CS 490: NATURAL LANGUAGE PROCESSING

Dan Goldwasser, Abulhair Saparov

Lecture 12: Syntax II

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).

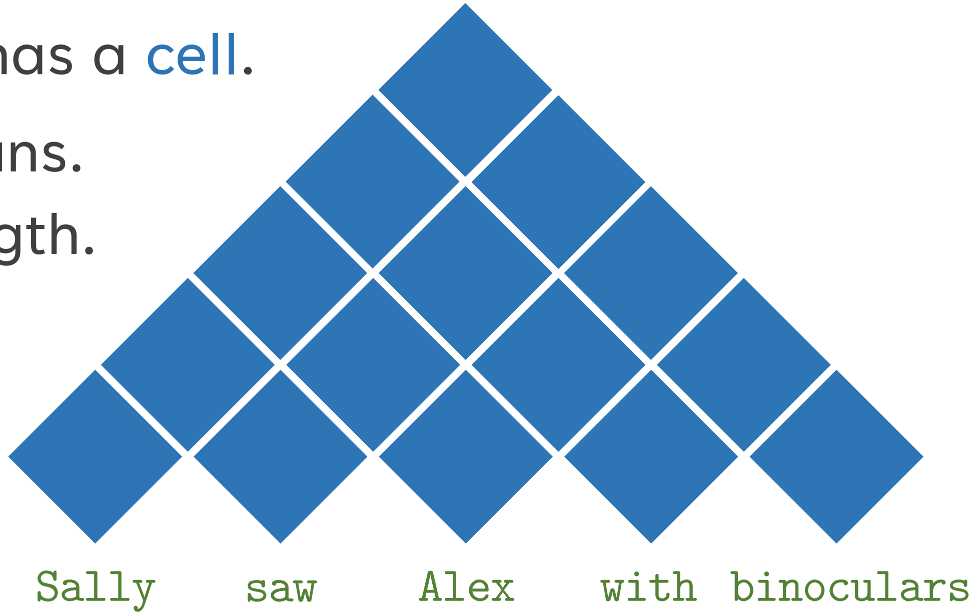
PARSING CFGS

- 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.

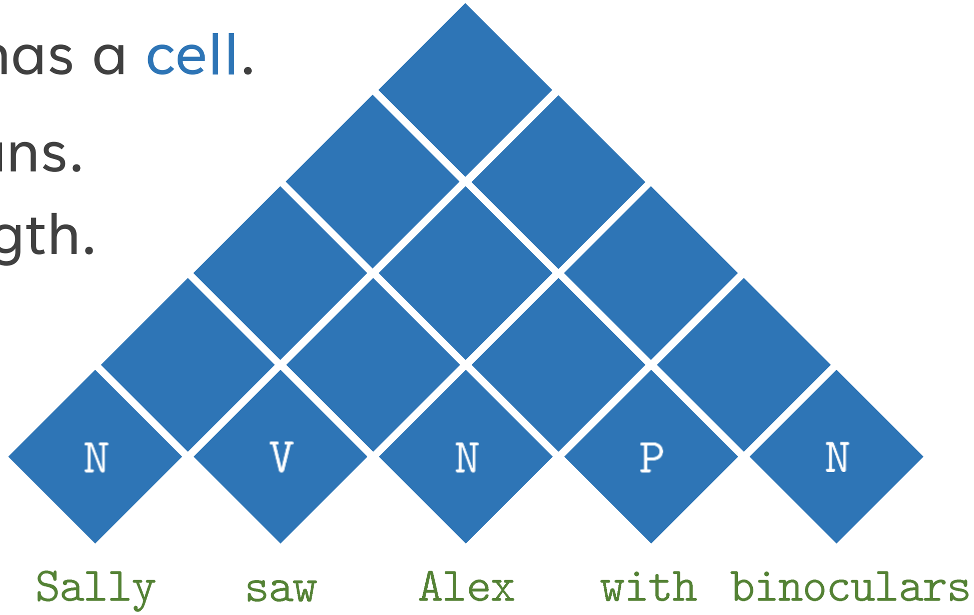
```
S  -> NP VP       V -> 'saw'
VP ->  V NP       P -> 'with'
VP -> VP PP       N -> 'binoculars'
PP ->  P NP       N -> 'Sally'
NP -> NP PP       N -> 'Alex'
NP ->  N
```



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.

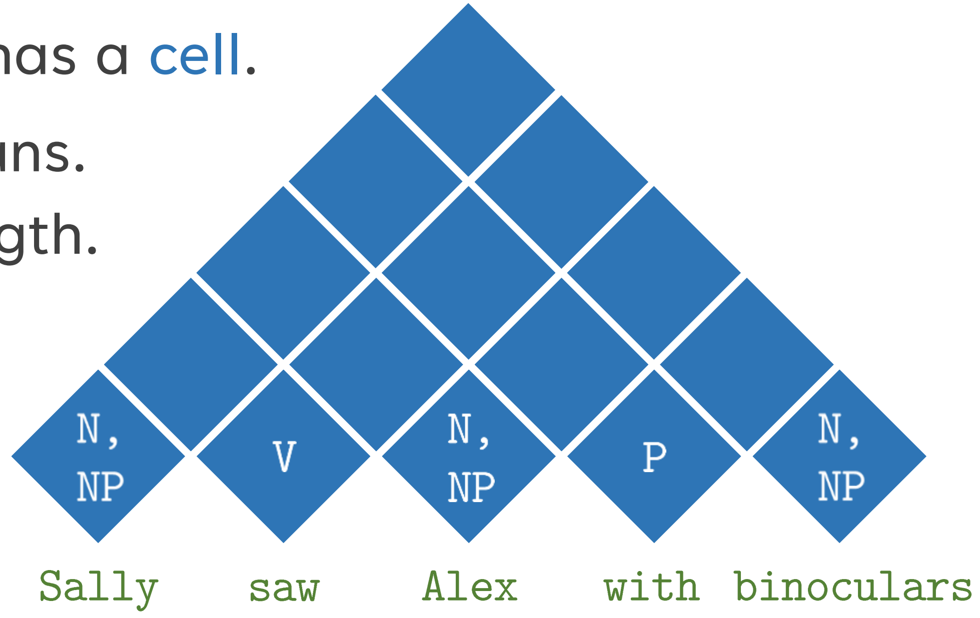
```
S  -> NP VP        V -> 'saw'
VP ->  V NP        P -> 'with'
VP -> VP PP        N -> 'binoculars'
PP ->  P NP        N -> 'Sally'
NP -> NP PP        N -> 'Alex'
NP ->  N
```



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.

