CS 577: NATURAL LANGUAGE PROCESSING

Abulhair Saparov

Lecture 22: Syntax III

SYNTAX (CONTINUED)

- Previously, we covered context-free grammars
- We described two efficient dynamic programming algorithms for parsing
  - CKY parsing
  - Earley parsing
- Is there a unified way to understand CKY and Earley parsing?
  - Or more generally, structured prediction?
- Can we utilize grammars to force LLM outputs to follow syntax?
- Beyond context-free grammars/languages
  - Is natural language context-free?

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 '`.`':
    - 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 $A \rightarrow B_1 \ldots B_m \; . \; B_{m+1} \ldots B_n$ with start position i and current position k.
  - The next step depends on the symbol following '.':
    - 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 $A \rightarrow B_1 \ldots B_{m+1} \; . \; \ldots B_n$ 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 '`.`':
    - 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 each span containing 1 token.
  - In Earley, we add an initial state for each 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:

    $S_1, \dots, S_n$

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

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.
  - E.g., given a language model and some prompt P, predict the most likely sequence of the next n tokens: $x_1, \ldots, x_n$.
  - The objective function is the joint probability:

$$p(x_1, \ldots, x_n | P) = p(x_1 | P) \; p(x_2 | P, x_1) \; \ldots \; p(x_n | P, x_1, \ldots, x_{n-1}),$$

$$\log p(x_1, \ldots, x_n | P) = \log p(x_1 | P) + \log p(x_2 | P, x_1) + \ldots + \log p(x_n | P, x_1, \ldots, x_{n-1}).$$

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 $x_1, \ldots, x_k$.
- 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 log probability so far:

$$\log p(x_1,\ldots,x_n|P) \le \log p(x_1,\ldots,x_k|P),$$
$$= \log p(x_1|P) + \log p(x_2|P,x_1) + \ldots + \log p(x_k|P,x_1,\ldots,x_{k-1}).$$

SEQUENCE PREDICTION

- This algorithm is too slow.
- Each branch step requires a forward pass of the LM.
- In the worst case, we need a number of branches exponential in n.
  - Suppose the optimal output has log probability -10.
  - The average log probability of each token for GPT-2 is about -2.4.
  - Therefore, sequences of length < $10/2.4$ (about 4 tokens) will, on average, have log probability > -10.
  - There are V^4 possible sequences of length 4, where V is the vocab size.

SEQUENCE PREDICTION

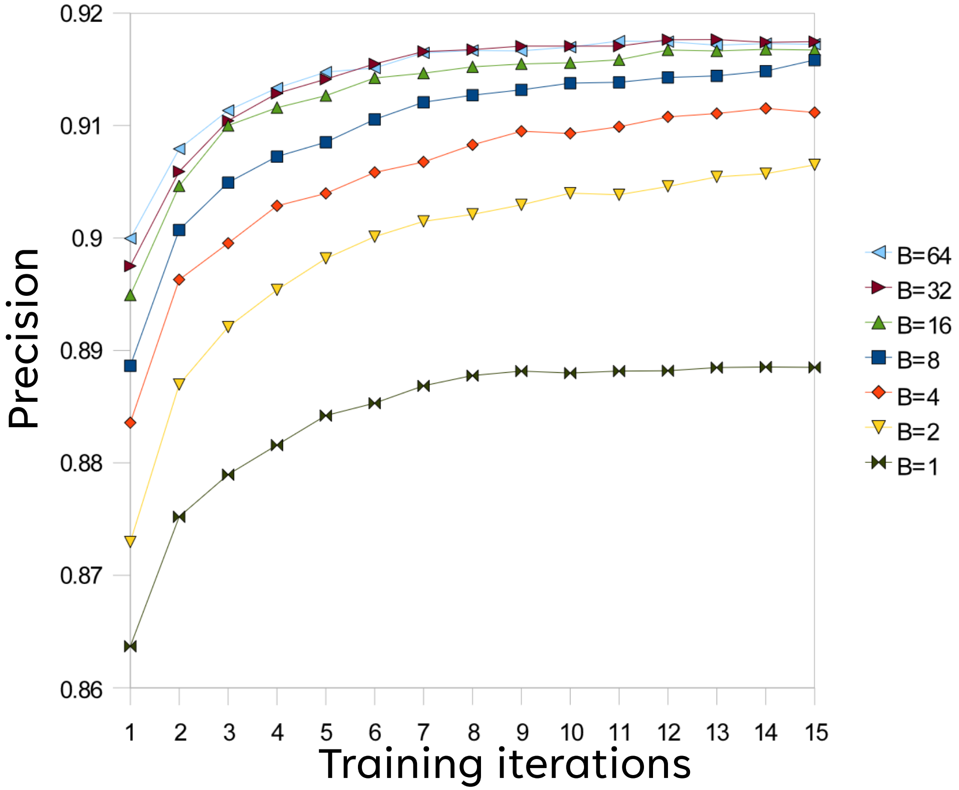
- How to make autoregressive sequence prediction faster?
- We can trade optimality for performance.
  - We can limit the capacity of the priority queue.
    - Let B be the capacity.
  - After adding new states to the priority queue, simply remove the lowest priority states until only B elements remain.
- 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

