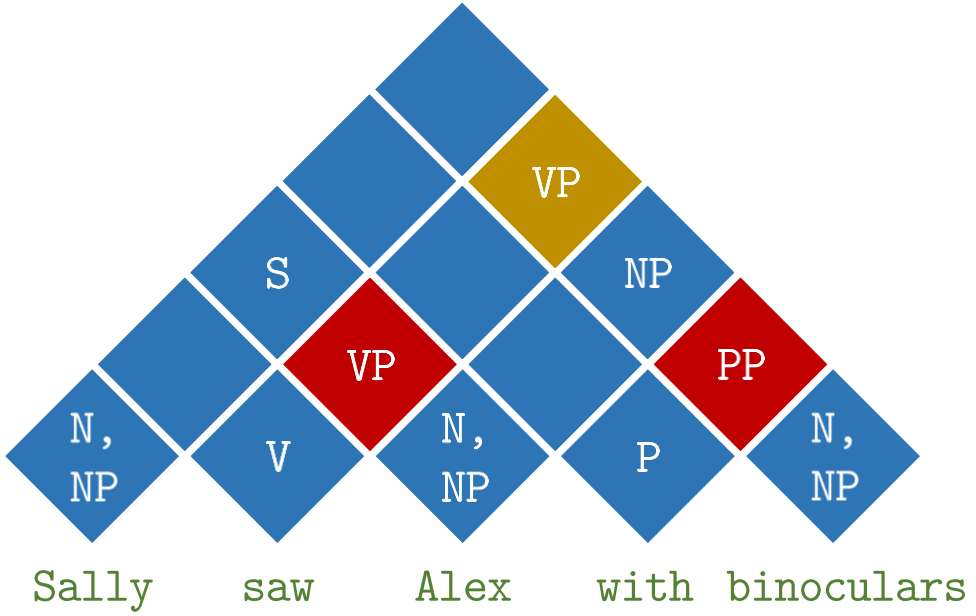
```
S  -> NP VP       V -> ‘saw’
VP ->  V NP       P -> ‘with’
VP -> VP PP       N -> ‘binoculars’
PP ->  P NP       N -> ‘Sally’
NP -> NP PP       N -> ‘Alex’
NP ->  N
```



CKY PARSING

- Similarly for the span ‘saw Alex with binoculars’ we must consider:
  - ‘saw’ and ‘Alex with binoculars’
  - ‘saw Alex’ and ‘with binoculars’

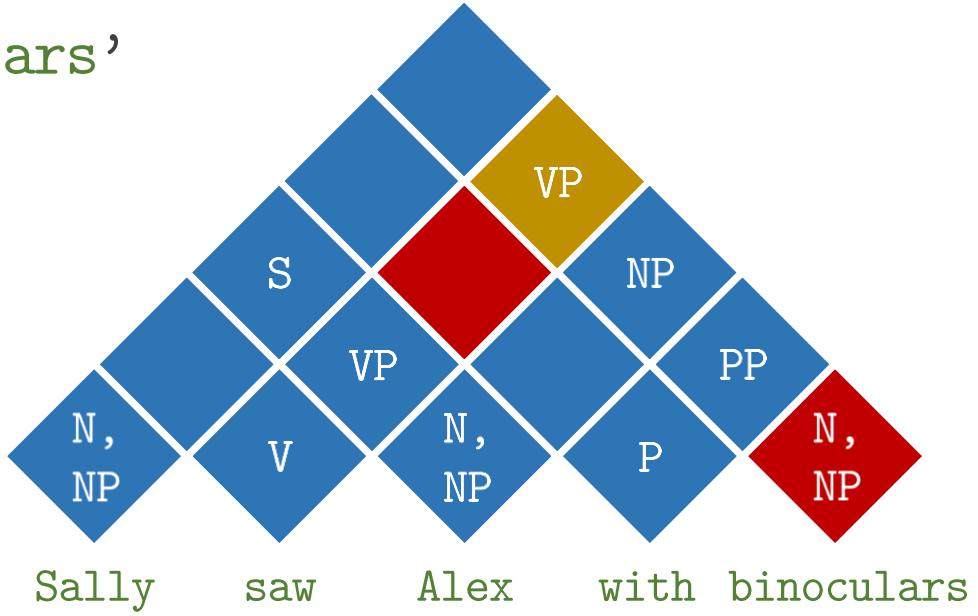
```
S  -> NP VP       V -> ‘saw’
VP ->  V NP       P -> ‘with’
VP -> VP PP       N -> ‘binoculars’
PP ->  P NP       N -> ‘Sally’
NP -> NP PP       N -> ‘Alex’
NP ->  N
```



CKY PARSING

- Similarly for the span ‘saw Alex with binoculars’ we must consider:
  - ‘saw’ and ‘Alex with binoculars’
  - ‘saw Alex’ and ‘with binoculars’
  - ‘saw Alex with’ and ‘binoculars’