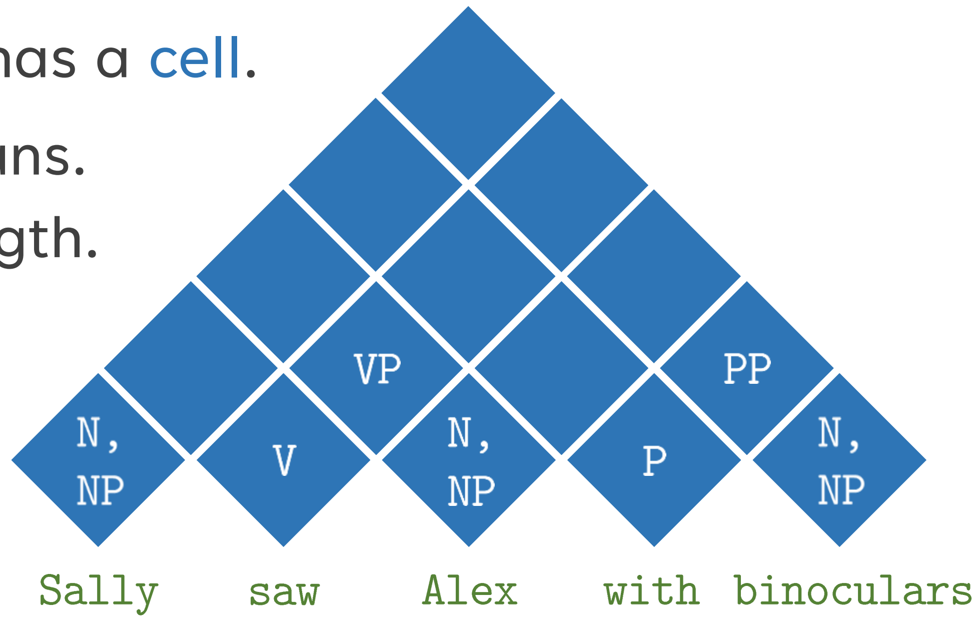
```
S  -> NP VP        V -> 'saw'
VP ->  V NP        P -> 'with'
VP -> VP PP        N -> 'binoculars'
PP ->  P NP        N -> 'Sally'
NP -> NP PP        N -> 'Alex'
NP ->  N
```



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
```

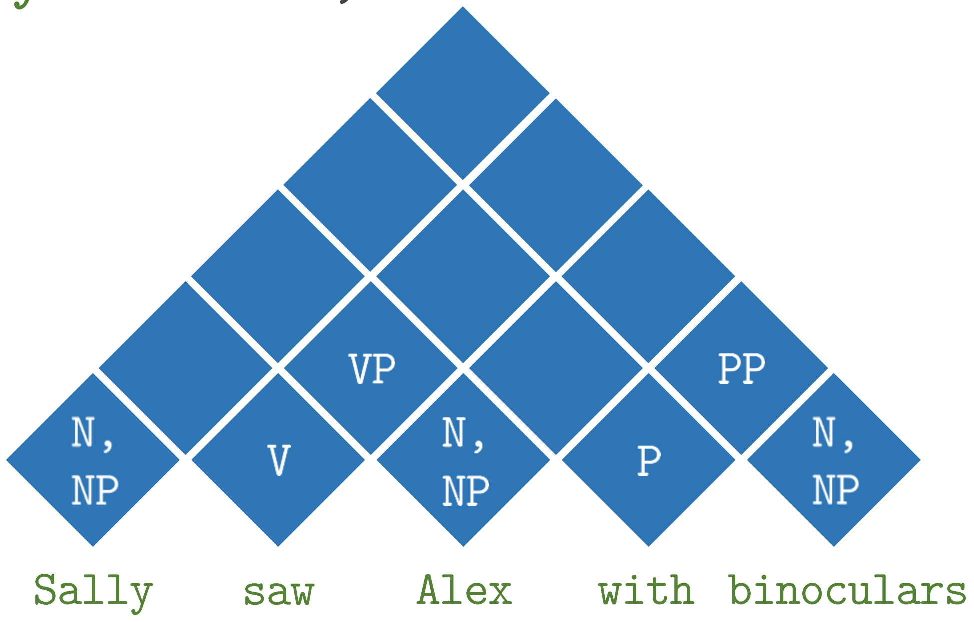


CKY PARSING

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

