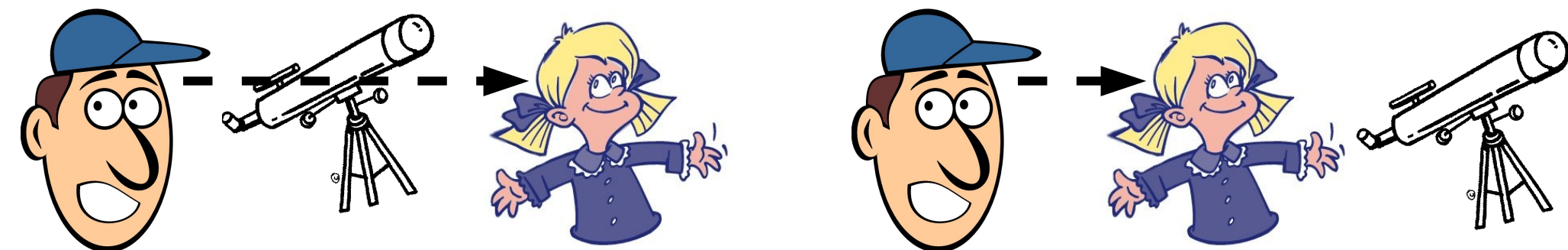


自然言語処理プログラミング勉強会 8 - 句構造解析

Graham Neubig
奈良先端科学技術大学院大学 (NAIST)

