Faculty of Computer Science, Dalhousie University *18-Nov-2024*

CSCI 4152/6509 — Natural Language Processing

Lecture 18: Deep Learning and NLP; DCG and PCFG

Location: Carleton Tupper Building Theatre C Instructor: Vlado Keselj
Time: 16:05 – 17:25

Previous Lecture

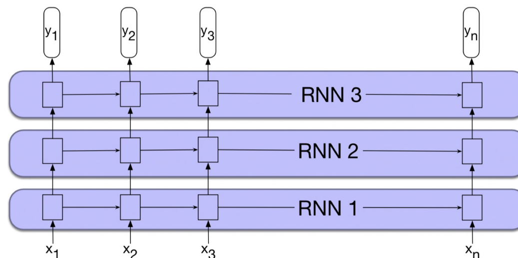
Neural networks and deep learning

- Applications
- Some main developments
- Large deep learning models
- Exponential growth in size of LLMs
- Biological neuron, perceptron, feed-forward network
- Activation functions, softmax function
- Neural language model, RNN

Slide notes:

Stacked RNN

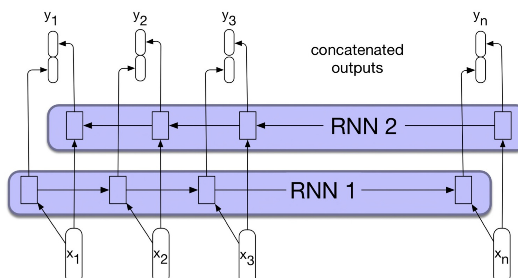
- Stacked RNN: Output from lower level is input to higher level; top level is final output (Jurafsky and Martin, Figure 9.10)



Slide notes:

Bidirectional RNN

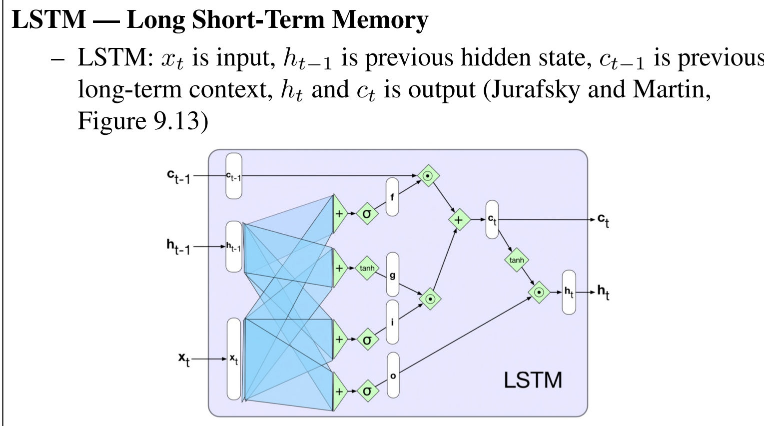
- Bidirectional RNN; trained forward and backward with concatenated output (Jurafsky and Martin, Figure 9.11)
- Output can be used for sequence labeling, for example



Slide notes:

LSTM — Long Short-Term Memory

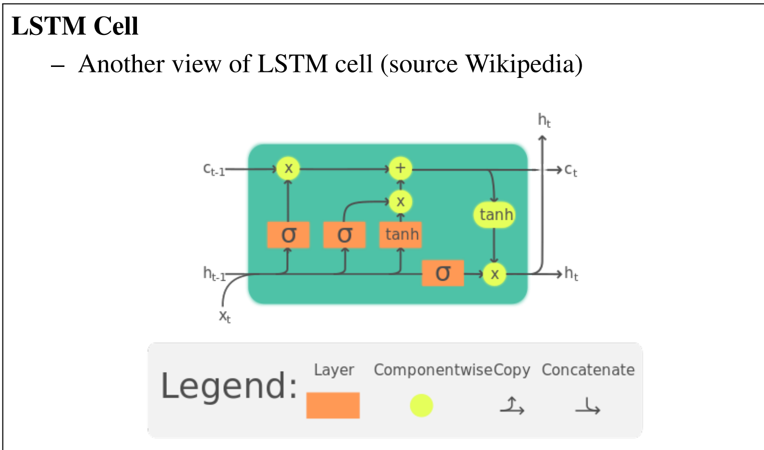
- LSTM: x_t is input, h_{t-1} is previous hidden state, c_{t-1} is previous long-term context, h_t and c_t is output (Jurafsky and Martin, Figure 9.13)