```
S  -> NP VP
VP ->  V NP
VP -> VP PP
PP ->  P NP
NP -> NP PP
NP ->  N
```

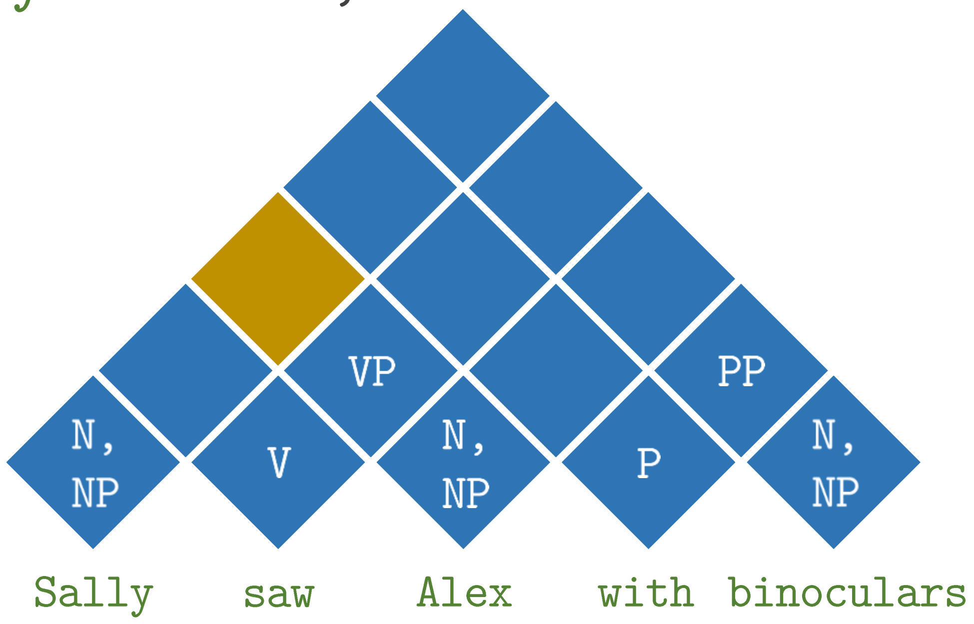
```
V -> 'saw'
P -> 'with'
N -> 'binoculars'
N -> 'Sally'
N -> 'Alex'
```



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:

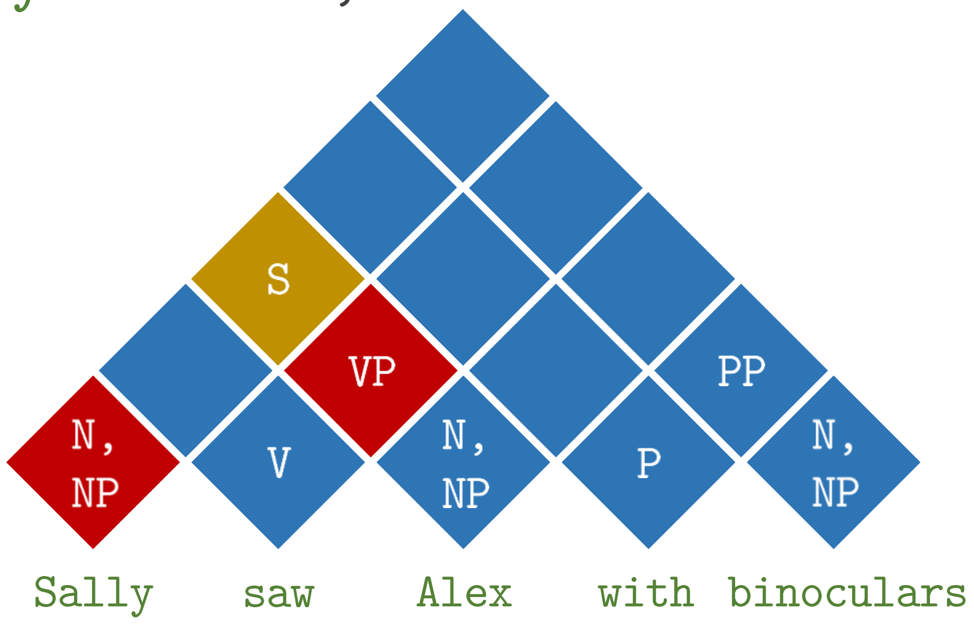
```
S  -> NP VP        V -> 'saw'
VP ->  V NP        P -> 'with'
VP -> VP PP        N -> 'binoculars'
PP ->  P NP        N -> 'Sally'
NP -> NP PP        N -> 'Alex'
NP ->  N
```



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'

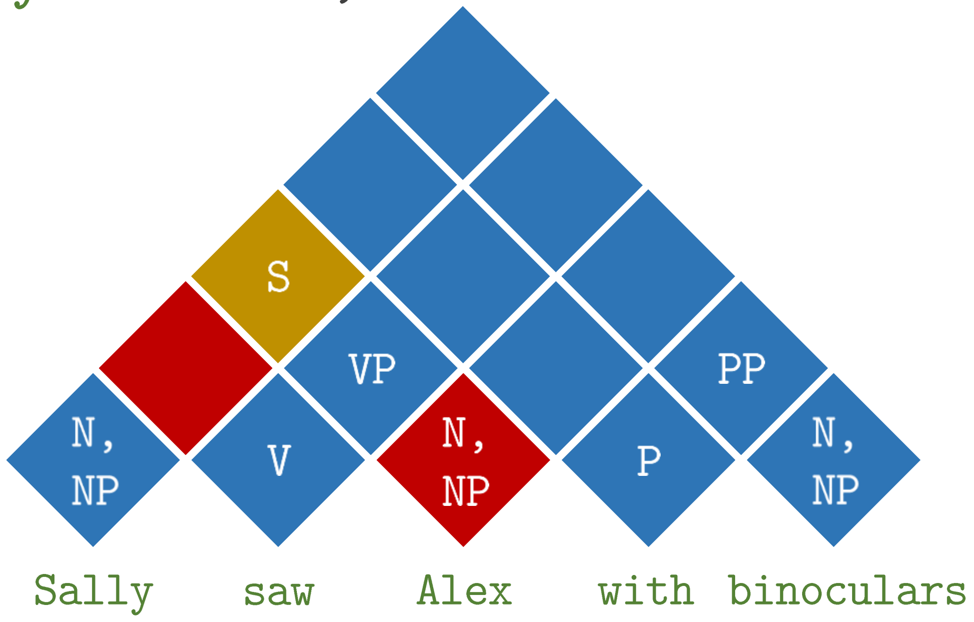
```
S  -> NP VP        V -> 'saw'
VP ->  V NP        P -> 'with'
VP -> VP PP        N -> 'binoculars'
PP ->  P NP        N -> 'Sally'
NP -> NP PP        N -> 'Alex'
NP ->  N
```



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'

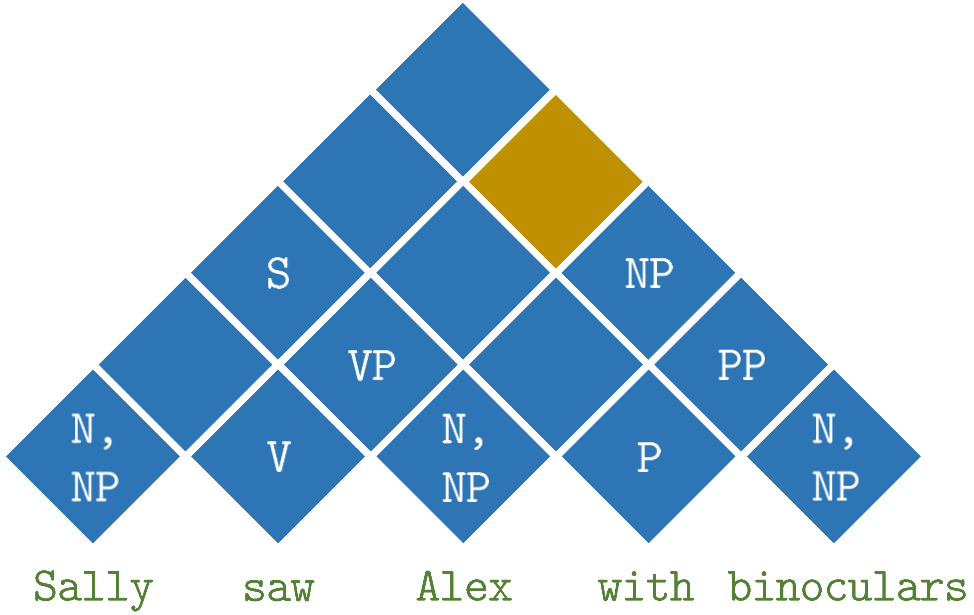
```
S  -> NP VP       V -> 'saw'
VP ->  V NP       P -> 'with'
VP -> VP PP       N -> 'binoculars'
PP ->  P NP       N -> 'Sally'
NP -> NP PP       N -> 'Alex'
NP ->  N
```



CKY PARSING

- Similarly for the span '`saw Alex with binoculars`' we must consider:

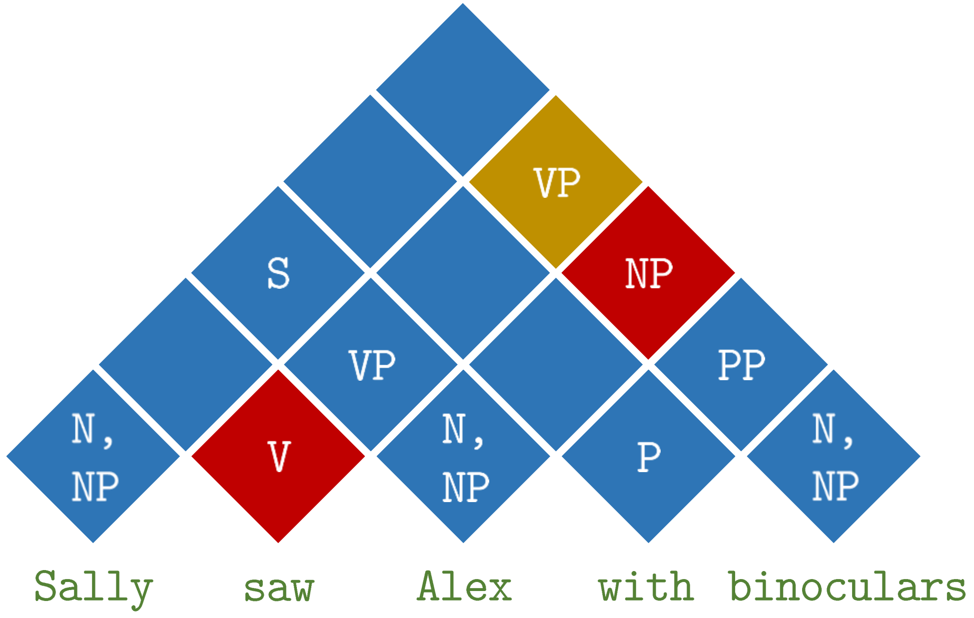
```
S  -> NP VP        V -> 'saw'
VP ->  V NP        P -> 'with'