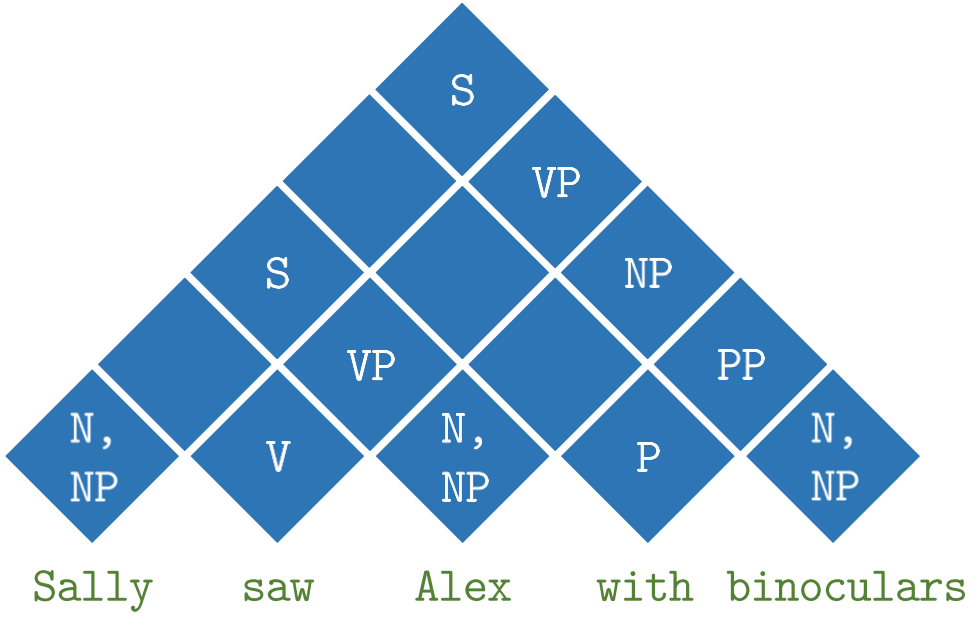
```
S  -> NP VP       V -> ‘saw’
VP ->  V NP       P -> ‘with’
VP -> VP PP       N -> ‘binoculars’
PP ->  P NP       N -> ‘Sally’
NP -> NP PP       N -> ‘Alex’
NP ->  N
```



CKY PARSING

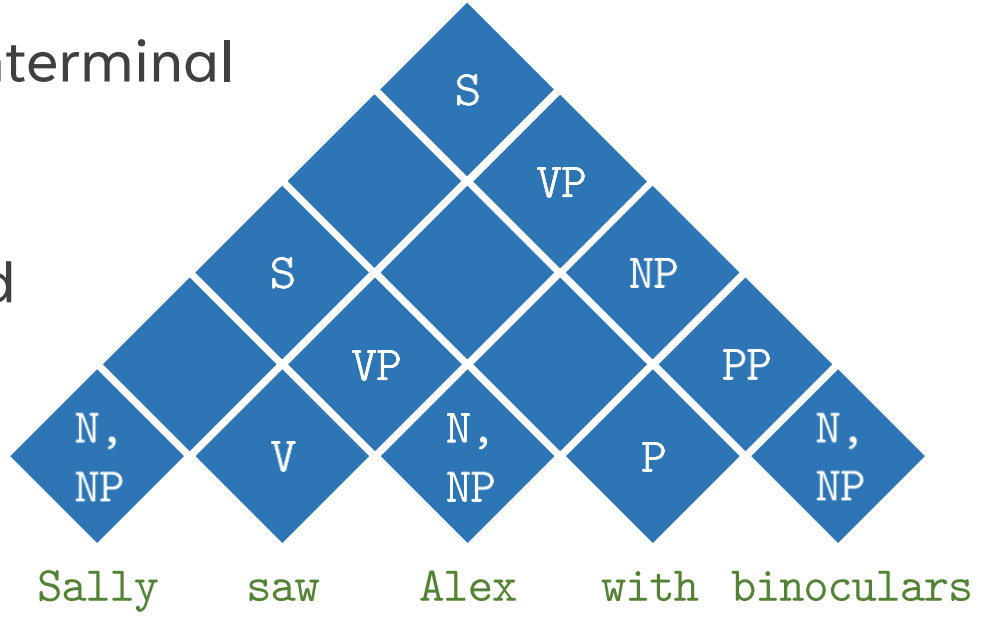
- We continue until we have processed all cells in the chart.
- In its simplest form, each cell simply records which nonterminals were parsed for that corresponding span.

```
S  -> NP VP        V -> 'saw'
VP ->  V NP        P -> 'with'
VP -> VP PP        N -> 'binoculars'
PP ->  P NP        N -> 'Sally'
NP -> NP PP        N -> 'Alex'
NP ->  N
```