自然言語は曖昧性だらけ！

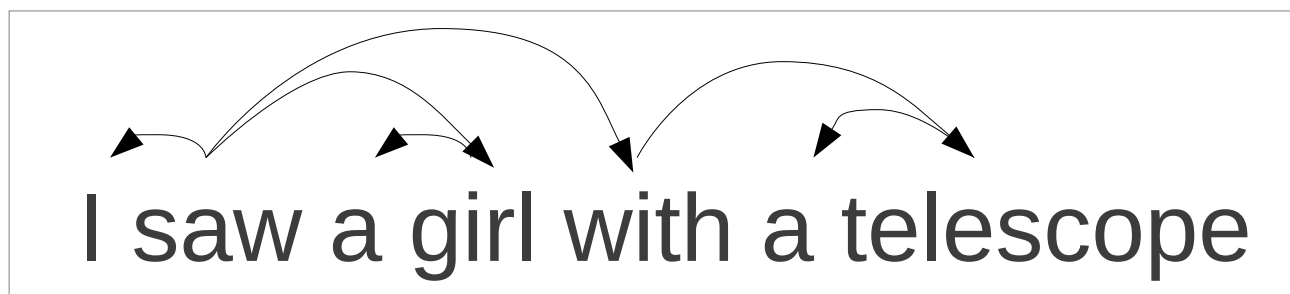
I saw a girl with a telescope



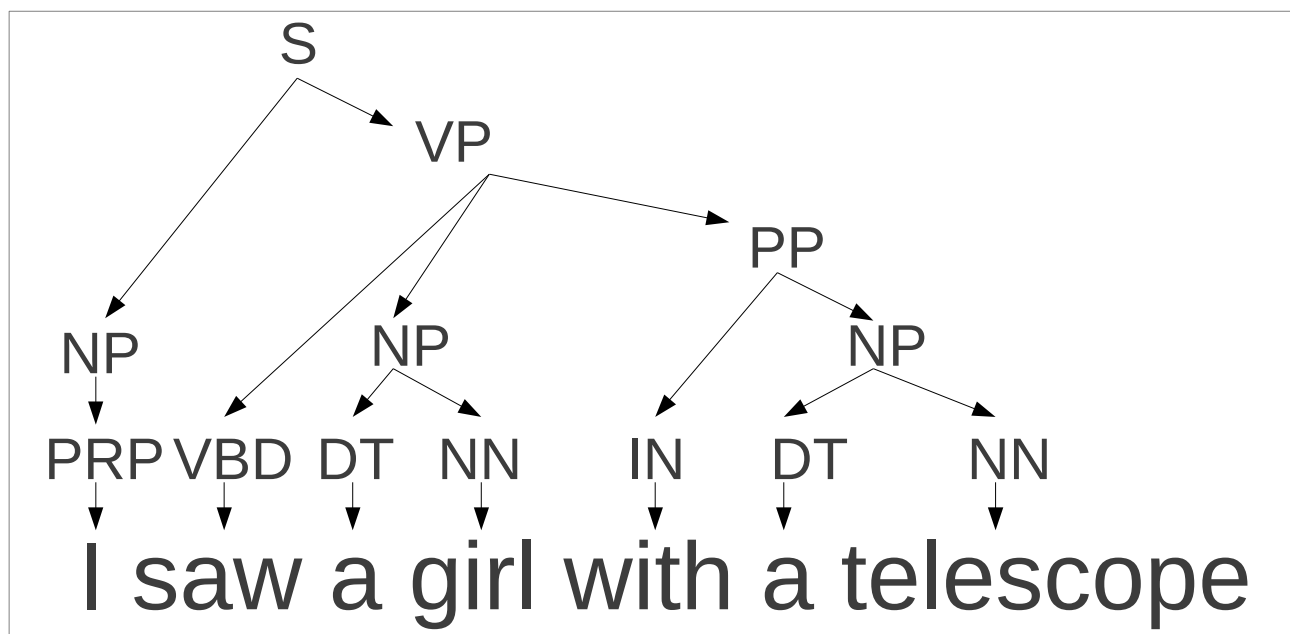
- 構文解析（パーズング）は構造的な曖昧性を解消

構文解析の種類

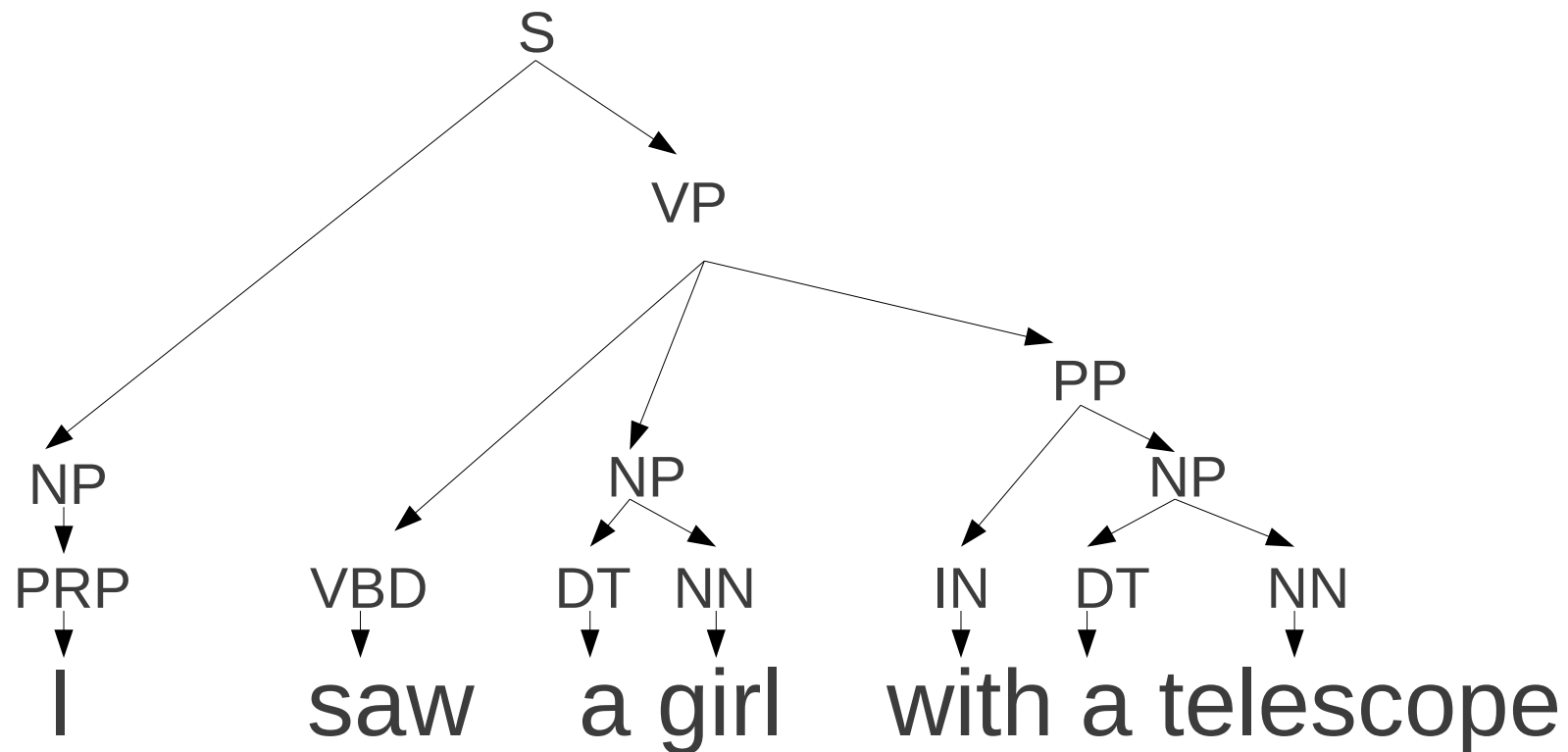
- 係り受け解析：単語と単語のつながりを重視



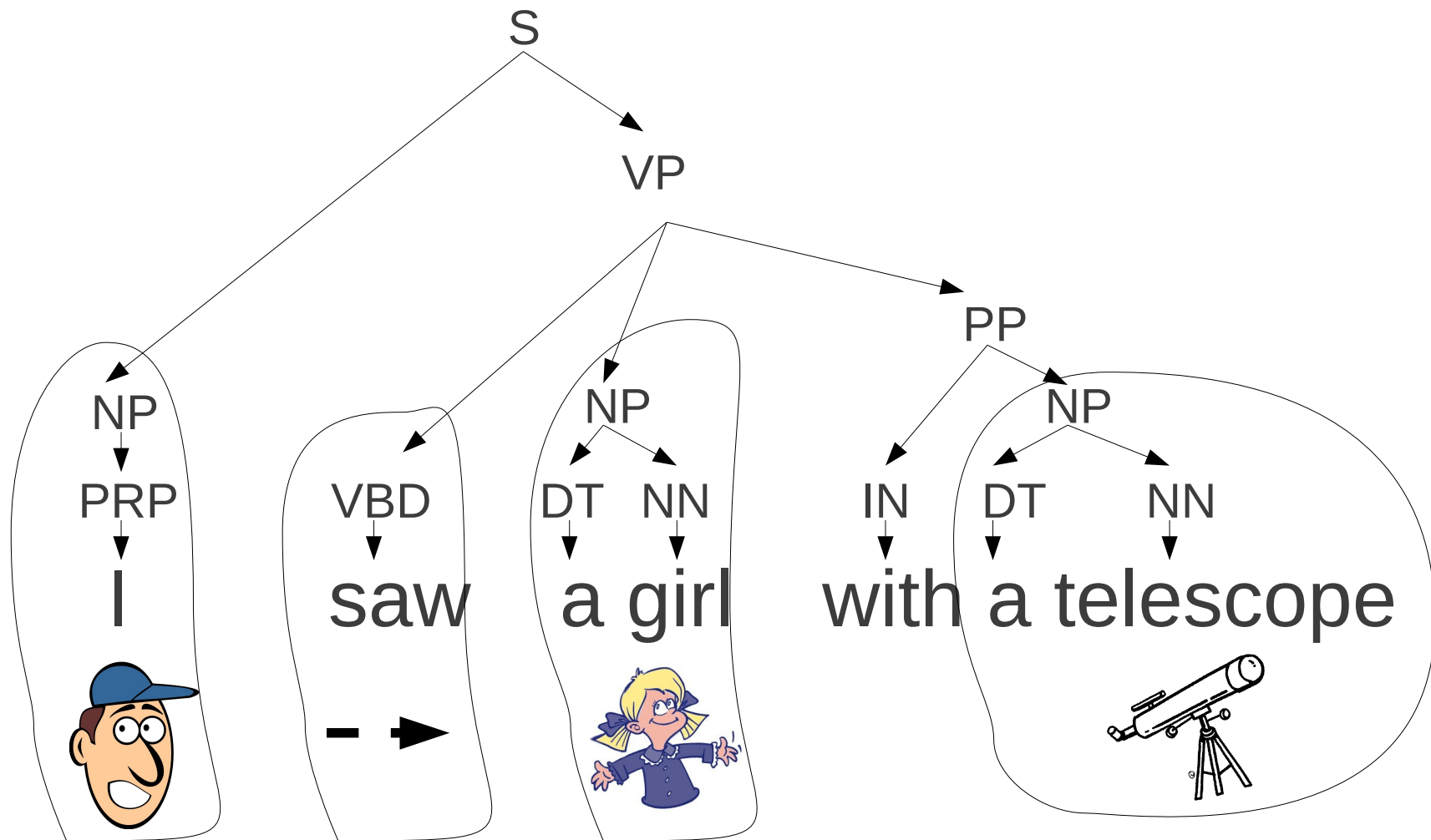
- 句構造解析：句とその再帰的な構造を重視



