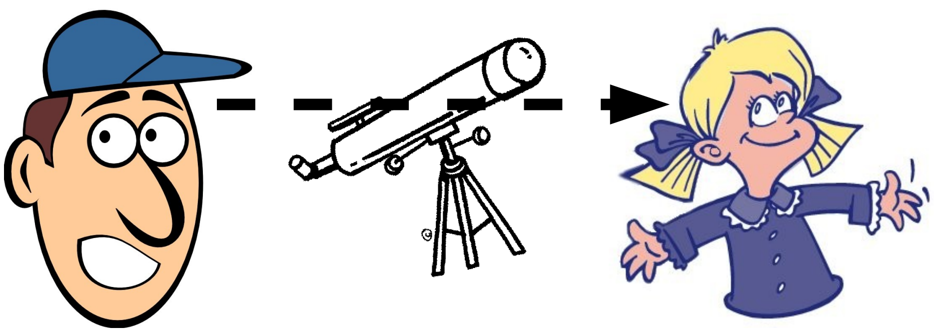
NLP Programming Tutorial 8 - Phrase Structure Parsing

Graham Neubig
Nara Institute of Science and Technology (NAIST)

Interpreting Language is Hard!

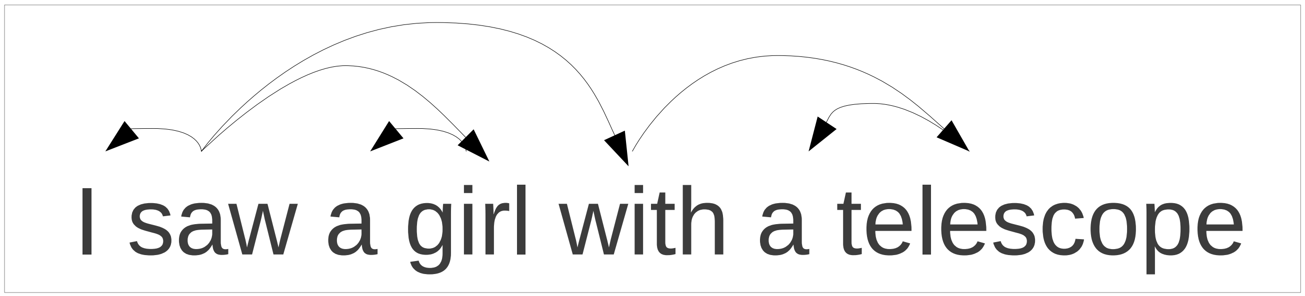
I saw a girl with a telescope



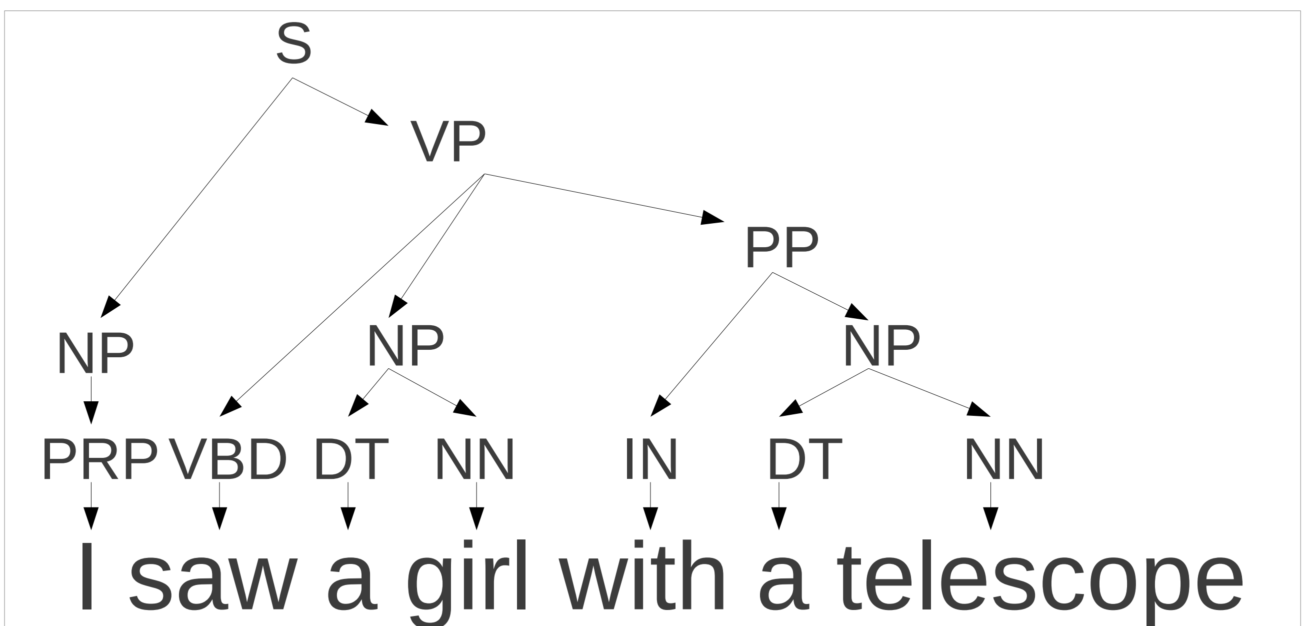
- “Parsing” resolves structural ambiguity in a formal way

Two Types of Parsing

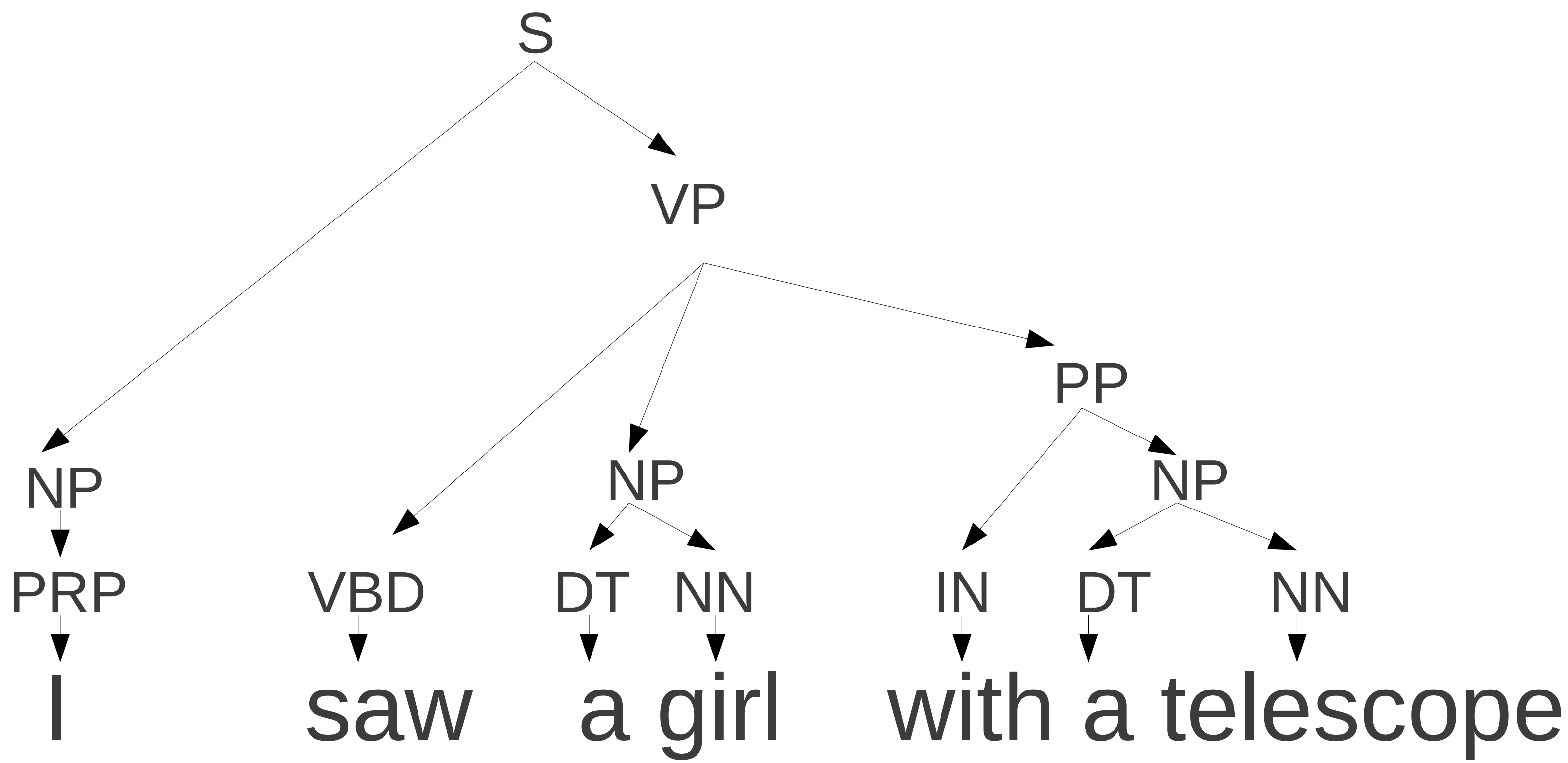
- Dependency: focuses on relations between words



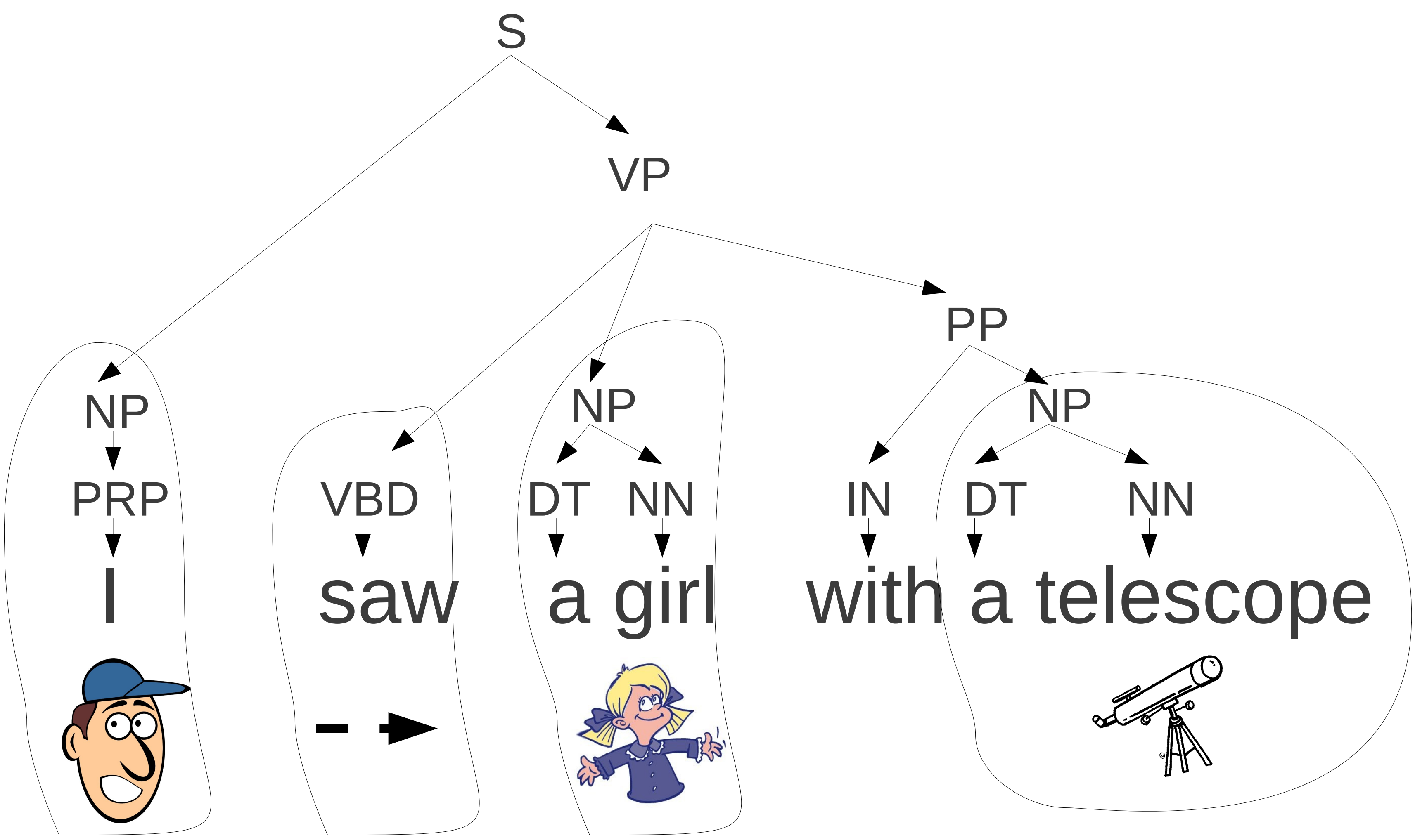
- Phrase structure: focuses on identifying phrases and their recursive structure



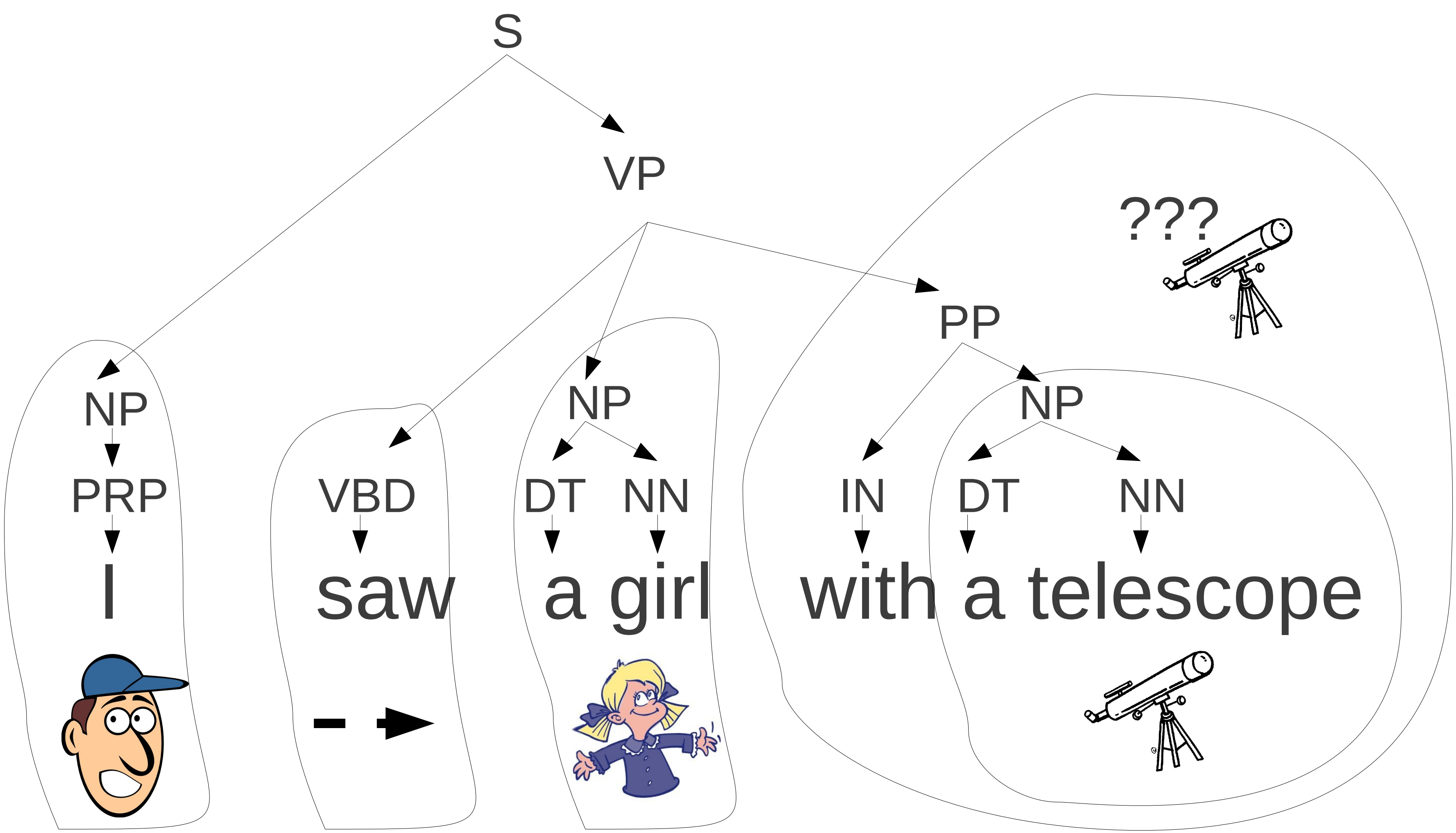
Recursive Structure?



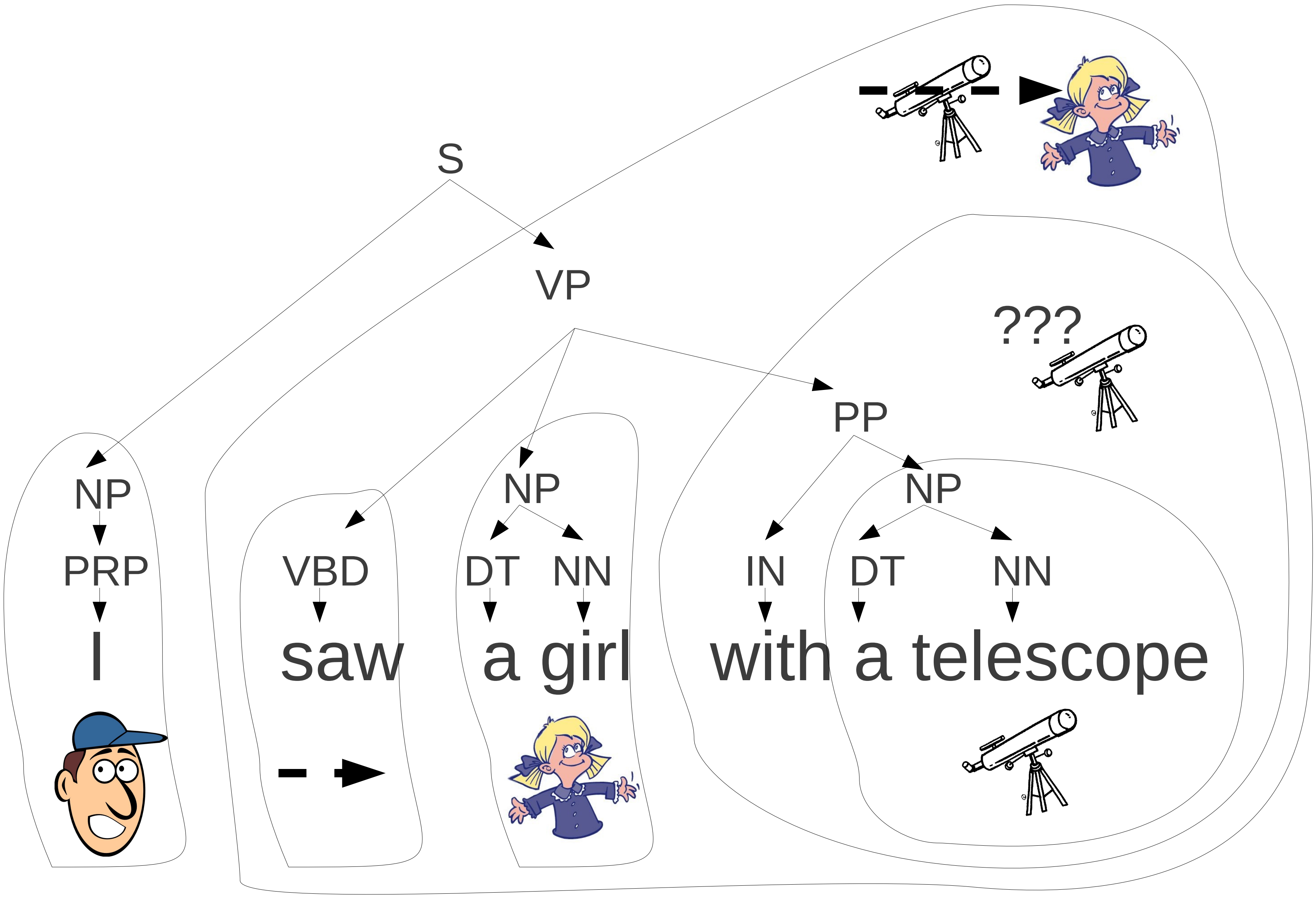
Recursive Structure?