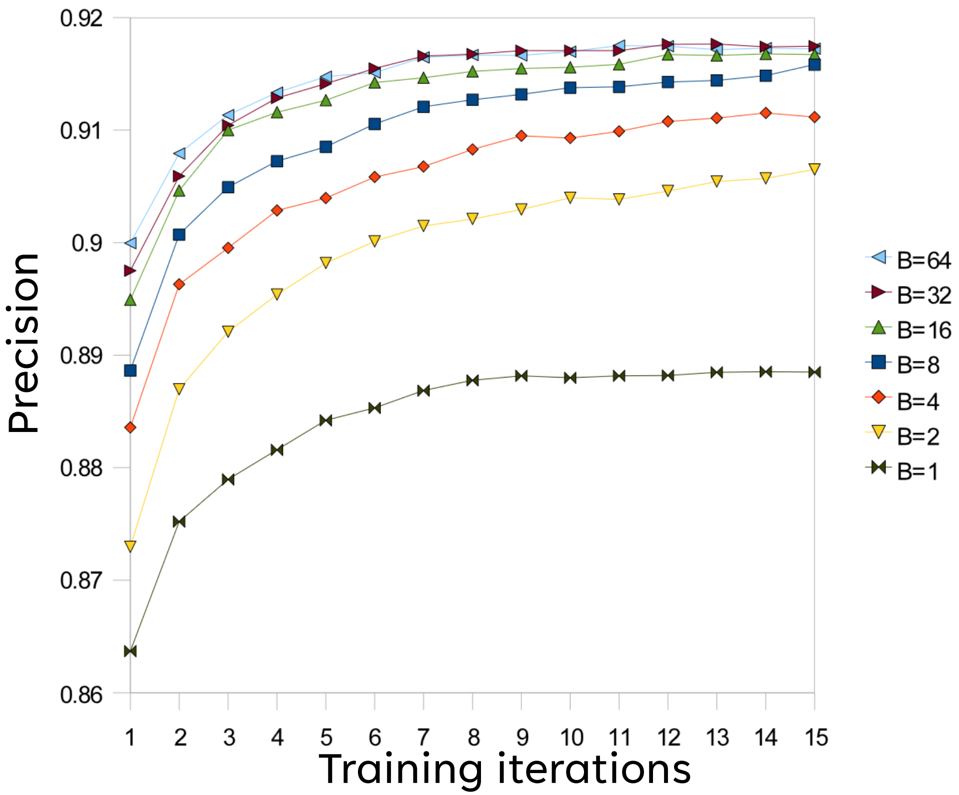
- Zhang and Clark (2008) used beam search for parsing.
- They used a neural model to assign probabilities to parser outputs.



[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]

CONSTRAINED DECODING

- For many NLP tasks, we require the output to follow a specific pattern.
  - E.g., code generation,
  - Tasks requiring output in JSON/Markdown/LaTeX format,
  - Generating figures in Matplotlib or Tikz.
- Autoregressive language models are probabilistic and may make errors with small probability.
  - When generating many tokens, the probability of error increases.
- If we have a grammar (e.g., the JSON or Python grammar), we can constrain the output of the model to follow the grammar.
  - This is called constrained decoding or structured generation.

CONSTRAINED DECODING

- Earley parsing can be adapted for constrained decoding.
- The main difference lies in the scanning step.
  - I.e., when the state looks like `A -> … . 't' …` for some terminal `t` and the current position is `k`.
  - During constrained decoding, we don't have the full input.
  - So the terminal at position `k` may not yet be available.
  - In this case, we add this state into a list of "waiting" scanning states,
  - And proceed with Earley parsing until the queue is empty.

```
S  -> N Op S      Op -> '+'
S  -> N Op N      Op -> '-'
N -> '0'          Op -> '*'
...               Op -> '/'
N -> '9'
```

`Input so far: '9 + 2'`

Waiting scan states:

`Op -> . '+'` i=3, k=3	`Op -> . '-'` i=3, k=3	`Op -> . '*'` i=3, k=3	`Op -> . '/'` i=3, k=3

CONSTRAINED DECODING

- Earley parsing can be adapted for constrained decoding.
- Next, we can inspect the list of “waiting” scanning states.
- These states tell us exactly which terminals are allowed for the next position.
- Then, after performing a forward pass with the LM, we can mask the tokens that do not match with any of the allowed terminals.
  - I.e., set their probabilities to zero.
- We then select the terminal symbol to add to the sequence.

```
S  -> N Op S      Op -> ‘+’
S  -> N Op N      Op -> ‘-’
N -> ‘0’          Op -> ‘*’
...               Op -> ‘/’
N -> ‘9’
```

Input so far: ‘9 + 2’

Waiting scan states:

Op -> . ‘+’	Op -> . ‘-’	Op -> . ‘*’	Op -> . ‘/’
i=3, k=3	i=3, k=3	i=3, k=3	i=3, k=3

CONSTRAINED DECODING

- Earley parsing can be adapted for constrained decoding.
- Once we have decoded the next terminal, we then resume Earley parsing by adding the waiting scanning states to the queue.
- Repeat until we have finished decoding.
  - E.g., the parser completes a syntax tree with the root nonterminal.

```
S  -> N Op S     Op -> '+'
S  -> N Op N     Op -> '-'
N -> '0'         Op -> '*'
...              Op -> '/'
N -> '9'
```

```
Input so far: '9 + 2 /'

Waiting scan    N -> . '0'     N -> . '1'          N -> . '9'
      states:   i=4, k=4       i=4, k=4     ...    i=4, k=4
```