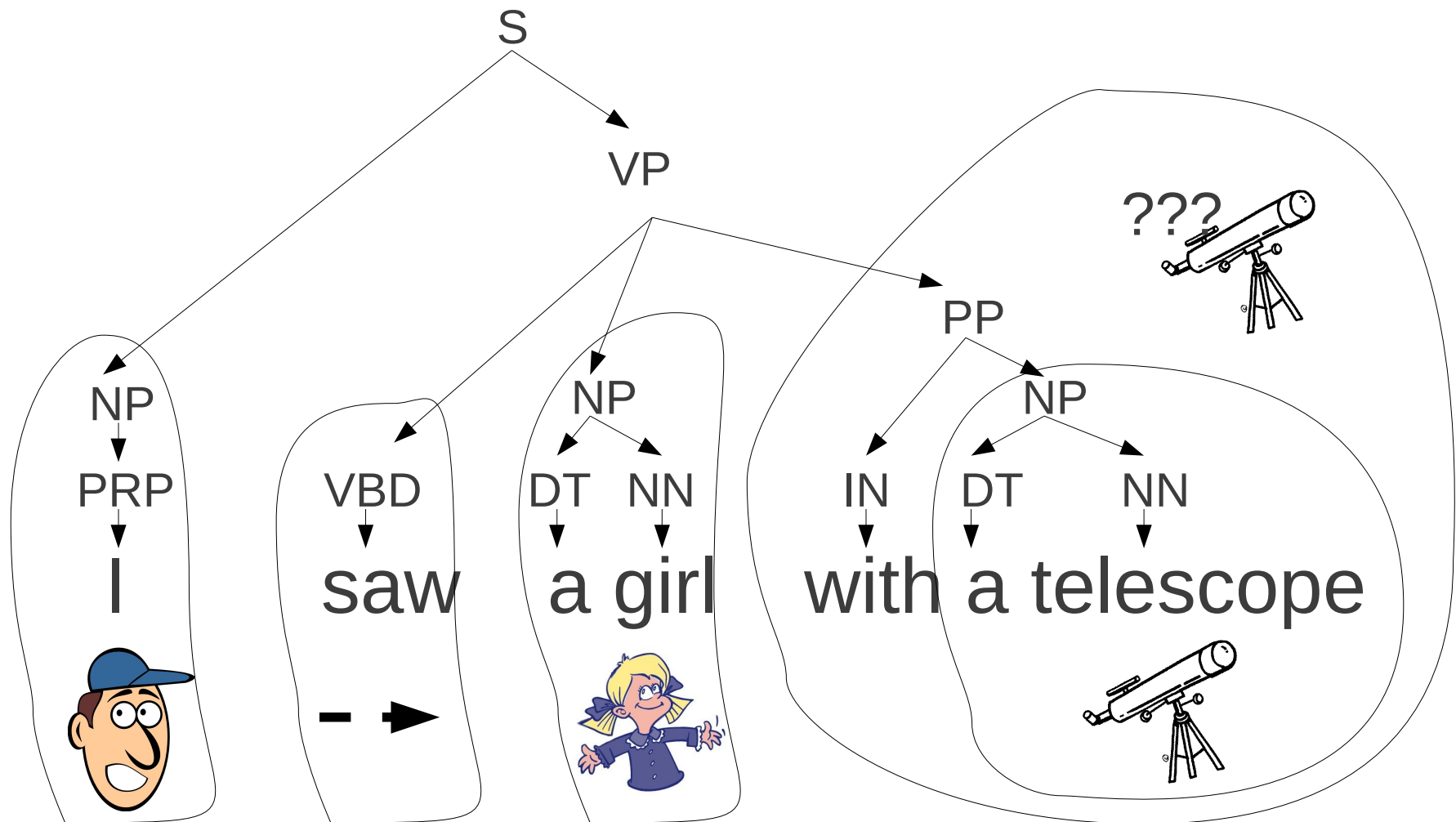
句の再帰的な構造



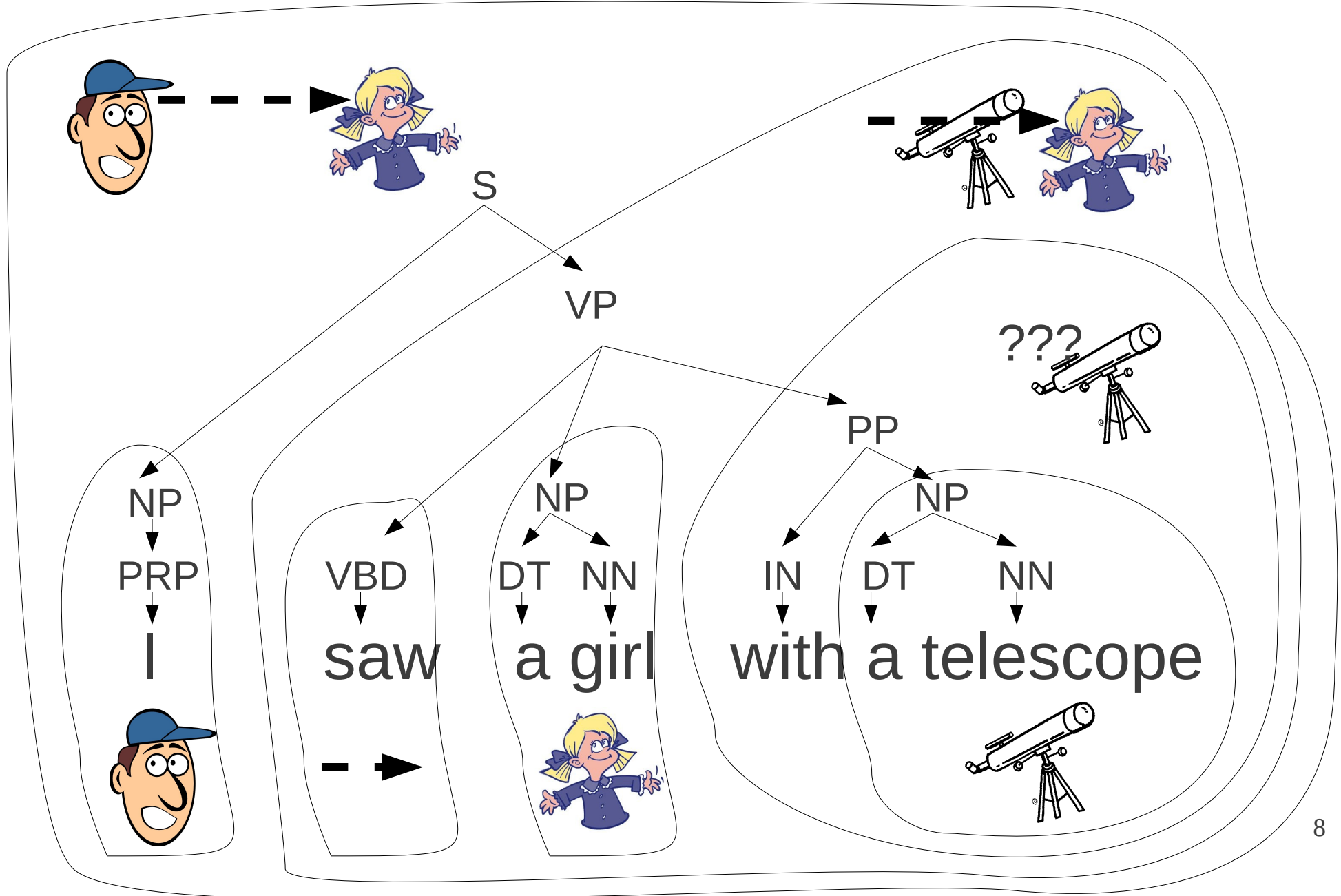
句の再帰的な構造



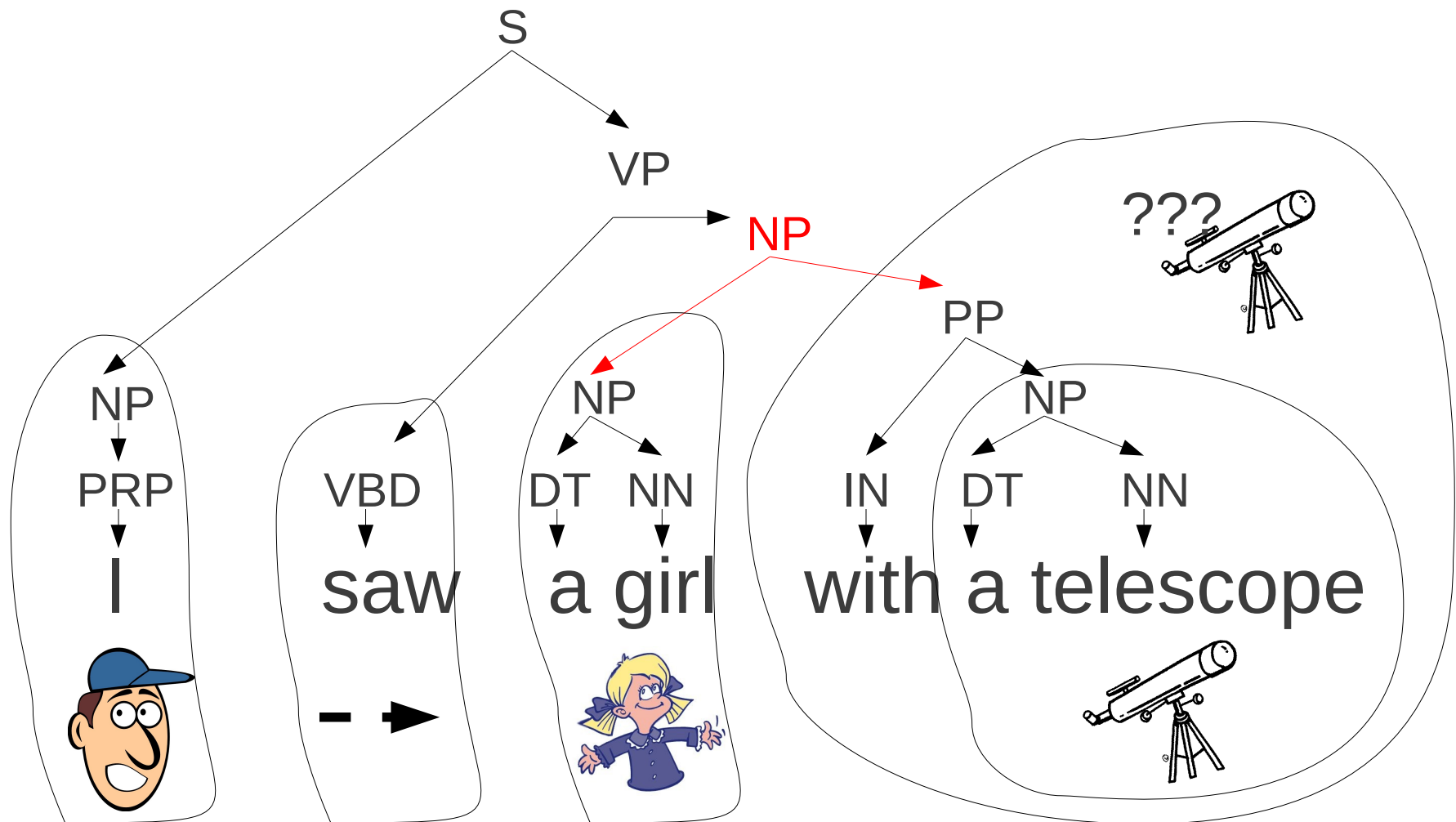
句の再帰的な構造



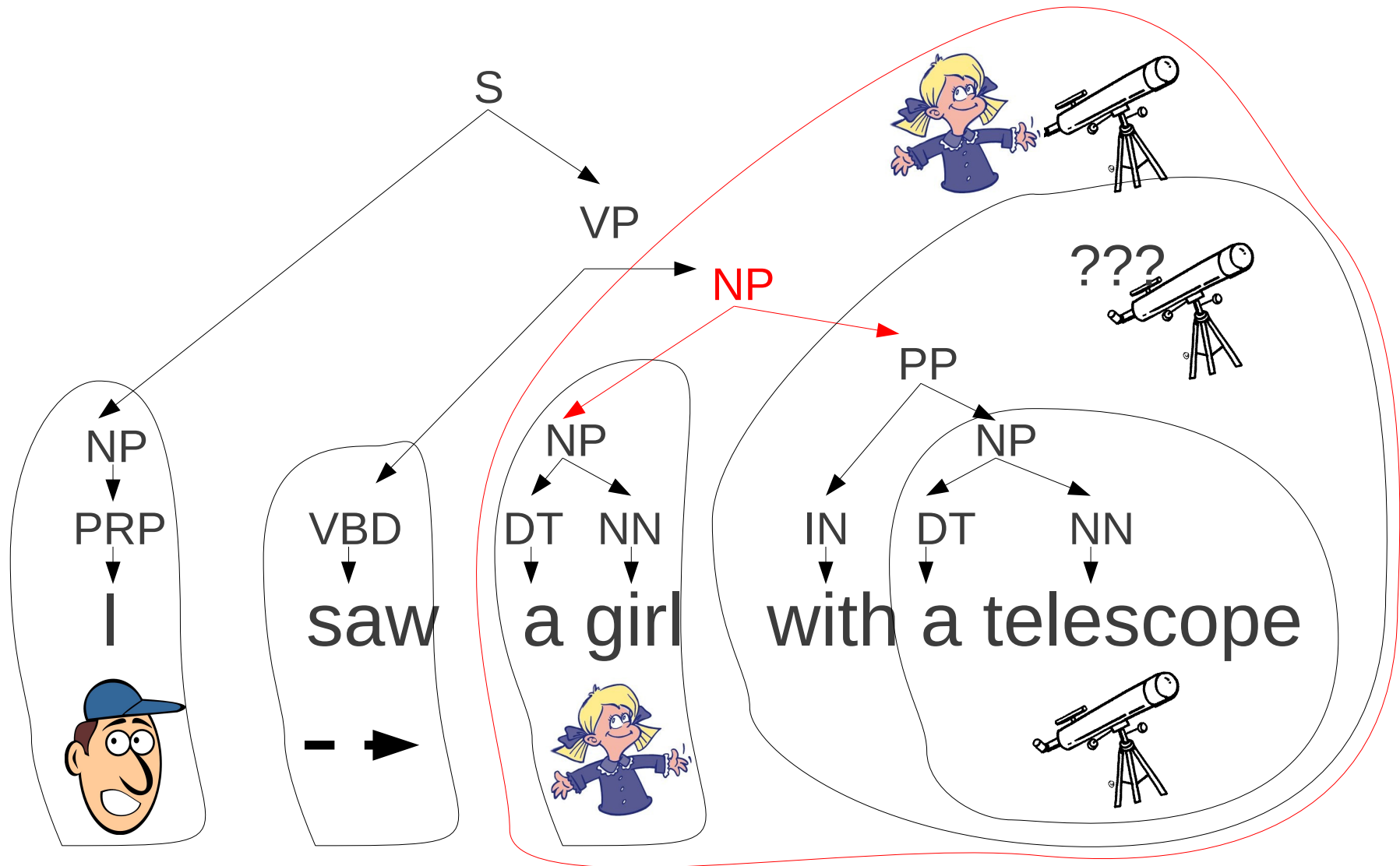
句の再帰的な構造



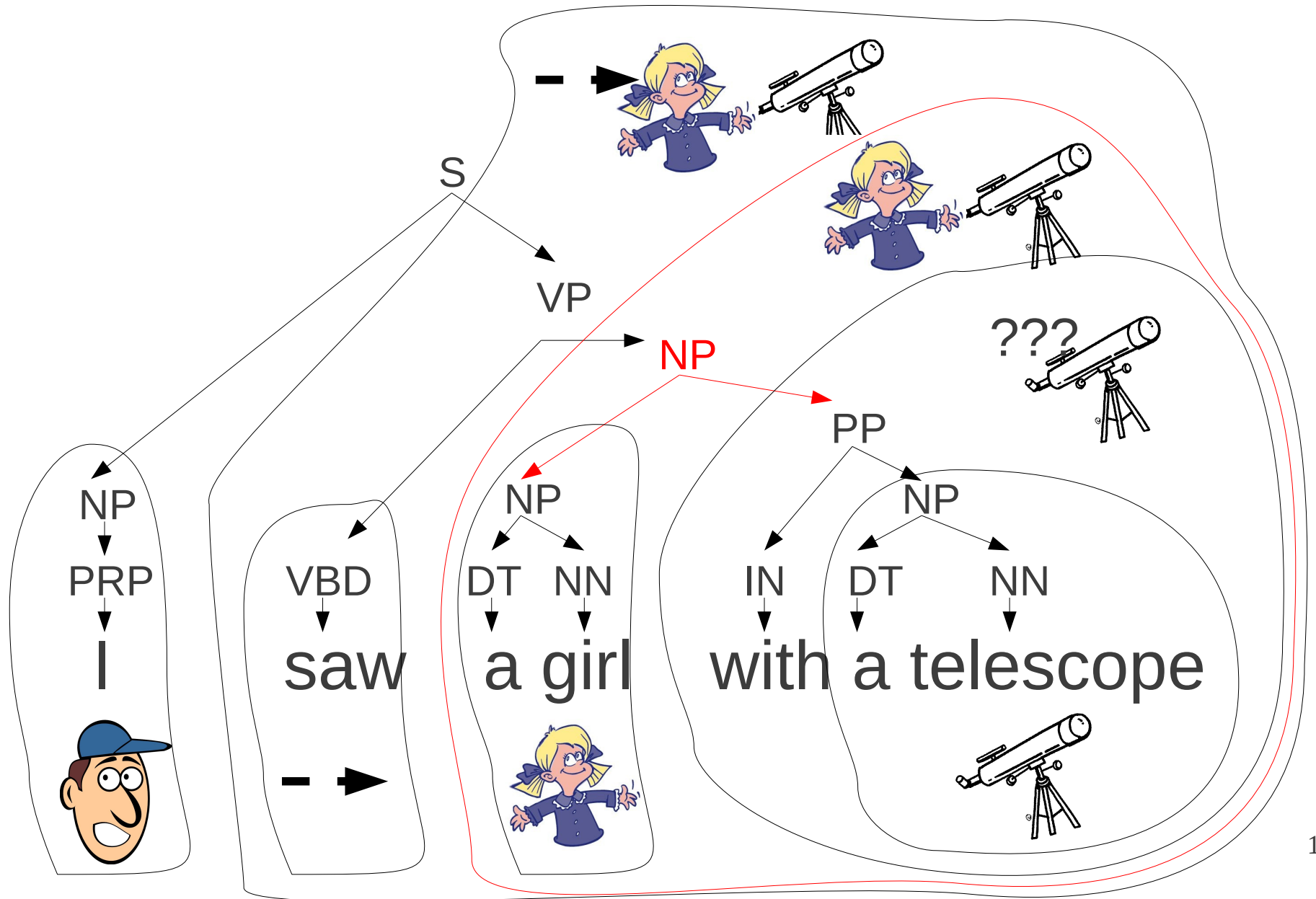
違う構造 → 違う解釈



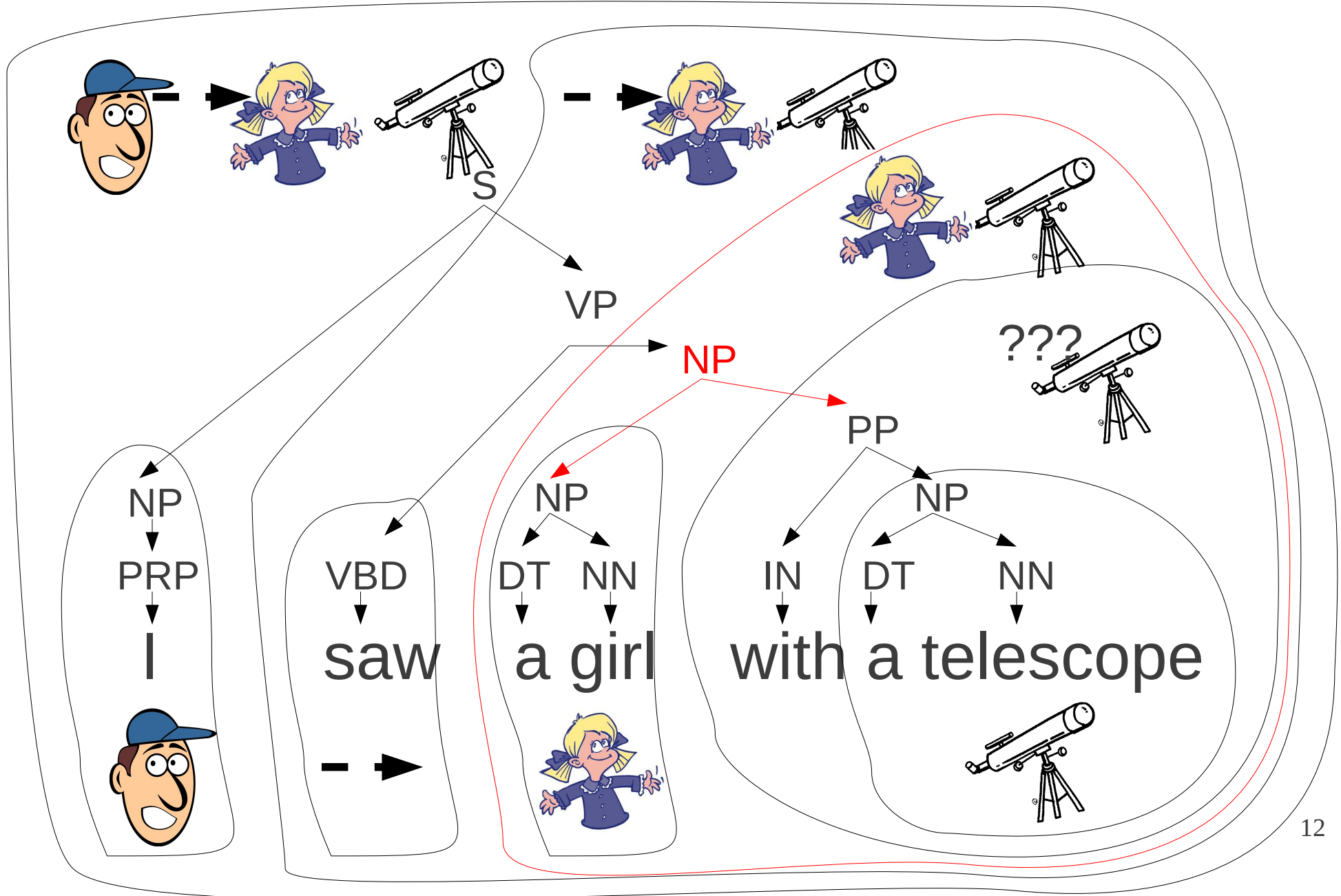
違う構造 → 違う解釈