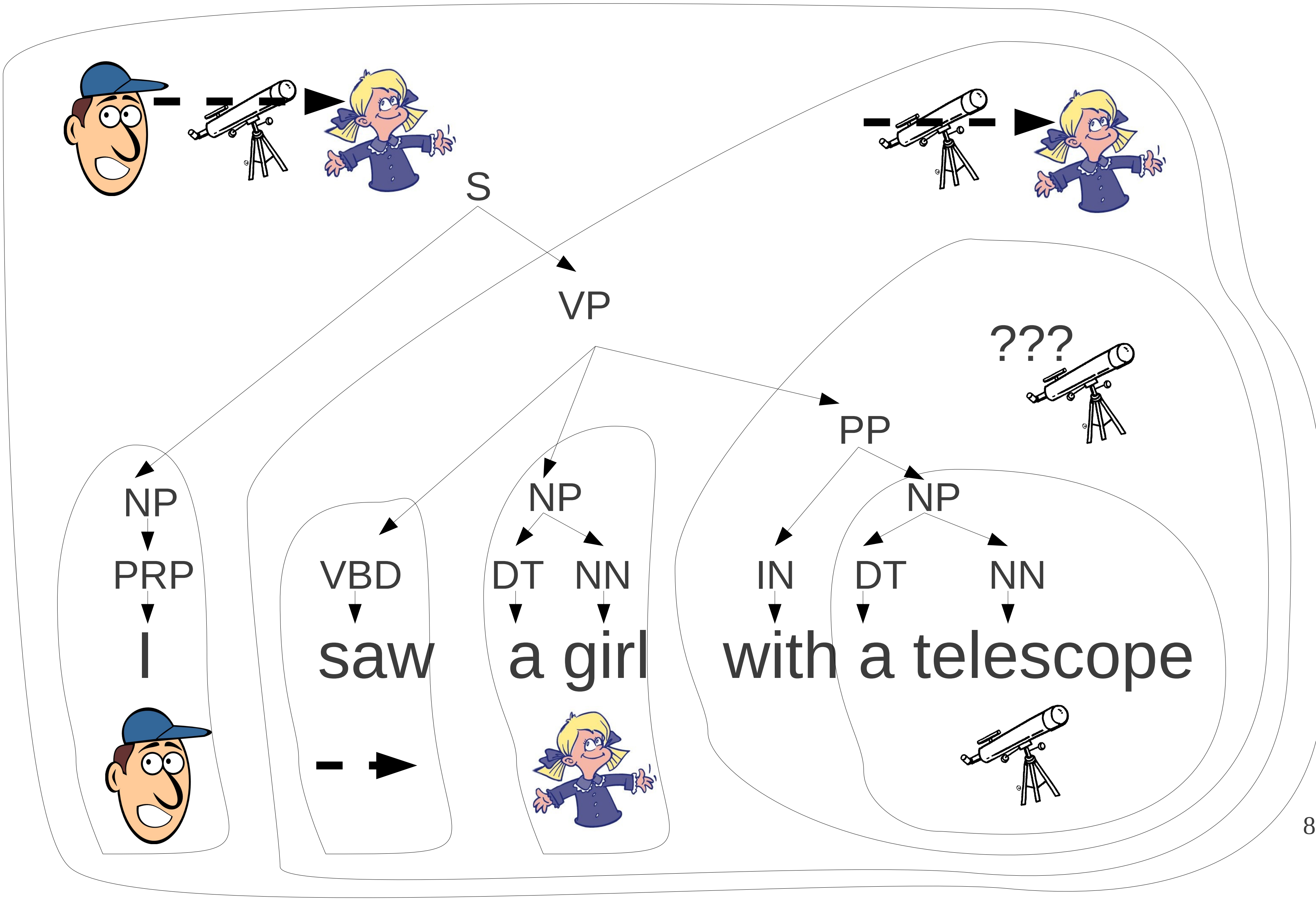
Recursive Structure?



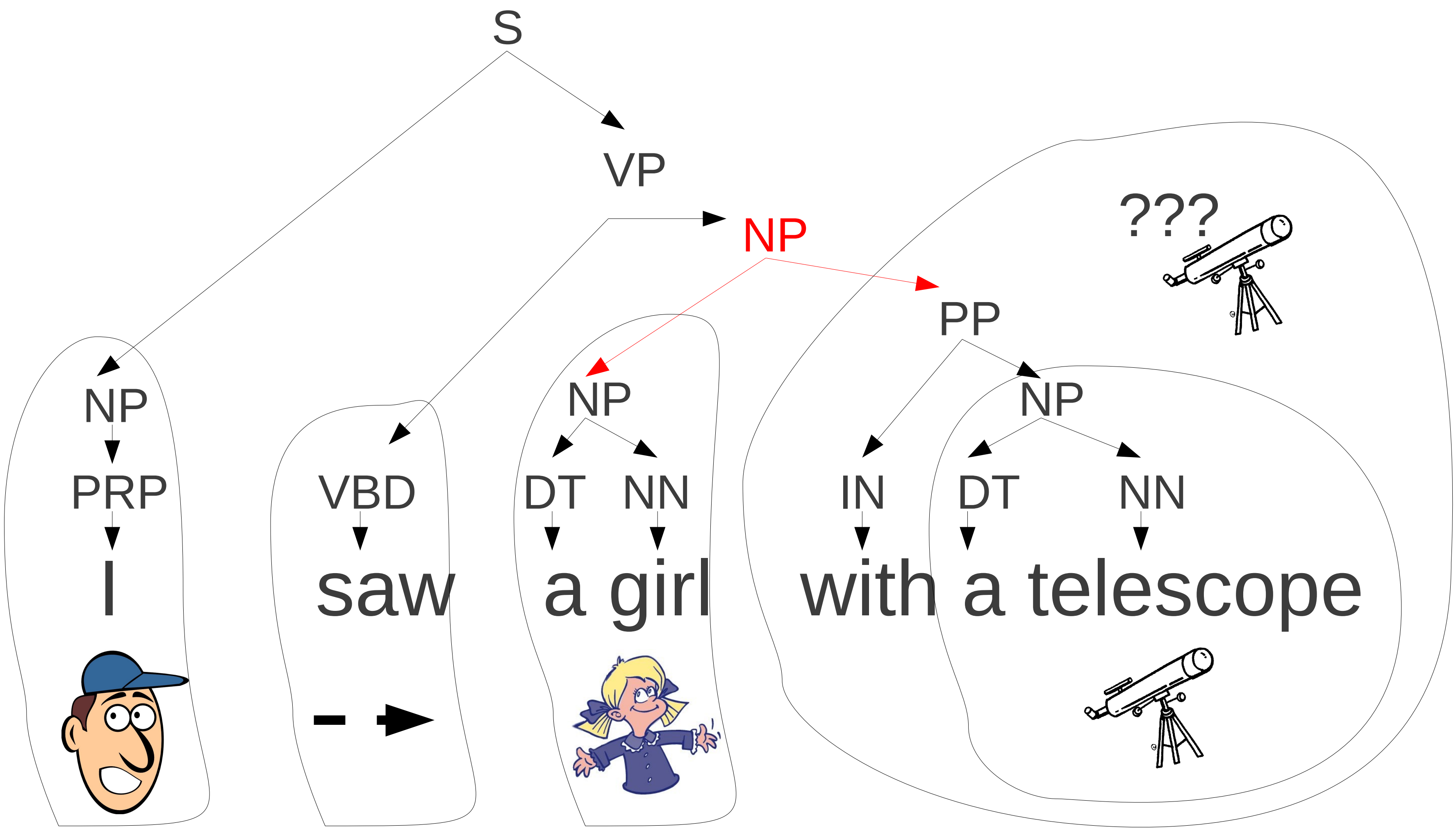
Recursive Structure?



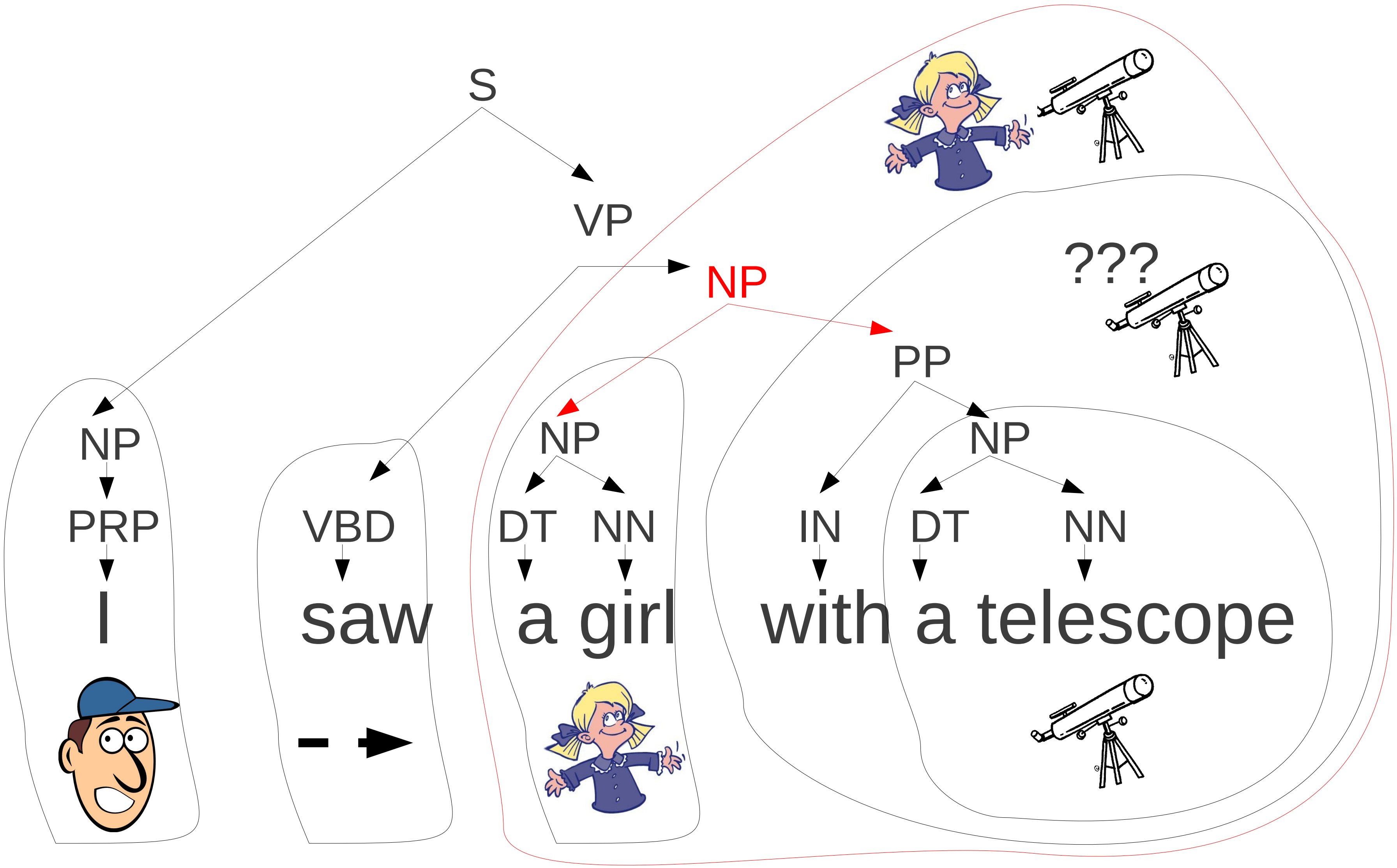
Recursive Structure?



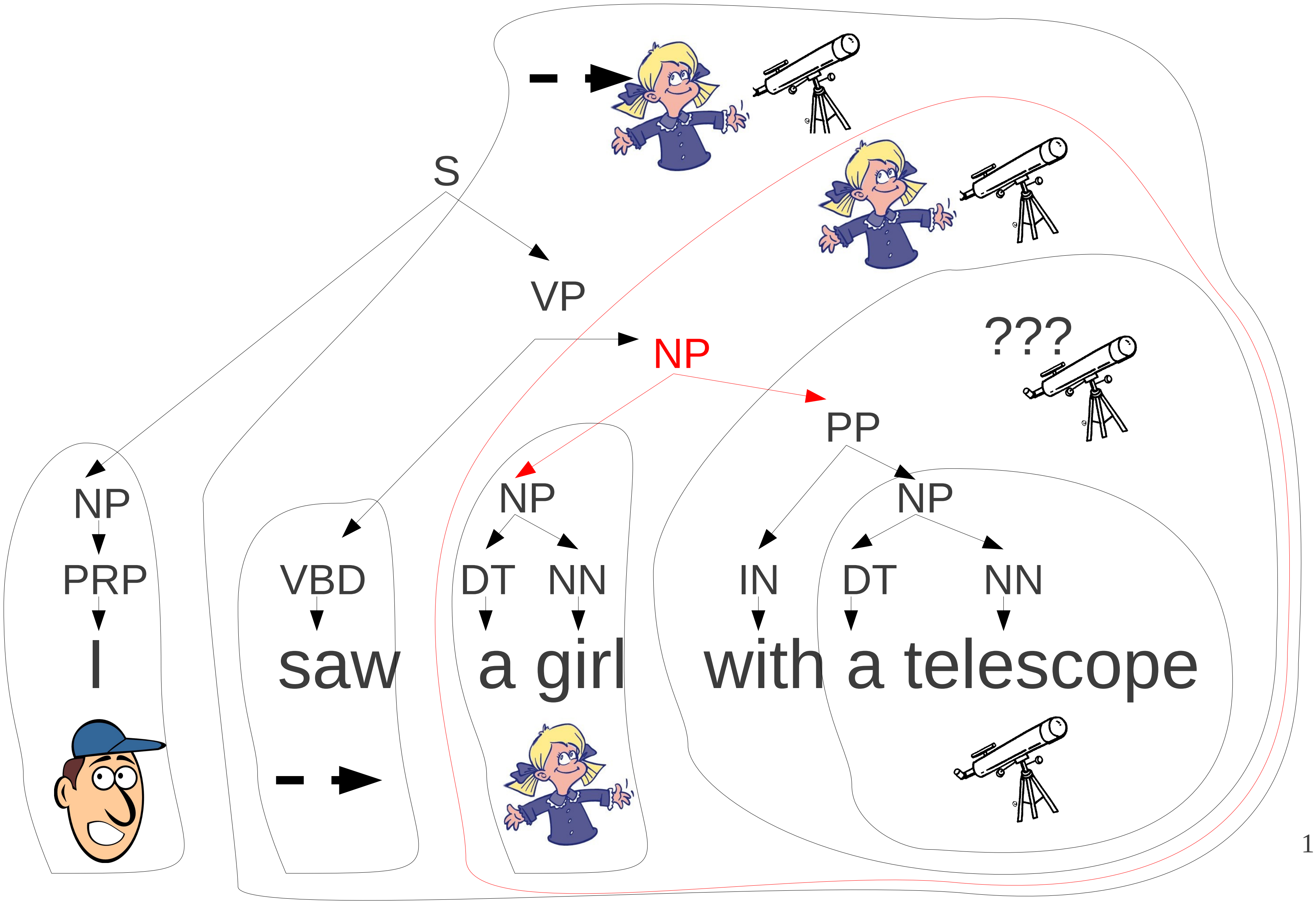
Different Structure, Different Interpretation



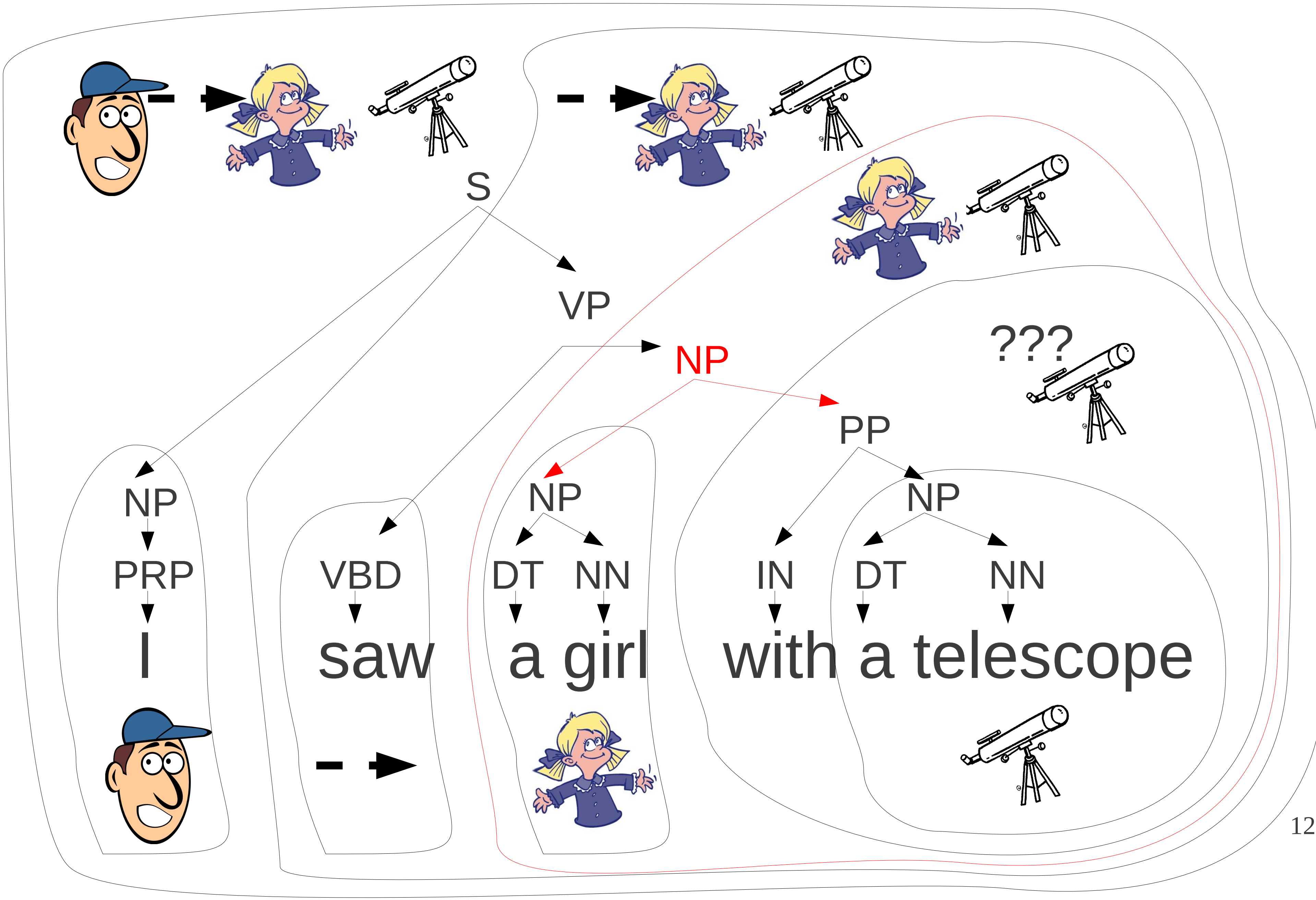
Different Structure, Different Interpretation