CONSTRAINED DECODING

- Earley parsing can be adapted for constrained decoding.
- Once we have decoded the next terminal, we then resume Earley parsing by adding the waiting scanning states to the queue.
- Repeat until we have finished decoding.
  - E.g., the parser completes a syntax tree with the root nonterminal.

```
S  -> N Op S     Op -> '+'
S  -> N Op N     Op -> '-'
N -> '0'         Op -> '*'
...              Op -> '/'
N -> '9'
```

Input so far: '9 + 2 / 7'

Waiting scan states:

Op -> . '+' i=5, k=5	Op -> . '-' i=5, k=5	Op -> . '*' i=5, k=5	Op -> . '/' i=5, k=5

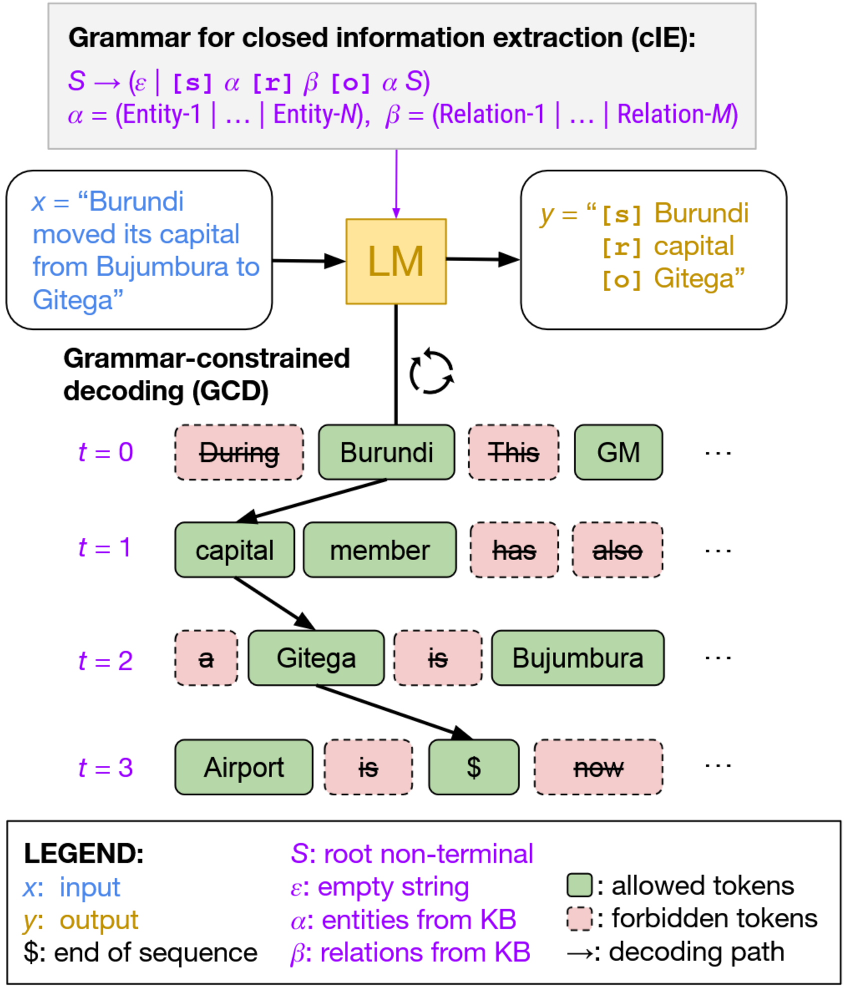
CONSTRAINED DECODING CHALLENGES

- One implementation challenge:
  - Terminals and tokens are not the same.
  - For example, the LM may have a token like ‘`+7`’ or ‘`-2`’, and the grammar has terminals ‘`+`’, ‘`-`’, ‘`2`’, and ‘`7`’.
  - Or the grammar may have a terminal like ‘`0 + 1`’ or ‘`public static`’. And the LM has tokens ‘`0`’ and ‘`pub`’.
- We need to take care to allow partial matching during scanning steps.
- Constrained decoding can only be applied when we have access to the logits.
  - Many API-based LM services do not provide this information.