違う構造 → 違う解釈

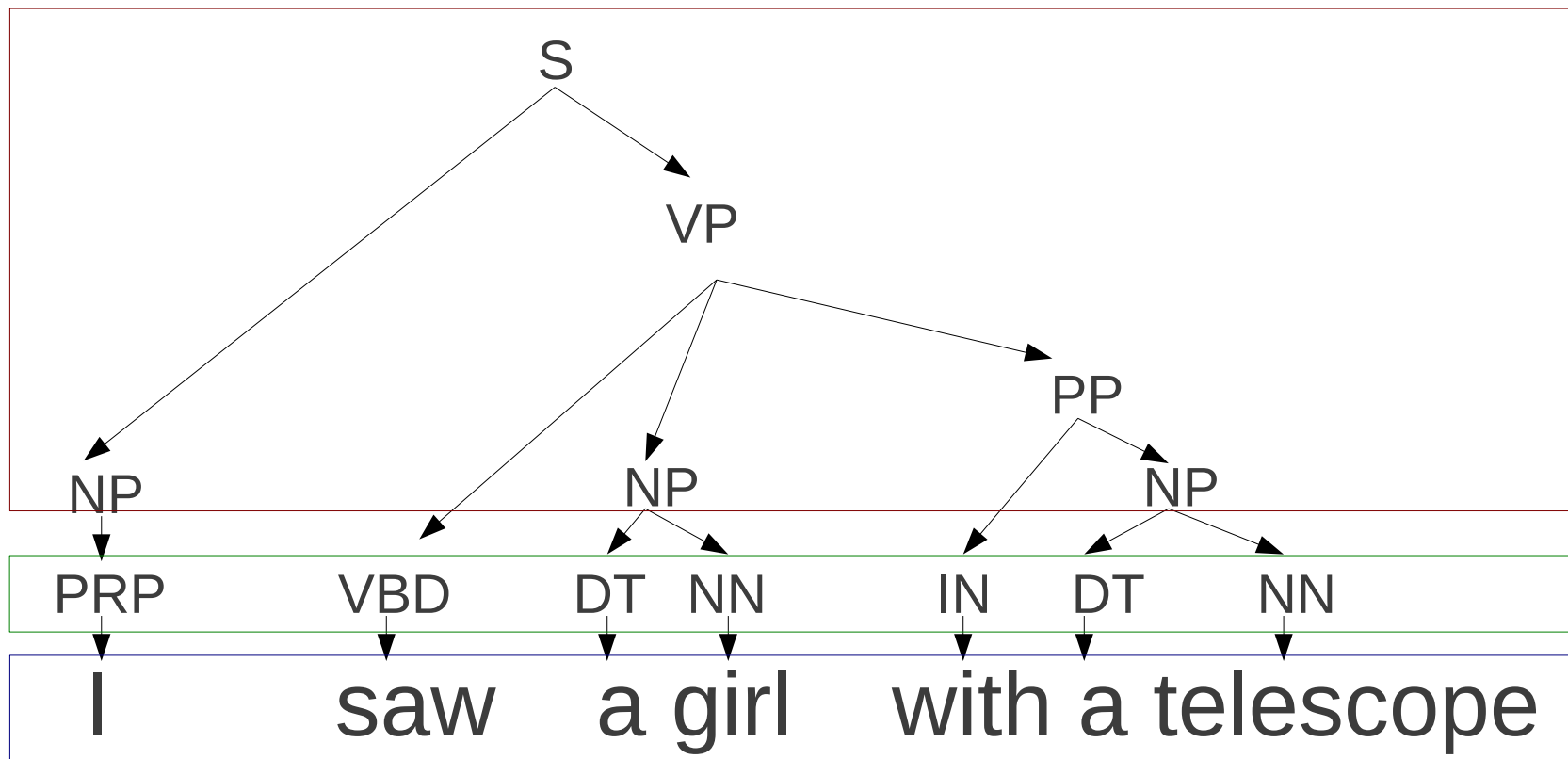


違う構造 → 違う解釈



非終端記号、前終端記号、終端記号

非終端記号

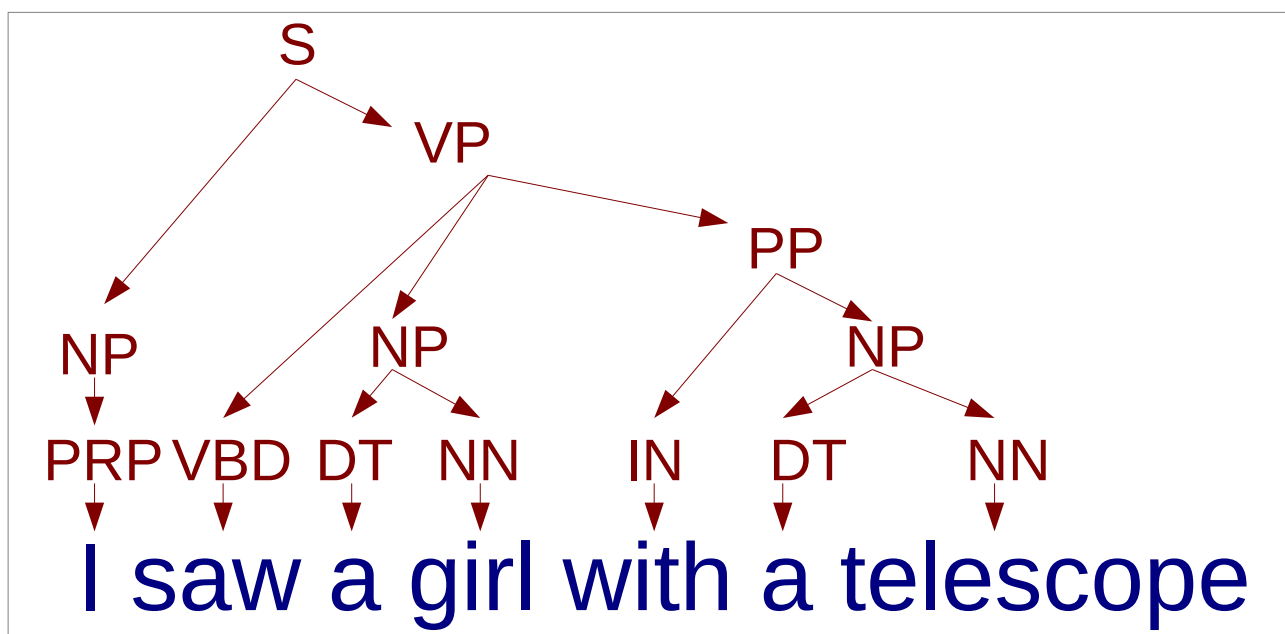


前終端記号

終端記号

予測問題としての構文解析

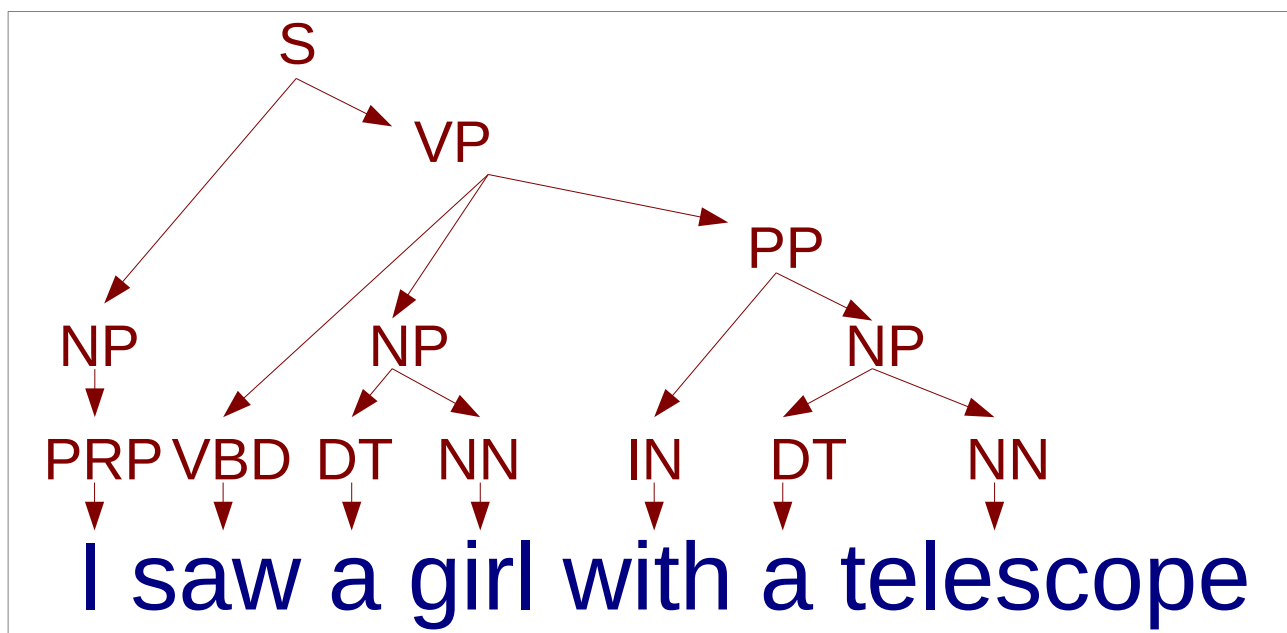
- 文 X が与えられ、構文木 Y を予測



- 「構造予測」の問題（品詞推定、単語分割と同様）

構文解析の確率モデル

- 文 X が与えられ、事後確率の最も高い構文木 Y を予測



