FNLP - Week 7: Evaluating Parsing, Improving PCFGs and Representing Meaning With Semantics

Antonio León Villares

March 2022

Contents

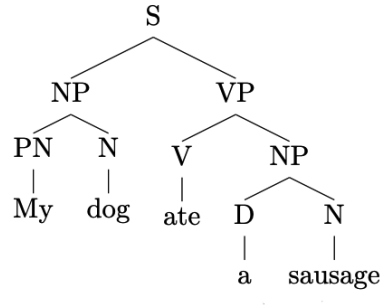
1 Evaluating Parsers

1.1 Evaluation Means for Parsing

- **What is intrinsic evaluation?**
  - compare parse results with some form of **gold label**:
    - **Automatic**: using human provided **gold standards** (good to keep track of **state of the art**)
    - **Manual**: human evaluate parse
- **What is extrinsic evaluation?**
  - use parsed **syntactic representation** in a downstream application
  - for example, compare performance of different parsers in a semantic analyser
- **What aspects of a parse are taken into account during evaluation?**
  - **exact match**: % of trees predicted correctly
  - **bracket score**: measure constituents in common between parse and gold label (**most common**)
  - **crossing brackets**: % of overlapping phrase boundaries (i.e the % of constituents with gold label $((AB)C)$ but parse $(A(BC))$)
  - **dependency metrics**: % of correctly identified heads in the **dependency structure** of the **constituent tree** (more on this later)

1.2 Bracket Scores

- **What is bracketing notation?**
  - a way of representing a **parse tree**




```
(S
  (NP  (PN My) (N Dog) )
  (VP  (V ate)
       (NP (D a )  (N sausage) )
  )
)
```

- **What exactly does bracket score measure?**
  - recall, CYK constructs a parse tree by considering whether a **span** of a sentence $span(min, max)$ has a given non-terminal C
  - in this regard, a tree can be thought of as a collection of **brackets**:

$$[min, max, C]$$

- **bracket score** compares the **brackets** predicted by a parser, versus the **gold label** brackets

- **How are brackets scores computed?**
  - **precision**:
  $$P = \frac{\#\ of\ brackets\ agreed\ on\ by\ parser\ and\ annotation}{\#\ of\ brackets\ predicted\ by\ parser}$$
  - **recall**:
  $$P = \frac{\#\ of\ brackets\ agreed\ on\ by\ parser\ and\ annotation}{\#\ of\ brackets\ in\ the\ annotation}$$
  - F_1 **Score**:
  $$\frac{2 \times P \times R}{P + R}$$

- **Why is bracket score more useful than accuracy/exact match?**
  - exact match can't distinguish between parses in which all constituents were wrong, and parses with only 1 wrong constituent
  - it is more useful and fine-grained to consider constituents, particularly when parsing long sentences, where the probability of a mistake is greater

- **What issues might be associated with using constituents to evaluate parsing?**
  - different parsers produce different trees $\implies$ need to convert to parse format of gold labels

2 Improving Vanilla PCFGs for Parsing

2.1 Recapping Weaknesses of PCFGs

- **What are the key issues of PCFGs?**
  - **strong independence assumption**: doesn't account for context/structural dependencies, so certain nuances (i.e distribution of NPs across **subjects** or **objects** in a sentence varies) are not considered
  - **no lexical preferences**: certain words are more likely to be grouped together (i.e certain prepositions with certain verbs)

- **How do Vanilla PCFGs perform for parsing?**

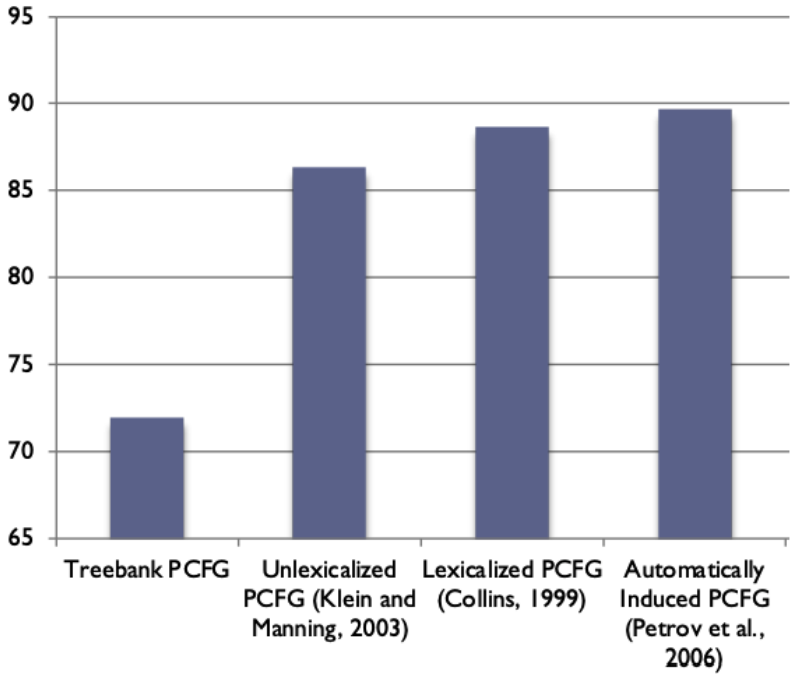


Figure 1: F_1 Scores for parsers trained on different grammars. We can see that vanilla PCFGs (directly from treebank rules) are the worst parsers, with around 0.72 F_1 score.

2.2 Improving PCFGs: Vertical Markovisation

- **What is Vertical Markovisation?**
  - in HMMs: **Markov Assumption** $\implies$ current tag depends on fixed history
  - **Vertical Markovisation**: non-terminal depends on previous non-terminals (beyond parents)

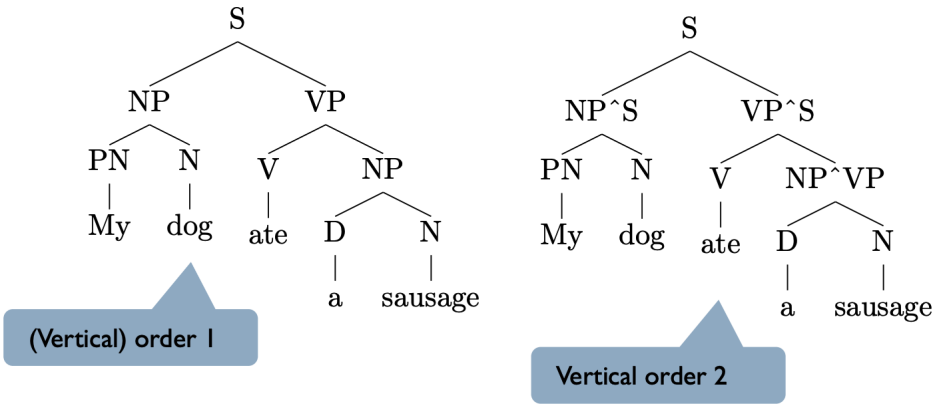


Figure 2: With second order markovisation, a non-terminal is derived based on its parent and grandparent.

- **How does Vertical Markovisation help?**

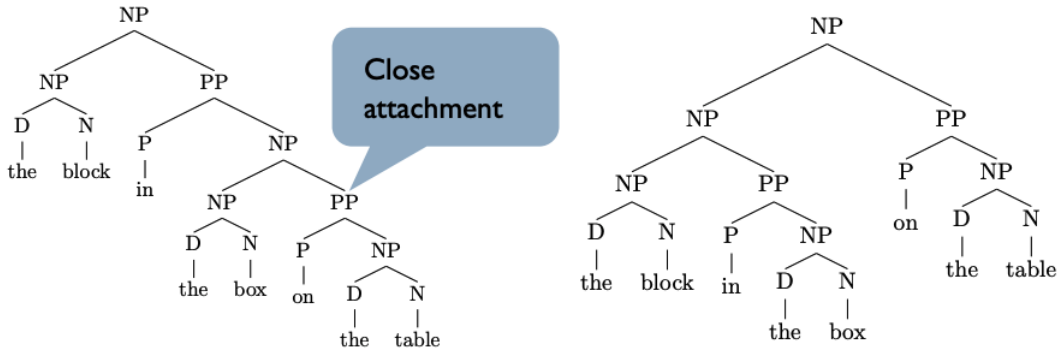


Figure 3: We can consider disambiguiting these 2 sentences. In Penn Treebank, close attachment is more common, but looking at these trees, they use the exact same non-terminal derivations, so the probability of both parses is exactly the same.

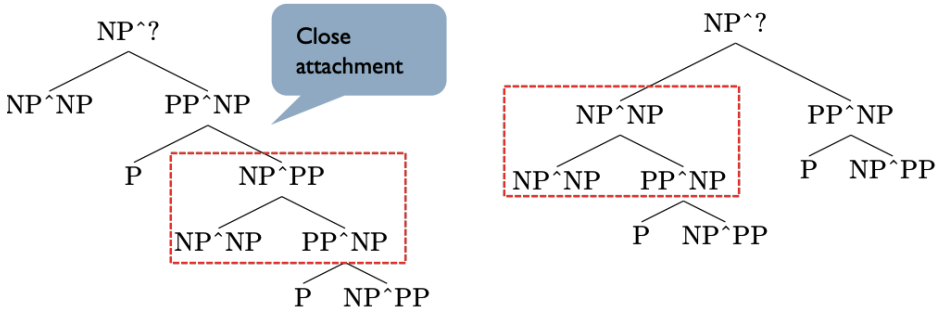


Figure 4: Introducing Vertical Markovisation changes this: now there are different rules involved, and close attachment could be preferred.