CONSTRAINED DECODING

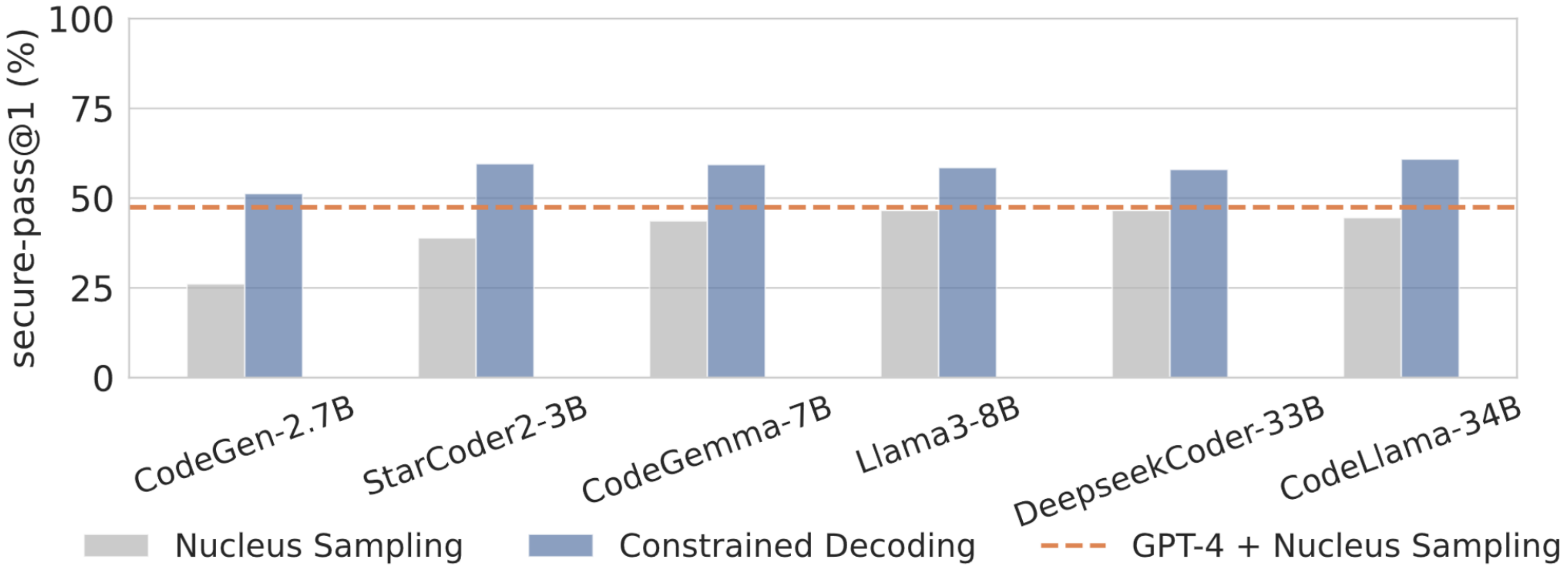
- Geng et al. (2023) applied constrained decoding to perform information extraction.

Method	Precision	Recall	F1
Weakly supervised			
GenIE T5-base	**49.6** ± 0.3	26.8 ± 0.2	34.8 ± 0.2
Few-shot unconstrained			
LLaMA-7B	10.2± 0.5	14.3± 0.7	11.9± 0.5
LLaMA-13B	10.3± 0.6	17.0± 0.9	12.9± 0.6
LLaMA-33B	14.1± 1.0	23.1± 1.4	17.5± 1.0
Vicuna-7B	12.5± 0.2	16.7± 0.1	14.3± 0.2
Vicuna-13B	13.4± 0.2	15.2± 0.2	14.4± 0.2
Few-shot constrained			
LLaMA-7B	27.9± 0.6	20.2± 0.5	23.5± 0.5
LLaMA-13B	36.2± 0.7	26.5± 0.5	30.6± 0.5
LLaMA-33B	39.3± 0.9	**33.2**± 0.8	**36.0** ± 0.7
Vicuna-7B	25.4± 0.5	15.8± 0.3	19.5± 0.3
Vicuna-13B	38.7± 1.0	19.8± 0.8	26.1± 0.8



CONSTRAINED DECODING

- Fu et al. (2024) applied constrained decoding and beam search for *secure code generation*.
- A 2.7B parameter model with constrained decoding outperformed GPT-4.



SEMANTICS-AWARE CONSTRAINED DECODING

- Semantic information can be incorporated into constrained decoding.
- Consider the following Java code fragment:

```
String foo = "foo";
int i = 7;
int j = i +
```

- The syntactic constraints (grammar) filters some possible outputs, such as '`{`', '`}`', '`;`', etc.
- Semantically, we can filter '`foo`' since Java is a strongly-typed language.
  - The type of the next symbol must be compatible with the '`+`' operation with an `int`, with an `int` output type.

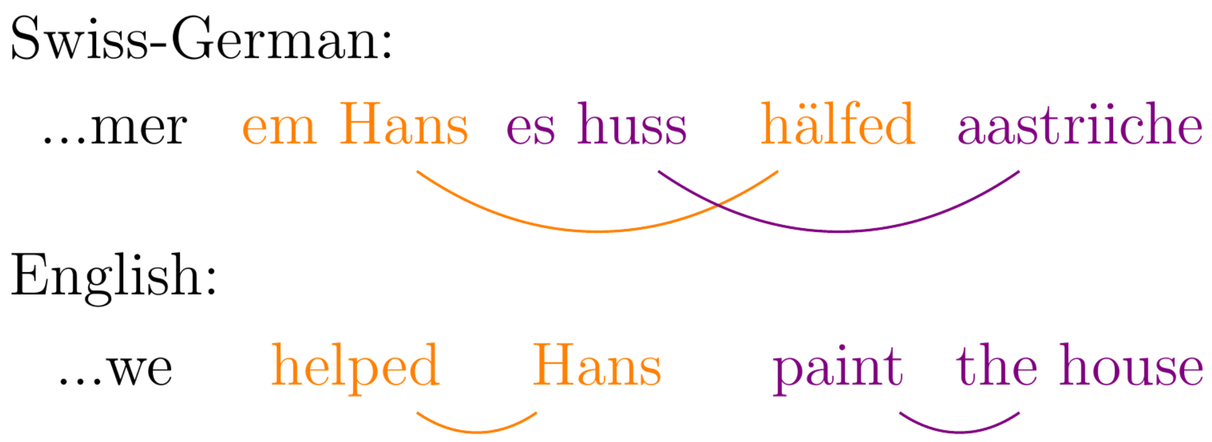
NON-CONTEXT-FREE GRAMMARS

- There are grammars and languages that are not context-free.
- One simple example:
  - The “copy” language: The set of all strings that contain two identical substrings side-by-side.