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

生成モデル

- 構文木 Y と文 X が同時に確率モデルにより生成されたとする

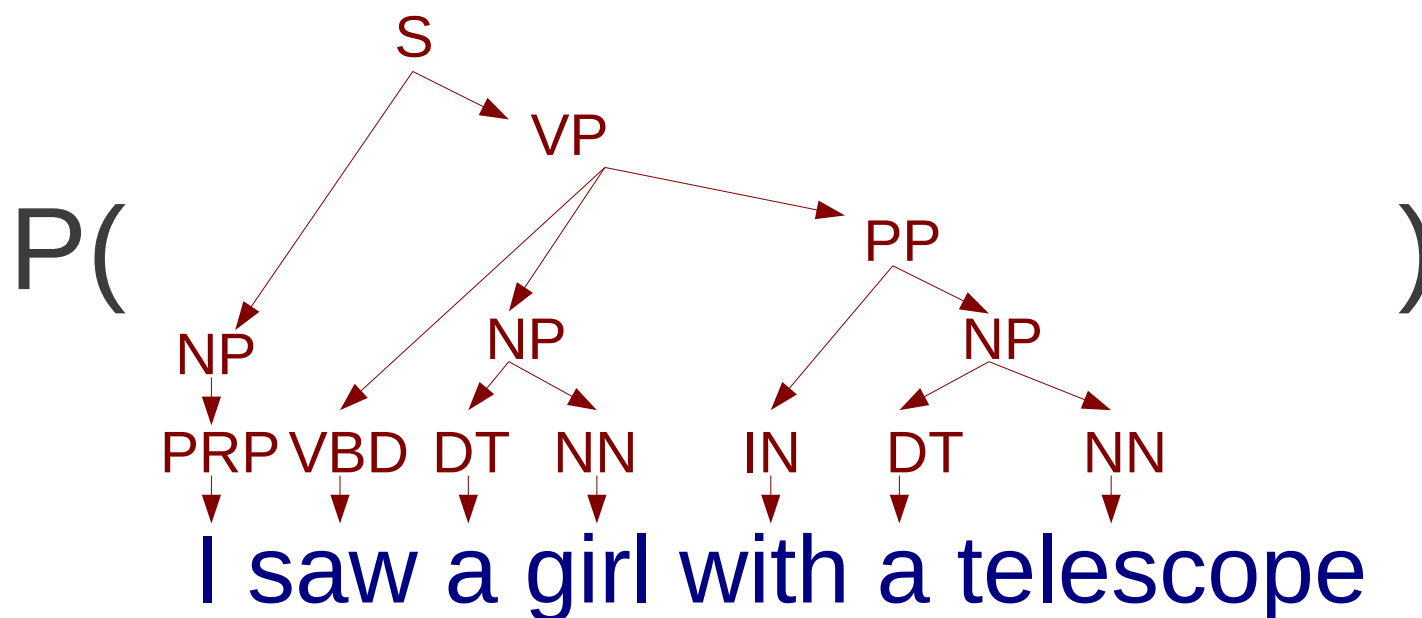
$$P(Y, X)$$

- X を固定すると、同時確率が最も高い Y は事後確率も最も高い

$$\operatorname{argmax}_Y P(Y|X) = \operatorname{argmax}_Y P(Y, X)$$

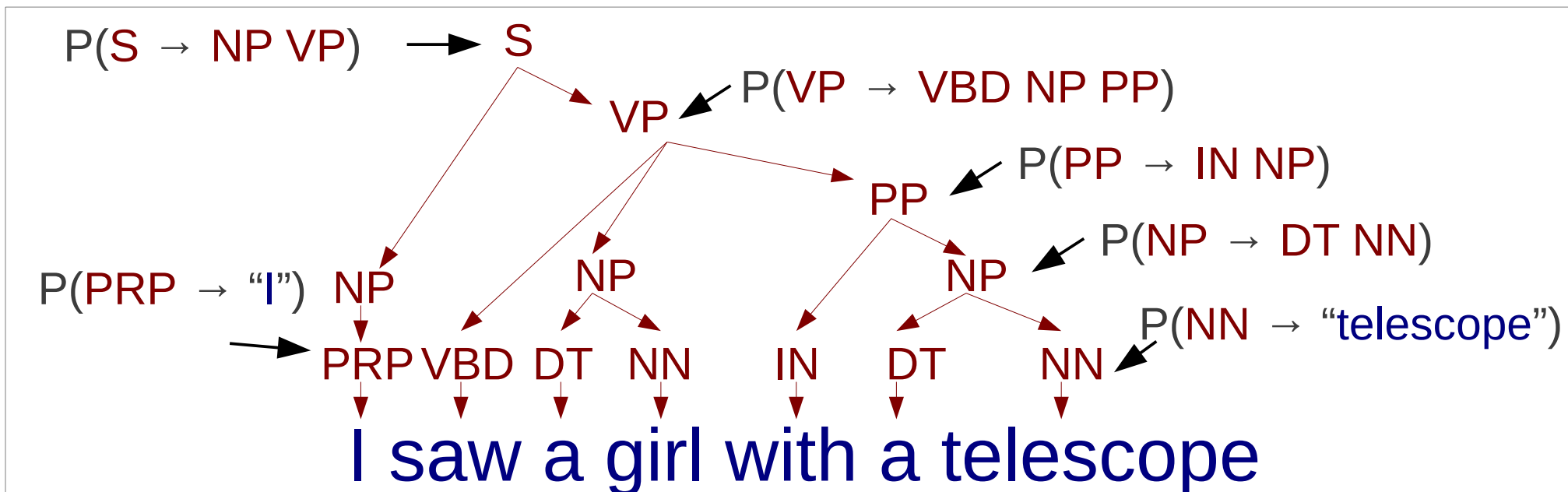
確率的文脈自由文法 (PCFG)