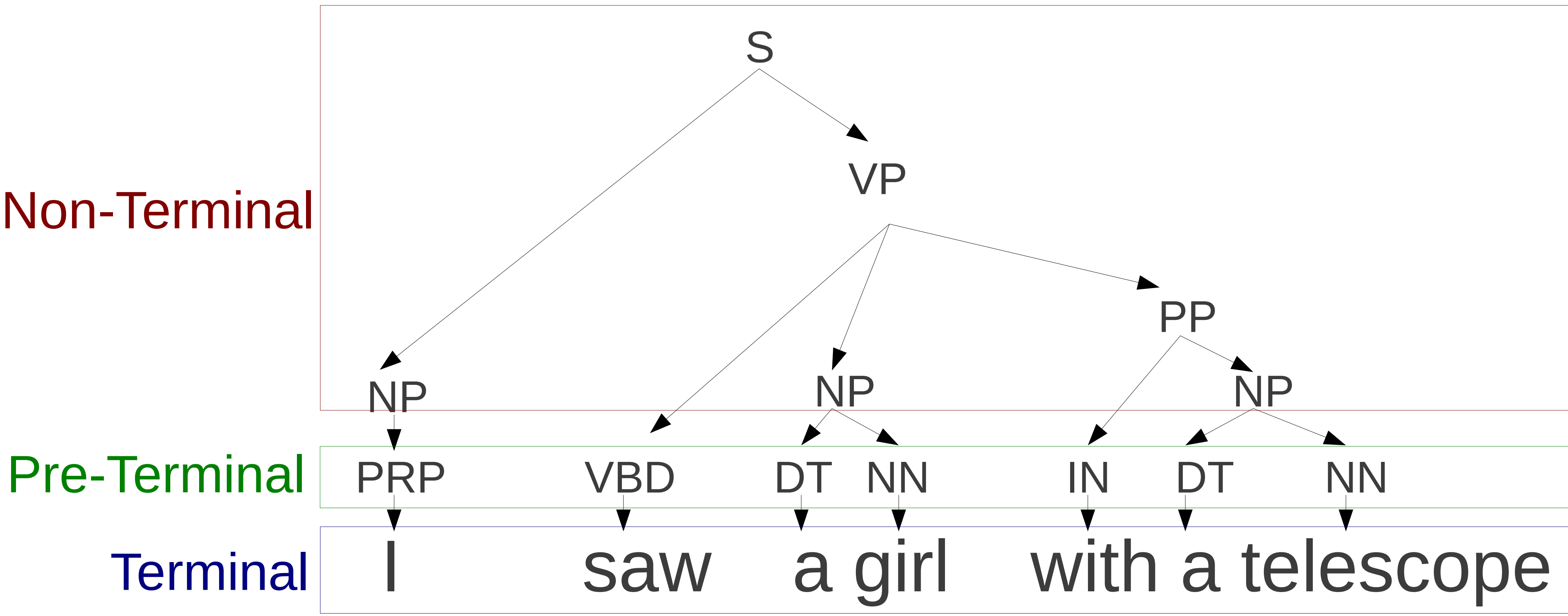
Different Structure, Different Interpretation



Different Structure, Different Interpretation

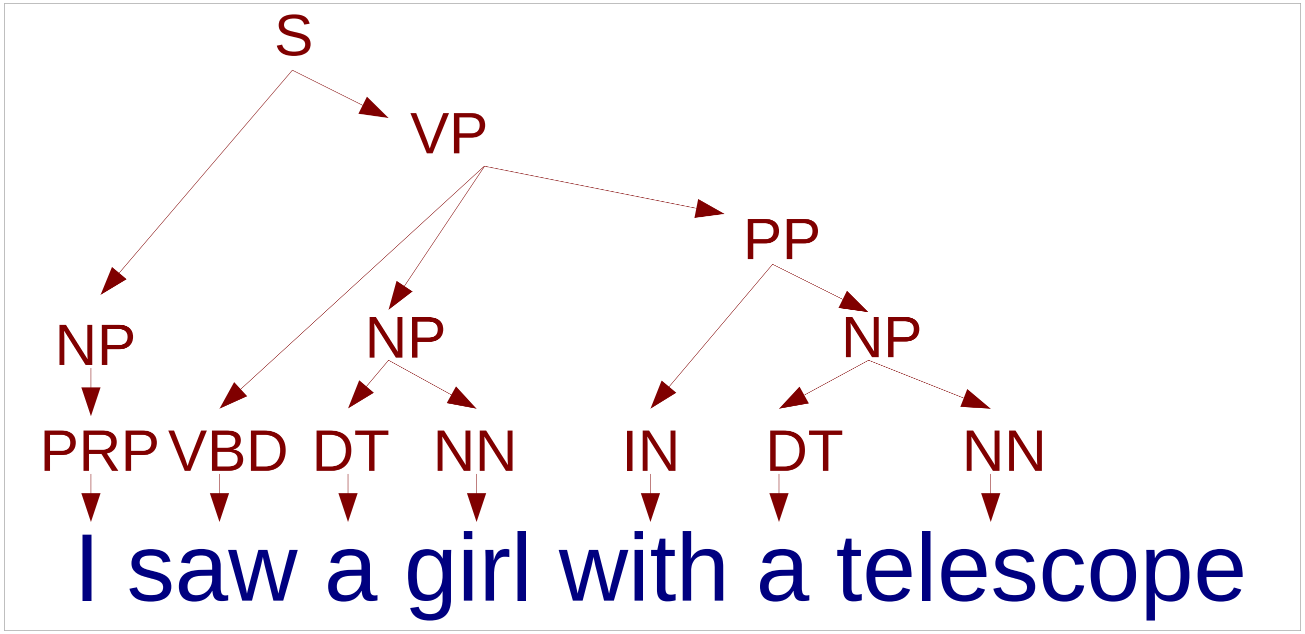


Non-Terminals, Pre-Terminals, Terminals



Parsing as a Prediction Problem

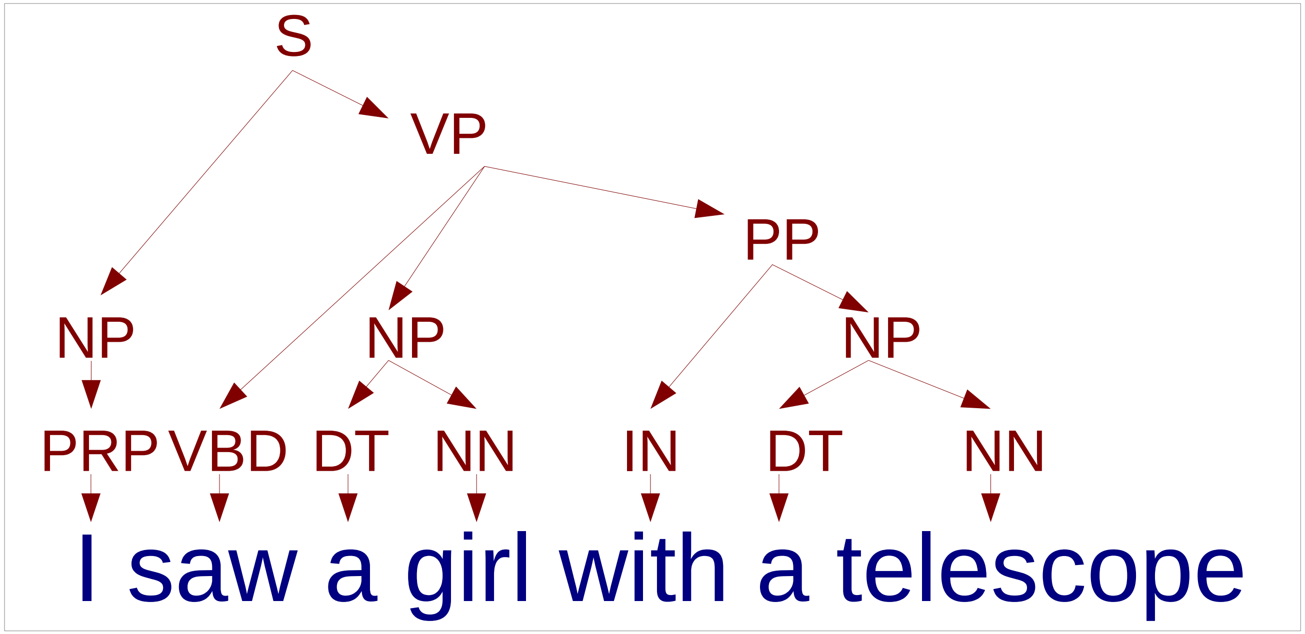
- Given a sentence X, predict its parse tree Y



- A type of “structured” prediction (similar to POS tagging, word segmentation, etc.)

Probabilistic Model for Parsing

- Given a sentence X, predict the most probable parse tree Y



$$\underset{Y}{\operatorname{argmax}} P(Y|X)$$

Probabilistic Generative Model

- We assume some probabilistic model generated the parse tree Y and sentence X jointly

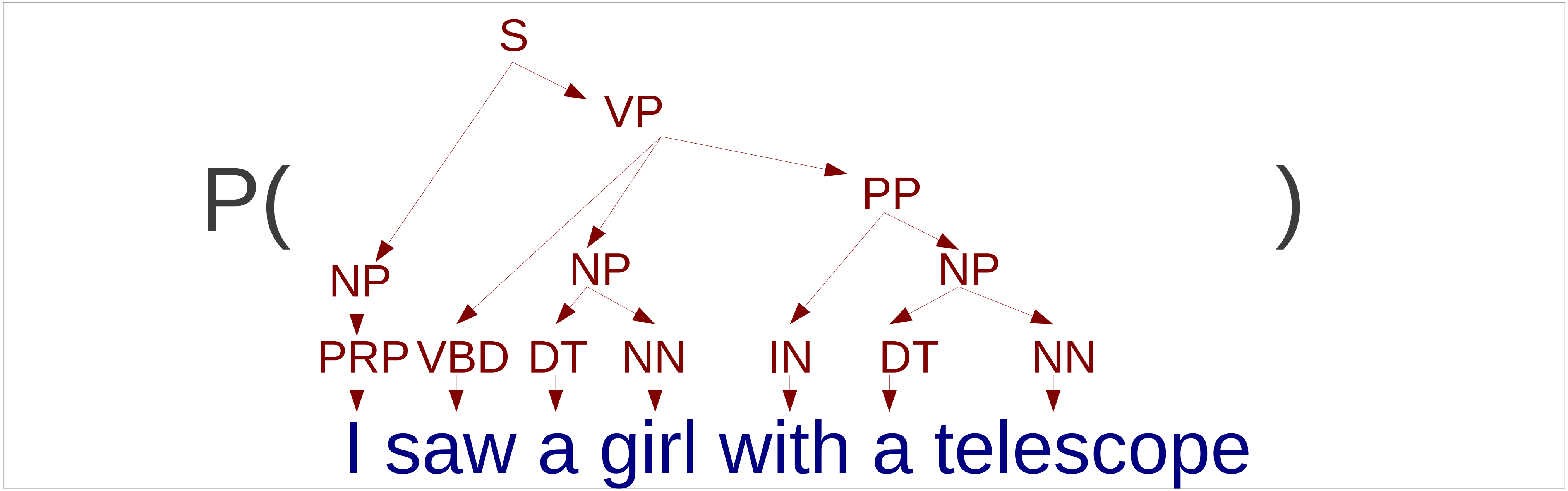
$$P(Y, X)$$

- The parse tree with highest joint probability given X also has the highest conditional probability

$$\underset{Y}{\operatorname{argmax}}\, P(Y \mid X) = \underset{Y}{\operatorname{argmax}}\, P(Y, X)$$

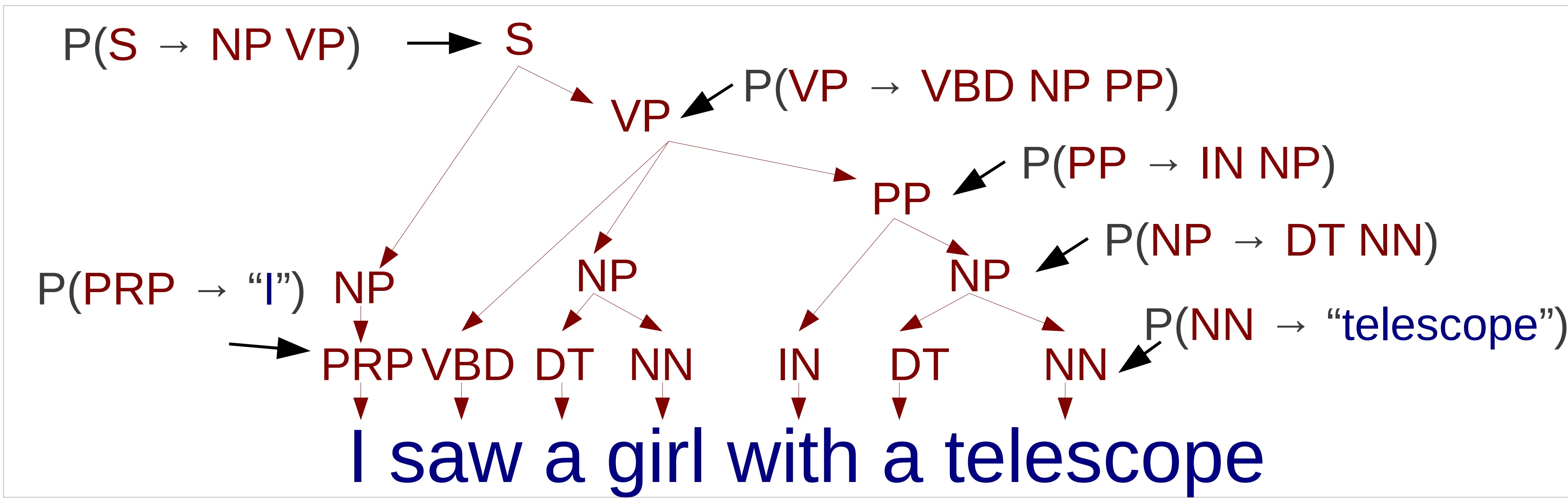
Probabilistic Context Free Grammar (PCFG)

- How do we define a joint probability for a parse tree?



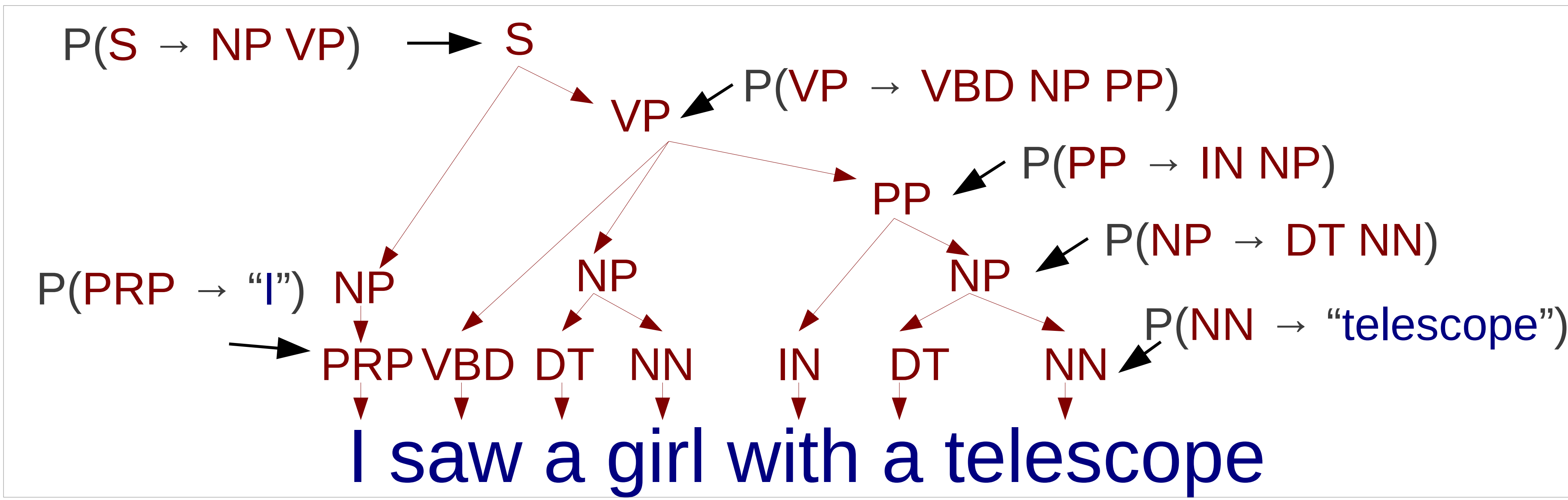
Probabilistic Context Free Grammar (PCFG)

- PCFG: Define probability for each node



Probabilistic Context Free Grammar (PCFG)

- PCFG: Define probability for each node



- Parse tree probability is product of node probabilities

P(S → NP VP) * P(NP → PRP) * P(PRP → “I”)
* P(VP → VBD NP PP) * P(VBD → “saw”) * P(NP → DT NN)
* P(DT → “a”) * P(NN → “girl”) * P(PP → IN NP) * P(IN → “with”)
* P(NP → DT NN) * P(DT → “a”) * P(NN → “telescope”)

Probabilistic Parsing

- Given this model, parsing is the algorithm to find

$$\underset{Y}{\operatorname{argmax}}\, P(Y, X)$$

- Can we use the Viterbi algorithm as we did before?

Probabilistic Parsing

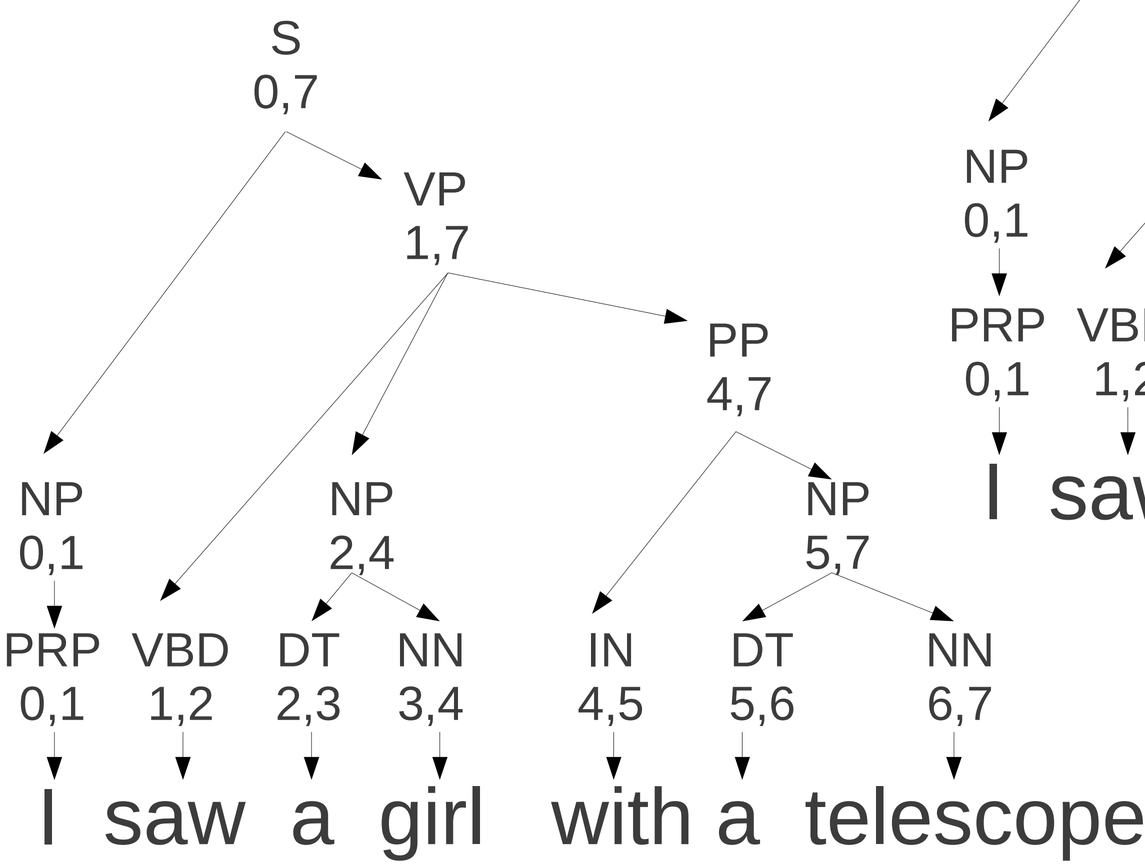
- Given this model, parsing is the algorithm to find

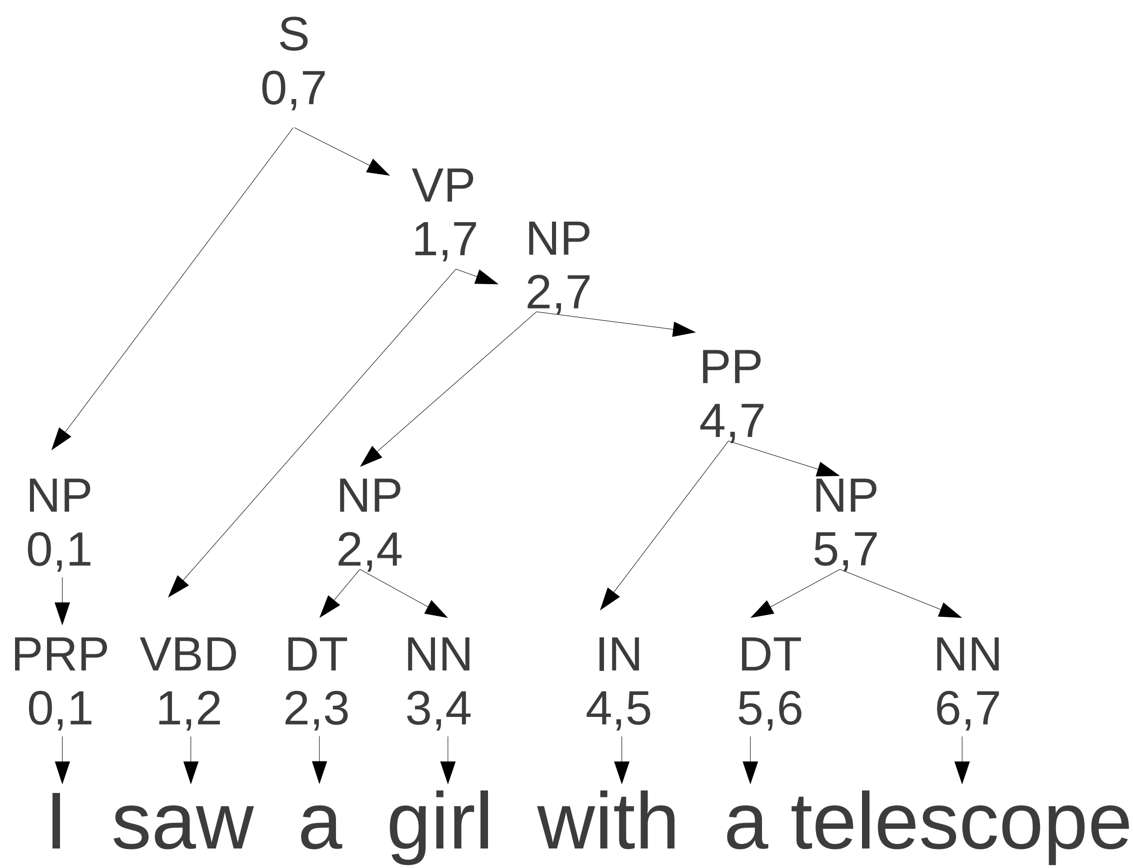
$$\underset{Y}{\operatorname{argmax}}\, P(Y, X)$$

- Can we use the Viterbi algorithm as we did before?
  - Answer: No!
  - Reason: Parse candidates are not graphs, but hypergraphs.