  `L = {‘aa’, ‘abcabc’, ‘this is a copy this is a copy ’, …}`
- Are natural languages context free?

IS NATURAL LANGUAGE CONTEXT-FREE?

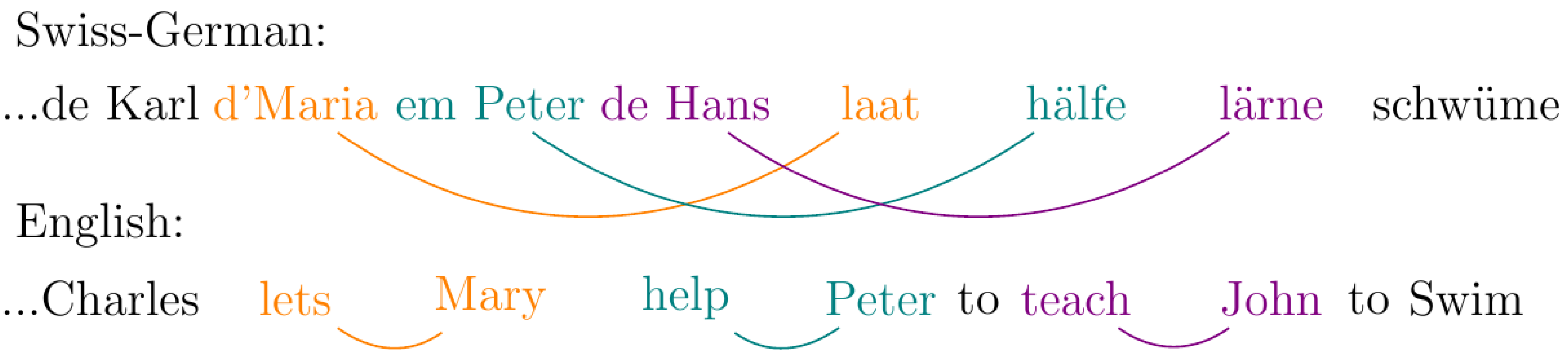
- There are some constructions in some natural languages that seem to require non-context-free modeling.
- E.g., the following examples in Swiss German:



[wikipedia.org/wiki/Cross-serial_dependencies]

IS NATURAL LANGUAGE CONTEXT-FREE?

- There are some constructions in some natural languages that seem to require non-context-free modeling.
- E.g., the following examples in Swiss German:
  - Some argue the cross-dependencies are semantic and not syntactic.
- These kinds of structures seem to be very rare.



[wikipedia.org/wiki/Cross-serial_dependencies]

COMBINATORY CATEGORIAL GRAMMAR

- Combinatory categorial grammar (CCG; Steedman 1987) is a grammar formalism that can be used to describe non-context-free grammars.
- Each item in the vocabulary is assigned a syntactic type or category.
  - A simple vocabulary containing 4 items:

$$\text{the} : NP/N \qquad \text{dog} : N \qquad \text{John} : NP \qquad \text{bit} : (S\backslash NP)/NP$$

- Syntactic types can be composed to create more complex syntactic types.
  - E.g., `NP/N` indicates that the word '`the`' can be combined with an `N` on the right side, and the resulting combination will have type `NP`.
  - So '`the dog`' will have syntactic type `NP`.

[wikipedia.org/wiki/Combinatory_categorial_grammar]

COMBINATORY CATEGORIAL GRAMMAR

- Combinatory categorial grammar (CCG; Steedman 1987) is a grammar formalism that can be used to describe non-context-free grammars.
- Each item in the vocabulary is assigned a syntactic type or category.
  - A simple vocabulary containing 4 items:

$$\text{the} : NP/N \qquad \text{dog} : N \qquad \text{John} : NP \qquad \text{bit} : (S\backslash NP)/NP$$

- Syntactic types can be composed to create more complex syntactic types.
  - E.g., `NP\N` indicates that the word can be combined with an `N` on the left side, and the resulting combination will have type `NP`.

[wikipedia.org/wiki/Combinatory_categorial_grammar]

COMBINATORY CATEGORIAL GRAMMAR

- Combinatory categorial grammar (CCG; Steedman 1987) is a grammar formalism that can be used to describe non-context-free grammars.
- Each item in the vocabulary is assigned a syntactic type or category.
  - A simple vocabulary containing 4 items:

$$\text{the} : NP/N \qquad \text{dog} : N \qquad \text{John} : NP \qquad \text{bit} : (S\backslash NP)/NP$$

$$\frac{\text{the}}{NP/N} \qquad \frac{\text{dog}}{N} \qquad \frac{\text{bit}}{(S\backslash NP)/NP} \qquad \frac{\text{John}}{NP}$$

[wikipedia.org/wiki/Combinatory_categorial_grammar]

COMBINATORY CATEGORIAL GRAMMAR

- Combinatory categorial grammar (CCG; Steedman 1987) is a grammar formalism that can be used to describe non-context-free grammars.
- Each item in the vocabulary is assigned a syntactic type or category.
  - A simple vocabulary containing 4 items:

$$\text{the} : NP/N \qquad \text{dog} : N \qquad \text{John} : NP \qquad \text{bit} : (S\backslash NP)/NP$$

$$\dfrac{\dfrac{\text{the}}{NP/N} \quad \dfrac{\text{dog}}{N}}{NP}> \qquad \dfrac{\text{bit}}{(S\backslash NP)/NP} \qquad \dfrac{\text{John}}{NP}$$