- **How do PCFGs constructed with Vertical Markovisation improve on Vanilla PCFGs?**

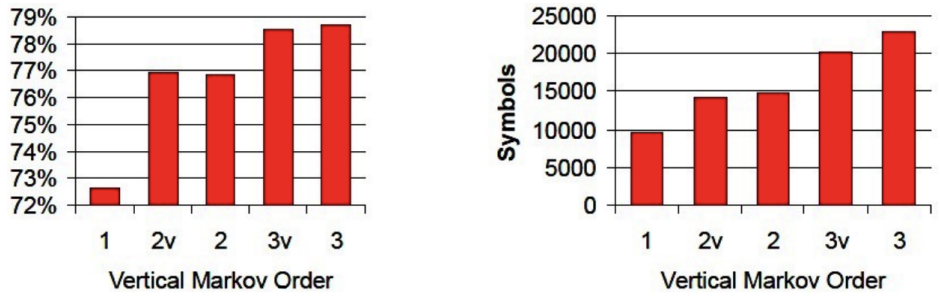


Figure 5: Generally, increasing the Markov Order Improves performance; however, this increases the number of symbols, which increases parse time (linearly for CYK).
However, clever tricks can be used, to only use certain histories (above represented with 2v and 3v), which reduce the number of symbols, whilst obtaining the benefits of markovisation.

2.3 Improving PCFGs: Better Binarisation

- **How can binarisation be streamlined to improved PCFGs?**
  - **binarisation**: process of converting PCFG to CNF

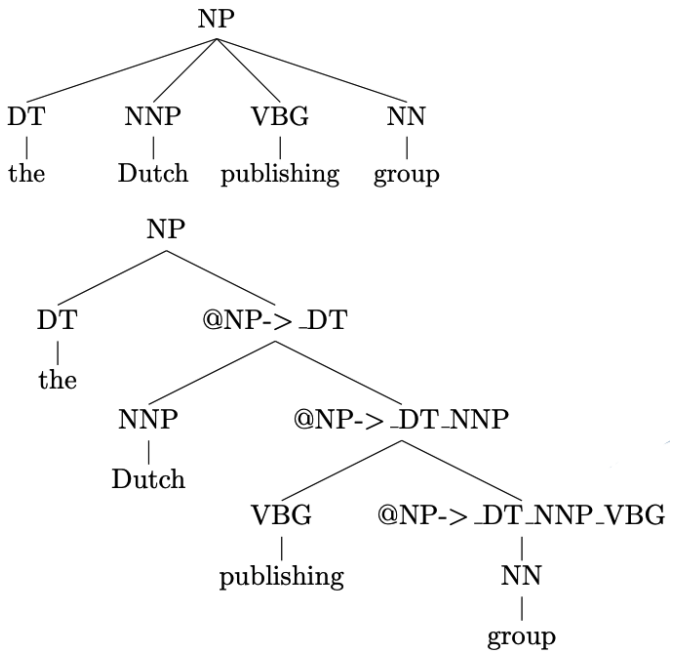


Figure 6: Binarisation can be thought as providing a **horizontal history** (as opposed to a **vertical history** like in markovisation.)

– we can be more selective about the **horizontal history** used

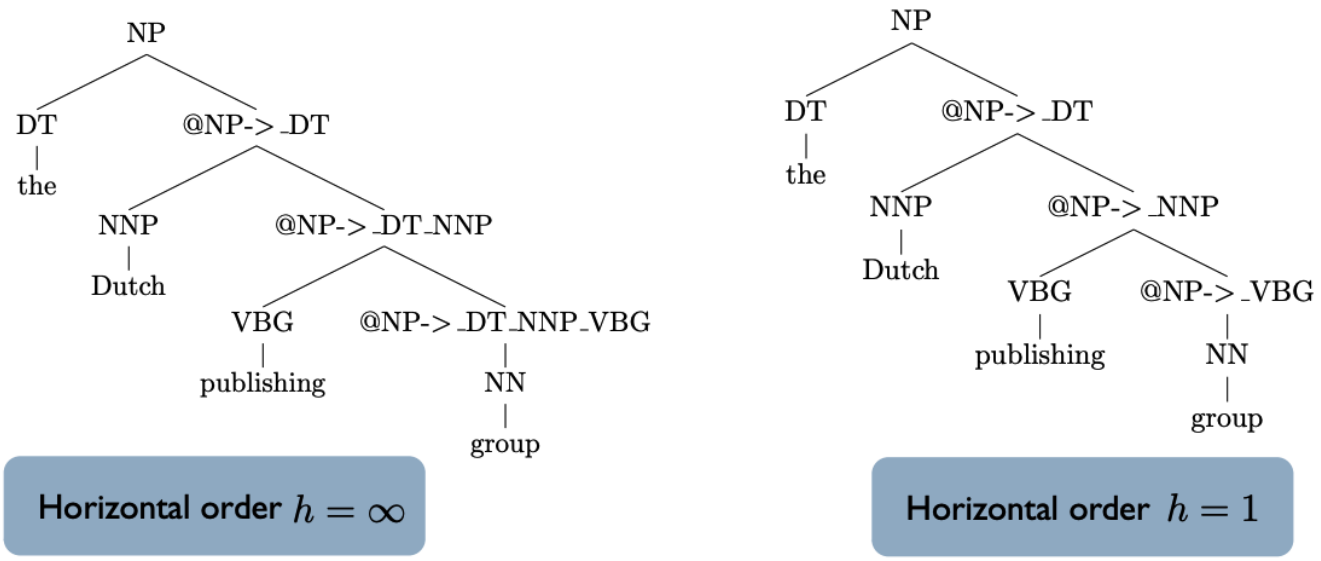
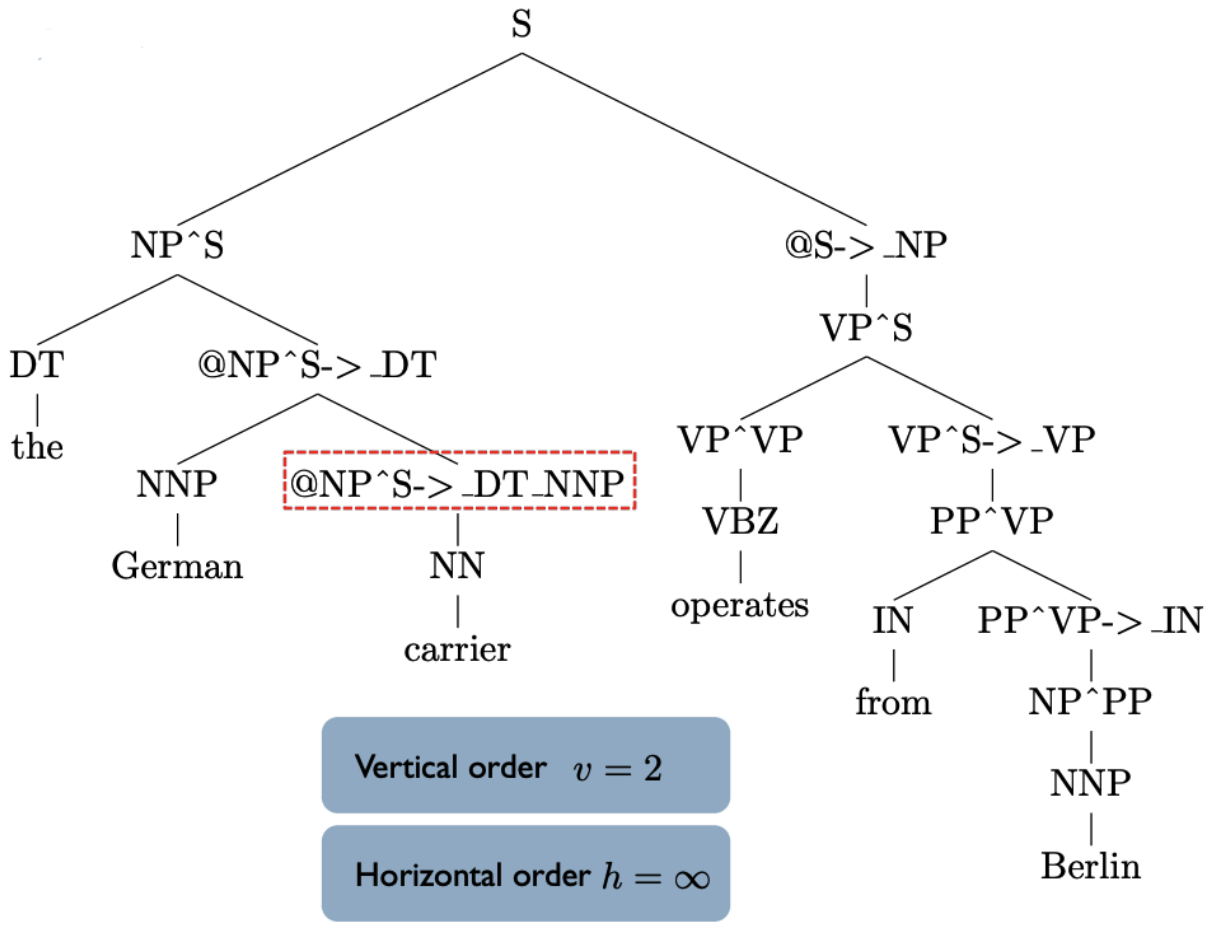


Figure 7: Instead of considering all previous derivations, just include the derivation which lead to the parent. For example, in "publishing group", we only note that $@NP- > \_NNP$ split into VBG (for "publishing"), instead of all the previous derivations (such as producing DT or NNP).