What is a Hypergraph?

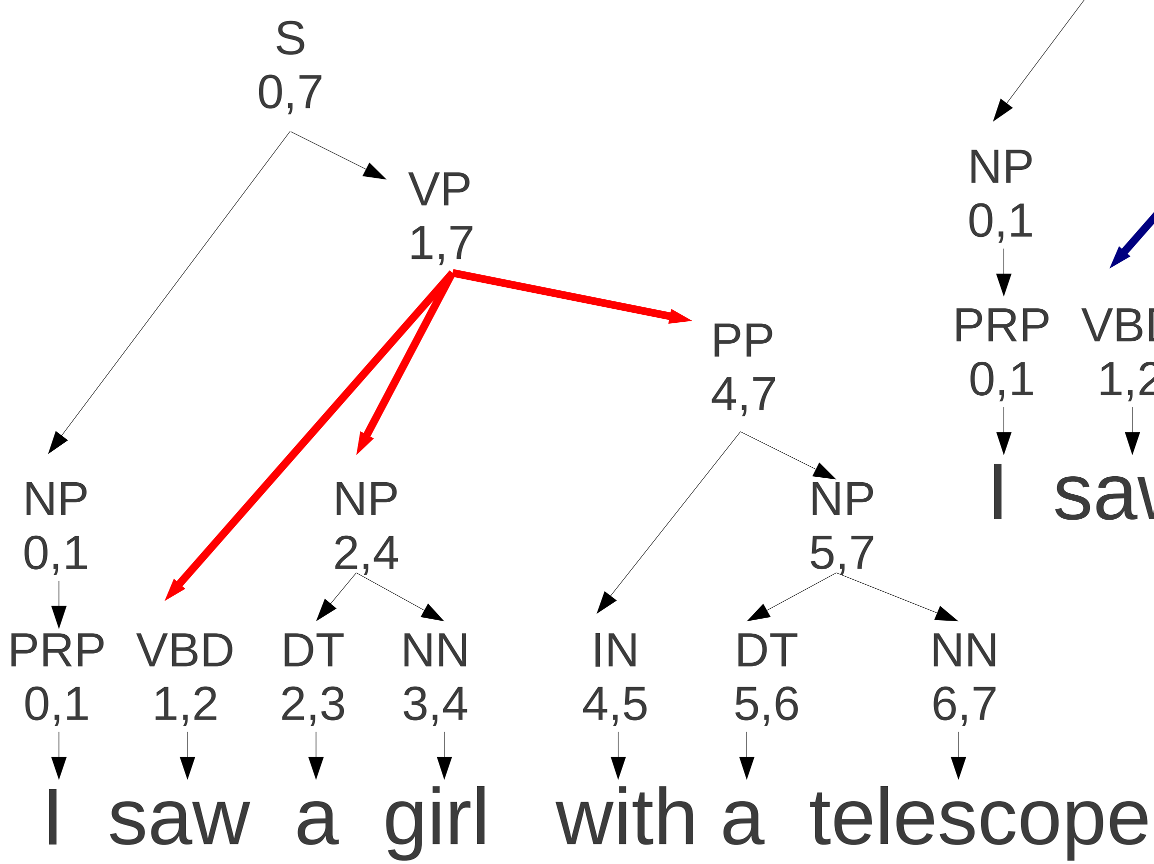
- Let's say we have two parse trees

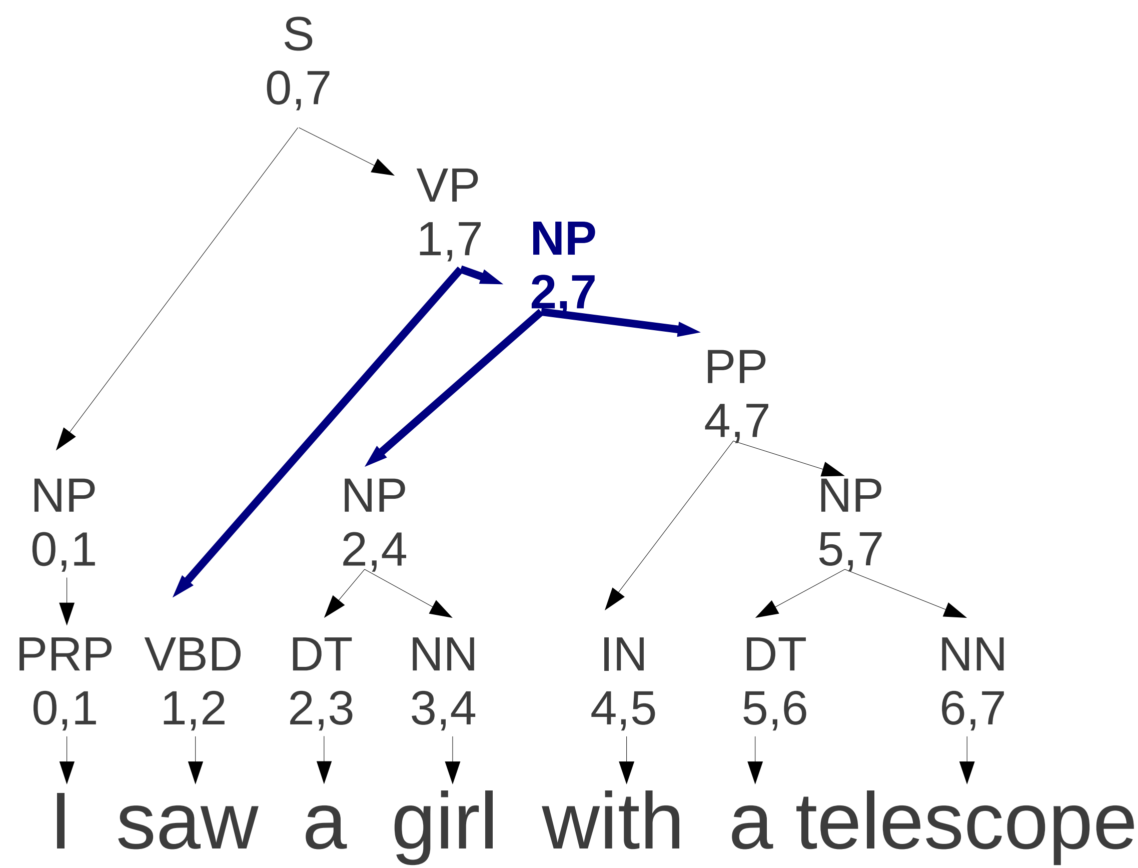




What is a Hypergraph?

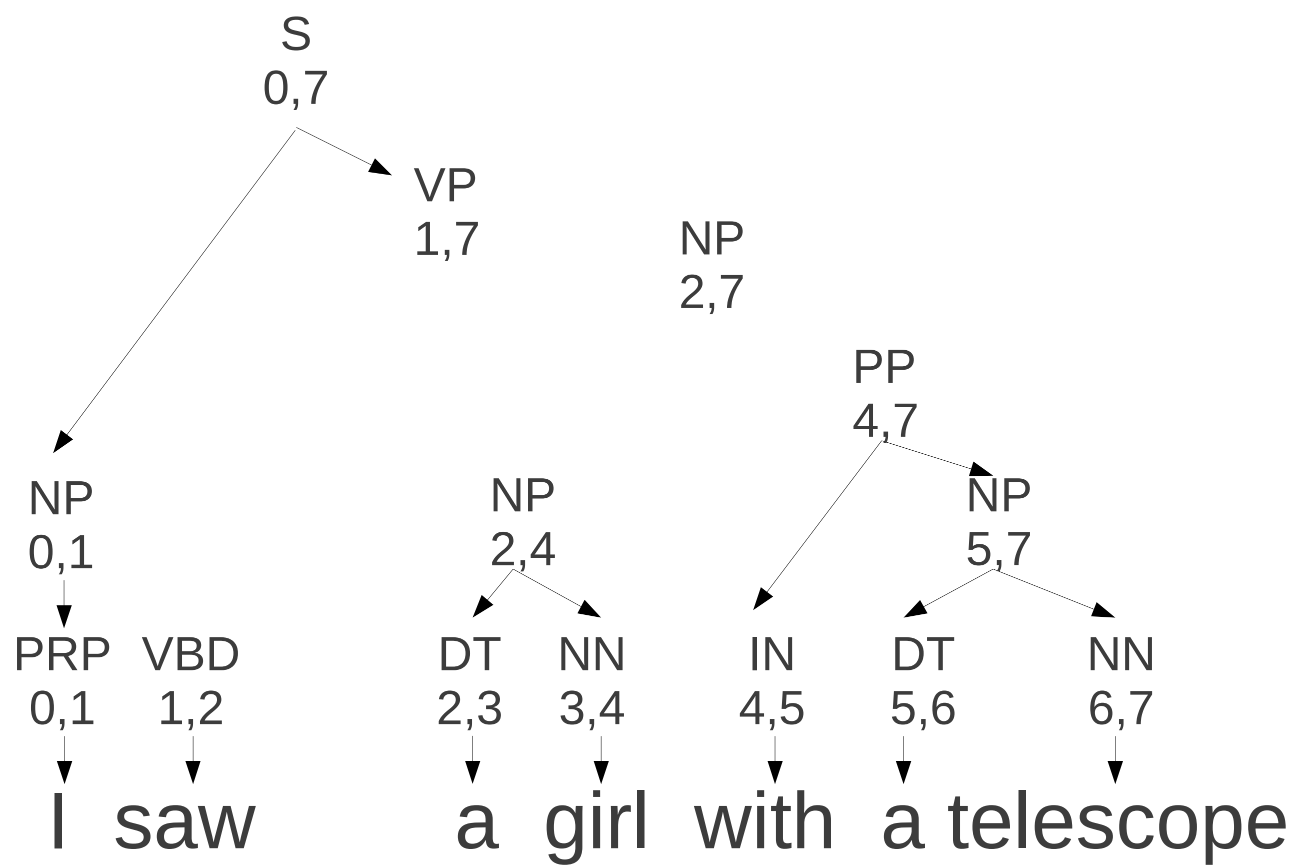
- Most parts are the same!





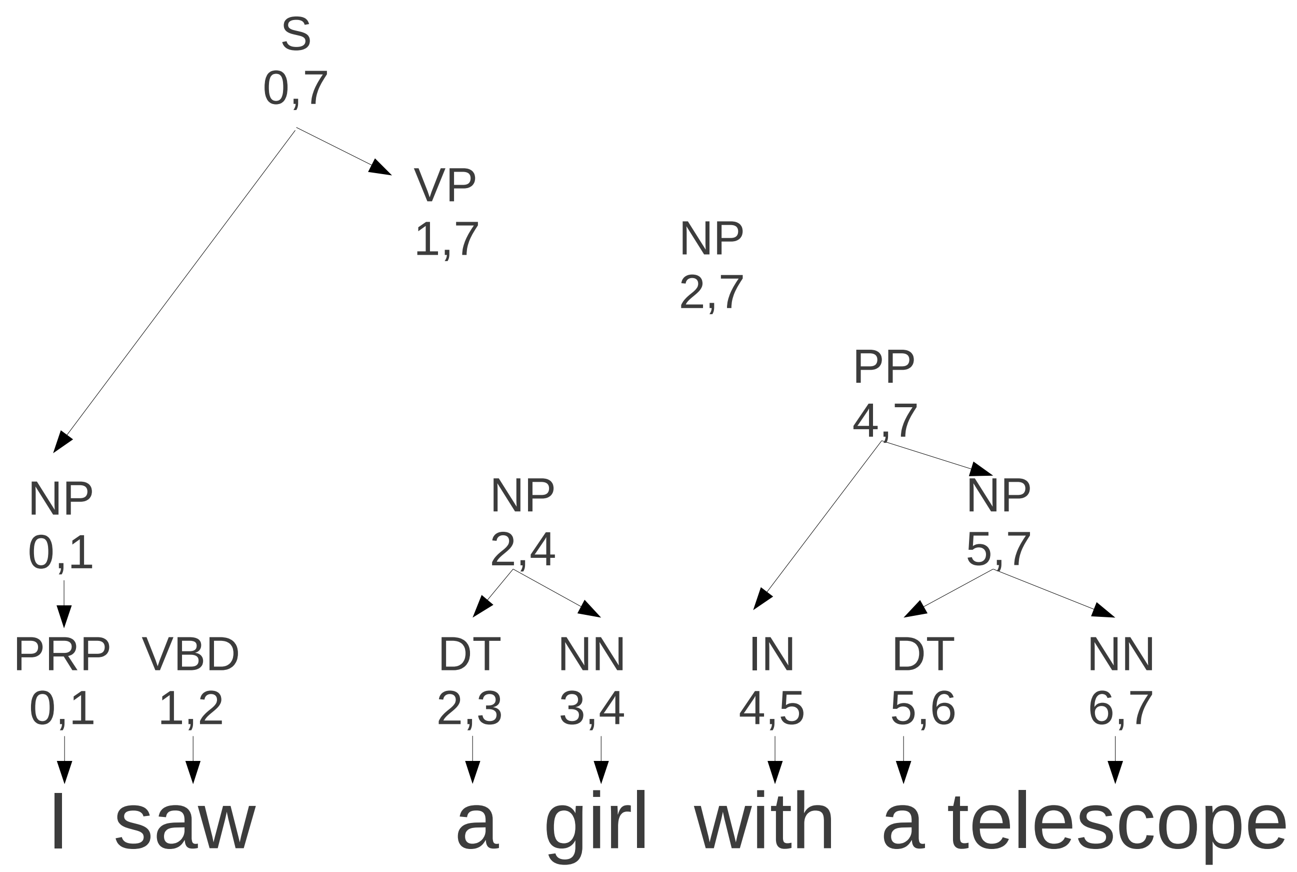
What is a Hypergraph?

- Graph with all same edges + all nodes



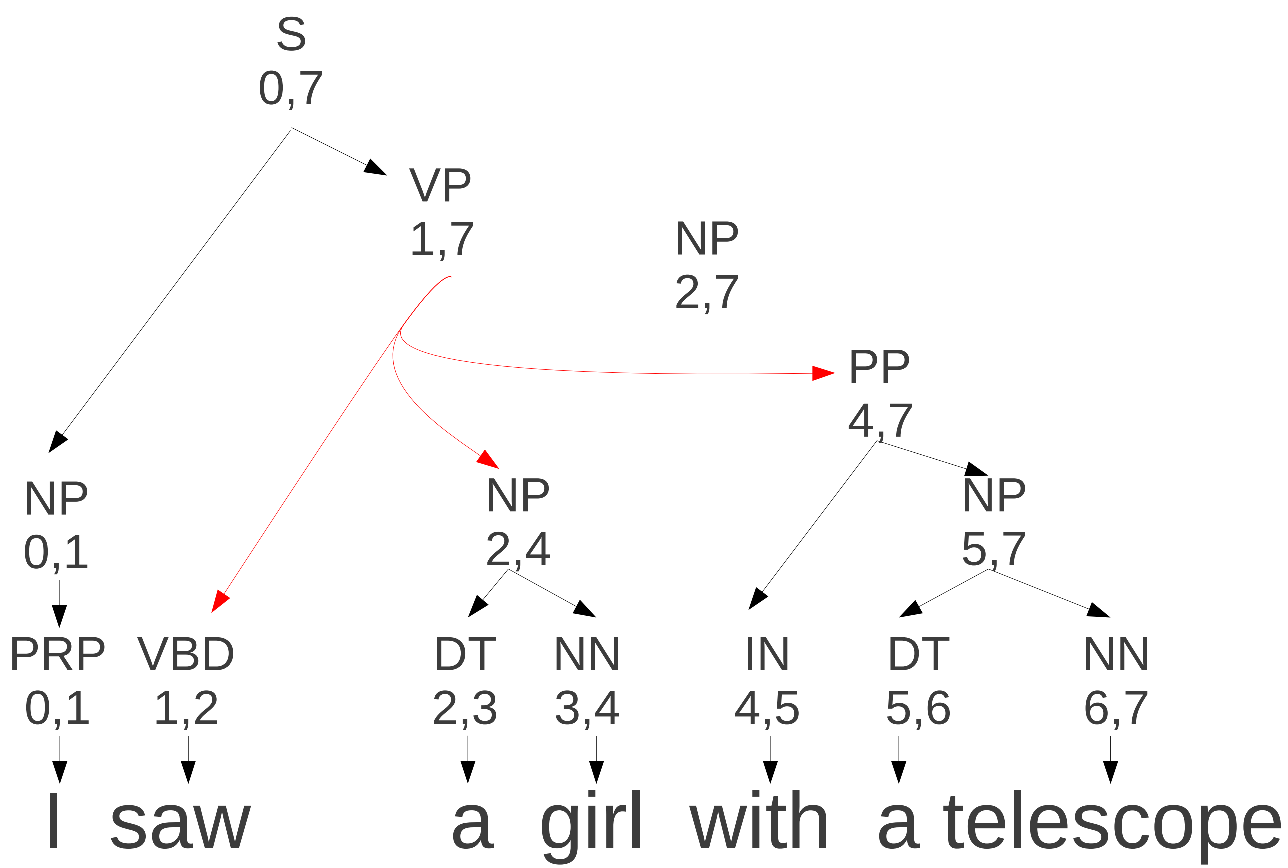
What is a Hypergraph?