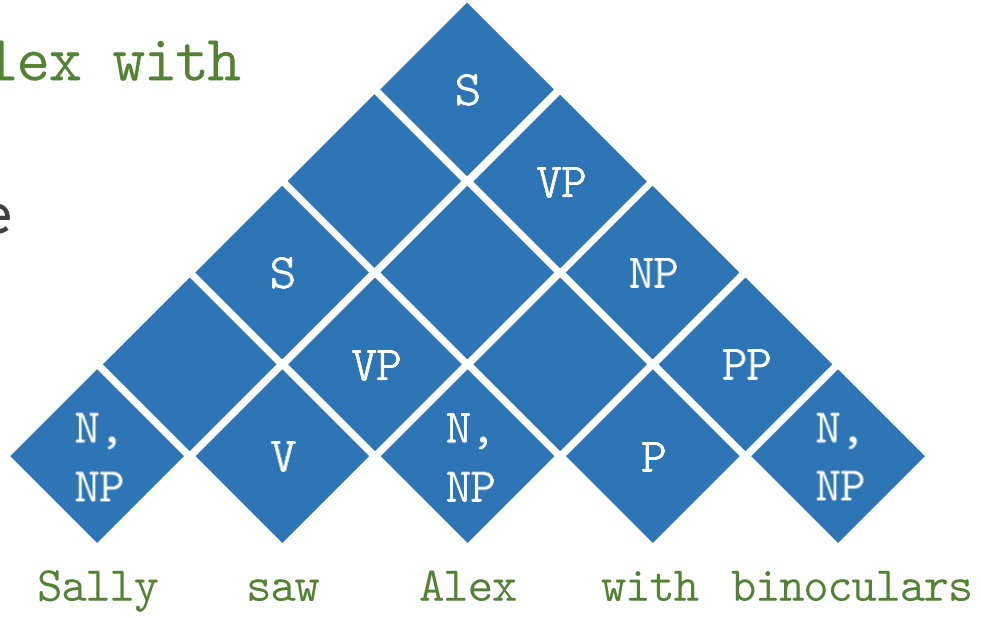
CKY PARSING

- We continue until we have processed all cells in the chart.
- In its simplest form, each cell simply records which nonterminals were parsed for that corresponding span.
- If instead, we record *how* each nonterminal was constructed in each cell,
- E.g., the VP for the span 'saw Alex with binoculars' was constructed from the V 'saw' and the NP 'Alex with binoculars,'
- We can reconstruct the syntax tree by following the pointers from the root of the chart.