- **How do PCFGs constructed with Streamlined Binarisation improve on Vanilla PCFGs?**

Figure 8: We can see that there is a sweet spot, whereby decreasing the horizontal order produces a better model. For free, we also get a reduction in the number of symbols, so as opposed to vertical markovisation, we get improved parsing time!

- **Can verticlal and horizontal histories be used together?**



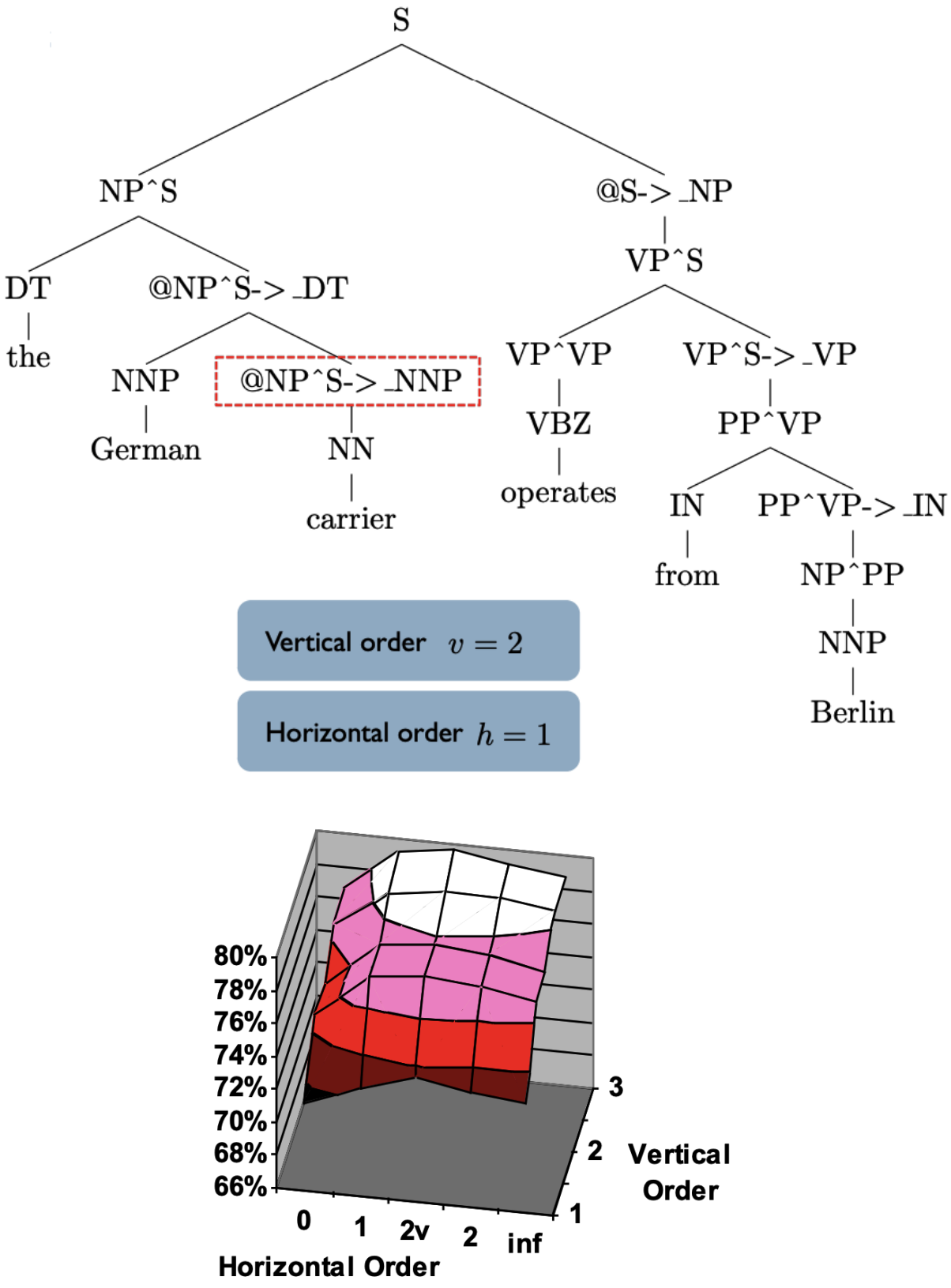


Figure 9: Applying both optimisation, we can attain up to 0.78 F_1, compared with Vanilla performance of 0.72.

2.4 Improving PCFGs: Splitting

- **What is splitting?**

- currently, the non-terminals are very **coarse**
- can augment labels:
  - in Penn: IN for both prepositions and subordinating conjunctions ("if") $\implies$ use non-terminal for each ($+0.2$ $F_1 \rightarrow 0.8$)
  - **split determiners** (demonstrative ["those"] vs other ["the", "a"])
  - **split adverbials** (phrasal or not ["quickly", "very"])
  - NP as subject or object
- overall, these small changes boost performance: 0.863 F_1

2.5 Advanced Improvements

- **What further improvements can be added?**

1. **Splits via Expectation Maximisation**: use EM to **automatically** learn non-terminals splits $\implies$ 0.9 F_1
2. **Neural Networks**: in CYK, probability of label C depends on children derivation:

$$P(C \rightarrow C_1\ C_2)$$

Use NN to incorporate additional data:

$$NN_\theta(C, C_1, C_2, min, max, mid)$$

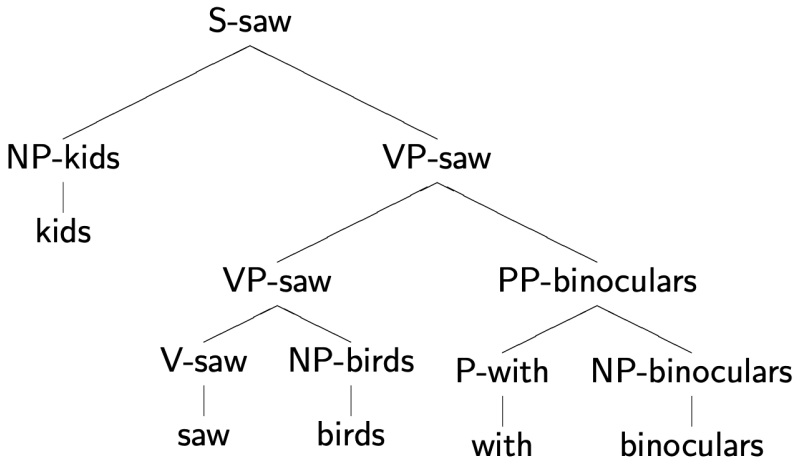
$\implies$ 0.96 F_1

3 Dependency Parsing

Above we focused on how to add context to constituent-based parsing. Here we discuss how lexical considerations can be applied: both to improve PCFGs, and to apply a completely new way of parsing - dependency parsing.

3.1 Adding Lexical Consideration to PCFGs

- **What is a lexical head?**
  - the word in a phrase which the **most important** (grammatically)
  - it is essential in transmitting the **core meaning** of a phrase
  - for example, in **noun phrases**, the head is the **noun**; in **verb phrases**, the head is the **verb**
- **What are the types of lexical heads?**
  - **content heads** (content words) vs **functional heads** (prepositions)
  - for example, in the PP "birds with fish":
    - **content head**: fish
    - **functional head**: with
- **What is lexicalisation?**
  - augment **non-terminals** by using their lexical head



- **Why is lexicalisation useful?**
  - with Vanilla PCFGs, we can replace a terminal with another one witht he same POS tag
  - this won't affect the produced parse, which isn't always right:

    kids saw birds with fish vs.
    kids saw birds with binoculars

    She stood by the door covered in tears vs.
    She stood by the door covered in ivy

    stray cats and dogs vs.
    Siamese cats and dogs

  - lexicalisation avoids this, since non-terminals depend on words
- **How can we evaluate lexicalisation?**
  - **Pros**:
    - more **specific** grammar (hopefully, $VP-saw \rightarrow VP-saw\ PP-fish$ is less likely than $VP-saw \rightarrow VP-saw\ PP-binoculars$, since it is strange to look at things through fish)
  - **Cons**:
    - very sparse grammar (need to use fancy smoothing; potentially create automatic subcategories to mitigate impact)

3.2 Purpose of Dependency Parsing

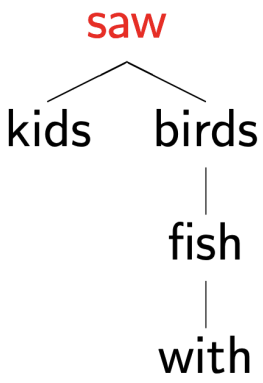
- **What is dependency parsing?**
  - produce a **parse tree** outlining **dependency relations** between words in a phrase
  - normal: meaning of words depends on other words, typically in an **asymmetric** and **binary** nature
  - useful for **morphologically rich** languages, languages with **free word order** (i.e Czech)
- **What are dependents?**

- parts of phrase which **modify** the **head** (sometimes called **modifiers**)
- include **direct** and **indirect** dependents
- for example:

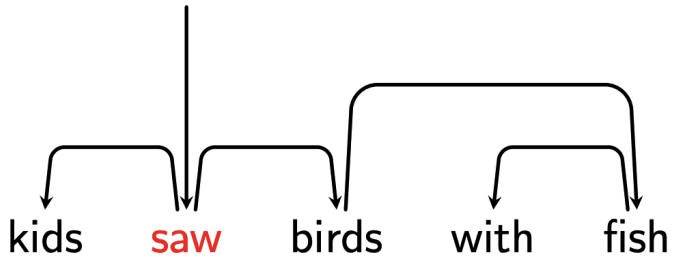
  "I prefer the morning flight through Denver."