- Create graph with all same edges + all nodes



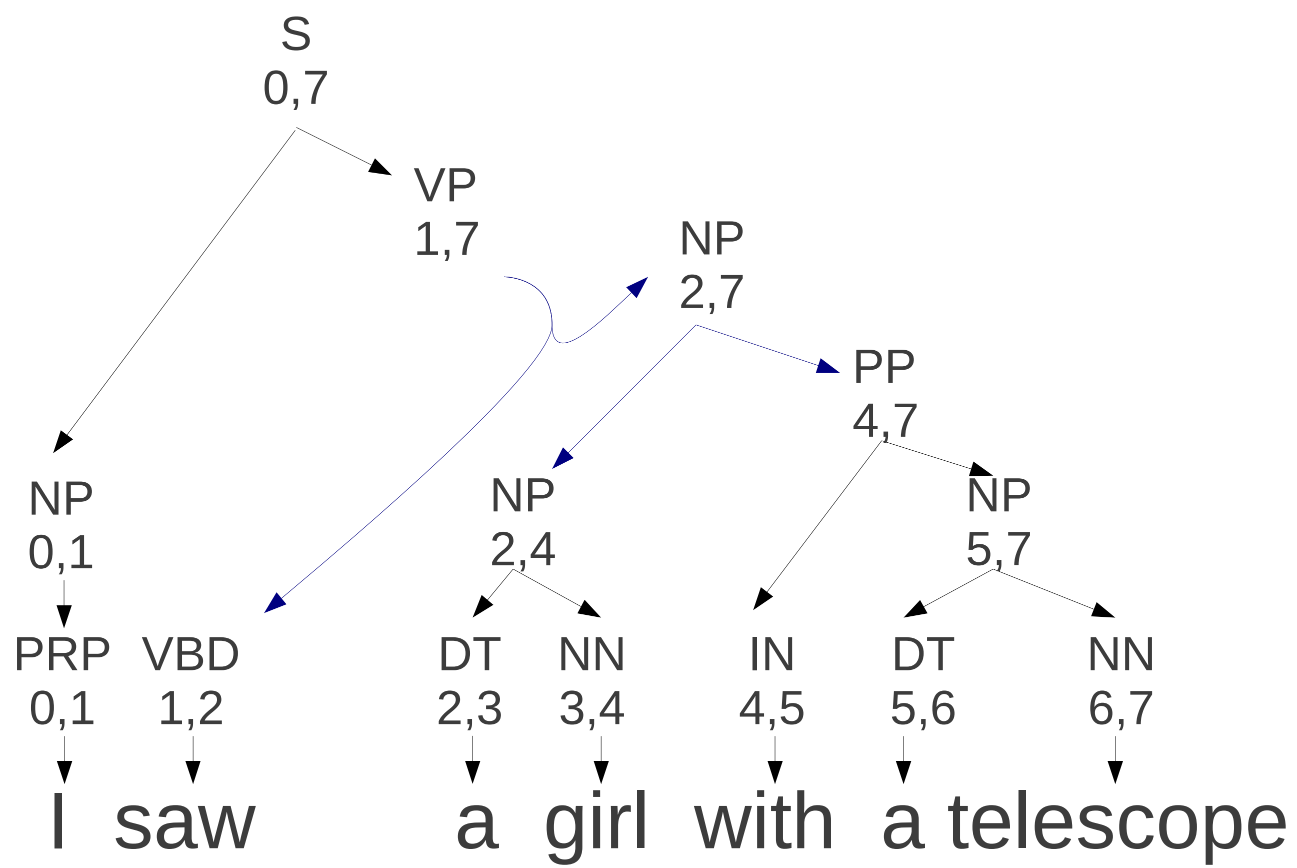
What is a Hypergraph?

- With the edges in the first trees:



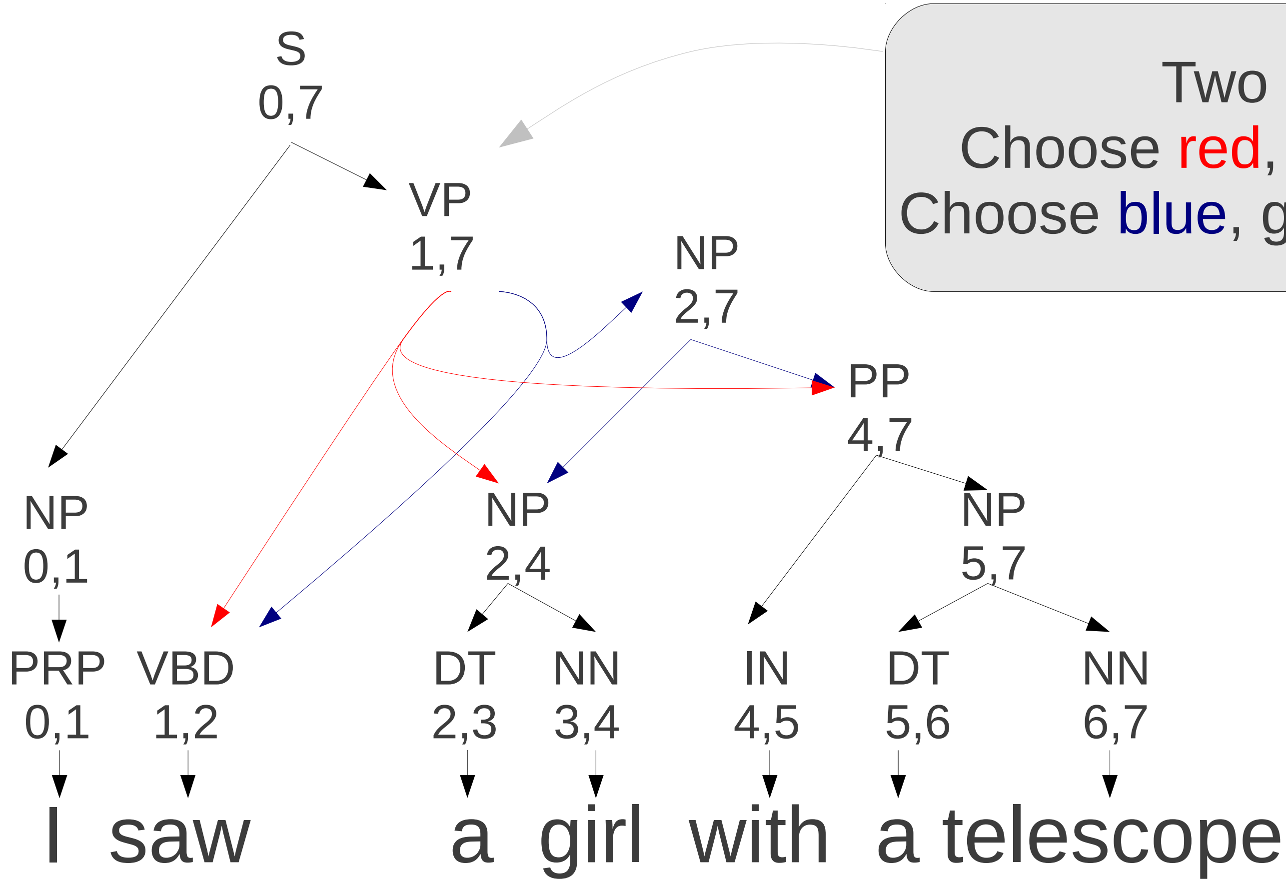
What is a Hypergraph?

- With the edges in the second tree:



What is a Hypergraph?

- With the edges in the first and second trees:

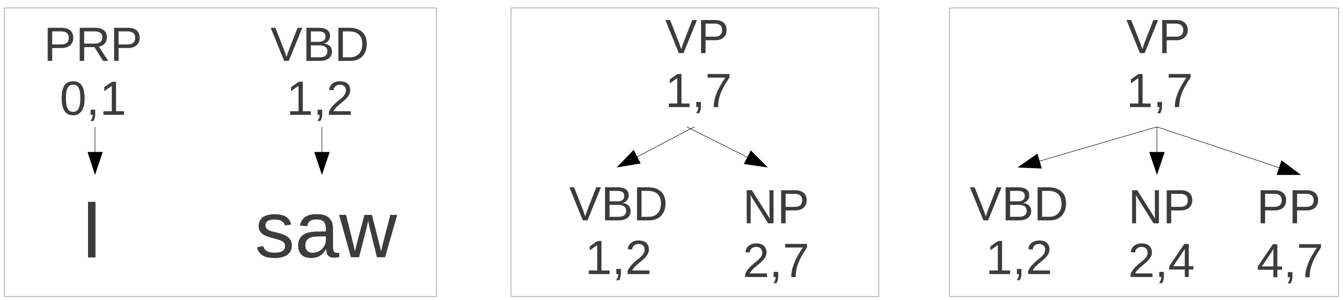


Two choices!
Choose red, get the first tree
Choose blue, get the second tree

Why a “Hyper”graph?

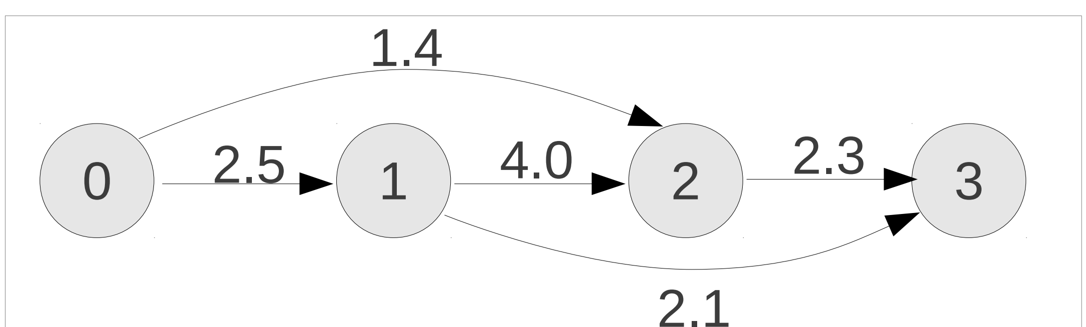
- The “degree” of an edge is the number of children

Degree 1 | Degree 2 | Degree 3



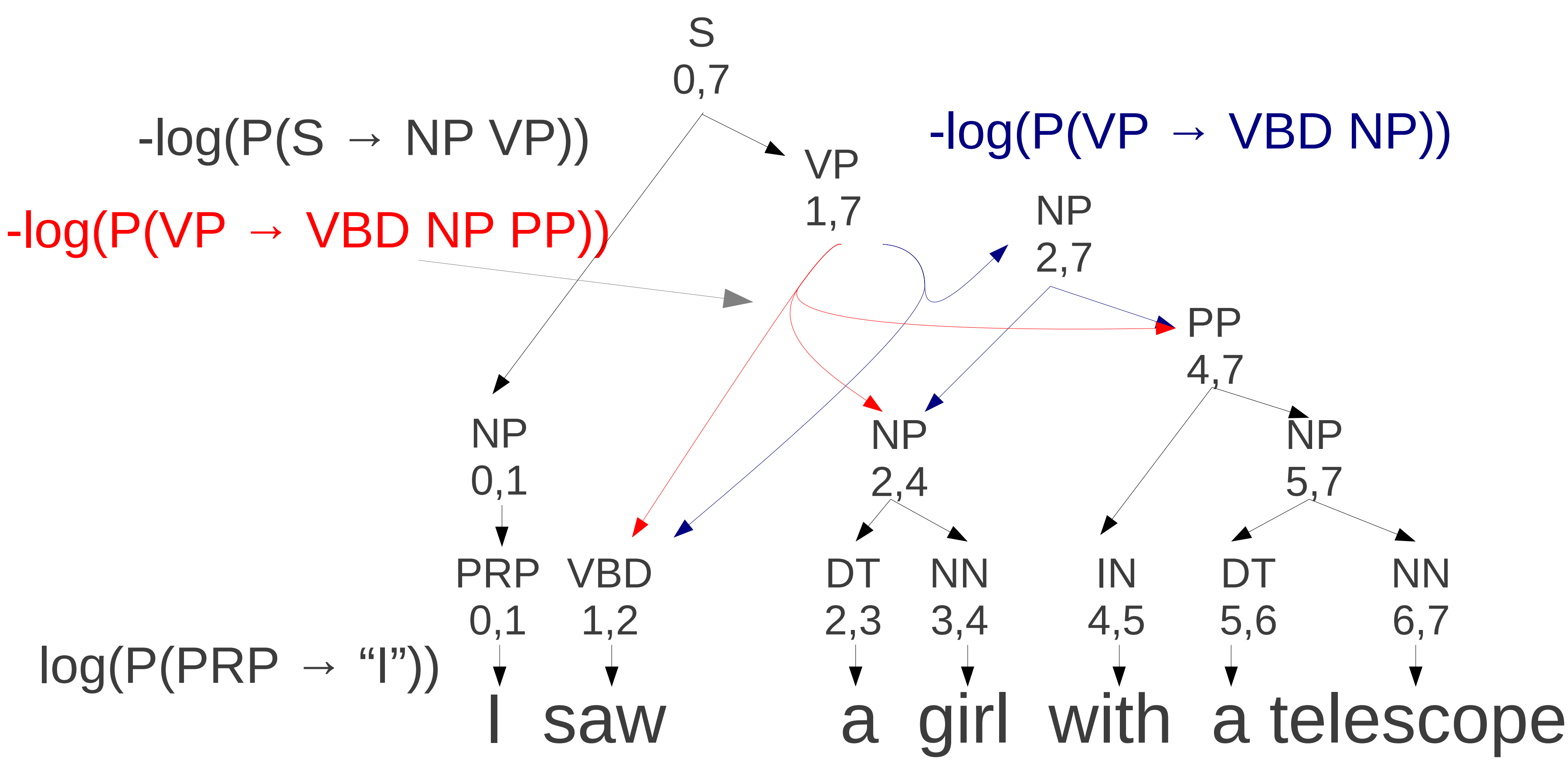
- The degree of a hypergraph is the maximum degree of all its edges
- A graph is a hypergraph of degree 1!

Example →



Weighted Hypergraphs

- Like graphs:
  - can add weights to hypergraph edges
  - use negative log probability of rule



Solving Hypergraphs

- Parsing = finding minimum path through a hypergraph

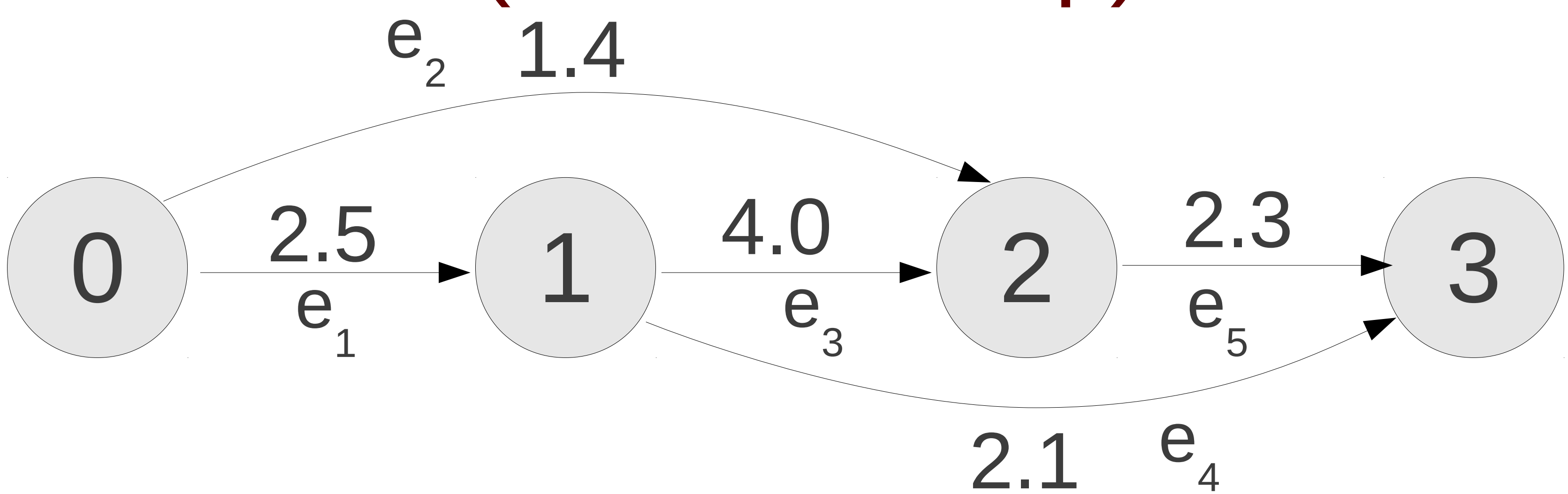
Solving Hypergraphs

- Parsing = finding minimum path through a hypergraph
- We can do this for graphs with the Viterbi algorithm
  - Forward: Calculate score of best path to each state
  - Backward: Recover the best path

Solving Hypergraphs

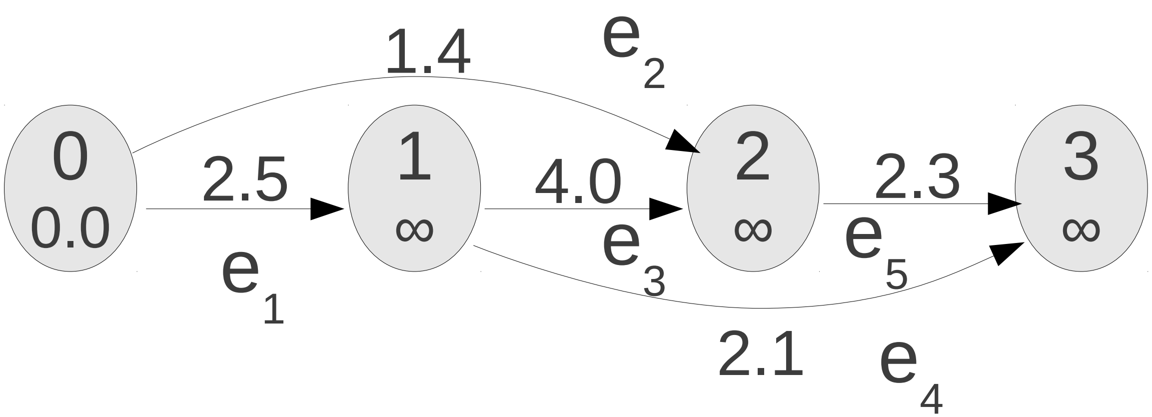
- Parsing = finding minimum path through a hypergraph
- We can do this for graphs with the Viterbi algorithm
  - Forward: Calculate score of best path to each state
  - Backward: Recover the best path
- For hypergraphs, almost identical algorithm!
  - Inside: Calculate score of best subtree for each node
  - Outside: Recover the best tree

Review: Viterbi Algorithm (Forward Step)



best_score[0] = 0
for each *node* in the *graph* (ascending order)
    best_score[*node*] = ∞
    for each incoming *edge* of *node*
        score = *best_score*[*edge.prev_node*] + *edge.score*
        if *score* < *best_score*[*node*]
            best_score[*node*] = *score*
            best_edge[*node*] = *edge*

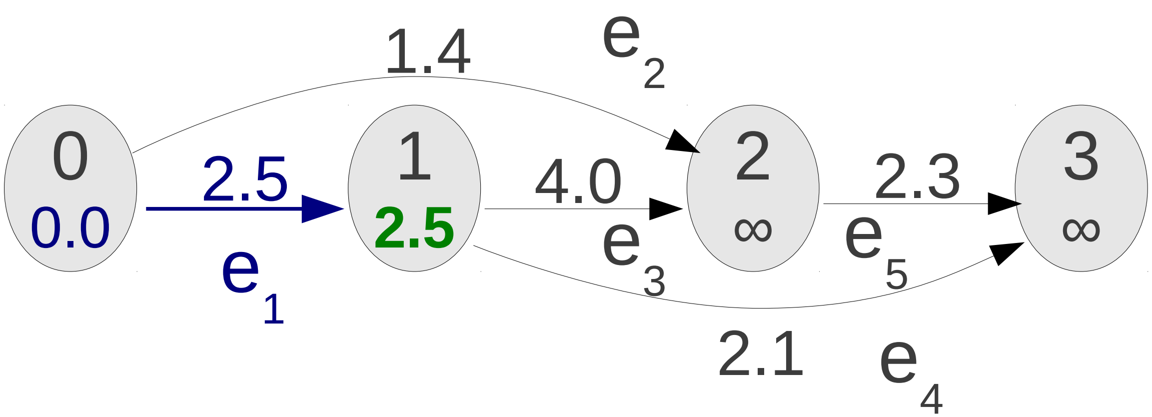
Example:



Initialize:

best_score[0] = 0

Example:



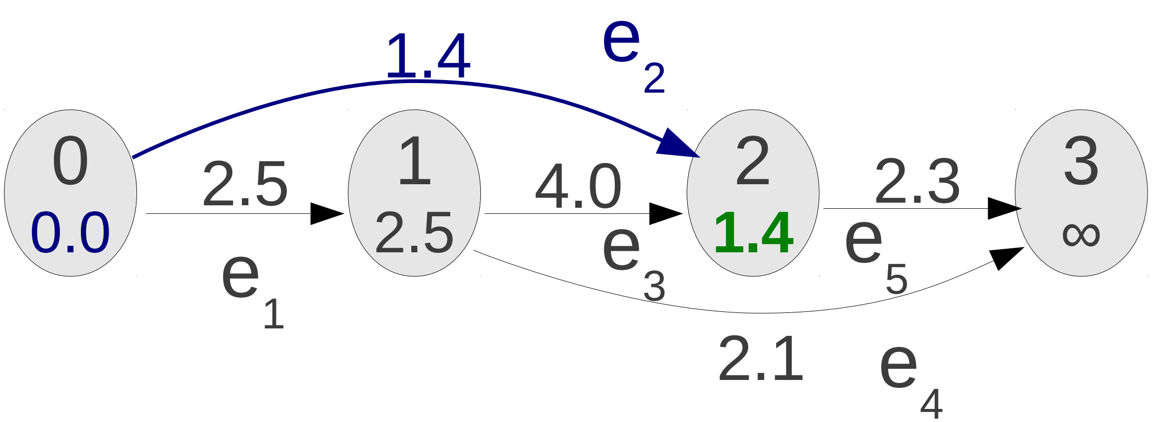
Initialize:

best_score[0] = 0

Check e_1:

score = 0 + 2.5 = 2.5 (< ∞)
best_score[1] = 2.5
best_edge[1] = e_1

Example:



Initialize:

best_score[0] = 0

Check e_1:

score = 0 + 2.5 = 2.5 (< ∞)

best_score[1] = 2.5

best_edge[1] = e_1

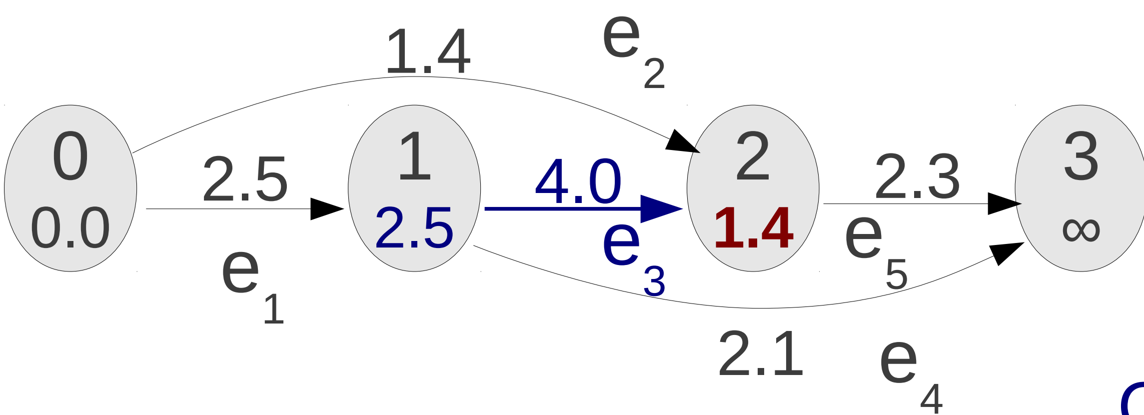
Check e_2:

score = 0 + 1.4 = 1.4 (< ∞)

best_score[2] = 1.4

best_edge[2] = e_2

Example:



Initialize:

best_score[0] = 0

Check e_1:

score = 0 + 2.5 = 2.5 (< ∞)

best_score[1] = 2.5

best_edge[1] = e_1

Check e_2:

score = 0 + 1.4 = 1.4 (< ∞)

best_score[2] = 1.4

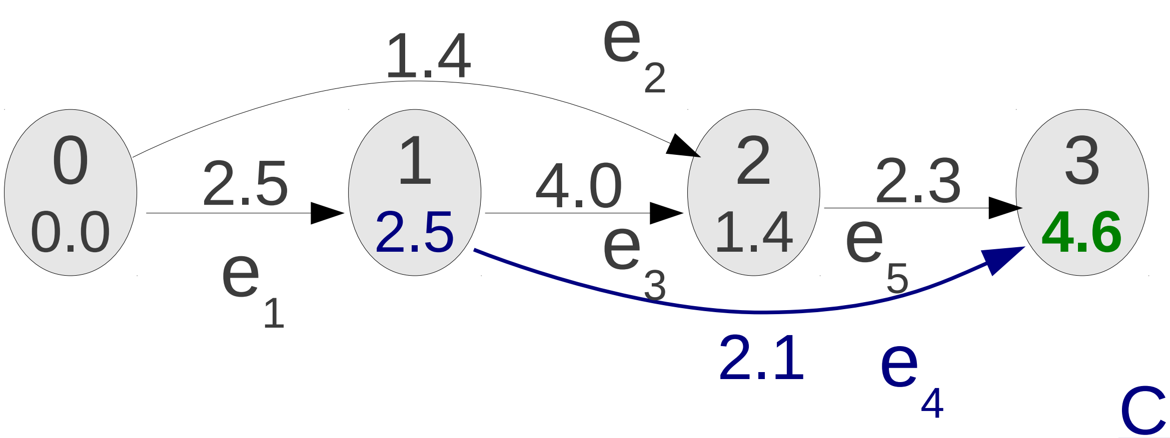
best_edge[2] = e_2

Check e_3:

score = 2.5 + 4.0 = 6.5 (> 1.4)

No change!

Example:



Initialize:

best_score[0] = 0

Check e_1:

score = 0 + 2.5 = 2.5 (< ∞)
best_score[1] = 2.5
best_edge[1] = e_1

Check e_2:

score = 0 + 1.4 = 1.4 (< ∞)
best_score[2] = 1.4
best_edge[2] = e_2

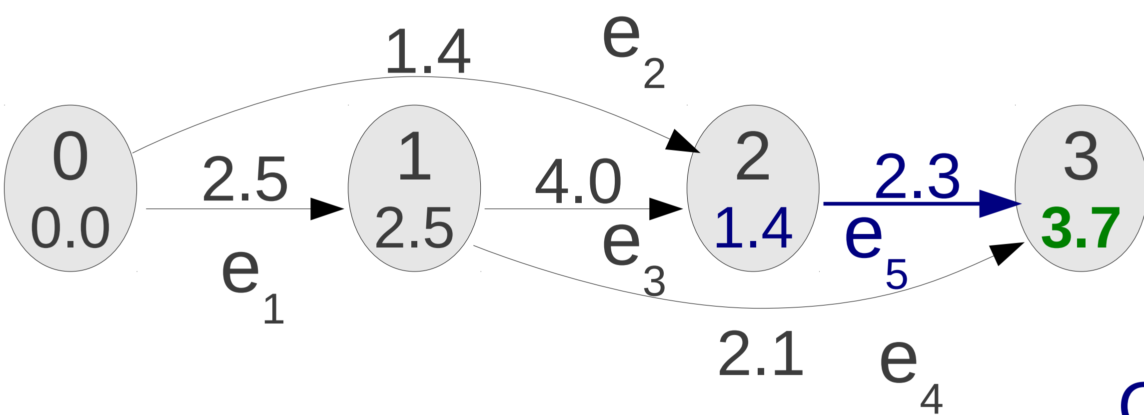
Check e_3:

score = 2.5 + 4.0 = 6.5 (> 1.4)
No change!

Check e_4:

score = 2.5 + 2.1 = 4.6 (< ∞)
best_score[3] = 4.6
best_edge[3] = e_4

Example:



Initialize:

best_score[0] = 0

Check e_1:

score = 0 + 2.5 = 2.5 (< ∞)

best_score[1] = 2.5

best_edge[1] = e_1

Check e_2:

score = 0 + 1.4 = 1.4 (< ∞)

best_score[2] = 1.4

best_edge[2] = e_2

Check e_3:

score = 2.5 + 4.0 = 6.5 (> 1.4)

No change!

Check e_4:

score = 2.5 + 2.1 = 4.6 (< ∞)

~~best_score[3] = 4.6~~

~~best_edge[3] = e_4~~

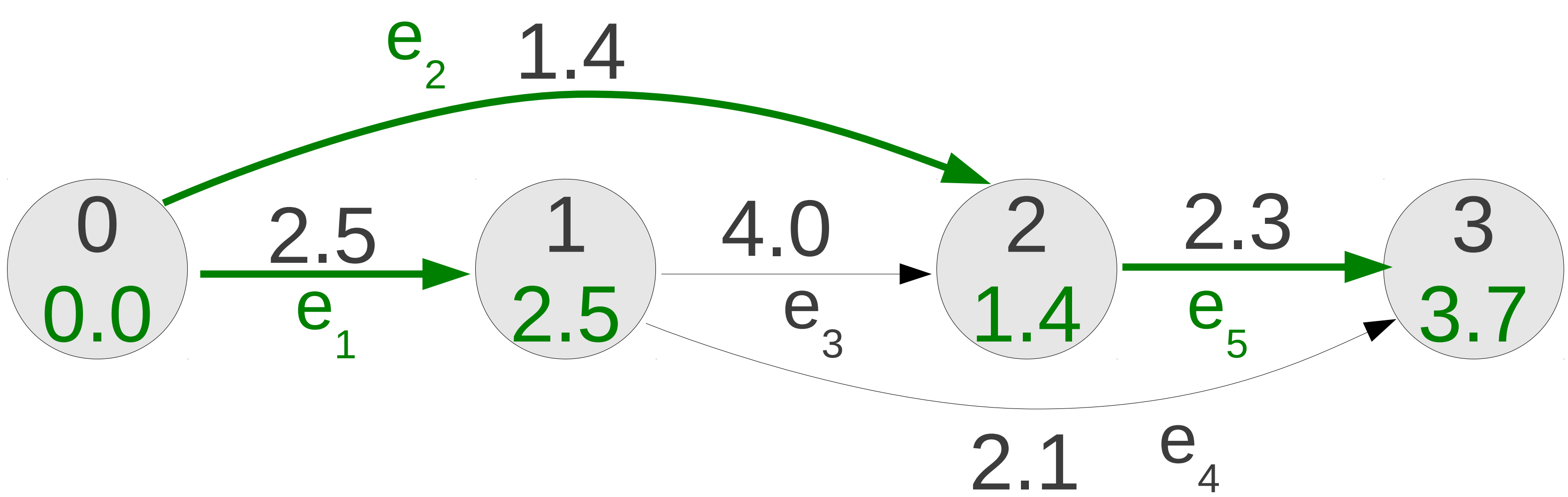
Check e_5:

score = 1.4 + 2.3 = 3.7 (< 4.6)

best_score[3] = 3.7

best_edge[3] = e_5

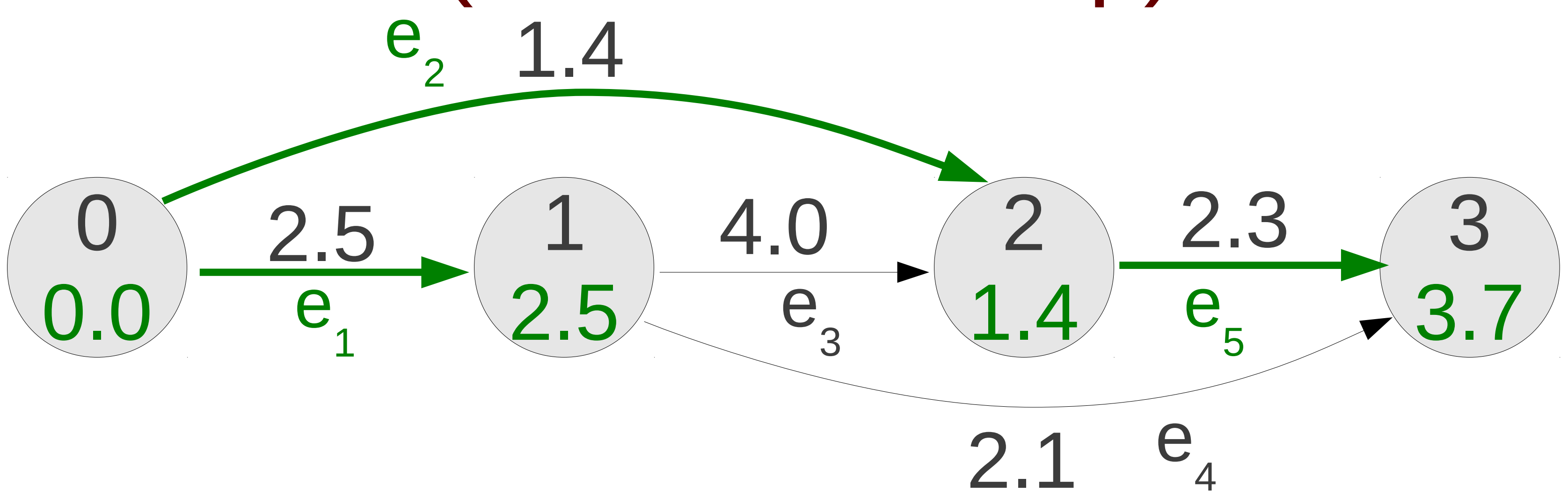
Result of Forward Step



best_score = (0.0, 2.5, 1.4, 3.7)

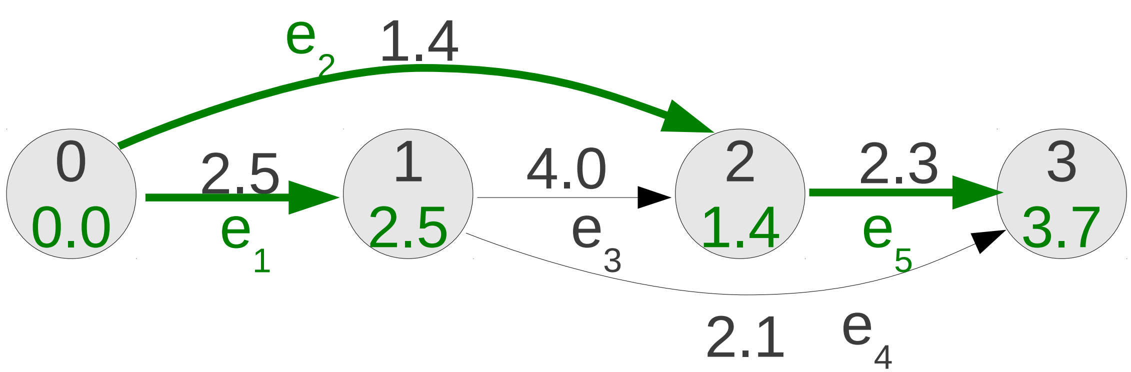
best_edge = (NULL, e_1, e_2, e_5)

Review: Viterbi Algorithm (Backward Step)



best_path = [ ]
next_edge = *best_edge*[*best_edge*.length – 1]
while *next_edge* != NULL
    add *next_edge* to *best_path*
    next_edge = *best_edge*[*next_edge.prev_node*]
reverse best_path

Example of Backward Step

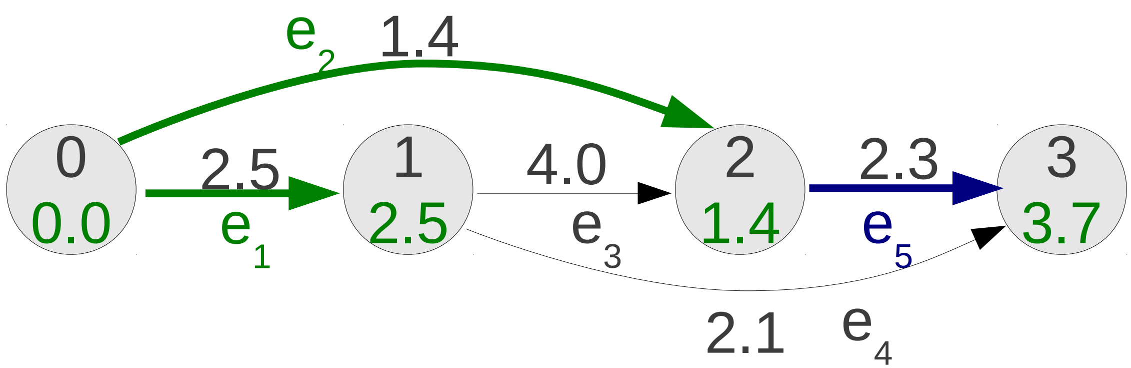


Initialize:

best_path = []

next_edge = best_edge[3] = e_5

Example of Backward Step



Initialize:

best_path = []

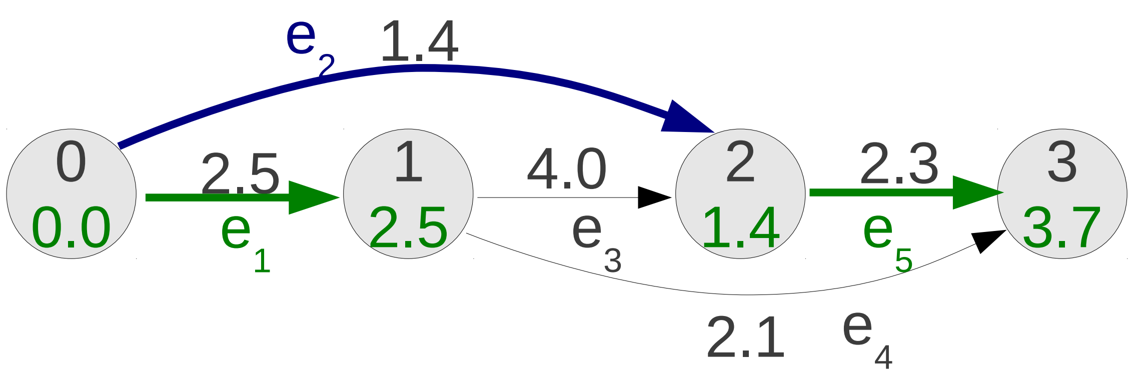
next_edge = best_edge[3] = e_5

Process e_5:

best_path = [e_5]

next_edge = best_edge[2] = e_2

Example of Backward Step



Initialize:

best_path = []
next_edge = best_edge[3] = e_5

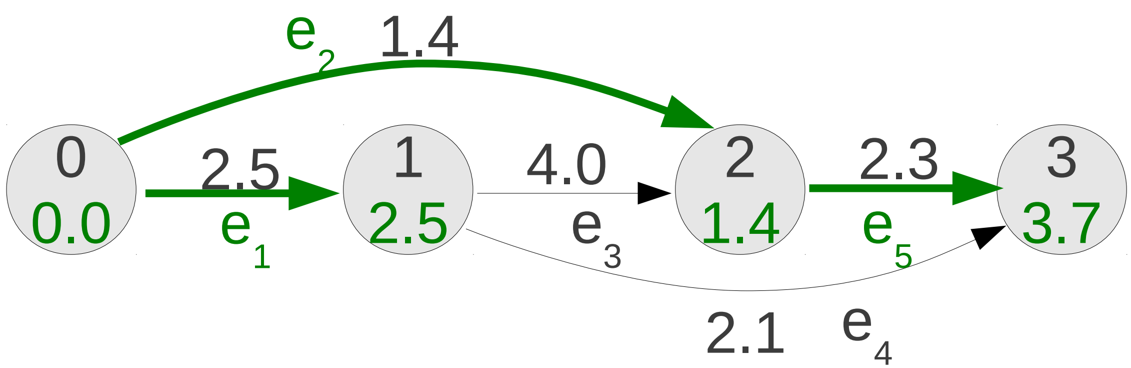
Process e_5:

best_path = [e_5]
next_edge = best_edge[2] = e_2

Process e_2:

best_path = [e_5, e_2]
next_edge = best_edge[0] = NULL

Example of Backward Step



Initialize:

best_path = []
next_edge = best_edge[3] = e_5

Process e_5:

best_path = [e_5]
next_edge = best_edge[2] = e_2

Process e_5:

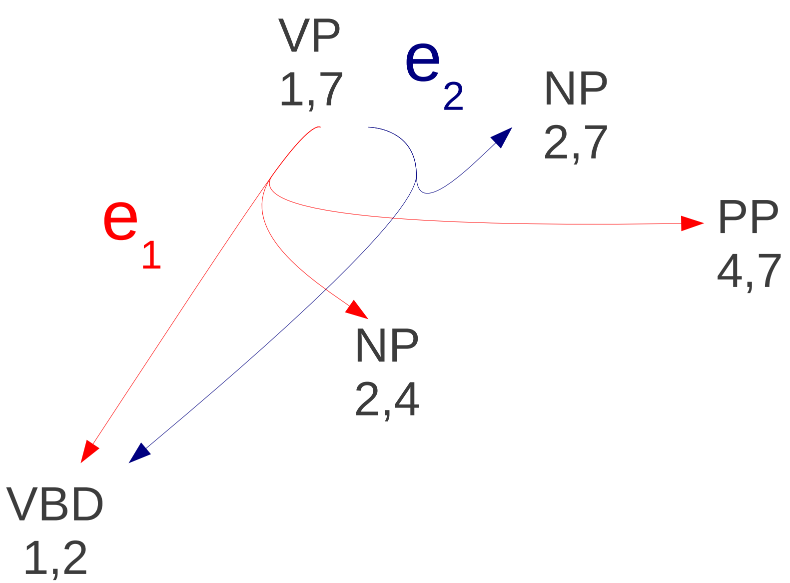
best_path = [e_5, e_2]
next_edge = best_edge[0] = NULL