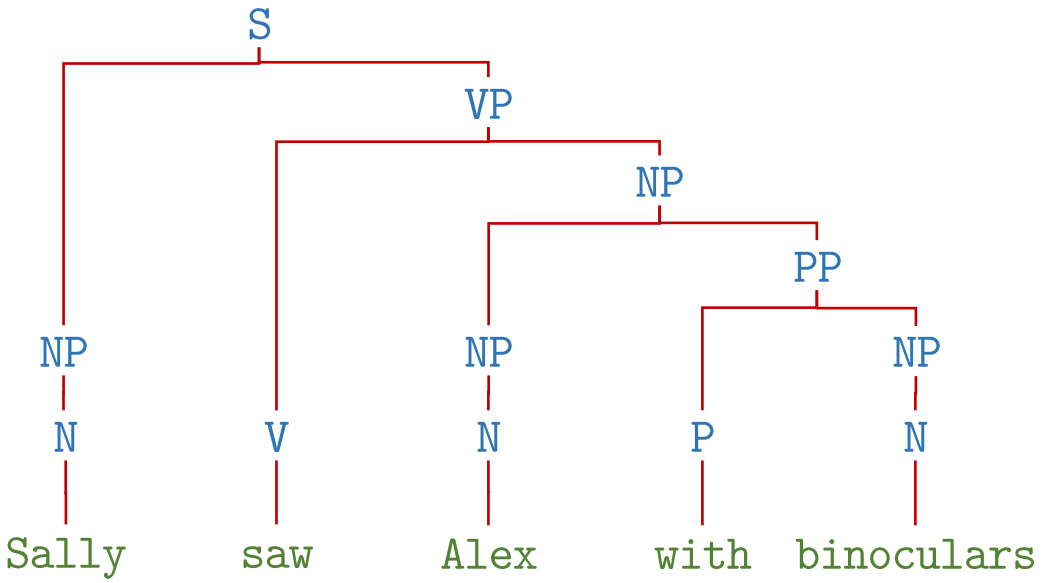
CKY PARSING

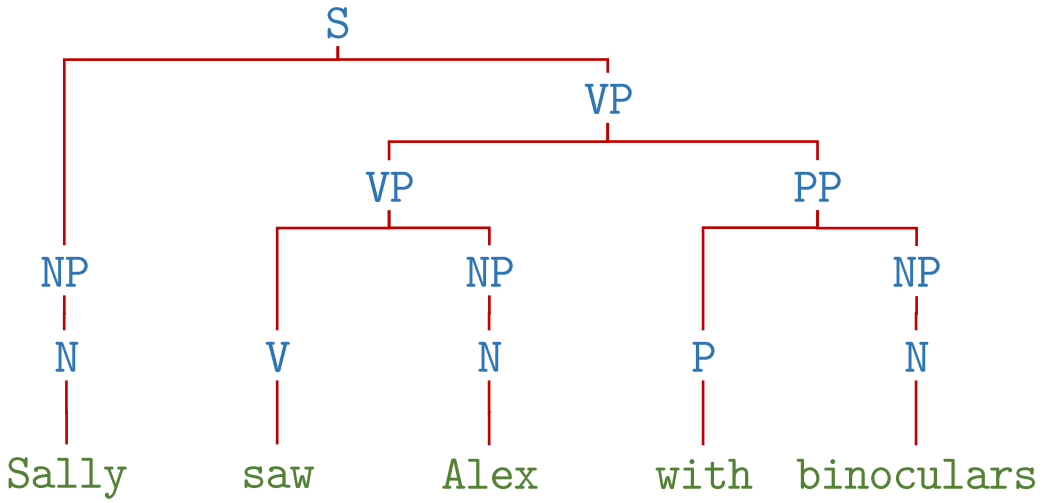
- CKY can also parse ambiguous sentences.
- Here, the VP for the span 'saw Alex with binoculars' can be constructed in two ways:
  1. from the V 'saw' and the NP 'Alex with binoculars,'
  2. from the VP 'saw Alex' and the PP 'with binoculars.'
- Both these constructions must be stored in the cell for 'saw Alex with binoculars.'



SYNTACTIC AMBIGUITY

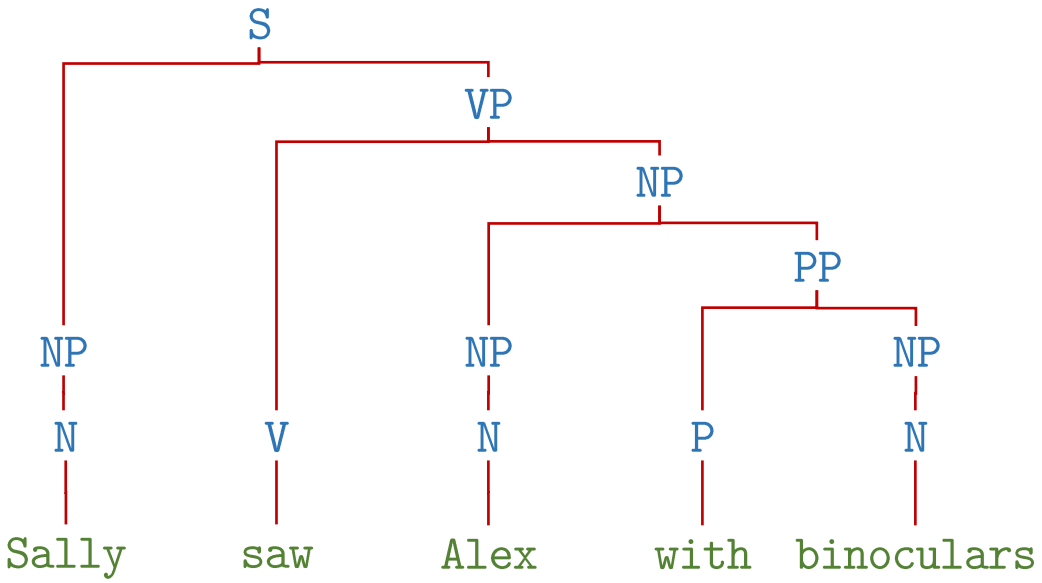
- Depending on which of the two ambiguous constructions we choose when reconstructing the syntax tree at that cell,
- We can obtain two different syntax trees:

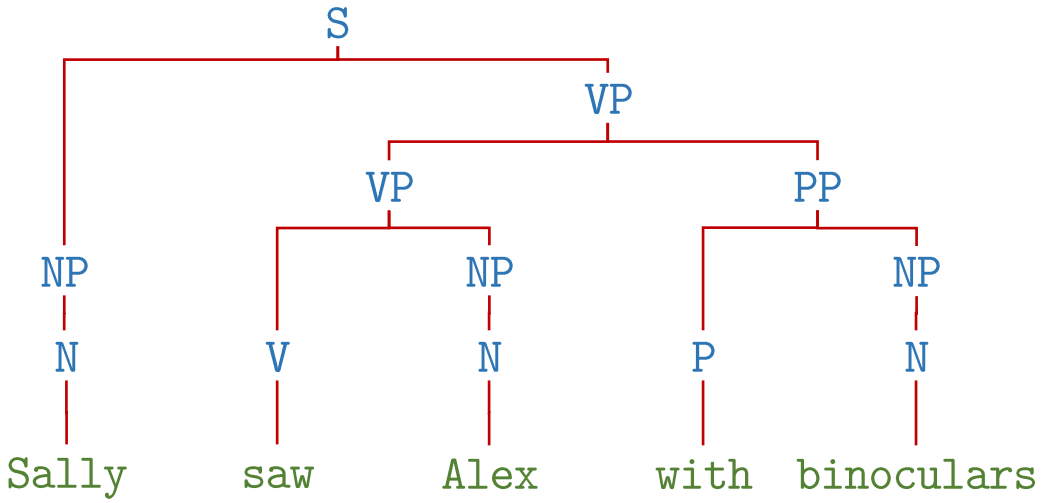




SYNTACTIC AMBIGUITY

- These ambiguous syntax trees correspond to two different interpretations:
  1. Alex has the binoculars, and Sally sees Alex.
  2. Sally uses binoculars to see Alex.