VP -> VP PP        N -> 'binoculars'
PP ->  P NP        N -> 'Sally'
NP -> NP PP        N -> 'Alex'
NP ->  N
```



CKY PARSING

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

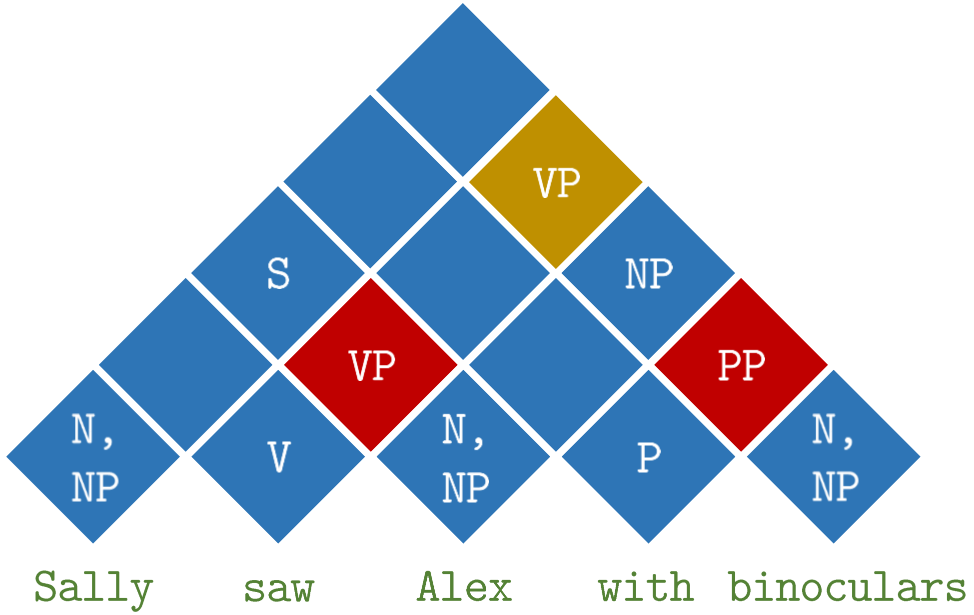
```
S  -> NP VP        V -> 'saw'
VP ->  V NP        P -> 'with'
VP -> VP PP        N -> 'binoculars'
PP ->  P NP        N -> 'Sally'
NP -> NP PP        N -> 'Alex'
NP ->  N
```



CKY PARSING

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

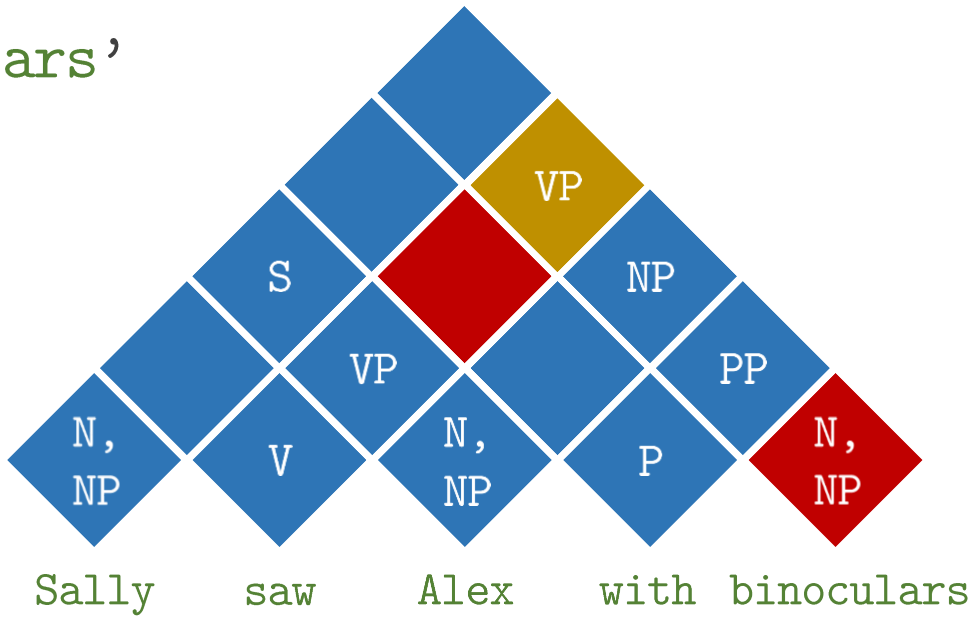
```
S  -> NP VP       V -> ‘saw’
VP ->  V NP       P -> ‘with’
VP -> VP PP       N -> ‘binoculars’
PP ->  P NP       N -> ‘Sally’
NP -> NP PP       N -> ‘Alex’
NP ->  N
```



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'

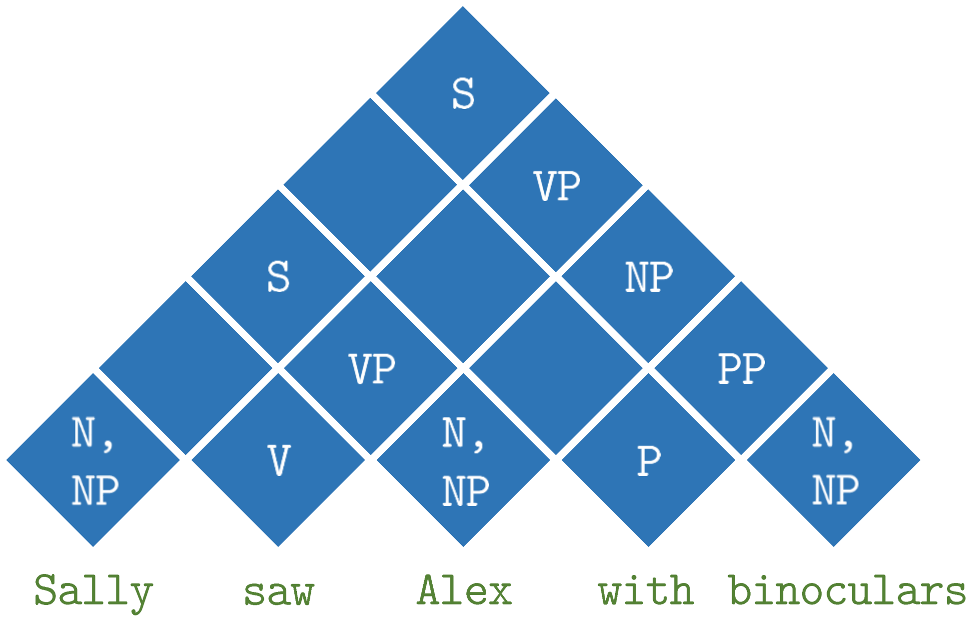
```
S  -> NP VP        V -> 'saw'
VP ->  V NP        P -> 'with'
VP -> VP PP        N -> 'binoculars'
PP ->  P NP        N -> 'Sally'
NP -> NP PP        N -> 'Alex'
NP ->  N
```



CKY PARSING

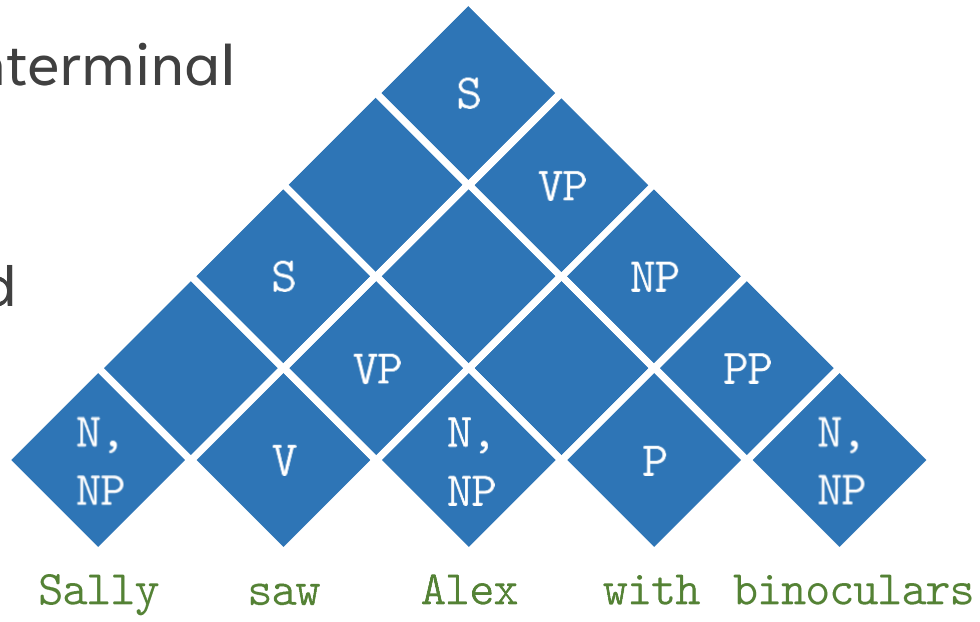
- We continue until we have processed all cells in the chart.
- In its simplest form, each cell simply records which nonterminals were parsed for that corresponding span.