[wikipedia.org/wiki/Combinatory_categorial_grammar]

COMBINATORY CATEGORIAL GRAMMAR

- Combinatory categorial grammar (CCG; Steedman 1987) is a grammar formalism that can be used to describe non-context-free grammars.
- Each item in the vocabulary is assigned a syntactic type or category.
  - A simple vocabulary containing 4 items:

$$\text{the} : NP/N \qquad \text{dog} : N \qquad \text{John} : NP \qquad \text{bit} : (S\backslash NP)/NP$$

$$\dfrac{\dfrac{\text{the}}{NP/N} \quad \dfrac{\text{dog}}{N}}{NP}> \qquad \dfrac{\dfrac{\text{bit}}{(S\backslash NP)/NP} \quad \dfrac{\text{John}}{NP}}{S\backslash NP}>$$

[wikipedia.org/wiki/Combinatory_categorial_grammar]

COMBINATORY CATEGORIAL GRAMMAR

- Combinatory categorial grammar (CCG; Steedman 1987) is a grammar formalism that can be used to describe non-context-free grammars.
- Each item in the vocabulary is assigned a syntactic type or category.
  - A simple vocabulary containing 4 items:

$$\text{the} : NP/N \qquad \text{dog} : N \qquad \text{John} : NP \qquad \text{bit} : (S\backslash NP)/NP$$

$$\dfrac{\dfrac{\dfrac{\text{the}}{NP/N} \quad \dfrac{\text{dog}}{N}}{NP}> \qquad \dfrac{\dfrac{\text{bit}}{(S\backslash NP)/NP} \quad \dfrac{\text{John}}{NP}}{S\backslash NP}>}{S}<$$

COMBINATORY CATEGORIAL GRAMMAR

- Thus, ‘`the dog bit John`’ is a string in the language, since we have successfully proved/derived the root S.
- Unlike CFGs, there are a small handful of rules (>, <, type-raising, etc),
  - And the allowed combinations are determined by the lexicon.

$$\text{the} : NP/N \qquad \text{dog} : N \qquad \text{John} : NP \qquad \text{bit} : (S\backslash NP)/NP$$

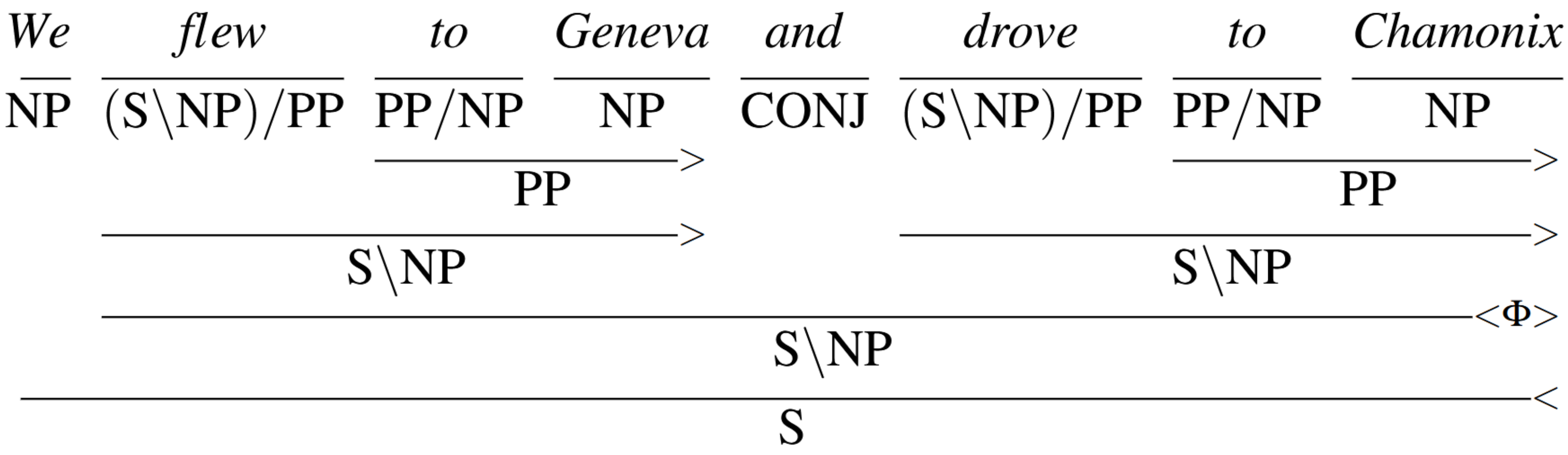
← lexicon

$$\dfrac{\dfrac{\dfrac{\text{the}}{NP/N} \quad \dfrac{\text{dog}}{N}}{NP}> \qquad \dfrac{\dfrac{\text{bit}}{(S\backslash NP)/NP} \quad \dfrac{\text{John}}{NP}}{S\backslash NP}>}{S}<$$

[wikipedia.org/wiki/Combinatory_categorial_grammar]

COMBINATORY CATEGORIAL GRAMMAR

- Aside from forward application (>) and backward application (<), there are a few more combination rules:
- Conjunction:



[Jurafsky and Martin, Speech and Language Processing, 2025]

COMBINATORY CATEGORIAL GRAMMAR

- Aside from forward application (>) and backward application (<), there are a few more combination rules:
- Composition:

$$\frac{\alpha : X/Y \qquad \beta : Y/Z}{\alpha\beta : X/Z} B_{>}$$

$$\frac{\beta : Y\backslash Z \qquad \alpha : X\backslash Y}{\beta\alpha : X\backslash Z} B_{<}$$