CKY RUNNING TIME

- What is the running time of CKY parsing?
- For each cell (i,j), we had to consider all pairs of cells $(i,k),(k,j)$ for all k such that $i \leq k \leq j$.
  - For each i, j, k, we iterate over each rule in the grammar to check for a match.
- We have (roughly half of) n^2 cells, where n is the length of the sentence.
- In the worst case, there are n possible values of k.
- Thus the running time is $O(|G|\, n^3)$,
  - Where $|G|$ is the number of production rules in the grammar G.

PARSING CFGS

- Another dynamic programming approach to CFG parsing was developed by Earley (1968, 1970).
  - Called Earley parsing.
- Unlike CKY which is a bottom-up parsing approach, Earley is top-down.
  - I.e., we start from the root of the syntax tree and work our way down to the leaves/terminals.

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- We start with the root nonterminal S.
  - Create an initial state for any rule of the form S -> ...:
- The dot ‘ . ’ indicates the current position of the parser.
  - In this example state, we haven’t parsed anything yet.
- Push this state onto a queue.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

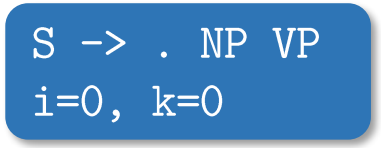
queue

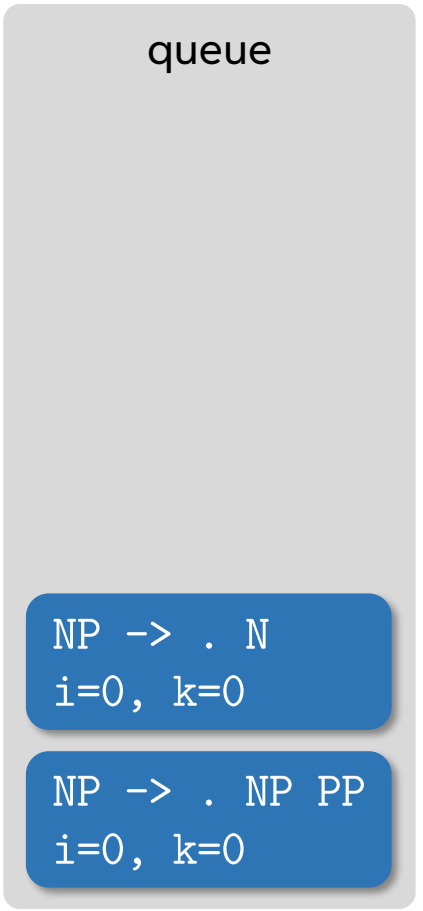
```
S -> . NP VP
i=0, k=0
```

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat the following:
  - Pop a state from the queue.
  - If the symbol following the ‘ . ’ is a nonterminal, do a prediction step.
  - Create a new state for any production rule of the form NP -> …
  - The old state is added to a list of “waiting” states.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```





EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat the following:
  - Pop a state from the queue.
  - We avoid repeating prediction steps for the same nonterminal at the same position (i.e., NP at position 0).
  - Otherwise we would have an infinite loop.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

queue

```
NP -> . NP PP
i=0, k=0
```

```
NP -> . N
i=0, k=0
```

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat the following:
  - Pop a state from the queue.
  - If the symbol following the ‘ . ’ is a nonterminal, do a prediction step.
  - Create a new state for any production rule of the form N -> …

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

NP -> . N
i=0, k=0

queue

N -> . ‘Sally’
i=0, k=0

N -> . ‘Alex’
i=0, k=0

N -> . ‘binoculars’
i=0, k=0

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat the following:
  - Pop a state from the queue.
  - If the symbol following ‘ . ’ is a terminal, do a scanning step.
  - If the token at position k in the sentence matches the terminal, push a new state where the ‘ . ’ moves forward.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

N -> . ‘binoculars’
i=0, k=0

queue

N -> . ‘Sally’
i=0, k=0

N -> . ‘Alex’
i=0, k=0

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat the following:
  - Pop a state from the queue.
  - If the symbol following ‘ . ’ is a terminal, do a scanning step.
  - If the token at position k in the sentence matches the terminal, push a new state where the ‘ . ’ moves forward.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

queue

N -> . ‘Alex’
i=0, k=0

N -> . ‘Sally’
i=0, k=0

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat the following:
  - Pop a state from the queue.
  - If the symbol following ‘ . ’ is a terminal, do a scanning step.
  - If the token at position k in the sentence matches the terminal, push a new state where the ‘ . ’ moves forward.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

queue

```
N -> . ‘Sally’
i=0, k=0
```

```
N -> ‘Sally’ .
i=0, k=1
```

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat the following:
  - Pop a state from the queue.
  - If the ‘ . ’ is at the end of the rule, do a completion step.
  - For all “waiting” states where an N follows the ‘ . ’, push a new state where the ‘ . ’ moves forward.
- Here, we have successfully parsed N at span (0,1).