```
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
```



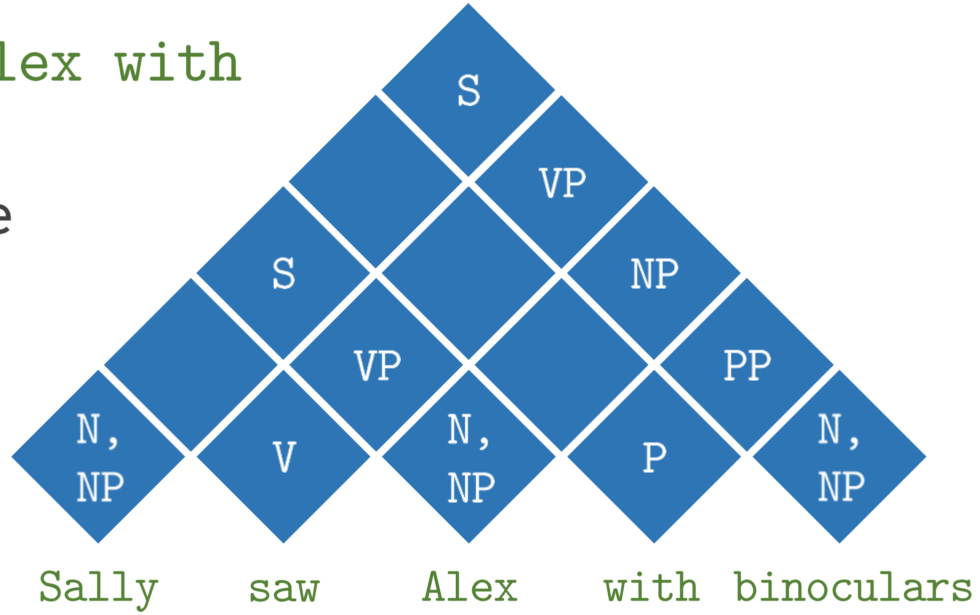
CKY PARSING

- 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.



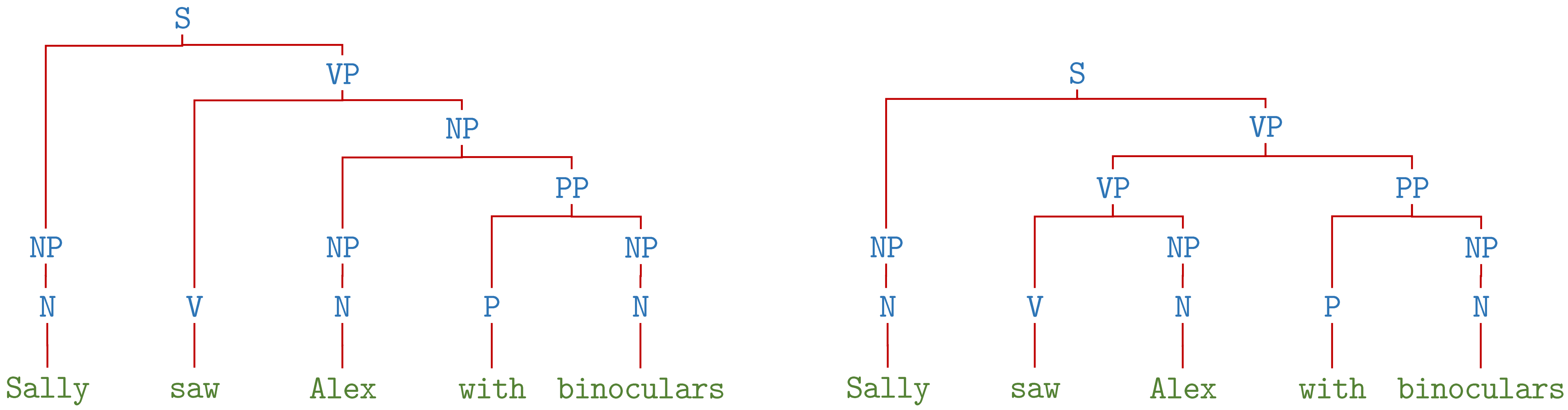
CKY PARSING

- 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.'



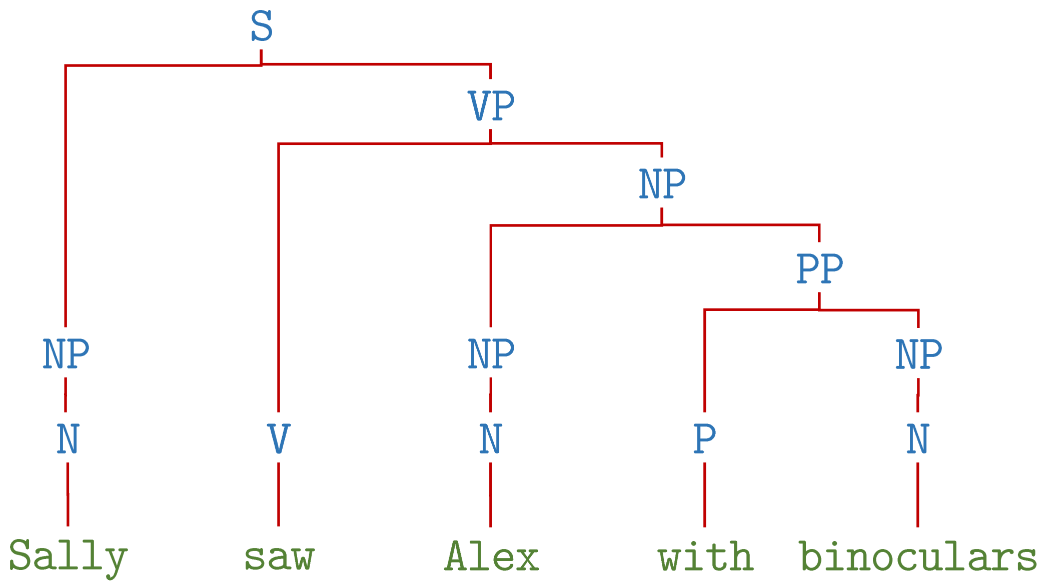
SYNTACTIC AMBIGUITY

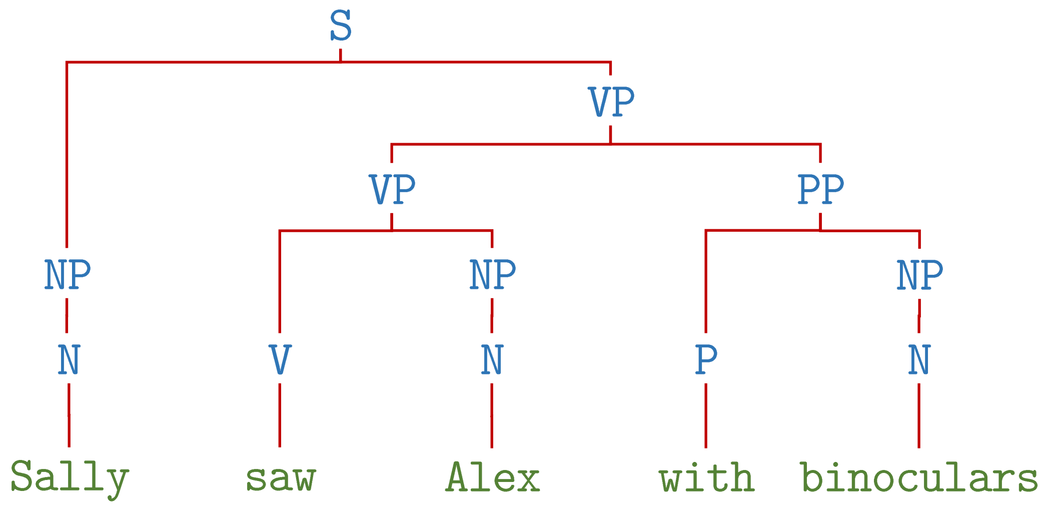
- Depending on which of the two ambiguous constructions we choose when reconstructing the syntax tree at that cell,
- We can obtain two different syntax trees:



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 ≤ k ≤ 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
```