[wikipedia.org/wiki/Combinatory_categorial_grammar]

COMBINATORY CATEGORIAL GRAMMAR

- Aside from forward application (>) and backward application (<), there are a few more combination rules:
- Type-raising:

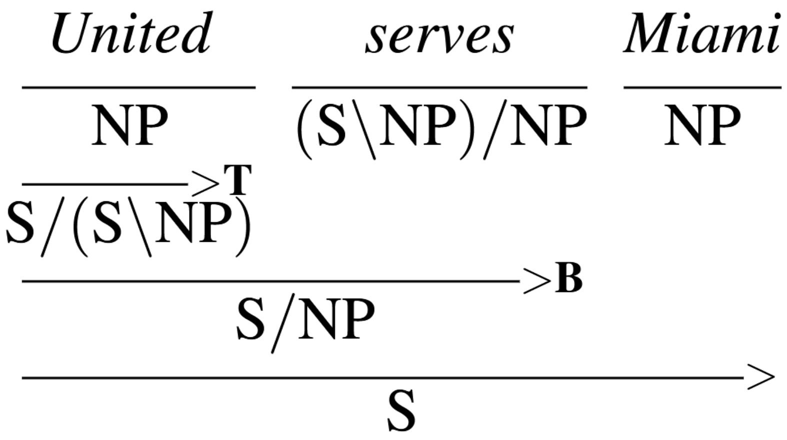
$$\frac{\alpha : X}{\alpha : T/(T\backslash X)}T_{>}$$

$$\frac{\alpha : X}{\alpha : T\backslash(T/X)}T_{<}$$

[wikipedia.org/wiki/Combinatory_categorial_grammar]

COMBINATORY CATEGORIAL GRAMMAR

- Why are the extra rules useful?

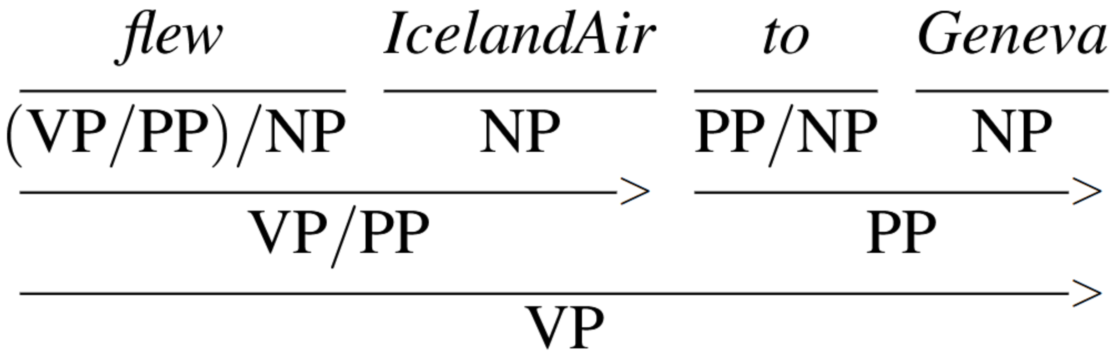


- This sentence could be parsed by first combining ‘`serves`’ and ‘`Miami`’.
- But through the use of type-raising and composition, we obtain an alternate left-to-right derivation.
  - This more closely mimics how humans parse natural language.

[Jurafsky and Martin, Speech and Language Processing, 2025]

COMBINATORY CATEGORIAL GRAMMAR

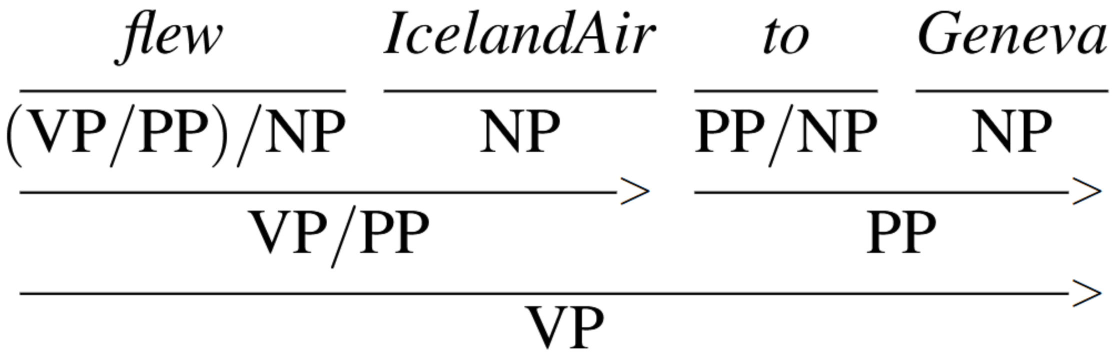
- Why are the extra rules useful?
- Consider the sentence ‘`I flew IcelandAir to Geneva.`’
- We could parse it as:



[Jurafsky and Martin, Speech and Language Processing, 2025]