- 構文木の同時確率をどう定義するか？



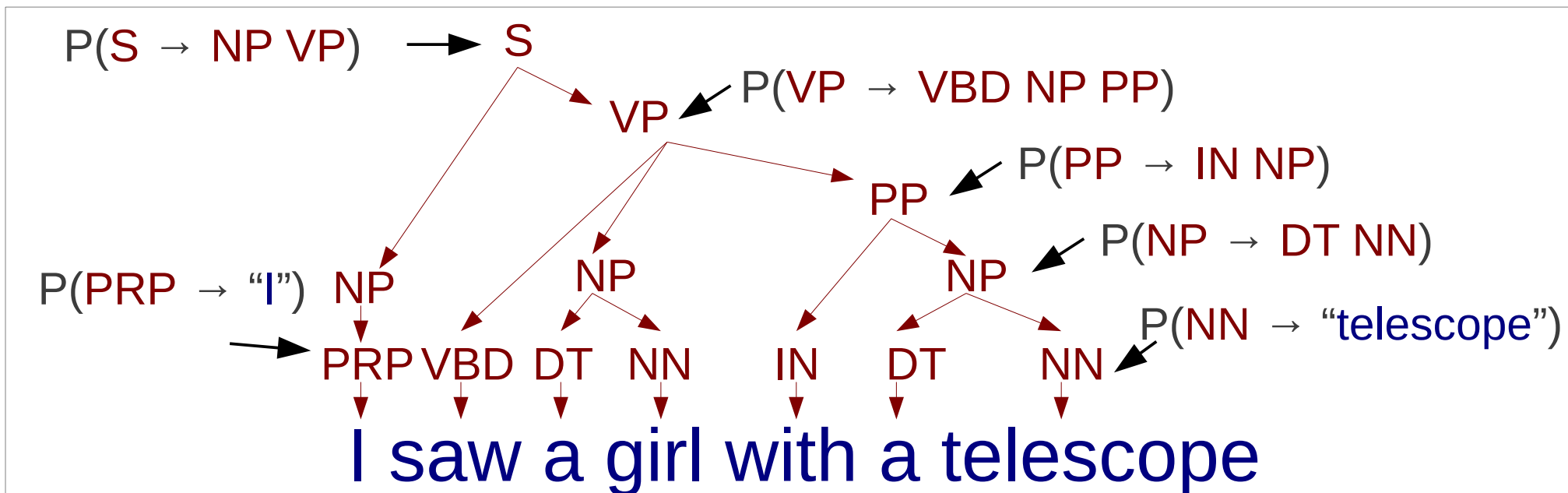
確率的文脈自由文法 (PCFG)

- PCFG : 各ノードの確率を個別に定義



確率的文脈自由文法 (PCFG)

- PCFG : 各ノードの確率を個別に定義



- 構文木の確率はノードの確率の積

$$\begin{aligned}
 &P(S \rightarrow NP VP) * P(NP \rightarrow PRP) * P(PRPR \rightarrow \text{"I"}) \\
 &* P(VP \rightarrow VBD NP PP) * P(VBD \rightarrow \text{"saw"}) * P(NP \rightarrow DT NN) \\
 &* P(DT \rightarrow \text{"a"}) * P(NN \rightarrow \text{"girl"}) * P(PP \rightarrow IN NP) * P(IN \rightarrow \text{"with"}) \\
 &* P(NP \rightarrow DT NN) * P(DT \rightarrow \text{"a"}) * P(NN \rightarrow \text{"telescope"})
 \end{aligned}$$