Slide notes:

LSTM Cell

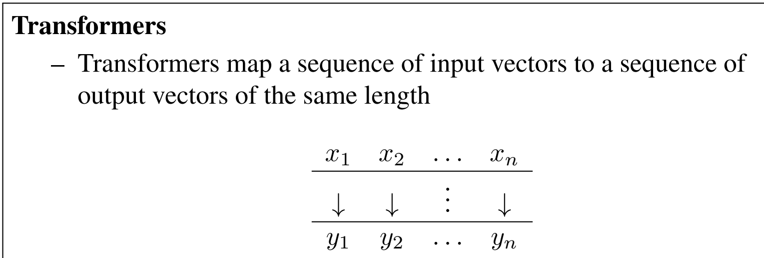
- Another view of LSTM cell (source Wikipedia)



Slide notes:

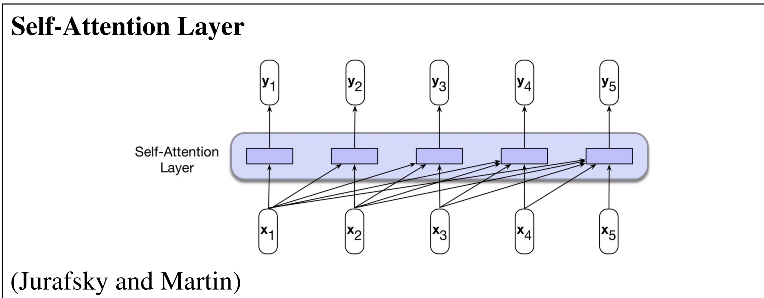
Transformers

- Transformers map a sequence of input vectors to a sequence of output vectors of the same length



Slide notes:

Self-Attention Layer



(Jurafsky and Martin)

Slide notes:

Self-Attention Training

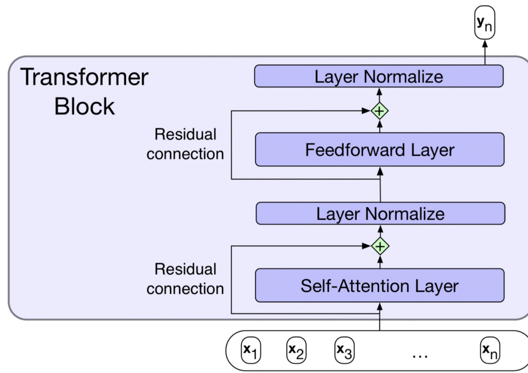
$$score(x_i, x_j) = x_i \cdot x_j$$

$$\alpha_{ij} = \text{softmax}(score(x_i, x_j)) \quad \forall j \leq i$$

$$y_i = \sum_{j \leq i} \alpha_{ij} x_j$$

Slide notes:

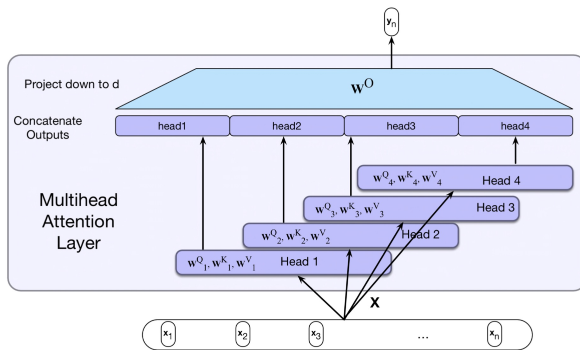
Transformer Block



(Jurafsky and Martin)