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

queue

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

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

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
```

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

queue

```
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
```

N -> . ‘Alex’
i=0, k=0

queue

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
```

N -> . ‘Sally’
i=0, k=0

queue

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
```

```
N -> 'Sally' .
i=0, k=1
```

queue

```
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
```

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

queue

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

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

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

queue

```
V -> . 'saw'
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
```

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

queue

```
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
```

```
V -> . 'saw'
i=1, k=1
```

queue

```
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
```

```
V -> ‘saw’ .
i=1, k=2
```

queue

```
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
```

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

queue

```
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
```

N -> . ‘Sally’
i=2, k=2

queue

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
```

```
N -> 'Alex' .
i=2, k=3
```

queue

```
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
```

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

queue

```
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.

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
```

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
```

P -> . ‘with’
i=3, k=3

queue

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
```

P -> 'with' .
i=3, k=4

queue

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
```

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

queue

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

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

queue

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.

queue

```
N -> . ‘Sally’
i=4, k=4

N -> . ‘Alex’
i=4, k=4

N -> . ‘binoculars’
i=4, k=4
```

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

```
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 until we complete a rule of the form `S -> ...`
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

queue

`N -> 'binoculars' .`
`i=4, k=5`

`N -> . 'Sally'`
`i=4, k=4`

`N -> . 'Alex'`
`i=4, k=4`

`N -> . 'binoculars'`
`i=4, k=4`

```
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 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
```

N -> . 'Sally'
i=4, k=4

queue

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
```

`N -> 'binoculars' .`
`i=4, k=5`

queue

`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
```

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

queue

```
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.

queue

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

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

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

```
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 until we complete a rule of the form S -> ...
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

queue

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

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

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

```
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 until we complete a rule of the form `S -> ...`
  - Depending on the popped state, we perform either prediction, scanning, or completion steps.

queue

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

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

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

```
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 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.

GENERALIZING CKY AND EARLEY

- CKY and Earley parsers are both dynamic programming solutions to the problem of parsing CFGs.
- But is there a way to view these algorithms as instances of the same framework?
- Consider the main loops of each algorithm:
  - In CKY, we loop over all spans `(i,j)` in order of increasing `j - i`.
  - So each state is a span `(i,j)`,
  - We consider all subspans `(i,k),(k,j)` such that `k` is between `i` and `j`.
  - And check whether we can construct a parse tree from the subtrees in `(i,k)` and `(k,j)`.

GENERALIZING CKY AND EARLEY

- CKY and Earley parsers are both dynamic programming solutions to the problem of parsing CFGs.
- But is there a way to view these algorithms as instances of the same framework?
- Consider the main loops of each algorithm:
  - In CKY, we loop over all spans `(i,j,A)` in order of increasing `j - i`.
  - So each state contains a span `(i,j)` and nonterminal `A`,
  - We consider all subspans `(i,k),(k,j)` such that `k` is between `i` and `j`.
  - And check whether we can construct a parse tree rooted at `A` from the subtrees in `(i,k)` and `(k,j)`.

GENERALIZING CKY AND EARLEY

- CKY and Earley parsers are both dynamic programming solutions to the problem of parsing CFGs.
- But is there a way to view these algorithms as instances of the same framework?
- Consider the main loops of each algorithm:
  - In Earley, we loop over states of the form `A -> B`$_1$ `…` `B`$_m$ `.` `B`$_{m+1}$ `…` `B`$_n$ with start position `i` and current position `k`.
  - The next step depends on the symbol following the '`.`':
    - If `B`$_{m+1}$ is a nonterminal, we do a prediction step.
    - I.e., create a new state for all rules of the form `B`$_{m+1}$ `-> …`

GENERALIZING CKY AND EARLEY

- CKY and Earley parsers are both dynamic programming solutions to the problem of parsing CFGs.
- But is there a way to view these algorithms as instances of the same framework?
- Consider the main loops of each algorithm:
  - In Earley, we loop over states of the form `A -> ` B_1 … B_m `.` B_{m+1} … B_n with start position `i` and current position `k`.
  - The next step depends on the symbol following the '`.`':
    - If B_{m+1} is a terminal, we do a scanning step.
    - I.e., check if the input sentence matches the terminal B_{m+1} at position `k`.
    - If so, create a new state `A -> ` B_1 … B_{m+1} `.` … B_n with incremented `k`.

GENERALIZING CKY AND EARLEY

- CKY and Earley parsers are both dynamic programming solutions to the problem of parsing CFGs.
- But is there a way to view these algorithms as instances of the same framework?
- Consider the main loops of each algorithm:
  - In Earley, we loop over states of the form `A -> B`$_1$ … `B`$_m$ `.` with start position `i` and current position `k`.
  - The next step depends on the symbol following the '`.`':
    - If there is no symbol after '`.`', do a completion step.
    - For every state "waiting" for `A` at position `i`, create a new state where the dot moves forward and `k` is updated appropriately.

GENERALIZING CKY AND EARLEY

- CKY and Earley parsers are both dynamic programming solutions to the problem of parsing CFGs.
- But is there a way to view these algorithms as instances of the same framework?
- Consider the main loops of each algorithm.
  - In both CKY and Earley, we can imagine the states being added to a priority queue.
  - At each iteration, we pop one state from the queue and process it.
- The priority of a state is computed differently in CKY and Earley.
  - In CKY, states with shorter spans are prioritized.
  - How are states prioritized in Earley?