  The head of the whole phrase is "flight", and it has a direct **dependency relation** with words like "prefer" (since "flight" is what is being modified by the preference) or "Denver" (since the act of flying involves going to Denver)

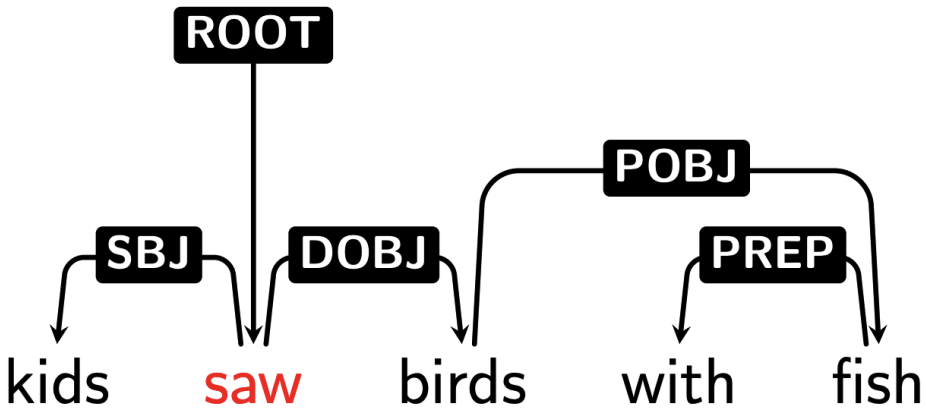
- **How are dependency parse trees represented?**
  - can be shown as a tree:



  - alternatively, use **arcs** to show **head-dependent** relations:



- **What are edge labels?**
  - in **constituent parsing**: use non-terminals to label nodes
  - we can label the **arcs** of **dependency parses**:



Clausal Argument Relations	Description
NSUBJ	Nominal subject
DOBJ	Direct object
IOBJ	Indirect object
CCOMP	Clausal complement
XCOMP	Open clausal complement
Nominal Modifier Relations	**Description**
NMOD	Nominal modifier
AMOD	Adjectival modifier
NUMMOD	Numeric modifier
APPOS	Appositional modifier
DET	Determiner
CASE	Prepositions, postpositions and other case markers
Other Notable Relations	**Description**
CONJ	Conjunct
CC	Coordinating conjunction

Figure 10: These are known as **Universal Dependency Relations**, which can be used to label the arcs.

- **What is a projective dependency parse?**
  - an arc *head* → *dependent* is **projective** if there is a **path** from *head* to all words between *head* and *dependent*
  - **dependency tree** projective ⟺ every arc is projective

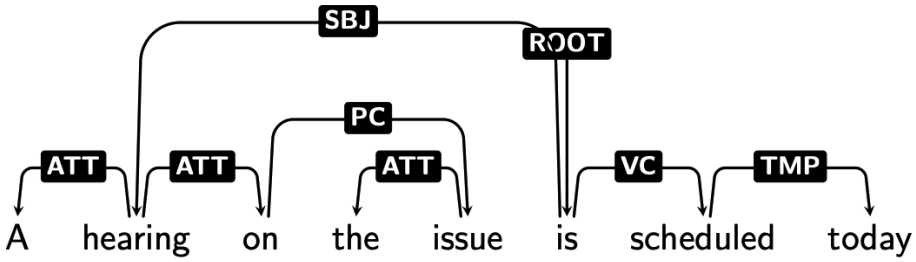


Figure 11: A projective dependency tree.

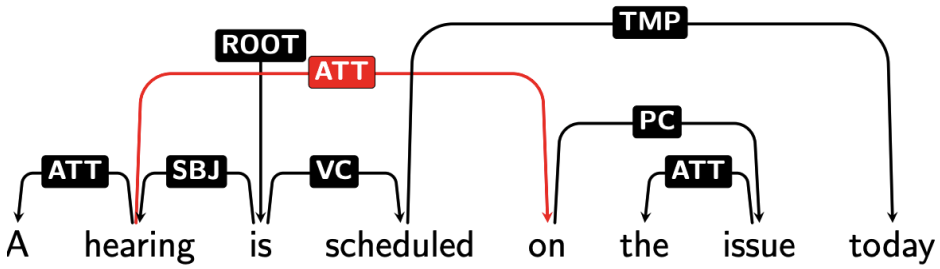
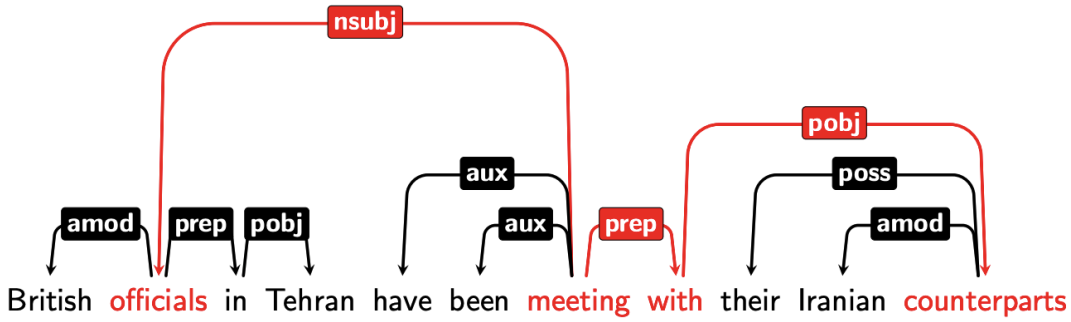


Figure 12: A non-projective dependency tree: there is no path between "hearing" and "schedule", even though there is an arc *hearing* → *on*.

  - intuitively: no arc intersects other arcs

- **Why is dependency parsing useful?**
  - can use head-dependent relations as **features**

– examples: **question-answering**, **information extraction** (for example, using **chains of dependencies**)



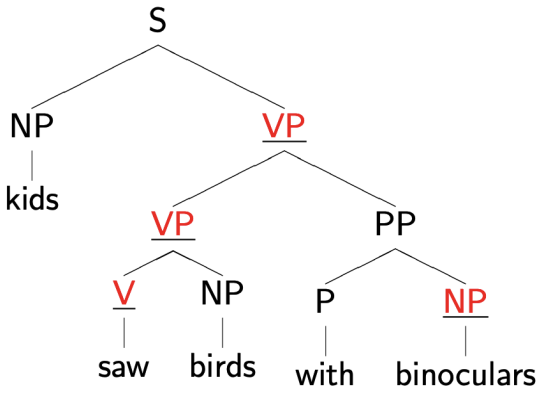
3.3 From Constituency Parse to Dependency Parse

- **What is a head rule?**
  - used to build dependency tree from constituent tree
  - in a PCFG, assigns the **head** of a phrase to one of the RHS non-terminals in a non-unary production:

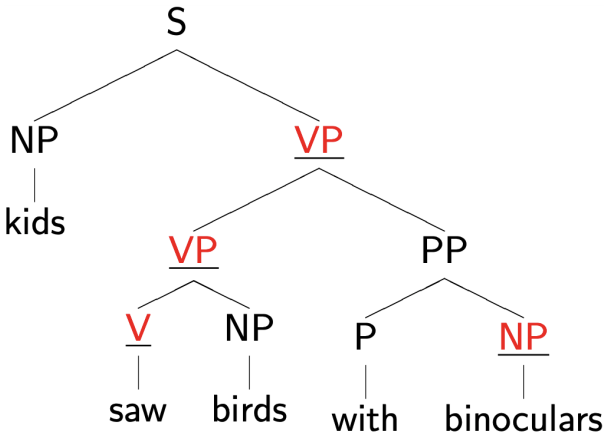
$$S \rightarrow NP\ VP$$

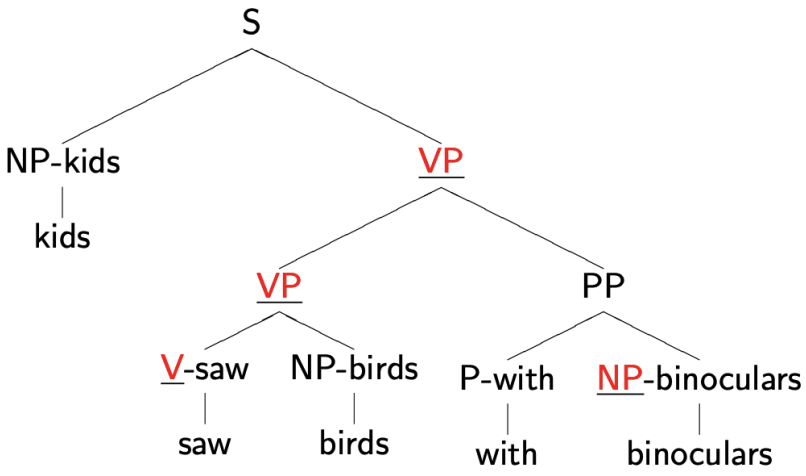
$$VP \rightarrow VP\ PP$$

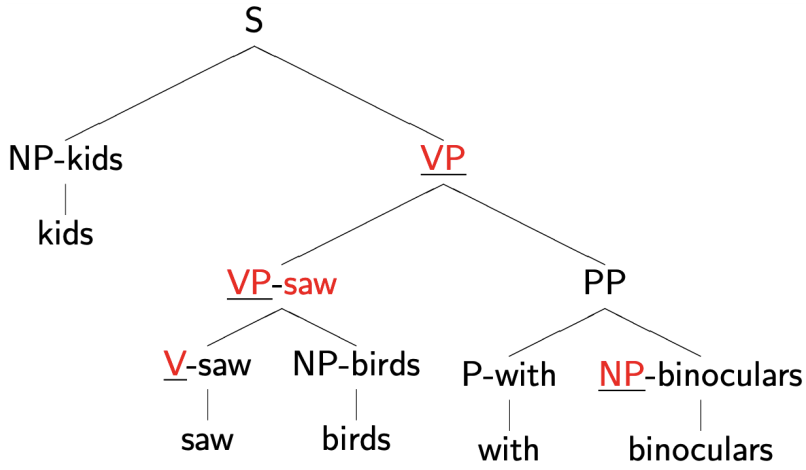
$$PP \rightarrow P\ NP$$

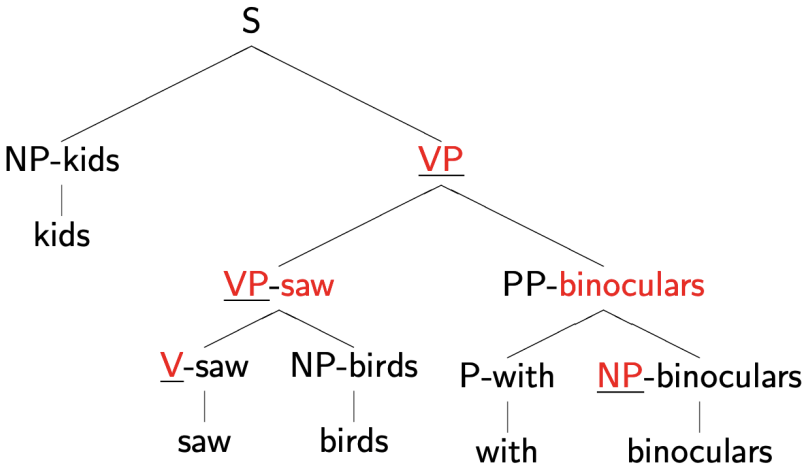


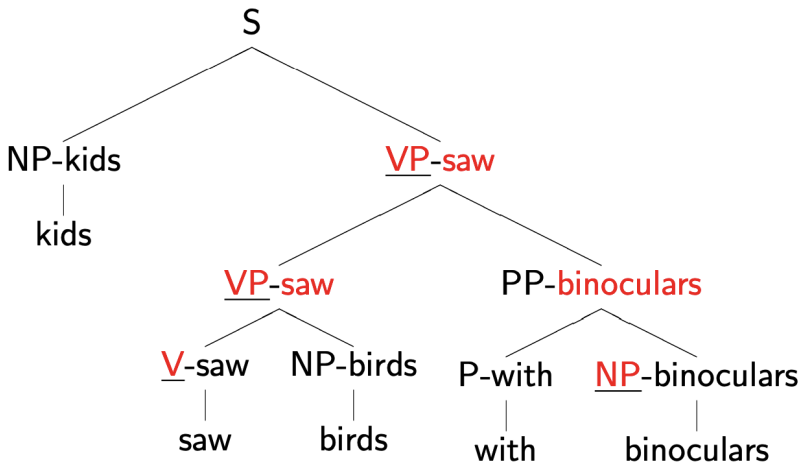
- **How can a constituent parse tree be converted into a dependency parse tree?**
  - apply **head rules** to constituent tree
  - propagate **head words** up the tree

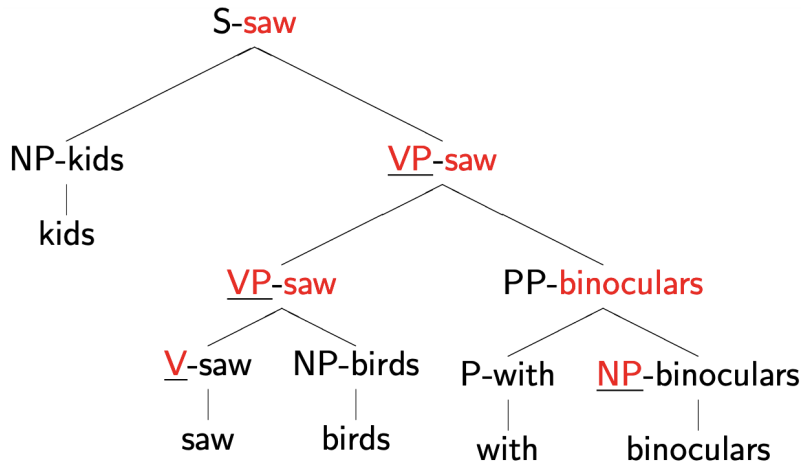



S
NP-kids
VP
kids
VP
PP
V-saw
NP-birds
P-with
NP-binoculars
saw
birds
with
binoculars

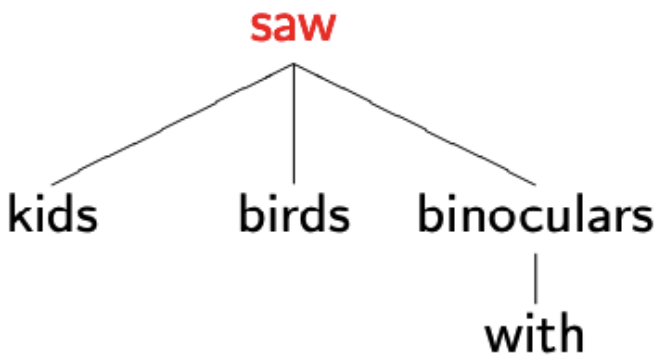

S
NP-kids
VP
kids
VP-saw
PP
V-saw
NP-birds
P-with
NP-binoculars
saw
birds
with
binoculars


S
NP-kids
VP
kids
VP-saw
PP-binoculars
V-saw
NP-birds
P-with
NP-binoculars
saw
birds
with
binoculars





- then, to construct a full dependency tree, we can "collapse" it: remove non-terminals, and join repeated head words:



3.4 Direct Dependency Parsing: Shift-Reduce

- **What is transition-based parsing?**
  - processes an input sentence and predicts a sequence of parsing actions in a left-to-right manner
- **What is the shift-reduce algorithm?**
  - **efficient** transition-based parser (no grammar required, $\mathcal{O}(n)$)
  - a **greedy** approach to dependency parsing (optimal parse not guaranteed)

– only produces **projective** trees

- **How does shift-reduce operate?**

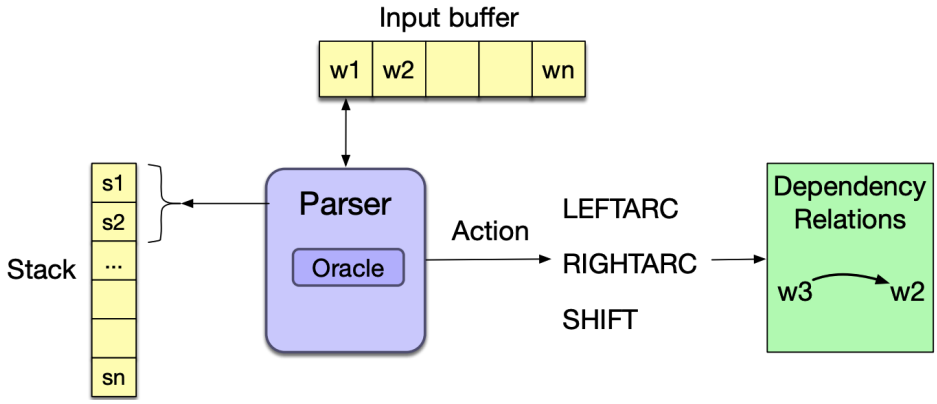


Figure 13:

– uses an **oracle** (machine learning model, a **classifier**) to predict 1 or 3 parse actions
  * `LeftArc`: create dependency relation $s_1 \rightarrow s_2$; pop s_2
  * `RightArc`: create dependency relation $s_2 \rightarrow s_1$; pop s_1
  * `Shift`: add w_1 to top of stack
– restrictions to actions:
  * `LeftArc`, `RightArc`: need at least 2 elements in stack
  * `LeftArc`: can't be applied to `Root` when its the second element of the stack (`Root` always a head; can't be dependent)

function DEPENDENCYPARSE(*words*) **returns** dependency tree

state ← {[root], [*words*], [] } ; initial configuration
while *state* **not final**
  t ← ORACLE(*state*) ; choose a transition operator to apply
  state ← APPLY(*t*, *state*) ; apply it, creating a new state
return *state*

3.4.1 Worked Example: Shift-Reduce Dependency Parsing

Step	Stack	Word List	Action	Relations
0	[root]	[Kim,saw,Sandy]		
1				
2				
3				
4				
5				

Figure 14: We try to produce a dependency parse for "Kim saw Sandy" (us acting as the oracle).

Step	Stack	Word List	Action	Relations
0	[root]	[Kim,saw,Sandy]	Shift	
1	[root,Kim]	[saw,Sandy]		
2				
3				
4				
5				

Figure 15: The stack only has the root, so only possibility is `Shift`.This adds "Kim" to the stack.