Slide notes:

Multihead Attention Layer



(Jurafsky and Martin)

Slide notes:

Encoding Word Positions in Transformers

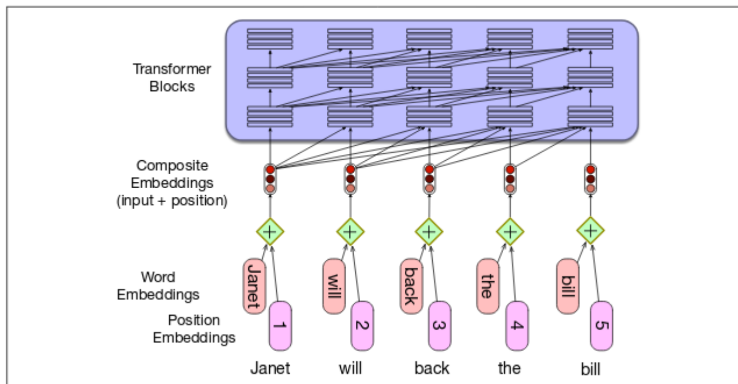


Figure 9.20 A simple way to model position: simply adding an embedding representation of the absolute position to the input word embedding.

from: Jurafsky and Martin, 3rd ed. draft

Slide notes:

Training Transformer as a Language Model

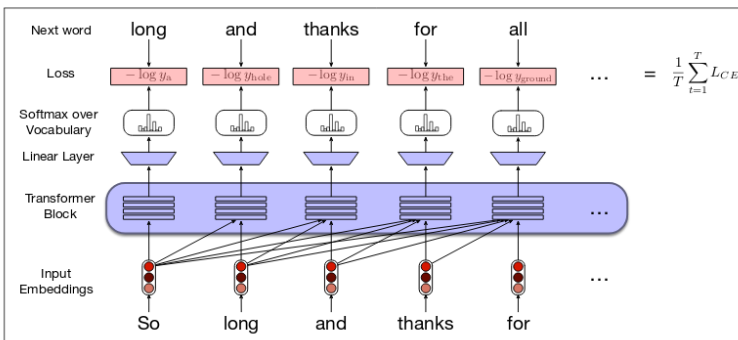


Figure 9.21 Training a transformer as a language model.

from: Jurafsky and Martin, 3rd ed. draft

Slide notes:

Text Completion with Transformers

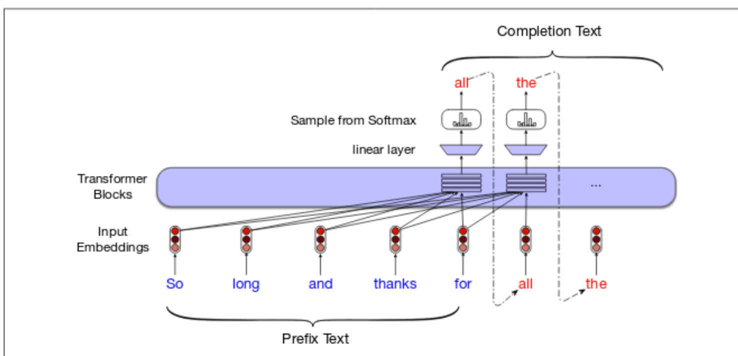


Figure 9.22 Autoregressive text completion with transformers.

from: Jurafsky and Martin, 3rd ed. draft

Part IV

Parsing

In this part, we will move a level above in processing natural languages—parsing, or syntactic processing. For some practical purposes, we will start with an brief introduction to the Prolog programming language.