ORDER OF STATES IN EARLEY PARSING

- In Earley parsing, states are removed from the queue in the same order they were added.
  - We can replicate this behavior in a priority queue by setting the priority of the state to be the iteration number.
- But it’s not necessary to follow this prioritization.
  - We can process states in a different order and still have a correct parsing algorithm.

ORDER OF STATES IN EARLEY PARSING

- But there is a minor caveat:
  - We assumed that whenever we have a completion step `A -> B`$_1$ `… B`$_m$ `.` with start position `i`,
  - All prediction steps of the form `C -> … . A …` have already been processed earlier in a prediction step.
- If we change the order of visited states, this may no longer be true.
  - But we can resolve this issue by adding an extra step during prediction.
  - Whenever we have a prediction step `C -> … . A …`, we check if there are any completed parses for `A` at the same position.
  - If there are, then create a new state `C -> … A . …` with `k` updated accordingly.

ORDER OF STATES IN EARLEY PARSING

- There is also an optimization available here:
  - Whenever we do a prediction step, `C -> … . A …` at position `k`, we create a new state for each rule of the form `A -> …` with start position `k`.
  - But we don't need to do this more than once.
  - So we can avoid doing this multiple times by keeping track of whether we have "expanded" `A` at position `k` in the past.
  - If we have, then avoid "expanding" `A` at `k` again.

CKY VS EARLEY INITIALIZATION

- There is one more major difference between CKY and Earley parsing:
- What are the initial states in the priority queue before starting the main loop?
  - In CKY, we add an initial state for *every span* containing 1 token.
  - In Earley, we add an initial state for *every rule* of the form `S -> . …` where S is the root nonterminal.
- Is related to the bottom-up vs top-down approach of CKY and Earley parsing.

BRANCH AND BOUND

- So it seems like CKY and Earley parsing share a lot of structure.
  - Is there a unifying description?
- Branch-and-bound is a general class of algorithms for discrete optimization/search.
  - Say we want to find a target object x in a large set of objects S that maximizes some priority function $f(x)$.
    - For search, this can be a simple indicator function.
    - $f(x) = 1$ if and only if x is the object we're searching for.
  - First, we partition (i.e., "branch") the set S into subsets:

    $S_1, \ldots, S_n$

  such that their union covers the full set: $S_1 \cup \ldots \cup S_n = S$

BRANCH AND BOUND

- Next, for each subset S_i, create a new state and add it to the priority queue.
  - What should we set its priority to?
  - Ideally, it should be $\max_{x \in S_i} f(x)$.
  - But this quantity can be intractable to compute,
    - Especially if S_i is very large.
  - Instead, we can use an easy-to-compute an upper bound on this quantity:

    $$g(S_i) \geq \max_{x \in S_i} f(x)$$

  - Just set the priority of the new state to $g(S_i)$.
    (i.e., the "bound" step)

BRANCH AND BOUND

- Then we just repeat:
- For each iteration of the main loop,
  - Pop a state from the priority queue.
  - Partition the set into subsets (branch).
  - Push a new state for each subset with priority given by $g(\cdot)$ (bound).
- Eventually, we will pop a state with a set containing a single element $\{x\}$.
- We can compute $f(x)$ and check if it's larger than the priority of the next state in the queue.
- If so, then x is necessarily the optimal object in S.
  - Since the priority of the next state in the queue is an upper bound on $f(y)$ for all other objects y.

BRANCH AND BOUND

- Then we just repeat:
- For each iteration of the main loop,
  - Pop a state from the priority queue.
  - Partition the set into subsets (branch).
  - Push a new state for each subset with priority given by $g(\cdot)$ (bound).
- Eventually, we will pop a state with a state containing a single element $\{x\}$.
- The first such x may not be strictly more optimal than all other objects y.
  - There may be other objects y such that $f(x) = f(y)$.
- We can continue the branch-and-bound main loop to find the top-k objects.

CKY PARSING AS BRANCH AND BOUND?

- How can we formulate CKY as a branch-and-bound algorithm?
- S is the set of all syntax trees (both valid and invalid) for a given sentence.

  (an invalid syntax tree would be one containing a rule not in the grammar)
- Recall each state in CKY is a span `(i,j)`.
  - This state represents the set of all syntax trees for the given sentence that contains a valid subtree for the subsequence starting at `i` and ending at `j`.
- Eventually, we reach the span `(0,n)` which represents the set of all valid syntax trees for the full sentence.
- What is the "branch" step?
  - When processing the state `(i,j)`, we add a new state to the queue for each `(i,m)` for `m > j` and for each `(m,j)` for `m < i`.

CKY PARSING AS BRANCH AND BOUND?

- How can we formulate CKY as a branch-and-bound algorithm?
- S is the set of all syntax trees (both valid and invalid) for a given sentence.
  (an invalid syntax tree would be one containing a rule not in the grammar)
- Recall each state in CKY is a span `(i,j)`.
  - This state represents the set of all syntax trees for the given sentence that contains a valid subtree for the subsequence starting at `i` and ending at `j`.
- What is the "bound" $g(x)$?
  - We can set it to $g(x)$ `= i - j` so that shorter spans have higher priority.
  - But we must make sure $g(x)$ is an upper bound of the objective function.
    (the objective function is 1 iff x is a valid parse of the whole sentence)
  - So we can simply set $g(x)$ `= i - j + n + 1`.

EARLEY PARSING AS BRANCH AND BOUND?

- How can we formulate Earley parsing as a branch-and-bound algorithm?
- Recall each state in Earley contains a rule `A -> ` B_1 … B_m . B_{m+1} … B_n with start position `i` and current position `k`.
  - This state represents the set of all syntax trees for the given sentence that contains a subtree for the subsequence starting at `i`,
  - Where the subtree has root `A`,
  - And this subtree has valid subtrees with roots B_1 … B_m up to position `k`,
  - And subtrees (valid or invalid) subtrees with roots B_{m+1} … B_n after position `k`.

EARLEY PARSING AS BRANCH AND BOUND?

- How can we formulate Earley parsing as a branch-and-bound algorithm?
- What is the "branch" step?
  - Depending on the current state, we either do prediction, scanning, or completion.
- What is the "bound"?
  - As stated earlier, we can be flexible about the order we visit states.
  - We can use a heuristic and frame the problem as an A* search.
    - E.g., Lee et al. (2016) train a neural network to predict the bound.
    - Resulting in a faster parser (with fewer iterations).
  - In general, tighter bounds leads to faster searching.
    - I.e, $g(S)$ is closer to $\max_{x \in S} f(x)$

STRUCTURED PREDICTION

- Structured prediction is the task where the output is structured.
  - E.g., syntax trees, sequences, graphs, tables, etc.
- This task usually involves discrete optimization/search,
  - For example via algorithms like branch-and-bound.
- Sequence prediction is a kind of structured prediction.
  - Let's say we have some objective function f over sequences of items.
    - E.g., this could be a sequence of words.
    - E.g., Traveling Salesman Problem: suppose we need to make n deliveries in n different cities. In what order should we visit the cities to minimize the overall distance traveled?
- Goal: Find the sequence $x_1, \ldots, x_n$ such that $f(x_1, \ldots, x_n)$ is maximized.
  - In traveling salesman, f is the negative distance.