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

queue

```
N -> ‘Sally’ .
i=0, k=1
```

```
NP -> N .
i=0, k=1
```

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat the following:
  - Pop a state from the queue.
  - If the ‘ . ’ is at the end of the rule, do a completion step.
  - For all “waiting” states where an N follows the ‘ . ’, push a new state where the ‘ . ’ moves forward.
- Here, we have successfully parsed NP at span (0,1).

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

NP -> N .
i=0, k=1

queue

S -> NP . VP
i=0, k=1

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat the following:
  - Pop a state from the queue.
  - If the ‘ . ’ is at the end of the rule, do a completion step.
  - For all “waiting” states where an N follows the ‘ . ’, push a new state where the ‘ . ’ moves forward.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

queue

VP -> . VP PP
i=1, k=1

VP -> . V NP
i=1, k=1

S -> NP . VP
i=0, k=1

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat the following:
  - Pop a state from the queue.
  - If the ‘ . ’ is at the end of the rule, do a completion step.
  - For all “waiting” states where an N follows the ‘ . ’, push a new state where the ‘ . ’ moves forward.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

queue

V -> . ‘saw’
i=1, k=1

VP -> . V NP
i=1, k=1

VP -> . VP PP
i=1, k=1

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> …
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

queue

```
VP -> . VP PP
i=1, k=1
```

```
V -> . ‘saw’
i=1, k=1
```

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> …
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

queue

V -> . ‘saw’
i=1, k=1

V -> ‘saw’ .
i=1, k=2

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> ...
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

queue

V -> ‘saw’ .
i=1, k=2

VP -> V . NP
i=1, k=2

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> …
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

VP -> V . NP
i=1, k=2

queue

NP -> . N
i=2, k=2

NP -> . NP PP
i=2, k=2

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> …
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

queue

```
NP -> . NP PP
i=2, k=2
```

```
NP -> . N
i=2, k=2
```

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> …
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

NP -> . N
i=2, k=2

queue

N -> . ‘Sally’
i=2, k=2

N -> . ‘Alex’
i=2, k=2

N -> . ‘binoculars’
i=2, k=2

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> …
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

N -> . ‘binoculars’
i=2, k=2

queue

N -> . ‘Sally’
i=2, k=2

N -> . ‘Alex’
i=2, k=2

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> …
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

N -> . ‘Alex’
i=2, k=2

queue

N -> ‘Alex’ .
i=2, k=3

N -> . ‘Sally’
i=2, k=2

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> …
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

queue

```
N -> . ‘Sally’
i=2, k=2
```

```
N -> ‘Alex’ .
i=2, k=3
```

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> …
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

queue

N -> ‘Alex’ .
i=2, k=3

NP -> N .
i=2, k=3

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> ...
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

NP -> N .
i=2, k=3

queue

NP -> NP . PP
i=2, k=3

VP -> V NP .
i=1, k=3

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> ...
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

VP -> V NP .
i=1, k=3

queue

VP -> VP . PP
i=1, k=3

S -> NP VP .
i=0, k=3

NP -> NP . PP
i=2, k=3

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> ...
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

NP -> NP . PP
i=2, k=3

queue

PP -> . P NP
i=3, k=3

VP -> VP . PP
i=1, k=3

S -> NP VP .
i=0, k=3

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> …
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.
- Here we successfully parsed S,
  - But we didn’t reach the end of the sentence,
  - So we continue.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

S -> NP VP .
i=0, k=3

queue

PP -> . P NP
i=3, k=3

VP -> VP . PP
i=1, k=3

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> …
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

queue

```
VP -> VP . PP
i=1, k=3
```

```
PP -> . P NP
i=3, k=3
```

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> …
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

queue

PP -> . P NP
i=3, k=3

P -> . ‘with’
i=3, k=3

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> …
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

queue

P -> . ‘with’
i=3, k=3

P -> ‘with’ .
i=3, k=4

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> …
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

queue

P -> ‘with’ .
i=3, k=4

PP -> P . NP
i=3, k=4

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> …
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

queue

NP -> . N
i=4, k=4

NP -> . NP PP
i=4, k=4

PP -> P . NP
i=3, k=4

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> …
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

queue

```
NP -> . NP PP
i=4, k=4
```

```
NP -> . N
i=4, k=4
```

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> …
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

NP -> . N
i=4, k=4

queue

N -> . ‘Sally’
i=4, k=4

N -> . ‘Alex’
i=4, k=4

N -> . ‘binoculars’
i=4, k=4

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> …
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