Parsing Natural Languages

- Must deal with possible ambiguities
- Decide whether to make a phrase structure or dependency parser
- When parsing NLP, there are generally two approaches:
  1. Backtracking to find all parse trees
  2. Chart parsing
- Prolog provides a very expressive way to NL parsing
- FOPL is also used to represent semantics

18 A Brief Introduction to Prolog

In this section, we will first go over a brief Prolog review. Prolog is described in some more details in the lab tutorial.

Slide notes:

Parsing with Prolog

- We will go over a brief Prolog review
  - more details are provided in the lab
- Implicative normal form:

$$p_1 \land p_2 \land \ldots \land p_n \Rightarrow q_1 \lor q_2 \lor \ldots \lor q_m$$

- If $m \leq 1$, then the clause is called a **Horn clause.**
- If resolution is applied to two Horn clauses, the result is again a Horn clause.
- Inference with Horn clauses is relatively efficient

An implicative normal form is a mathematical logic formula, which is a conjunction of smaller formulae called clauses, where each clause is in the following form:

$$p_1 \land p_2 \land \ldots \land p_n \Rightarrow q_1 \lor q_2 \lor \ldots \lor q_m$$

where p_i and q_i are simple logical statements called propositions.

Note: Just as a reminder, the operator $\land$ is the logical AND, operator $\lor$ is the logical OR, and the operator $\Rightarrow$ is the logical "implies" operator.

If $m \leq 1$, then the clause is called a **Horn clause**.

When resolution is applied to two Horn clauses, the result is again a Horn clause. Inference on Horn clauses is relatively efficient.

Rules

A Horn clause with $m = 1$ is called a **rule**:

$$p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q_1$$

It is expressed in Prolog as:

```
q1 :- p1, p2, ..., p_n.
```

Facts

A clause with $m = 0$ is called a **fact**:

$$p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow \top$$

is expressed in Prolog as:

```
p1, p2, ..., p_n.
```

or

```
:- p1, p2, ..., p_n.
```

and it is called a **fact.**

Running Prolog

It is covered in more details in the lab how to run Prolog interpreter. We use a Prolog interpreter called SWI Prolog and it is available on the `timberlea` server. The lab also covers how to write a program, load it and execute it using interpreter.

Rabbit and Franklin Example

The 'rabbit and franklin' example in Prolog:

```
hare(rabbit).
turtle(franklin).
faster(X,Y) :- hare(X), turtle(Y).
```

Save the program in a file, e.g., named `file.prolog` and load the file using the command `['file.prolog']`. The Prolog interpreter uses prompt '`?-`'. After loading the file, on Prolog prompt, type:

```
faster(rabbit,franklin).
```

After this there is some difference between Prolog interpreters. The newest SWI-Prolog will simply print 'true' and go back to the prompt. The previous version of SWI-Prolog would print 'Yes' waiting for user input. The user should type semicolon (`;`) and then the Prolog prompt would appear.

Try `faster(X,franklin).` and `faster(X,Y).` in the similar fashion (keep pressing the semicolon if user input is required until the Prolog prompt is obtained in the both cases).

Slide notes:

Unification and Backtracking

- Two important features of Prolog: unification and backtracking
- Prolog expressions are generally mathematical symbolic expressions, called *terms*
- **Unification** is an operation of making two terms equal by substituting variables with some terms
- **Backtracking:** Prolog uses backtracking to satisfy given goal; i.e., to prove given term expression, by systematically trying different rules and facts, which are given in the program

Example in Unification and Backtracking

- What happens after we type:
  `?- faster(rabbit,franklin).`
- Prolog will search for a 'matching' fact or head of a rule:
  `faster(rabbit,franklin)` and
  `faster(X,Y) :- ...`
- 'Matching' here means **unification**
- After unifying `faster(rabbit,franklin)` and `faster(X,Y)` with substitution `X`←`rabbit` and `Y`←`franklin`, the rule becomes:
  `faster(rabbit,franklin) :-   hare(rabbit), turtle(franklin).`

Example (continued)

- Prolog interpreter will now try to satisfy predicates at the right hand side: `hare(rabbit)` and `turtle(franklin)` and it will easily succeed based on the same facts
- If it does not succeed, it can generally try other options through **backtracking**

Variables

Variable names in Prolog start with an uppercase letter or an underscore character ('_]). The variable name _ (just an underscore) is special because it denotes a special, so-called *anonymous* variable. Two occurrences of this variable can represent arbitrary different values, and there is no connection between them. This variable is used a placeholder in terms for part that is generally ignored.