Step	Stack	Word List	Action	Relations
0	[root]	[Kim,saw,Sandy]	Shift	
1	[root,Kim]	[saw,Sandy]	Shift	
2	[root,Kim,saw]	[Sandy]		
3				
4				
5				

Figure 16: Again, can only `Shift` (if we applied `RightArc`, we'd just get that "Kim" is the head word - which it isn't).
Hence, "saw" is added to the stack.

Step	Stack	Word List	Action	Relations
0	[root]	[Kim,saw,Sandy]	Shift	
1	[root,Kim]	[saw,Sandy]	Shift	
2	[root,Kim,saw]	[Sandy]	LeftArc	nsubj(saw,Kim)
3	[root,saw]	[Sandy]		
4				
5				

Figure 17: Since "saw" is the head word, the oracle should create a dependency $saw \rightarrow Kim$ by applying `LeftArc`. This then pops "Kim".

Step	Stack	Word List	Action	Relations
0	[root]	[Kim,saw,Sandy]	Shift	
1	[root,Kim]	[saw,Sandy]	Shift	
2	[root,Kim,saw]	[Sandy]	LeftArc	nsubj(saw,Kim)
3	[root,saw]	[Sandy]	Shift	
4	[root,saw,Sandy]	[]		
5				

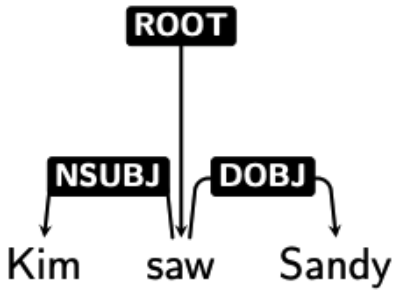
Figure 18: As above, can only `Shift`, and add "Sandy" to the stack.

Step	Stack	Word List	Action	Relations
0	[root]	[Kim,saw,Sandy]	Shift	
1	[root,Kim]	[saw,Sandy]	Shift	
2	[root,Kim,saw]	[Sandy]	LeftArc	nsubj(saw,Kim)
3	[root,saw]	[Sandy]	Shift	
4	[root,saw,Sandy]	[]	RightArc	dobj(saw,Sandy)
5	[root,saw]	[]		

Figure 19: "saw" is the head word, so apply `RightArc` to produce $saw \rightarrow Sandy$, and pop "Sandy".

Step	Stack	Word List	Action	Relations
0	[root]	[Kim,saw,Sandy]	Shift	
1	[root,Kim]	[saw,Sandy]	Shift	
2	[root,Kim,saw]	[Sandy]	LeftArc	nsubj(saw,Kim)
3	[root,saw]	[Sandy]	Shift	
4	[root,saw,Sandy]	[]	RightArc	dobj(saw,Sandy)
5	[root,saw]	[]	RightArc	root→saw
6	[root]	[]		

Figure 20: Nothing left in the word list, and "saw" is the head word, so apply a final `RightArc` to produce $root \rightarrow saw$, and pop "saw".
This completes the parse.



3.5 Additional Dependency Parsing Methods

- **Can CYK be applied to dependency parsing?**
  - can be adapted:
    - naive: $\mathcal{O}(Gn^5)$
    - Eisner algorithm: $\mathcal{O}(Gn^3)$
- **What is graph based parsing?**
  - construct **fully connected digraph**
  - assign a **score** to each edge
  - use **maximum spanning tree** to find dependency tree with highgest score
  - $\mathcal{O}(n^2)$
- **How do the different dependency parsers compare?**
  1. **Conversion**:
     - constituent-parses lead to accurate dependency parses

  - slower than direct dependency parsing
  - treebanks may only have dependency form
2. **Transition**:
  - no optimality guarantee
  - linear
  - projective only (unless tweaked)
3. **Graph**:
  - optimal
  - quadratic
  - not necessarily projective

- **What considerations should be used when picking a parser?**
  - **constituent** or **dependency**
  - how **runtime/memory** efficient is it
  - can you obtain partial parses (i.e as the sentence is inputted), or do you parse sentence at once
  - can you retrain the system

4 Semantics from Syntax

4.1 Formalising Meaning

- **What are semantics?**
  - the study of meaning of words
- **What is semantic parsing?**
  - process of producing **meaning representations**
  - for example, when reading a menu in a restaurant, we need to understand meaning (what is food? how is food cooked? what is the result of applying a recipe?)
  - **semantic parsing** takes sentences, and produces a representation of its meaning
- **What exactly is meaning?**
  - in NLP, focus on whether a computer can **behave** as an understanding entity
    - can it maintain a dialogue (**Turing test)**
    - can it perform effective **machine translation**
    - how well does it engage in **question answering**
- **What considerations should be made when defining a question answering agent?**
  - agent which **answers** question using **natural language**, given a **knowledge base** and **English text**
  - needs to understand:
    - **sentential semantics**: meaning derived from **word combinations**; understand the **whos, whats wheres, whys** and **whens**
      - understands impact of word order on meaning:

        $$\textit{“John loves Mary.”} \neq \textit{“Mary loves John.”}$$

        If someone asks “Who loves Mary?”, this should answer “John”.
      - understands **inference**:

        $$\textit{“John loves Mary.”} \implies \textit{“Someone loves Mary.”}$$

        If someone asks “Is Mary loved?”, this should answer “yes”.

    - **lexical semantics**: meaning derived from **word meaning**
        - if asked "Is snow white?", system should know what "snow" means in our world, that "white" is a colour, what "colour" is, etc ...
        - understands that "John" represents a man, that the act of "love" is very dissimilar to the act of "hate", but similar to that of "like"

- **How are syntax and semantics related?**
    - **sentence meaning** is very related to **sentence syntax**
        - consider

          "John loves Mary." *"Mary loves John."*
        - **syntax**: different; in one, "John" is the subject; in other, "John" is the object
        - **semantics**: in one, "John" is loving; in other "John" is loved
    - **syntactic ambiguity** and **semantic ambiguity** are related
        - intended meaning used to disambiguate **syntactic ambiguity** when annotating parse trees
        - **syntactic ambiguity** often ledas to **semantic ambiguity**
        - however, lack of **syntactic ambiguity** doesn't mean lack of **semantic ambiguity**:
            - *word sense*: "bank" (as financial institution or side of a river) can form a **grammatical** sentence, whilst being meaningless semantically:

              "I placed my money in a bank (of a river)"
            - *semantic scope*: for example:

              "Every man loves a woman"

              is syntactically **unambiguous**, but **semantically** it has 2 meanings (every man loves the same woman, or every man has a unique woman who they love)
            - *anaphoric expression*: for example:

              "I love this woman. She makes me happy"

              we need to understand that "she" refers to the woman who is loved

4.2 First Order Logic and Semantic Representations

- **What are desired properties of semantic representations?**
    - we focus on representations of **literal meaning** (as opposed to **metaphorical meaning**)
    - any representation of **meaning** should satisfy:
        1. **Unambiguity**: a sentence must be represented **uniquely** (i.e the different interpretations of "I made her duck" should have **different semantic representations)**
        2. **Automated Inference**: derive true conclusions from **meaning representations**
        3. **Verifiability**: truth value of a sentence must be **verifiable**, given a world model
        4. **Canonical Forms**: **distinct sentences** with the **same meaning** should be mapped to the **same meaning representation** (i.e "John loves Mary", "Mary is loved by John", "John is in love with Mary" should all have the same **semantic representation**)
        5. **Expressiveness**: able to handle diverse subject matters, from any natural language

- **Why is prepositional logic not apt for semantic representations?**

    1. **Prepositions** encode the meaning of a whole phrase, so no understanding of the composition of the preposition (i.e if P = "Fred ate rice", the variable P has no understanding of what "Fred", "ate" or "rice" indicate)

2. Lack of **universal/existential quantifier** (can't express simple notions, such as "everyone eats rice", nor derive inferential knowledge):

$$\textit{"Everyone eats rice"} \implies \textit{"Someone eats rice"}$$

3. Hard to produce **canonical forms**
4. Not **expressive**

- **Why is FOL a suitable knowledge representation?**
  - *For a recap of FOL, this link is fantastic.*
  - it satisfies all the requirements for producing a powerful **semantic representation**
- **What are Davidsonian Semantics?**
  - an extension of FOL to include **events**
  - particularly useful for **temporal relations** and **tense logic**
    - the action of "eating" can be undertaken at many temporal stages (i.e "I ate", "I'm eating", "I will eat")
    - we can represent the now using n; then we can write:

$$\textit{"Fred ate rice"} \implies \exists e(eat(e, fred, rice) \ \wedge \ e \prec n)$$

      to say that the action of eating occurred before the current time
    - we can apply **modifiers** to the event variable:

$$\begin{aligned} &\textit{"Fred ate rice with a fork at midnight"} \\ \implies &\exists e(eat(e, fred, rice) \ \wedge \ e \prec n \ \wedge \ \exists x(with(e, x) \ \wedge \ fork(x)) \ \wedge \ at(e, midnight))) \end{aligned}$$

      We are adding extra information to the information regarding the event, such as it happening during midnight, and involving the use of a fork. Notice, **wedge elimination** means that this expression implies "Fred ate rice".