N -> . ‘binoculars’
i=4, k=4

queue

N -> ‘binoculars’ .
i=4, k=5

N -> . ‘Sally’
i=4, k=4

N -> . ‘Alex’
i=4, k=4

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> …
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

N -> . ‘Alex’
i=4, k=4

queue

N -> ‘binoculars’ .
i=4, k=5

N -> . ‘Sally’
i=4, k=4

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> …
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

queue

N -> . ‘Sally’
i=4, k=4

N -> ‘binoculars’ .
i=4, k=5

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> ...
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

queue

N -> ‘binoculars’ .
i=4, k=5

NP -> N .
i=4, k=5

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> ...
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

queue

```
NP -> N .
i=4, k=5
```

```
PP -> P NP .
i=3, k=5
```

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> ...
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

PP -> P NP .
i=3, k=5

queue

VP -> VP PP .
i=1, k=5

NP -> NP PP .
i=2, k=5

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> ...
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

NP -> NP PP .
i=2, k=5

queue

VP -> V NP .
i=1, k=5

VP -> VP PP .
i=1, k=5

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> ...
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

VP -> VP PP .
i=1, k=5

queue

S -> NP VP .
i=0, k=5

VP -> V NP .
i=1, k=5

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> ...
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

VP -> V NP .
i=1, k=5

queue

S -> NP VP .
i=0, k=5

S -> NP VP .
i=0, k=5

EARLEY PARSING

- Consider the example: ‘Sally saw Alex with binoculars’
- Repeat until we complete a rule of the form S -> ...
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.
- We have found a valid parse of S for the full sentence.
- If instead, the queue became empty, then the input sentence is not part of the language.

```
S  -> NP VP      V -> ‘saw’
VP ->  V NP      P -> ‘with’
VP -> VP PP      N -> ‘binoculars’
PP ->  P NP      N -> ‘Sally’
NP -> NP PP      N -> ‘Alex’
NP ->  N
```

S -> NP VP .
i=0, k=5

queue

S -> NP VP .
i=0, k=5

EARLEY PARSING

- If, for each state, we keep track of substates that completed it,
  - E.g., in the state `S -> NP VP .`, we keep a pointer to the completed state for `NP`, and another for `VP`,
  - We can reconstruct the syntax tree by following the backpointers from the completed state for `S`.
  - Similar to CKY.
- Unlike CKY, Earley can be applied to any CFG,
  - Including those not in Chomsky normal form.

EARLEY VS CKY

- Since Earley is a top-down parser, it can avoid producing phantom parses.
  - These are parses that are useless in the final parse.
  - E.g., ‘It traveled 90% of the speed of light.’
    - ‘speed’ and ‘light’ are nouns here,
    - But ‘speed’ can be a verb (e.g., ‘Don’t speed on the highway.’).
    - And ‘light’ can be an adjective (e.g., ‘the light jacket’).
  - CKY would produce all valid parses of these phrases (noun, verb, and adjective phrases).
  - But Earley would only produce the phrases that are valid in the context of the whole sentence.
    - In this example, ‘speed’ and ‘light’ would only be parsed as nouns.

EARLEY RUNNING TIME

- Is Earley faster than CKY?
- In the worst case, Earley would also require $O(|G|n^3)$ time.
- However, for unambiguous grammars, Earley runs in $O(|G|n^2)$.
- There is a smaller class of simpler CFGs called deterministic CFGs on which Earley can run in $O(|G|n)$ time.

DO LLMS UNDERSTAND SYNTAX?

- Not too much work studying this question.
- Zhou et al. (2023) tested various LLMs on syntax questions.

Sentence: Pierre Vinken ***will join*** the board as a nonexecutive director Nov. 29.

Question: In the above sentence, the ***grammatical subject*** of "**will join**" is _____.

Options:

A. The board
B. Pierre Vinken
C. 61 years old
D. A nonexecutive director

Answer: **B**

[Zhou et al., 2023]

DO LLMS UNDERSTAND SYNTAX?

- They produced these questions using the Penn TreeBank.
  - Which is a large corpus of sentences from the Wall Street Journal which were hand-annotated with syntax trees.

Sentence: Pierre Vinken ***will join*** the board as a nonexecutive director Nov. 29.

Question: In the above sentence, the ***grammatical subject*** of "**will join**" is _____.

Options:

A. The board
B. Pierre Vinken
C. 61 years old
D. A nonexecutive director

Answer: **B**

[Zhou et al., 2023]

DO LLMS UNDERSTAND SYNTAX?

- They tested the models on questions about many different aspects of syntax:
  - They asked true/false, multiple choice, and fill-in-the-blank questions.