Slide notes:

Variables in Prolog

- Variable names start with uppercase letter or underscore ('_')
- _ is a special, *anonymous variable*
- Examples:

```
?- faster(rabbit,franklin).
Yes ;
...
?- faster(rabbit,X).
X = franklin ;
...
?- hare(X).
X = rabbit ;
```

Lists (Arrays), Structures.

Lists are implemented as linked lists. Structures (records) are expressed as terms. Examples:

In program: `person(john,public,'123-456').`

Interactively: `?- person(john,X,Y).`

`[]` is an empty list.

A list is created as a nested term, usually a special function '`.`' (dot):

```
?- is_list(.(a, .(b, .(c, [])))).
```

List Notation

`(.(a, .(b, .(c, []))))` is the same as `[a,b,c]`

This is also equivalent to:

```
[ a | [ b | [ c | [] ]]]
```

or

```
[ a, b | [ c ] ]
```

A frequent Prolog expression is: `[H|T]`
where H is head of the list, and T is the tail, which is another list.

Example: Calculating Factorial

```
factorial(0,1).
factorial(N,F) :- N>0, M is N-1, factorial(M,FM),
    F is FM*N.
```

After saving in `factorial.prolog` and loading to Prolog:

```
?- ['factorial.prolog'].
% factorial.prolog compiled 0.00 sec, 1,000 bytes

Yes
?- factorial(6,X).

X = 720 ;
```

Example: List Membership

Example (testing membership of a list):

```
member(X, [X|_]).
member(X, [_|L]) :- member(X,L).
```

19 Natural Language Syntax

Slide notes:

Natural Language Syntax

- Syntax — NLP level of processing
  - Syntax = sentence structure; i.e., study of the phrase structure
- sýntaxis (Greek) — "setting out together, arrangement"
- Words are not randomly ordered — word order is important and non-trivial
- There are "free-order" languages (e.g., Latin, Russian), but they are not completely order free.
- Reading: Chapter 12 (JM book) or Ch.17 (JM on-line)

Phrase Structure and Dependency Structure

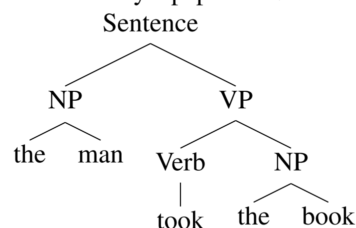
- Two ways of organizing sentence structure:
  - phrase structure
  - dependency structure
- Phrase structure
  - nested consecutive groupings of words
- Dependency structure
  - dependency relations between words
- The main NLP task at the syntax level: parsing
  - given a sentence, find the correct structure

Phrase Structure

- Phrase Structure Grammars or Context-Free Grammars
- A hierarchical view of sentence structure:
  - words form phrases
  - phrases form clauses
  - clauses form sentences
- Parsing: given a sentence find the context-free parse tree; a.k.a. phrase structure parse tree

Phrase Structure Parse Tree Examples

- Phrase Structure parse trees are also called Context-Free parse trees
- This example is from the seminal Noam Chomsky's paper in 1956:

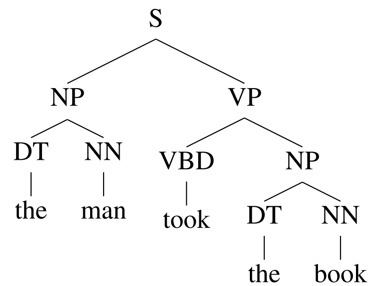


The above example is from the seminal article of Noam Chomsky, "Three models for the description of language" published in IRE Transactions on Information Theory in 1956.