4.3 Lambda Calculus

- **What is the Lambda Calculus?**
  - the basis of the best programming language: Haskell
  - a **model of computation**, based on function abstraction and application using variable binding and substitution
  - used to represent simple addition, to produce **Turing machines**
  - particularly useful to **abstract** FOL expressions, which is used in **semantic analysis**
- **How is Lambda Calculus used in FOL?**
  - allow us to substitute values into **free variables** of FOL expressions:

$$\lambda x.\phi(a) = \phi[x/a]$$

  - the **function** $\lambda x.\phi$ is an example of **lambda abstraction**
  - the expression $\phi[x/a]$ is an example of **beta reduction**
  - for example:

$$\begin{aligned} &\lambda y.\lambda x.(\exists e(eat(e, x, y) \wedge e \prec n))(rice) \\ = \ &\lambda x.(\exists e(eat(e, x, rice) \wedge e \prec n)) \end{aligned}$$

  - we can even use **lambda expressions** as **arguments**:

$$\begin{aligned} &\lambda P.(P(fred))(\lambda x.(\exists e(eat(e, x, rice) \wedge e \prec n)) \\ = \ &\lambda x.(\exists e(eat(e, x, rice) \wedge e \prec n))(fred) \\ = \ &\exists e(eat(e, fred, rice) \wedge e \prec n) \end{aligned}$$

4.4 Compositional Semantics: Building Semantics from Syntax

- **What is compositionality?**
  - solid theory of meaning:

> *"The meaning of a complex expression is a function of the meaning of its parts, alongside the rules used to combine them."*

  - this means that **semantic parsing** can be built by using **syntactic parsing**: we just need
    - **Lexical Meanings**: associate a FOL expression to each word in lexicon
    - **Composition Rules**: augmenting CFGs with ways of **composing** the FOL expressions
- **How can semantic attachment be used to augment CFGs?**
  - in standard CFG, we have rules:

$$A \rightarrow \alpha_1 \ \alpha_2 \ \ldots \ \alpha_n$$

  - we can denote the **semantic meaning** of a non-terminal via:

$$\alpha_i.Sem$$

  - the **semantic meaning** for the rule above can be described as a **function** of the **semantic meanings** of its derived non-terminals:

$$A.Sem = f(\alpha_1.Sem, \ldots, \alpha_n.Sem)$$

  - in some cases, this can be summarised in the following way. Consider the production:

$$S \rightarrow NP \ VP$$

    This can have a **semantic meaning**:

$$S.Sem = VP.Sem(NP.Sem)$$

    that is: $VP.Sem$ is a **functor**, which takes as argument $NP.Sem$ (in particular, we can think of $VP.Sem$ as a **lambda expression**, and $NP.Sem$ as its argument)
  - if we have **unary** rules, the semantic meanings gets "passed up":

$$MassN \rightarrow rice \quad \{rice.Sem = rice\} \implies MassN.Sem = rice$$

- **How can syntactic parsing lead to semantic parsing?**
  - we can construct a **syntactic parse**, and then apply the **semantic rules** of the augment CFG, as described above:

S → NP VP *VP.Sem(NP.Sem)*	**(Sentences)**
NP → MassN *MassN.Sem* | PropN *PropN.Sem*	**(Noun phrases)**
VP → Vi *Vi.Sem* | Vt NP *Vt.Sem(NP.Sem)*	**(Verb phrases)**
PropN → Fred *fred* | Jo *jo*. . .	**(Proper nouns)**
MassN → rice *rice* | wood *wood* . . .	**(Mass nouns)**
Vi → talked $\lambda x \exists e(talk(e,x) \wedge e \prec n)$ | . . .	**(Intransitive verbs)**
Vt → ate $\lambda y \lambda x . \exists e(eat(e,x,y) \wedge e \prec n)$ | . . .	**(Transitive verbs)**

Figure 21: Lambda calculus lends itself very well for composing FOL expressions, according to some augmented CFG.

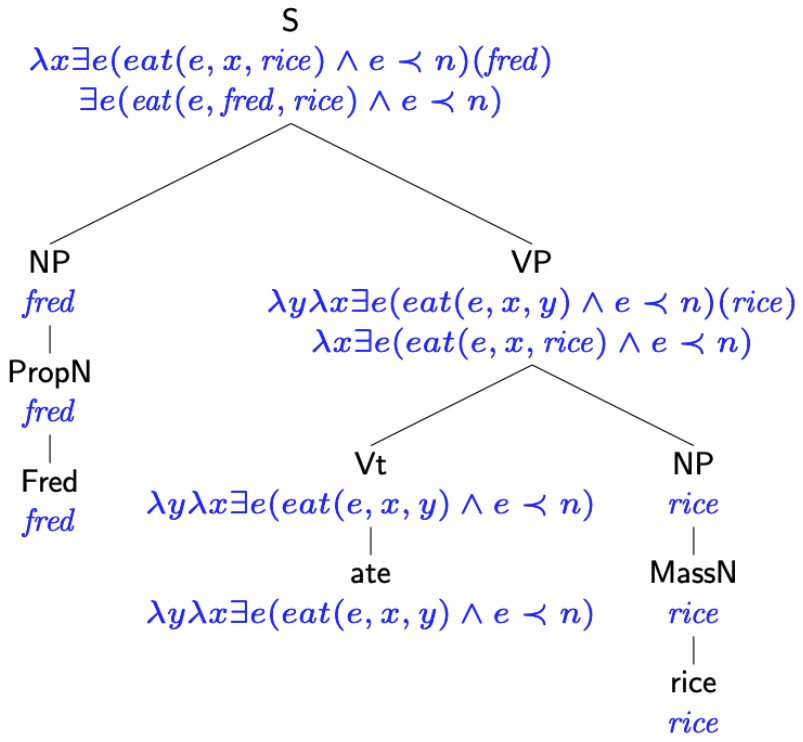


Figure 22: From the syntactic tree of "Fred ate rice", we can obtain a semantic expression, based on Lambda Calculus.

4.4.1 Issues with Naive CFG Augmentation

Every man ate rice: $\forall x(man(x) \rightarrow \exists e(eat(e, x, rice) \wedge e \prec n))$

Breaking it down:

- What is the meaning of *Every man* anyway?
  $\forall x(man(x) \rightarrow Q(x))$
- If so, the subject NP needs to be:
  $\lambda Q \forall x(man(x) \rightarrow Q(x))$
- But in our grammar we had the VP as the functor:
  S $\rightarrow$ NP VP *VP.Sem(NP.Sem)*
- $\lambda z \exists e(eat(e, z, rice) \wedge e \prec n)(\lambda Q \forall x(man(x) \rightarrow Q(x)))$ becomes
  $\lambda z \exists e(eat(e, \lambda Q \forall x(man(x) \rightarrow Q(x)), rice) \wedge e \prec n)$
- That's not even syntactically well-formed!!

Figure 23: The problem with the above grammar is it can create invalid lambda expressions (the second argument of eat should be a variable or symbol, not a lambda expression).

Make NP the functor and VP the argument.

S → NP VP *NP.Sem(VP.Sem)*

$\lambda Q \forall x(man(x) \rightarrow Q(x))(\lambda z \exists e(eat(e, z, rice) \wedge e \prec n))$
$\forall x(man(x) \rightarrow \lambda z \exists e(eat(e, z, rice) \wedge e \prec n))(x))$
$\forall x(man(x) \rightarrow \exists e(eat(e, x, rice) \wedge e \prec n))$

But this means NPs must all look like this: $\lambda P.P(x)$.
Fred $\mapsto \lambda P.P(fred)$ etc.

Figure 24: To solve the above, we can modify how we define the semantic meaning of S, by applying $NP.Sem$ as a functor (instead of as an argument, as above). For this to produce well-formed expressions, we modify how noun phrases are defined (so now simple nouns, like "Fred", are defined as properties).

ate: $\lambda y \lambda z \exists e(eat(e, z, y) \wedge e \prec n)$

every grape: $\lambda Q \forall x(grape(x) \rightarrow Q(x))$

NP.Sem(Vt.Sem) is ill formed!

$\lambda Q \forall x(grape(x) \rightarrow Q(x))(\lambda y \lambda z \exists e(eat(e, z, y) \wedge \prec n))$ becomes
$\forall x(grape(x) \rightarrow \lambda y \lambda z \exists e(eat(e, z, y) \wedge e \prec n)(x)$ becomes
$\forall x(grape(x) \rightarrow \lambda z \exists e(eat(e, z, x) \wedge e \prec n))$

It should be: $\lambda z \forall x(grape(x) \rightarrow \exists e(eat(e, z, x) \wedge e \prec n))$

Figure 25: The above modification still produces a problem: "every grape is eaten" requires that λz is on the outside of the expression, but the current grammar has it as part of the implication, which is ill-formed.

VP → Vt NP *Vt.Sem(NP.Sem)*
Vt → ate $\lambda R.\lambda z.R(\lambda y.\exists e(eat(e, z, y) \wedge e \prec n))$

ate every grape:

$\lambda R.\lambda z.R(\lambda y.\exists e(eat(e, z, y) \wedge e \prec n))(\lambda Q \forall x(grape(x) \rightarrow Q(x)))$ becomes
$\lambda z \lambda Q \forall x(grape(x) \rightarrow Q(x))(\lambda y.\exists e(eat(e, z, y) \wedge e \prec n))$ becomes
$\lambda z \forall x(grape(x) \rightarrow \lambda y.\exists e(eat(e, z, y) \wedge e \prec n)(x))$ becomes
$\lambda z \forall x(grape(x) \rightarrow \exists e(eat(e, z, x)))$

Figure 26: We can modify how transitive verbs are applied semantically, which solves the issue above.