Syntactic Knowledge Points	Abbr.	#TF	#MC	#FITB	Example
Grammatical **S**ubject	GS	130	105	105	**Desks** *are cleared* by John.
Subject **C**omplement	SC	130	85	85	John *is* **a teacher**.
Direct **O**bject	DO	150	145	145	John *gave* me **a book**.
Indirect **O**bject	IO	30	20	20	John *gave* **me** a book.
Main **V**erb **P**hrase	MVP	440‡	170	170	John **gave** me a book.
ADJectival modifier†	ADJ	185	165	135	I enjoy *the book* **John gave me**.
ADVerbial modifier (Adjunct)	ADV	165	125	115	I *read* the book **quickly**.
COordination	CO	165	160	155	We **will play** football and **watch** TV.
Prepositional **P**hrase **A**ttachment	PPA	110	100	100	I like **the book** *on my shelf*. I **hide** the book *on my shelf*.

[Zhou et al., 2023]

DO LLMS UNDERSTAND SYNTAX?

	Zero-shot					Few-shot				
	TF	MC	FITB		OA	TF	MC	FITB		OA
	Acc.	Acc.	Acc.	F_1		Acc.	Acc.	Acc.	F_1	
Random	50.00	25.00	0.68	23.21	28.66	50.00	25.00	0.68	23.21	28.66
Mistral 7B	51.08	50.42	40.19	57.01	50.03	56.50	56.59	55.60	69.58	58.56
Mistral 7B (Instruct)	57.65	52.93	36.12	53.17	51.74	56.06	54.60	46.05	62.68	55.01
Baichuan2 13B	52.11	54.98	36.21	53.84	50.71	52.05	57.67	52.59	66.39	56.40
Baichuan2 13B (Chat)	59.53	55.91	26.60	46.05	50.59	57.12	57.46	44.69	60.83	55.78
Falcon 40B	52.68	48.56	27.57	45.11	45.86	57.65	54.23	46.34	62.07	55.36
Falcon 40B (Instruct)	58.03	48.37	29.22	45.65	47.95	55.77	53.71	46.22	62.39	54.59
Llama 65B	58.59	56.00	45.63	62.62	56.24	52.24	55.23	61.10	74.11	58.36
Llama2 70B	57.09	66.14	46.21	63.57	59.37	57.34	66.95	61.59	75.11	64.21
Llama2 70B (Chat)	57.00	61.58	42.33	60.30	56.63	60.09	68.65	55.86	70.63	64.00
GPT3.5	59.53	58.70	55.34	71.44	60.54	63.38	73.58	57.28	72.36	67.26 🏆
GPT4	**81.88**	**88.19**	**63.98**	**77.78**	**80.32** 🏆	**88.83**	**92.28**	**69.32**	**83.10**	**85.77** 🏆

[Zhou et al., 2023]

DO LLMS UNDERSTAND SYNTAX?

Models	GS	SC	DO	IO	MVP	ADJ	ADV	PPA	CO
Mistral 7B	62.81	57.68	63.22	68.76	59.66	64.06	55.13	38.26	60.74
Mistral 7B (Instruct)	61.89	52.80	60.76	55.42	53.42	58.51	58.19	33.16	56.21
Baichuan2 13B	59.61	58.66	61.41	67.90	60.68	62.15	54.39	30.45	55.96
Baichuan2 13B (Chat)	63.81	58.52	59.97	65.04	57.31	61.78	50.79	33.13	56.05
Falcon 40B	61.38	55.45	57.21	64.90	60.36	60.11	50.36	36.05	55.71
Falcon 40B (Instruct)	57.50	56.17	57.40	58.26	60.78	62.55	49.54	36.55	51.67
Llama 65B	63.42	60.58	62.67	71.35	59.34	65.38	56.15	39.86	55.27
Llama2 70B	70.83	65.67	63.36	82.20	65.59	74.58	61.82	44.54	60.68
Llama2 70B (Chat)	68.86	56.22	67.14	68.76	70.36	75.04	60.00	49.72	56.33
GPT3.5	75.95	69.93	70.55	80.42	69.94	70.57	62.98	58.94	58.71
GPT4	**89.74**	**86.70**	**86.99**	**96.67**	**85.29**	**92.44**	**73.55**	**81.50**	**87.63**
Avg.	55.44	53.58	55.23	58.35	54.00	58.53	49.65	36.43	51.62

[Zhou et al., 2023]

DO LLMS UNDERSTAND SYNTAX?

- Larger models are better at answering questions about syntax.
- However, Penn TreeBank sentences contaminate the training set.
- The depth of syntax trees is also limited.
  - If a model truly understands a CFG, then it should be able to generalize to arbitrarily deep syntax trees.
  - On the other hand, humans are not able to parse sentences with arbitrarily deep syntax trees.
  - We have memory constraints.

DO LLMS UNDERSTAND SYNTAX?

- Answering questions about syntax is not the same as understanding syntax.
- Understanding syntax is required to answer questions correctly,
  - But not vice versa.
- Answering questions about syntax also requires additional self-reflection ability, and knowledge about linguistics.
- However, a child who doesn’t know linguistics is able to understand syntax.
- More research is needed.

QUESTIONS?