Reverse:

best_path = [e_2, e_5]

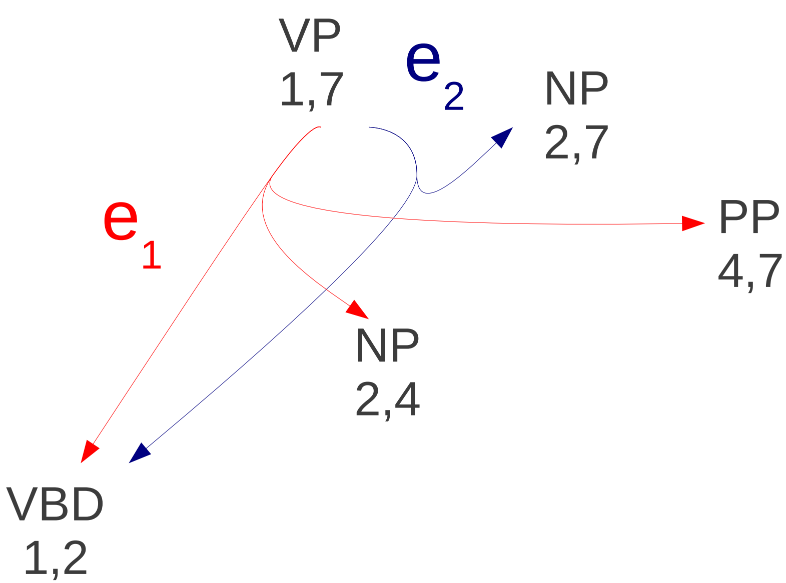
Inside Step for Hypergraphs:

- Find the score of best subtree of VP1,7



Inside Step for Hypergraphs:

- Find the score of best subtree of VP1,7

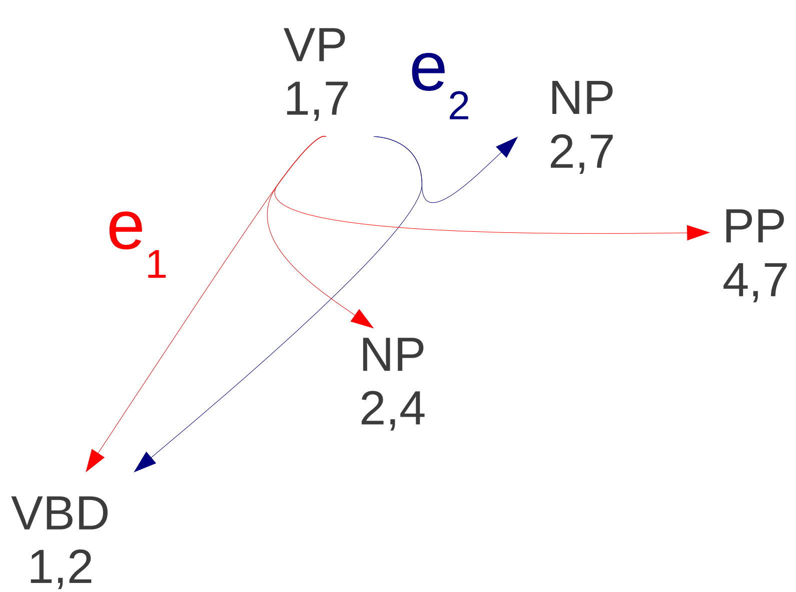


score(e_1) =
-log(P(VP → VBD NP PP)) +
best_score[VBD1,2] +
best_score[NP2,4] +
best_score[NP2,7]

score(e_2) =
-log(P(VP → VBD NP)) +
best_score[VBD1,2] +
best_score[VBD2,7]

Inside Step for Hypergraphs:

- Find the score of best subtree of VP1,7



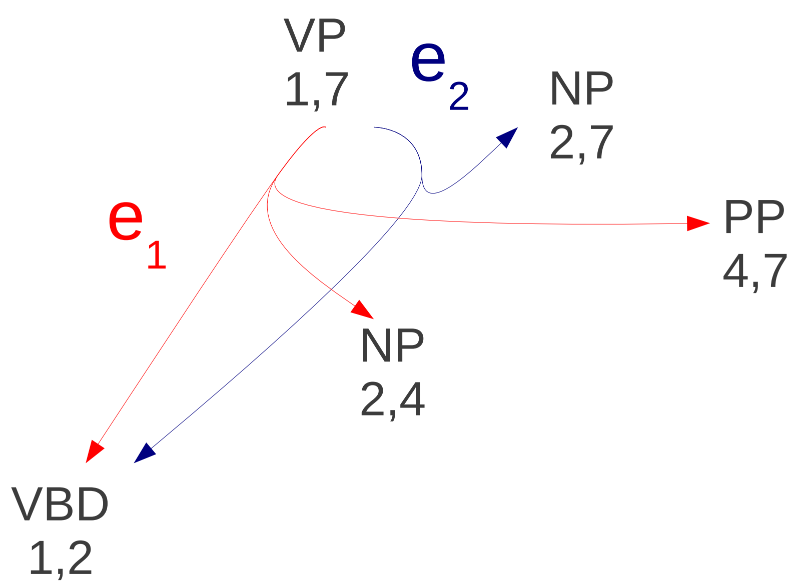
score(e_1) =
-log(P(VP → VBD NP PP)) +
best_score[VBD1,2] +
best_score[NP2,4] +
best_score[NP2,7]

score(e_2) =
-log(P(VP → VBD NP)) +
best_score[VBD1,2] +
best_score[VBD2,7]

best_edge[VB1,7] = argmin$_{e1,e2}$ score

Inside Step for Hypergraphs:

- Find the score of best subtree of VP1,7



score(e_1) =
-log(P(VP → VBD NP PP)) +
best_score[VBD1,2] +
best_score[NP2,4] +
best_score[NP2,7]

score(e_2) =
-log(P(VP → VBD NP)) +
best_score[VBD1,2] +
best_score[VBD2,7]

best_edge[VB1,7] = $\text{argmin}_{e1,e2}$ score

best_score[VB1,7] =
score(best_edge[VB1,7])

Building Hypergraphs from Grammars

- Ok, we can solve hypergraphs, but what we have is:

A Grammar	A Sentence
P(S → NP VP) = 0.8 P(S → PRP VP) = 0.2 P(VP → VBD NP PP) = 0.6 P(VP → VBD NP)= 0.4 P(NP → DT NN) = 0.5 P(NP → NN) = 0.5 P(PRP → "I") = 0.4 P(VBD → "saw") = 0.05 P(DT → "a") = 0.6 ...	I saw a girl with a telescope

- How do we build a hypergraph?

CKY Algorithm

- The CKY (Cocke-Kasami-Younger) algorithm creates and solves hypergraphs
- Grammar must be in Chomsky normal form (CNF)
  - All rules have two non-terminals or one terminal on right

OK	OK	Not OK!
S → NP VP	PRP → “I”	VP → VBD NP PP
S → PRP VP	VBD → “saw”	NP → NN
VP → VBD NP	DT → “a”	NP → PRP

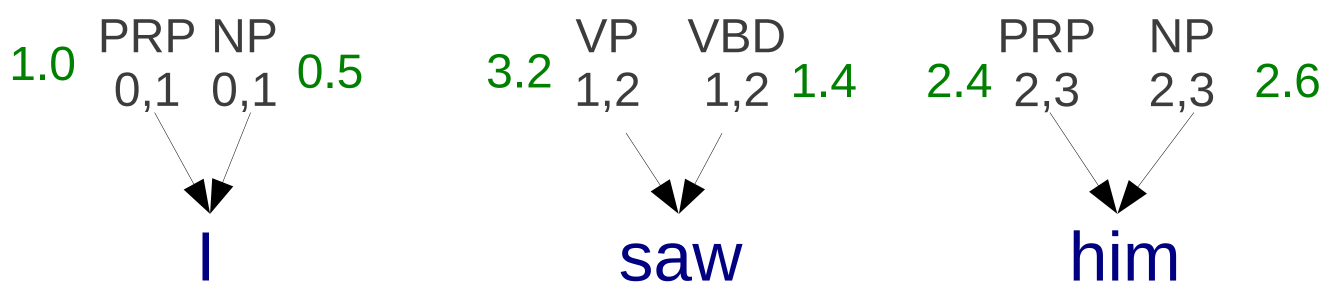
  - Can convert rules into CNF

VP → VBD NP PP ➔ VP → VBD VP'
VP' → NP PP

NP → PRP + PRP → “I” ➔ NP_PRP → “I”