If we follow more closely the Penn treebank tagset, we would rewrite the above parse tree as follows:

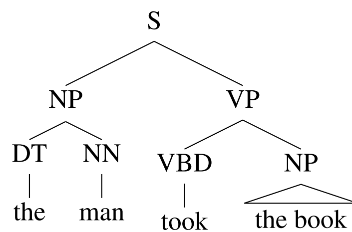
Parse Tree Examples (Penn treebank tagset)

- Using Penn treebank tagset:



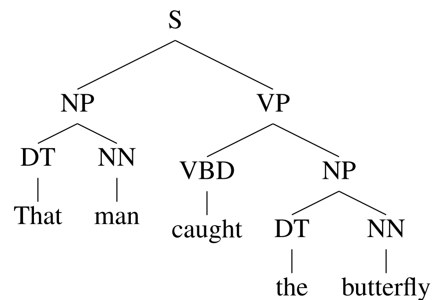
Parse Tree Examples ('triangle' notation)

- Sometimes we simplify a parse tree by ignoring a part of the structure, as in:



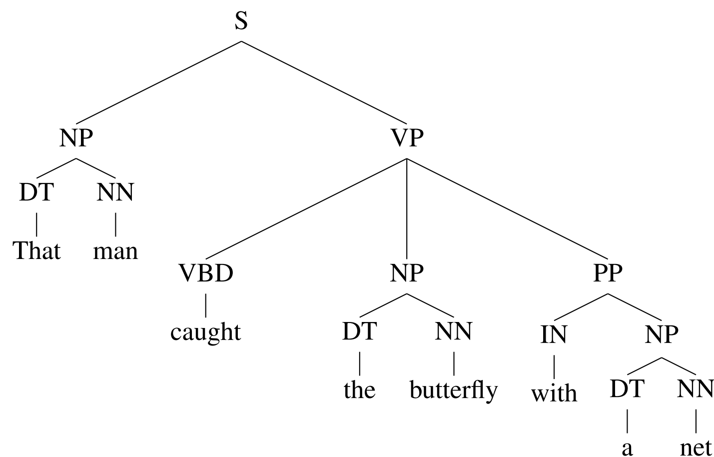
Parse Tree Example 2 ('butterfly')

- Another example:



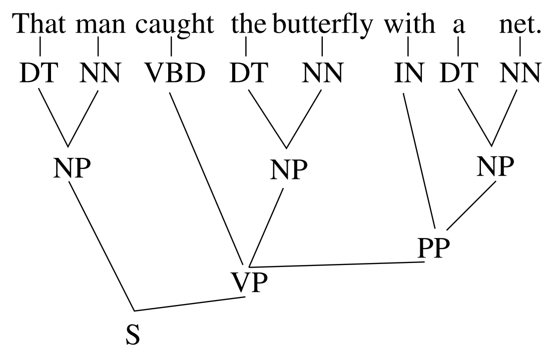
Parse Tree Example3 ('butterfly' extended)

- Extending the previous example:



Parse Tree Example (root bottom)

- Representing parse trees in the bottom-up direction:



Some Basic Notions in Context-Free Trees

- Context-free trees, also called phrase structure trees, parse trees, syntactic trees
- Node relations: root, leaf, parent (mother), child (daughter), sibling, ancestor, descendant, dominate
- Context-free grammar
- Consider for example the context-free grammar induced by the last parse tree shown

20 Context-Free Grammars (CFG) Review

Context-Free Grammars (CFG) Review

CFG is a tuple (V, T, P, S), where

- V is a finite set of **variables** or **non-terminals**; e.g., $V = \{S, NP, DT, NN, VP, VBD, PP, IN\}$
- T is a finite set of **terminals**, words, or lexemes; e.g., $T = \{$That, man, caught, the, butterfly, with, a, net$\}$
- P is a set of **rules** or **productions** in the form $X \to \alpha$, where $X \in V$ and $\alpha \in (V \cup T)^*$; e.g.,
  $P = \{S \to NP\ VP,\ \ NP \to DT\ NN,\ \ DT \to \text{That},\ \ NP \to \epsilon\}$
- S is the **start symbol** $S \in V$