SEQUENCE PREDICTION

- We can apply branch-and-bound to autoregressive sequence prediction.
- The set S is the set of all sequences of length n.
- Each state is a partial sequence: $x_1, \ldots, x_k$
  - Represents the set of all sequences of length n that start with $x_1, \ldots, x_k$.
  - E.g., the first k cities that we visit to make deliveries.
- What is the "branch" step?
  - For each possible next symbol x_{k+1}, we create a new state $x_1, \ldots, x_{k+1}$.
- What is the "bound"?
  - The simplest bound is the total cost so far:

$$-\text{distance}(x_1, \ldots, x_n) \leq -\text{distance}(x_1, \ldots, x_k),$$

$$= -\text{distance}(x_1, x_2) - \text{distance}(x_2, x_3) - \ldots - \text{distance}(x_{k-1}, x_k).$$

SEQUENCE PREDICTION

- This algorithm is too slow.
- In the worst case, we need a number of branches exponential in n.
  - We would need to search over all possible sequences of cities.
  - How many such sequences are there?
  - $n(n-1)(n-2)\ldots(2)(1) = n!$
  - (there are n possible values for the first city, then there are $n-1$ possible values for the second city, etc...)

SEQUENCE PREDICTION

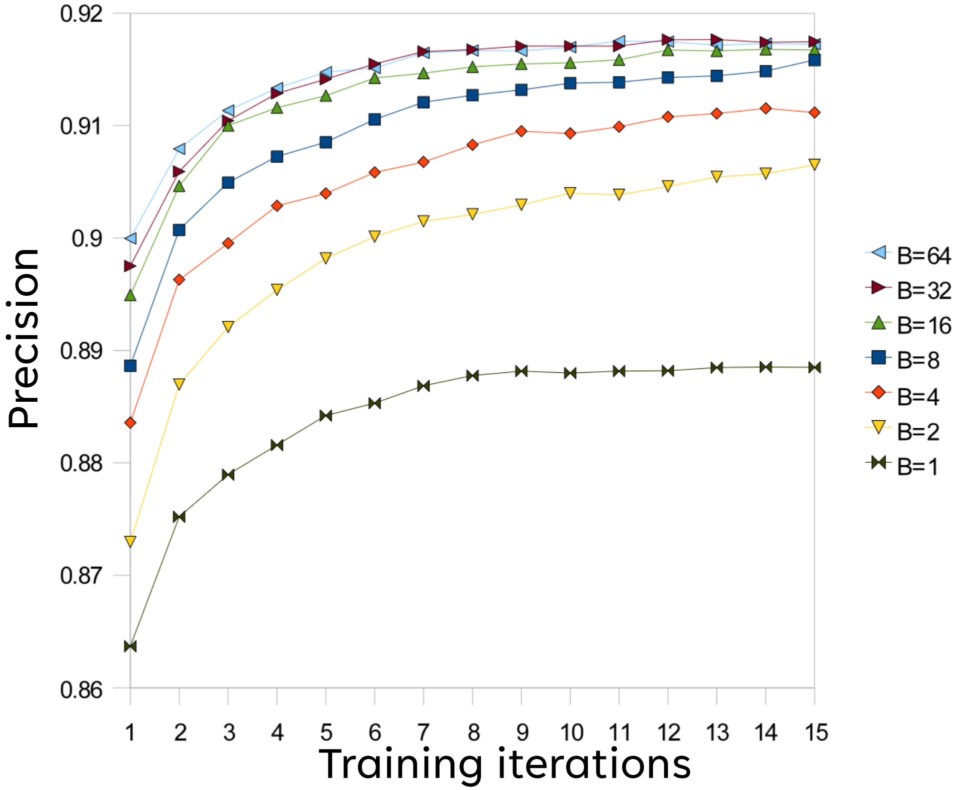
- How to make sequence prediction faster?
- We can trade optimality for performance.
  - We can limit the capacity of the priority queue.
    - Let B be the capacity.
  - After adding new states to the priority queue, simply remove the lowest priority states until only B elements remain.
- This is called beam search,
  - And B is the beam width or beam size.
- If $B = 1$, we have greedy search.
- If $B = \infty$, we recover exact search.

BEAM SEARCH IN PARSING

- Beam search can also be used in parsing.
- Used when there is an objective function over syntax trees.
  - E.g., a model that assigns probabilities to syntax trees.
  - This would be very useful in choosing among ambiguous parses.
  - E.g., in ‘`Sally caught a butterfly with a net,`’ who has the net?
- Can significantly increase parsing speed if there are many ambiguous parses.

BEAM SEARCH IN PARSING

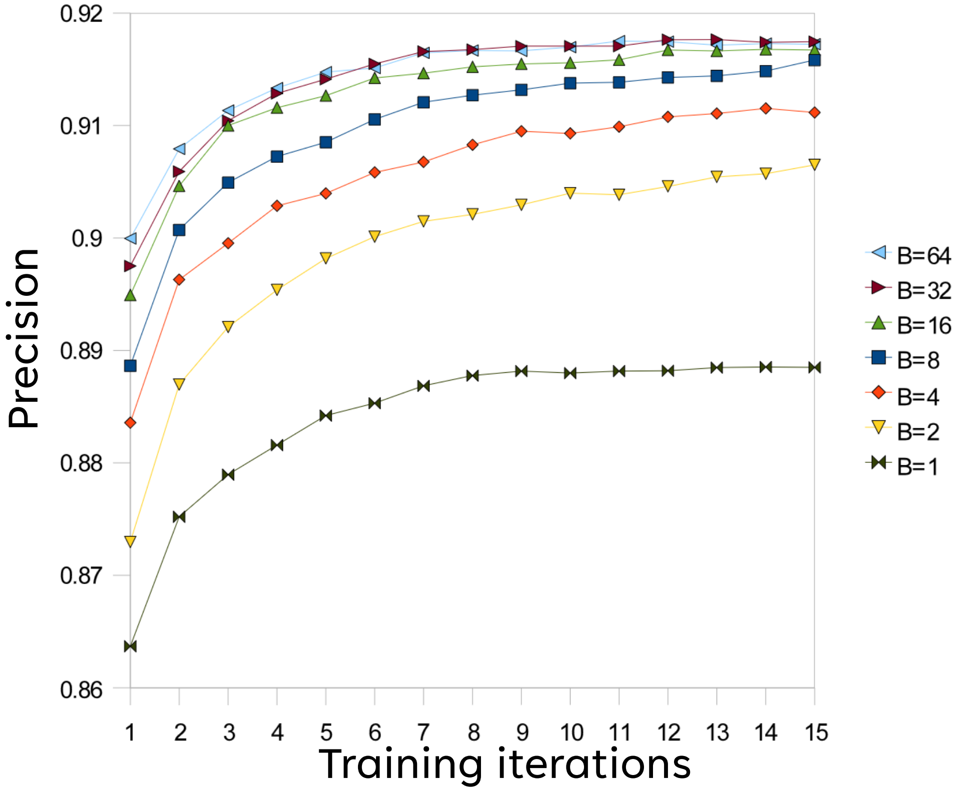
- Zhang and Clark (2008) used beam search for parsing.
- They used a neural model to assign probabilities to parser outputs.



[Zhang and Clark, 2008]

BEAM SEARCH IN PARSING

- They measured precision vs number of training iterations for the neural model vs beam size.



[Zhang and Clark, 2008]

QUESTIONS?