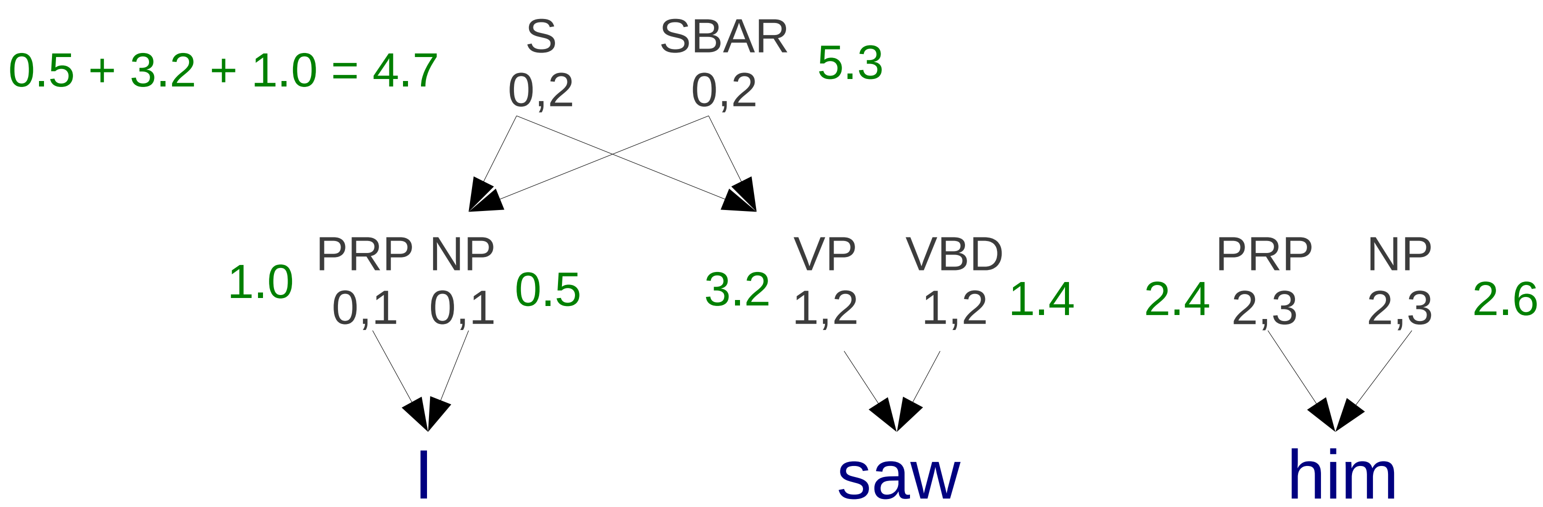
CKY Algorithm

- Start by expanding all rules for terminals with scores



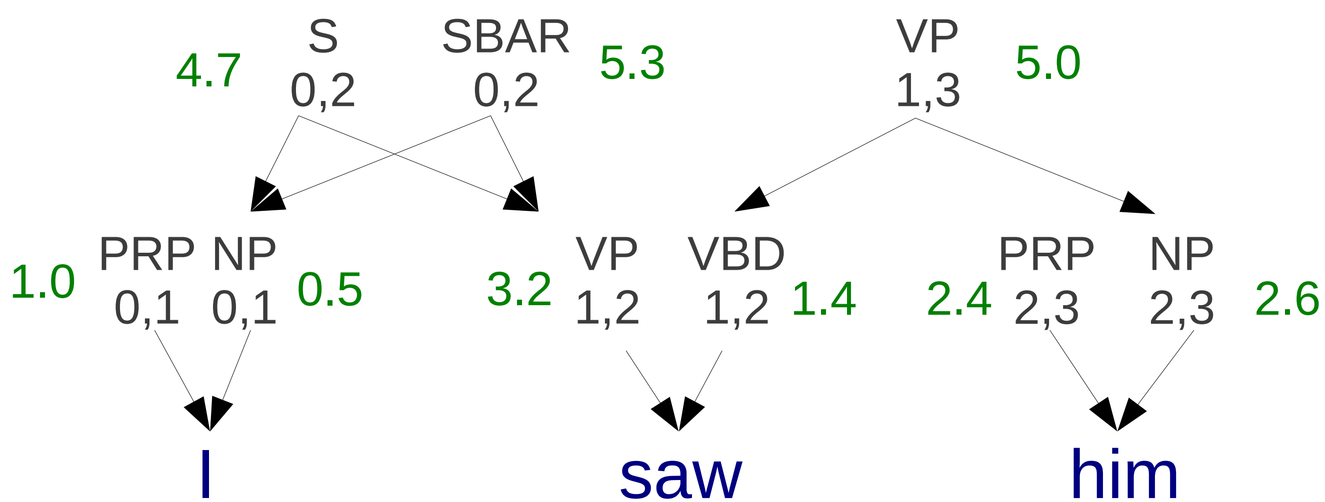
CKY Algorithm

- Expand all possible nodes for 0,2



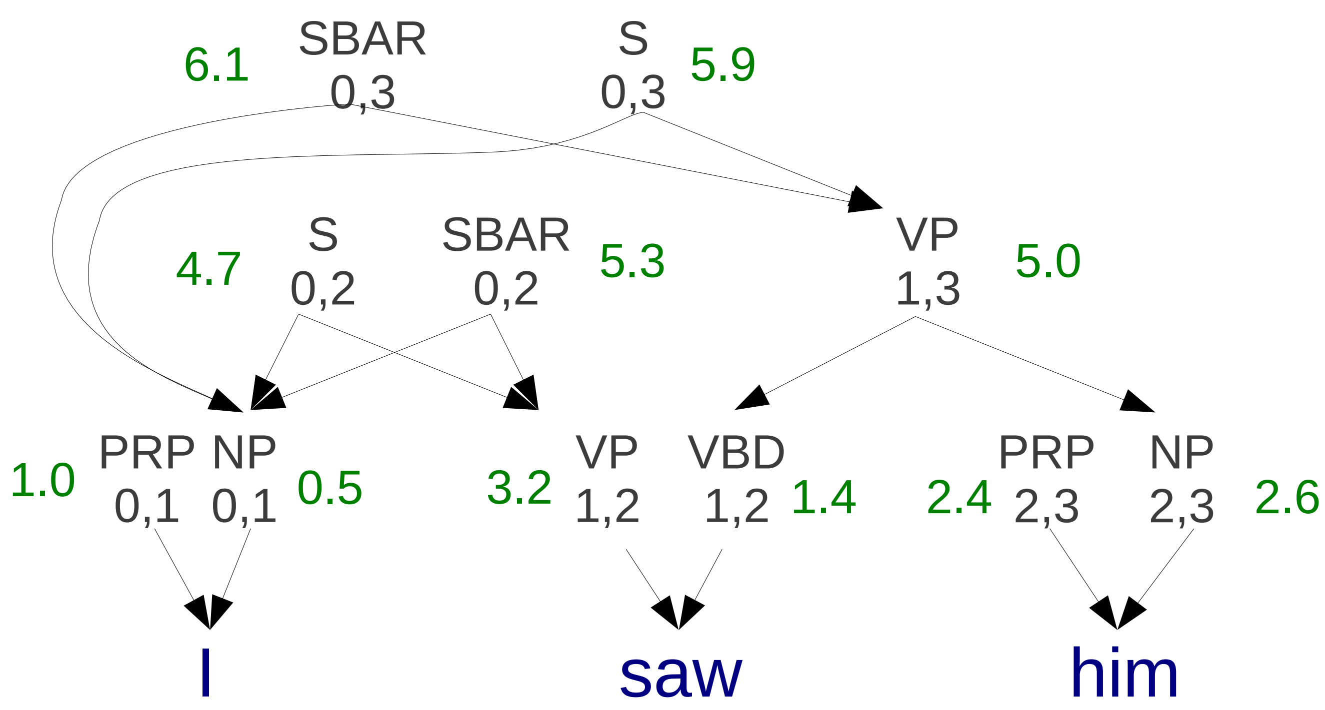
CKY Algorithm

- Expand all possible nodes for 1,3



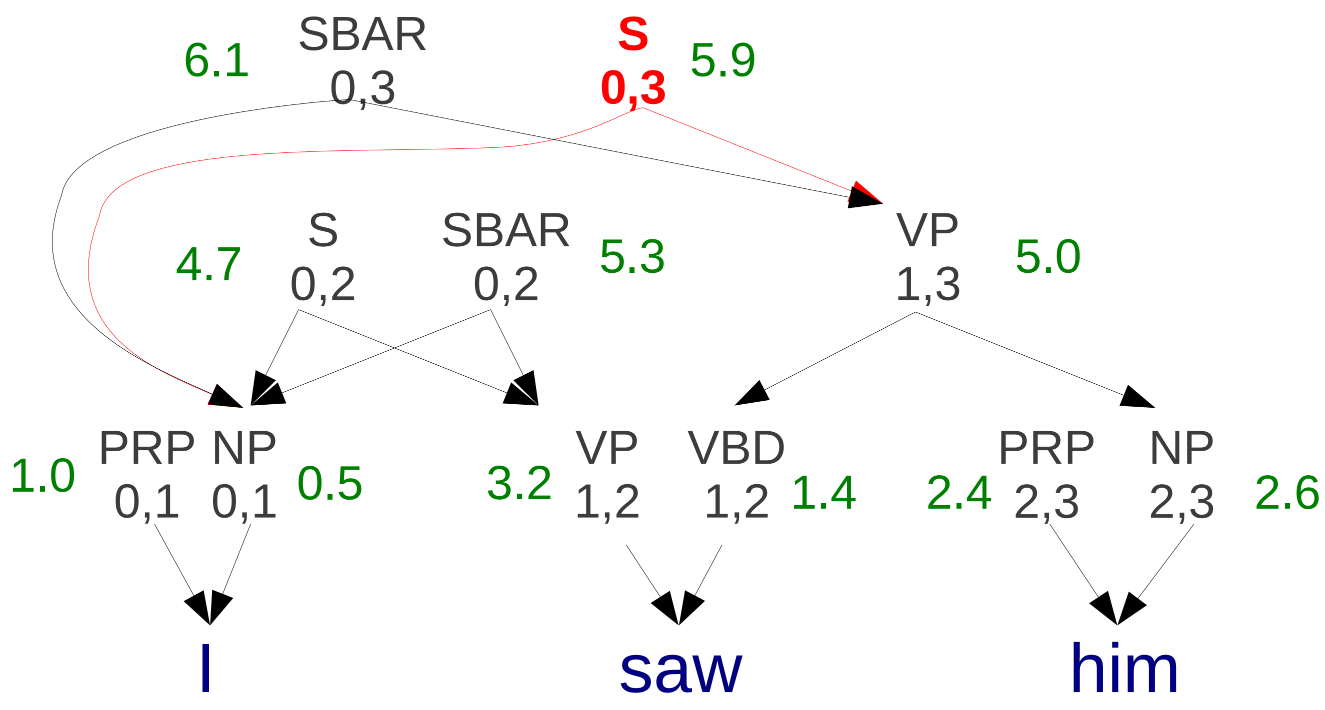
CKY Algorithm

- Expand all possible nodes for 0,3



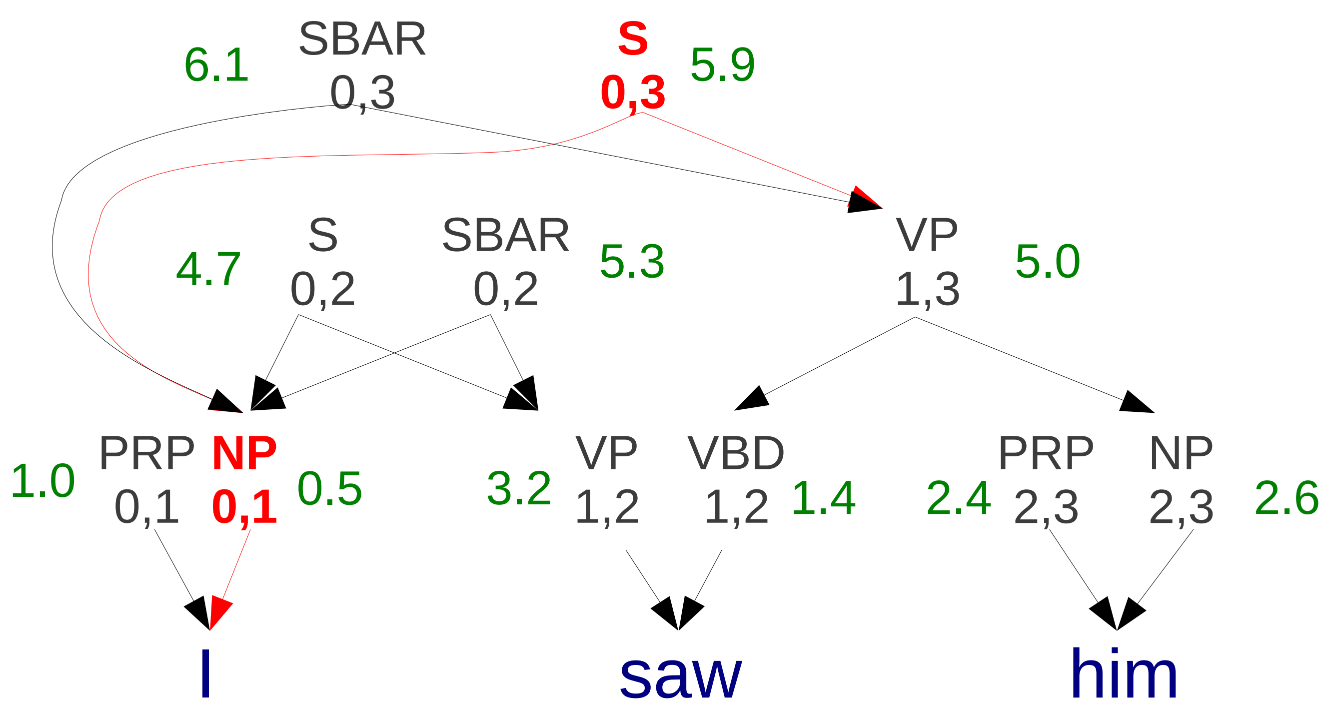
CKY Algorithm

- Find the S that covers the entire sentence and its best edge



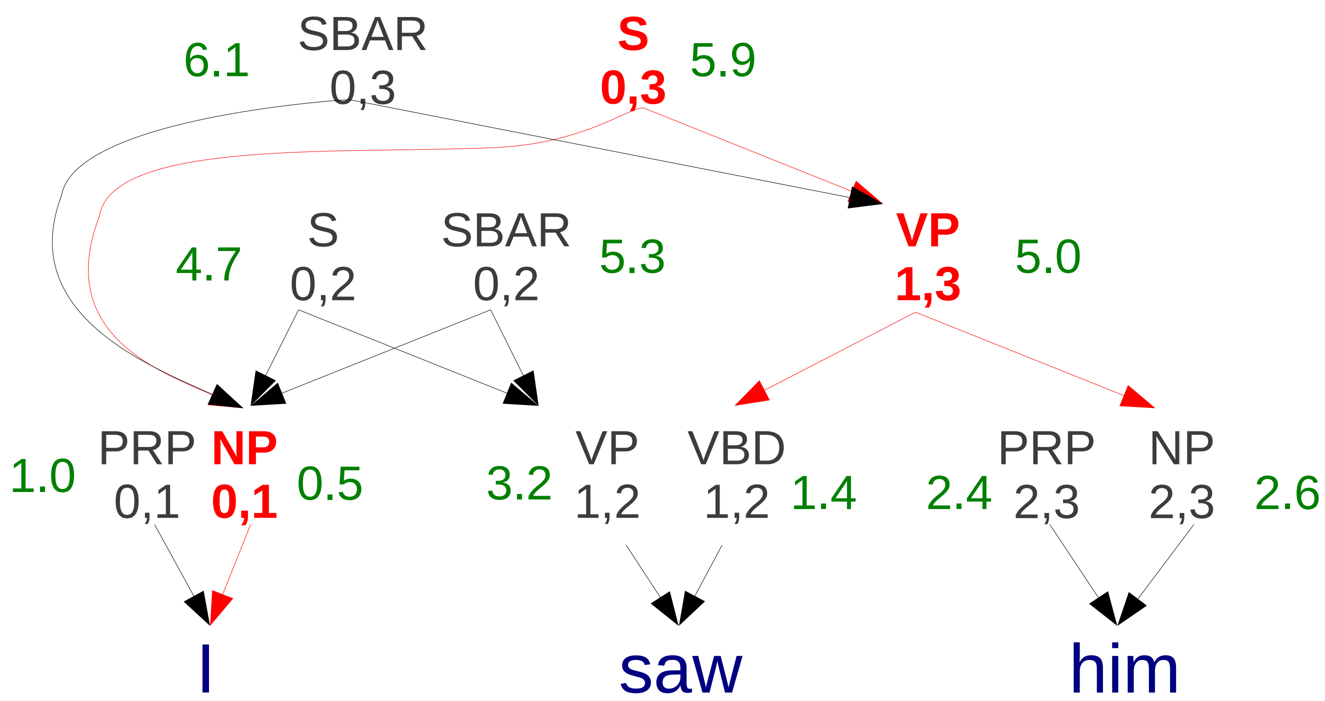
CKY Algorithm

- Expand the left child, right child recursively until we have our tree



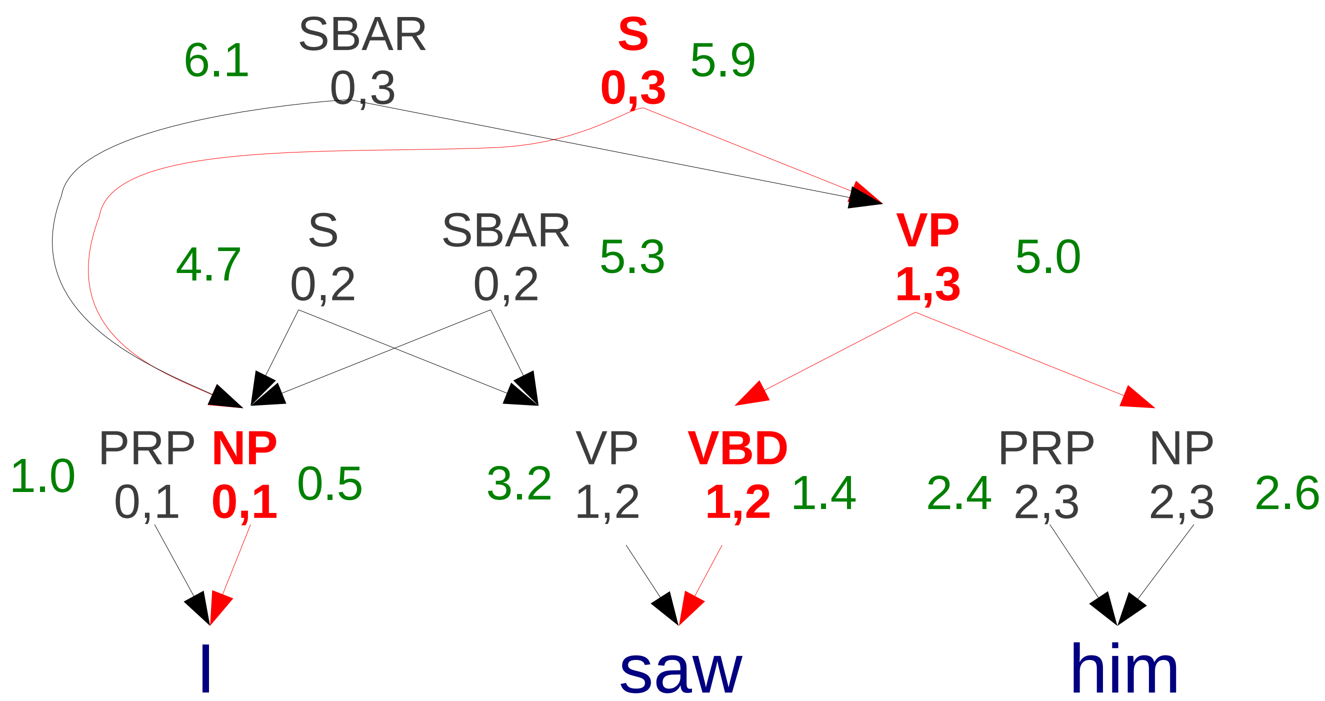
CKY Algorithm

- Expand the left child, right child recursively until we have our tree



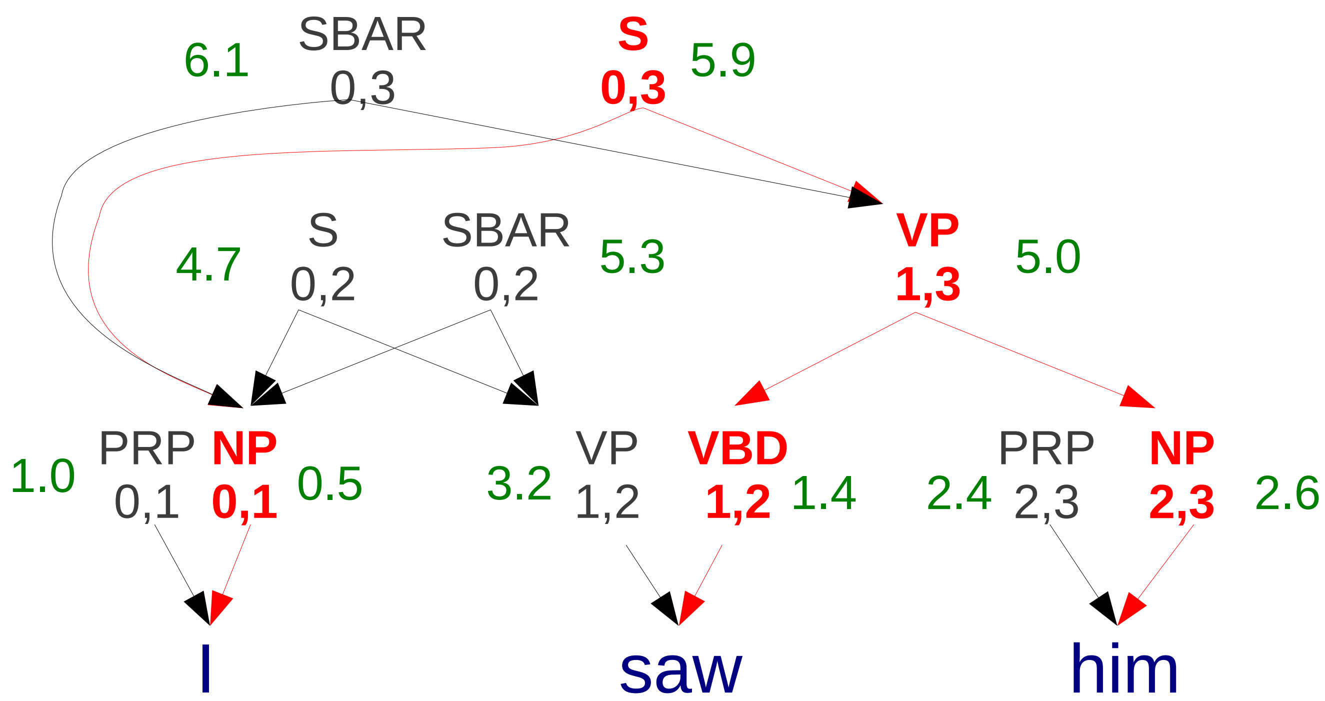
CKY Algorithm

- Expand the left child, right child recursively until we have our tree



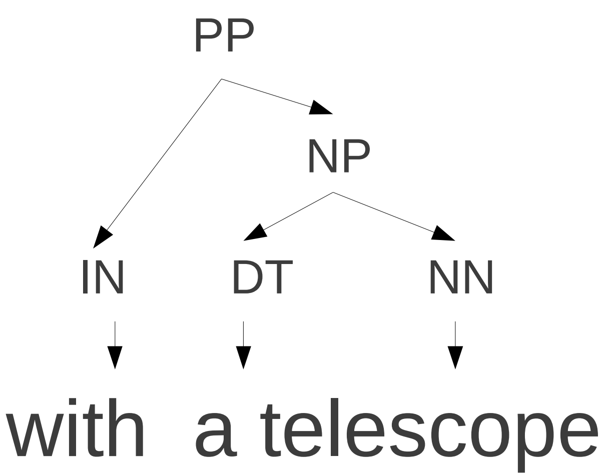
CKY Algorithm

- Expand the left child, right child recursively until we have our tree



Printing Parse Trees

- Standard text format for parse tree: “Penn Treebank”




```
(PP
  (IN with)
  (NP
    (DT a)
    (NN telescope)))
```

Printing Parse Trees

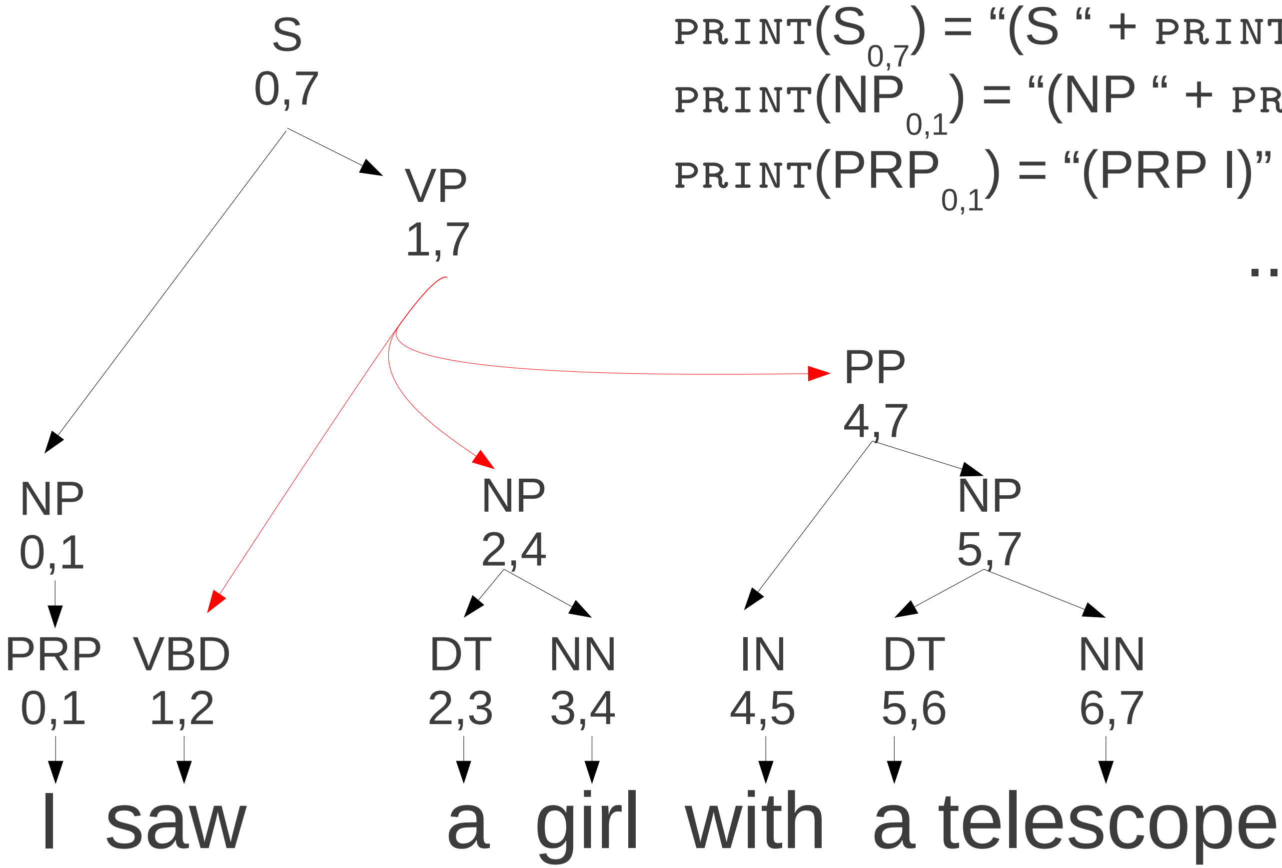
- Hypergraphs printed recursively, starting at top:

$\text{PRINT}(S_{0,7})$ = “(S “ + $\text{PRINT}(NP_{0,1})$ + “ “ + $\text{print}(VP_{1,7})$+”)”

$\text{PRINT}(NP_{0,1})$ = “(NP “ + $\text{PRINT}(PRP_{0,1})$ + ”)”

$\text{PRINT}(PRP_{0,1})$ = “(PRP I)”

...



Pseudo-Code

CKY Pseudo-Code: Read Grammar

```
# Read a grammar in format "lhs \t rhs \t prob \n"
make list nonterm # Make list of (lhs, rhs1, rhs2, prob)
make map preterm # Make a map preterm[rhs] = [ (lhs, prob) ...]
for rule in grammar_file
  split rule into lhs, rhs, prob (with "\t") # Rule P(lhs → rhs)=prob
  split rhs into rhs_symbols (with " ")
  if length(rhs) == 1:  # If this is a pre-terminal
    add (lhs, log(prob)) to preterm[rhs]
  else: # Otherwise, it is a non-terminal
    add (lhs, rhs[0], rhs[1], log(prob)) to nonterm
```

CKY Pseudo-Code: Add Pre-Terminals

```
split line into words
make map best_score # index: sym_{i,j} value = best log prob
make map best_edge  # index: sym_{i,j}  value = (lsym_{i,k}, rsym_{k,j})
# Add the pre-terminal sym
for i in 0 .. length(words)-1:
  for lhs, log_prob in preterm where P(lhs → words[i]) > 0:
    best_score[lhs_{i,i+1}] = [log_prob]
```

CKY Pseudo-Code: Combine Non-Terminals

```
for j in 2 .. length(words):  # j is right side of the span
  for i in j-2 .. 0:          # i is left side (Note: Reverse order!)
    for k in i+1 .. j-1:      # k is beginning of the second child
      # Try every grammar rule log(P(sym → lsym rsym)) = logprob
      for sym, lsym, rsym, logprob in nonterm:
        # Both children must have a probability
        if best_score[lsym_{i,k}] > -∞ and best_score[rsym_{k,j}] > -∞:
          # Find the log probability for this node/edge
          my_lp = best_score[lsym_{i,k}] + best_score[rsym_{k,j}] + logprob
          # If this is the best edge, update
          if my_lp > best_score[sym_{i,j}]:
            best_score[sym_{i,j}] = my_lp
            best_edge[sym_{i,j}] = (lsym_{i,k}, rsym_{k,j})
```

CKY Pseudo-Code: Print Tree

PRINT($S_{0,length(words)}$) # Print the “S” that spans all words

```
subroutine PRINT(sym_{i,j}):
  if sym_{i,j} exists in best_edge:   # for non-terminals
    return “(“+sym+” “
            + PRINT(best_edge[0]) + “ ” +
            + PRINT(best_edge[1]) + “)”
  else: # for terminals
    return “(“+sym+“ ”+words[i]+“)”
```

Exercise

Exercise

- Write cky.py
- Test the program
  - Input: `test/08-input.txt`
  - Grammar: `test/08-grammar.txt`
  - Answer: `test/08-output.txt`
- Run the program on actual data:
  - `data/wiki-en-test.grammar, data/wiki-en-short.tok`
- Visualize the trees
  - `script/print-trees.py < wiki-en-test.trees`
  - (Requires NLTK: http://nltk.org/)
- Challenge: think of a way to handle unknown words

Thank You!