Some Notions about CFGs

- CFG, also known as Phrase-Structure Grammar (PSG)
- Equivalent to BNF (Backus-Naur form)
- Idea from Wundt (1900), formally defined by Chomsky (1956) and Backus (1959)
- Typical notation (V, T, P, S); also (N, Σ, R, S)

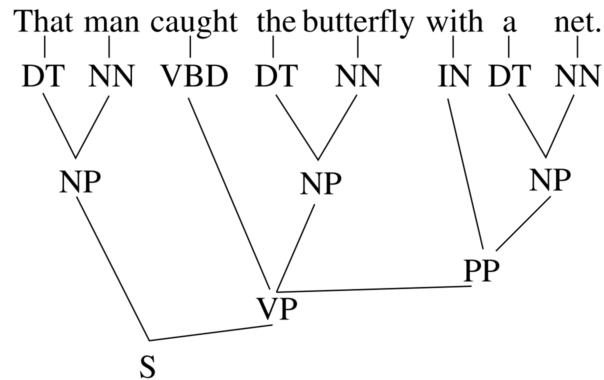
CFG (Context-Free Grammar) is also knows as the Phrase-Structure Grammar (PCG). It is usually referred to as CFG in the Formal Language theory and Computer Science in general, while the term PCG is used in some Computational Linguistic and Linguistic circles. The idea of CFG is traced to Wundt in 1900, but the exact created of the formalism is attributed to Noam Chomsky (1956), who worked in the area of Natural Language Processing, and also to John Backus, who worked in the area of Programming Languages.

CFG Derivations

CFG Derivations

- Direct derivation, derivation
- Example of a direct derivation: $S \Rightarrow NP\ VP$
- Example of a derivation (beginning of):
  $S \Rightarrow NP\ VP \Rightarrow DT\ NN\ VP \Rightarrow$ That $NN\ VP \Rightarrow \ldots$
- Left-most and right-most derivation

Parse Tree Example (revisited)



Leftmost Derivation Example

$$
\begin{aligned}
S &\Rightarrow NP\ VP \Rightarrow DT\ NN\ VP \Rightarrow \text{That } NN\ VP \Rightarrow \text{That man } VP \\
&\Rightarrow \text{That man } VBD\ NP\ PP \\
&\Rightarrow \text{That man caught } NP\ PP \\
&\Rightarrow \text{That man caught } DT\ NN\ PP \\
&\Rightarrow \text{That man caught the } NN\ PP \\
&\Rightarrow \text{That man caught the butterfly } PP \\
&\Rightarrow \text{That man caught the butterfly } IN\ NP \\
&\Rightarrow \text{That man caught the butterfly with } NP \\
&\Rightarrow \text{That man caught the butterfly with } DT\ NN \\
&\Rightarrow \text{That man caught the butterfly with a } NN \\
&\Rightarrow \text{That man caught the butterfly with a net}
\end{aligned}
$$

Some Notions about CFGs (continued)

- Language generated by a CFG
- Context-Free languages
- Parsing task
- Ambiguous sentences
- Ambiguous grammars
- Inherently ambiguous languages

A language over a set of words (or terminals) in this context (i.e., the formal language context) is any set of strings of words, where a string of words is any finite, possibly empty, sequence of words.

The language generated by a CFG is the set of all strings of words that can be derived from the start symbol using a derivation.

A language is a context-free language, if there is a CFG that generates this language.

The parsing problem for a CFG is the problem of finding all parse trees of an arbitrary string of words, which may include an empty set of trees if the string does not belong to the language generated by the grammar.