Grammar Refined! (Changes in purple)

S →	NP VP *NP.Sem(VP.Sem)*	**(Sentences)**
NP →	MassN *MassN.Sem* | PropN *PropN.Sem* | Det N *Det.Sem(N.Sem)*	**(Noun phrases)**
VP →	Vi *Vi.Sem* | Vt NP *NP.Sem(Vt.Sem)*	**(Verb phrases)**
PropN →	Fred $\lambda P.P(fred)$ | . . .	**(Proper nouns)**
MassN →	rice $\lambda P.P(rice)$ | . . .	**(Mass nouns)**
Vi →	talked $\lambda x \exists e(talk(e,x) \wedge e \prec n)$ | . . .	**(Intransitive verbs)**
Vt →	ate $\lambda R.\lambda z.R(\lambda y.\exists e(eat(e,z,y) \wedge e \prec n))$	**(Transitive verbs)**
N →	man $\lambda x.man(x)$	**(Count Nouns)**
Det →	a $\lambda P \lambda Q \exists x(P(x) \wedge Q(x))$ | every $\lambda Q \lambda Q \exists x(P(x) \rightarrow Q(x))$	**(Determiners)**

Figure 27: All the above allows us to define a more robust grammar.

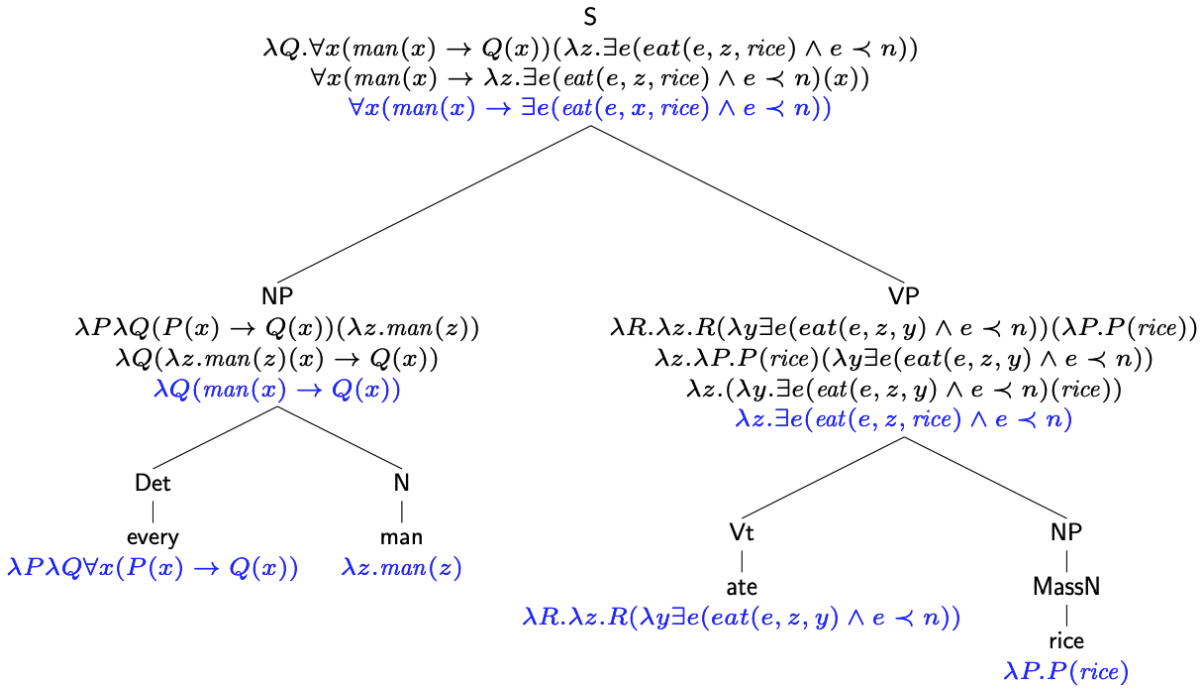


Figure 28: The derivation for "Every man ate rice" now looks like.

4.5 Ambiguity and Underspecification

- **Do semantics attached grammars deal with semantic ambiguity?**
  - can't deal with **scope ambiguity** (i.e "Every man loves a woman" is syntactically unambiguous, but has 2 semantic interpretations)
  - it could be useful to derive grammars which **encode** both semantic representations in a **general form**
- **How can scope ambiguity be handled?**
  1. **Enumerate All Interpretations**: unfeasible - computations grow exponentially with scope operators ("Every student

Every student at **some** university has access to **a** laptop.

1. Not necessarily same laptop, not necessarily same university

$\forall x(stud(x) \wedge \exists y(univ(y) \wedge at(x,y)) \rightarrow \exists z(laptop(z) \wedge have\_access(x,z)))$

2. Same laptop, not necessarily same university

$\exists z(laptop(z) \wedge \forall x(stud(x) \wedge \exists y(univ(y) \wedge at(x,y)) \rightarrow have\_access(x,z)))$

3. Not necessarily same laptop, same university

$\exists y(univ(y) \wedge \forall x((stud(x) \wedge at(x,y)) \rightarrow \exists z(laptop(z) \wedge have\_access(x,z))))$

4. Same university, same laptop $\exists y(univ(y) \wedge \exists z(laptop(z) \wedge \forall x((stud(x) \wedge at(x,y)) \rightarrow have\_access(x,z))))$

5. Same laptop, same university $\exists z(laptop(z) \wedge \exists y(univ(y) \wedge \forall x((stud(x) \wedge at(x,y)) \rightarrow have\_access(x,z))))$

where 4 & 5 are equivalent

Every student at **some** university does **not** have access to **a** computer.
$\rightarrow$ 18 interpretations

2. **Semantic Underspecification**: build FOL formulae which **underspecifies** the semantic scopes of quantifiers (since the quantifiers define how the FOL "bits" are combined together, by underspecifying them, we don't restrict ourselves to any one interpretation)

- **How do we apply underspecification in practice?**
  - as an example, we can consider "Every man loves a woman":

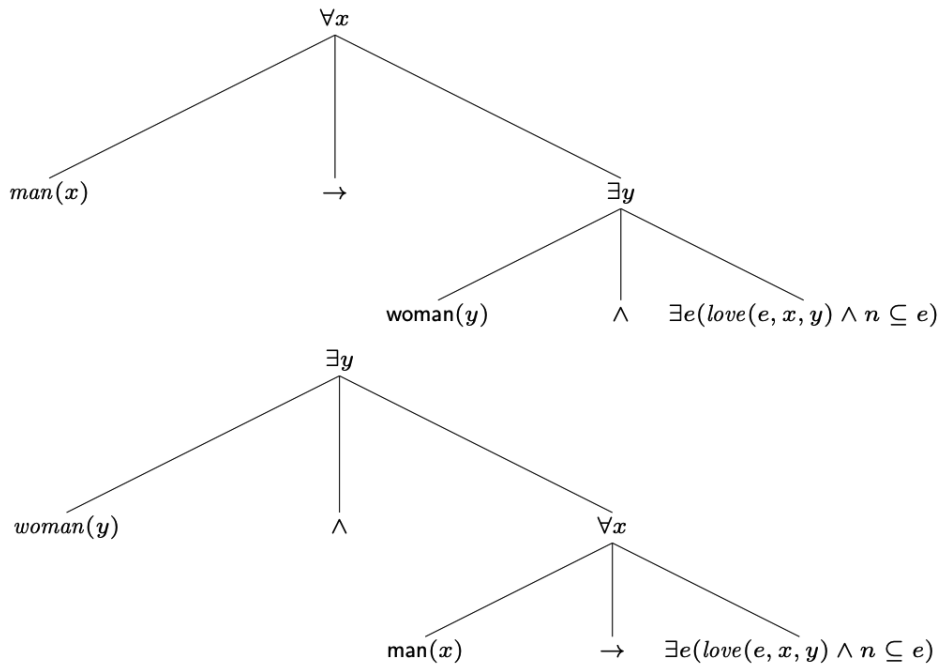


Figure 29: Both interpretations have similar trees; the presence of the scopes is what defines what we have at the terminals, and thus, defines the different meanings that can be derived.

  - we can **label** each node of the tree $l_1, \ldots, l_n$
  - then, provide constraints on the FOL "bits" which can appear at each label

$$
\begin{aligned}
l_1 &: \quad \forall x(h_2 \rightarrow h_3) \\
l_2 &: \quad man(x) \\
l_3 &: \quad love(e, x, y) \\
l_4 &: \quad \exists y(h_4 \wedge h_5) \\
l_5 &: \quad woman(y) \\
h_2 &=_q l_2, h_4 =_q l_5
\end{aligned}
$$

Figure 30: Here, we have set constraints, such that we have:

$$l_1 : \forall x(man(x) \implies h_3) \qquad l_3 : \exists y(woman(y) \wedge h_5)$$

If we put l_1 first, then we can assign $h_3 = l_4$ $[\exists y(woman(y) \wedge h_5)]$ and $h_5 = l_3$ $[love(e, x, y)]$ to obtain "For every man, there is a woman whom that man loves" (so $\forall$ outscopes $\exists$).
Alternatively, by putting l_4 first, and assigning $h_5 = l_1$ $[\forall x(man(x) \implies h_3)]$ and $h_3 = l_3$ $[love(e, x, y)]$, we obtain "There exists a woman such that every man loves that woman" (so $\exists$ outscopes $\forall$).