COMBINATORY CATEGORIAL GRAMMAR

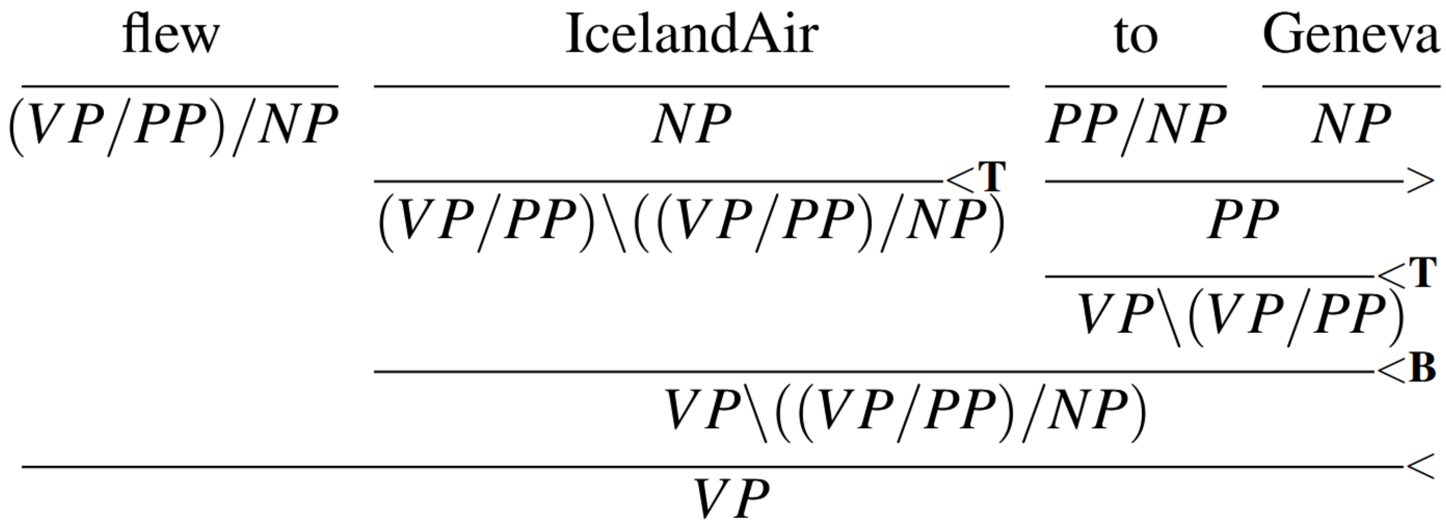
- But what about the sentence ‘`I flew IcelandAir to Geneva and SwissAir to London`’?
- To parse this correctly, we need to apply the conjunction rule to ‘`IcelandAir to Geneva`’ and ‘`SwissAir to London`’.



- But the above derivation combines ‘`IcelandAir`’ with ‘`flew`’.

COMBINATORY CATEGORIAL GRAMMAR

- But we can workaround this by using type-raising and composition rules:

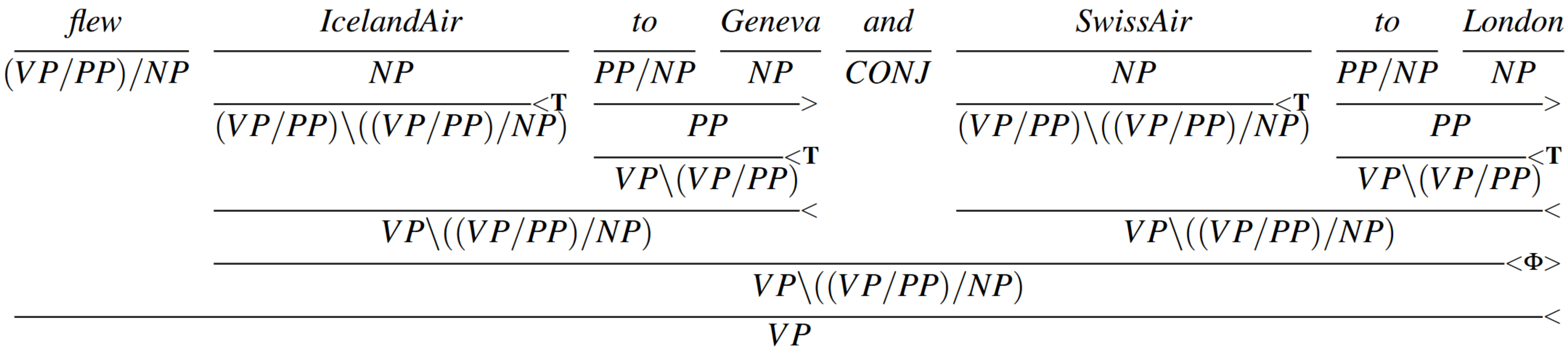


- Now ‘IcelandAir to Geneva’ is combined before combining with ‘flew’.

[Jurafsky and Martin, Speech and Language Processing, 2025]

COMBINATORY CATEGORIAL GRAMMAR

- And we can correctly parse '`I flew IcelandAir to Geneva and SwissAir to London`'.



PARSING CCG

- How do you parse CCG?
- Since CCG operations are unary or binary, we can extend CKY parsing.
  - CCG is well-suited for bottom-up parsing.
  - Vijay-Shanker and Weir (1993) describe this algorithm and show that it has running time $O(n^6)$.
- This can be made practically faster using beam search and better heuristics (e.g., A*).
  - But better heuristics do not change the worst-case running time.
  - And beam search sacrifices exactness/accuracy.

NEXT TIME

- Next time, we will wrap up our discussion of syntax.
- We move onto semantics.
  - How can we describe the meaning of sentences?
  - What are different representations of meaning?
  - Can we use some representations to do reasoning?
    - E.g., logic

QUESTIONS?