確率的構文解析

- 構文解析は確率が最大の構文木を探索すること

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

- ビタビアルゴリズムは利用可能か？

確率的構文解析

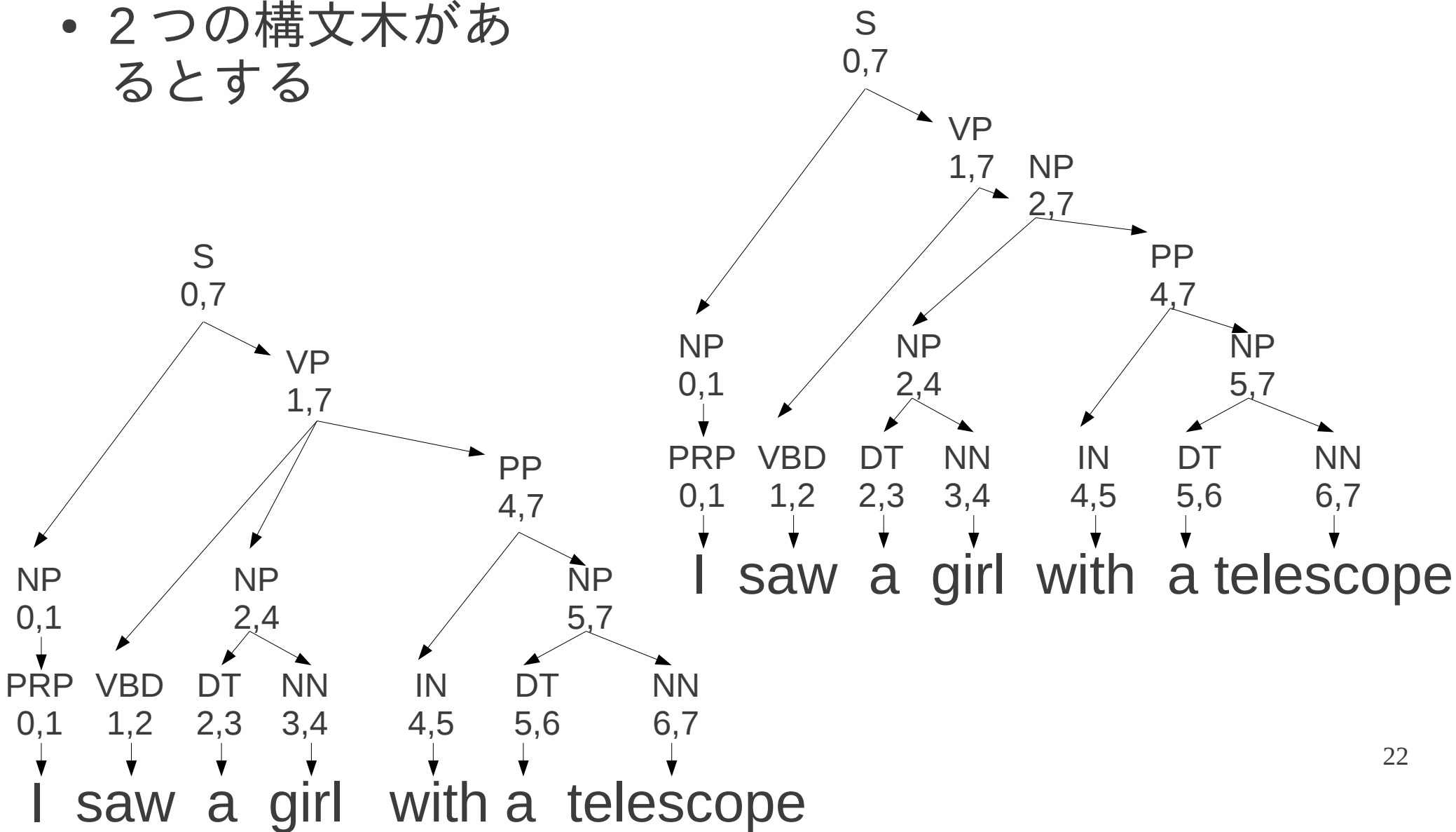
- 構文解析は確率が最大の構文木を探索すること

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

- ビタビアルゴリズムは利用可能か？
 - 答え：いいえ！
 - 理由：構文木の候補はグラフで表せず超グラフとなる

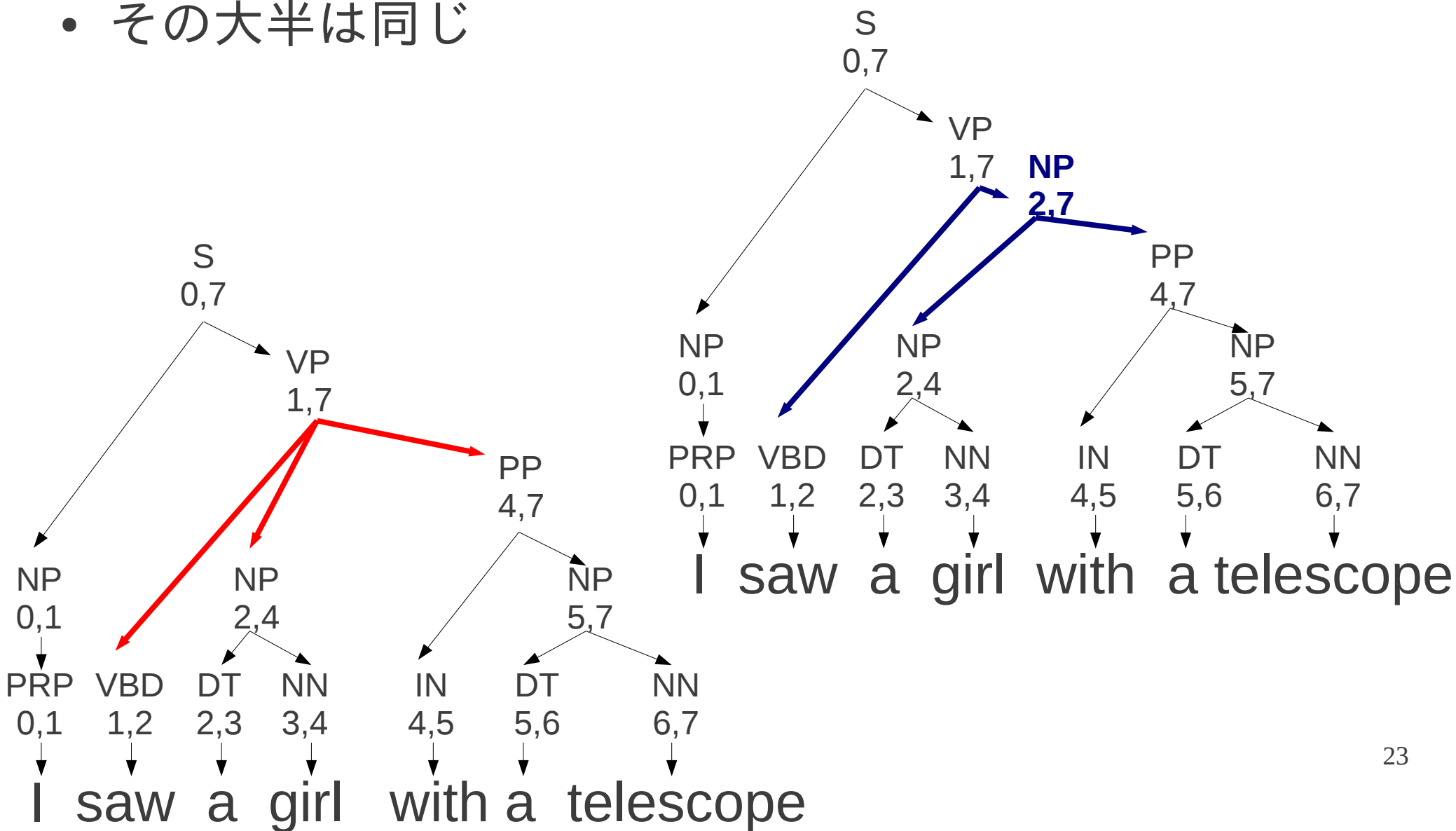
超グラフとは？

- 2つの構文木があるとする



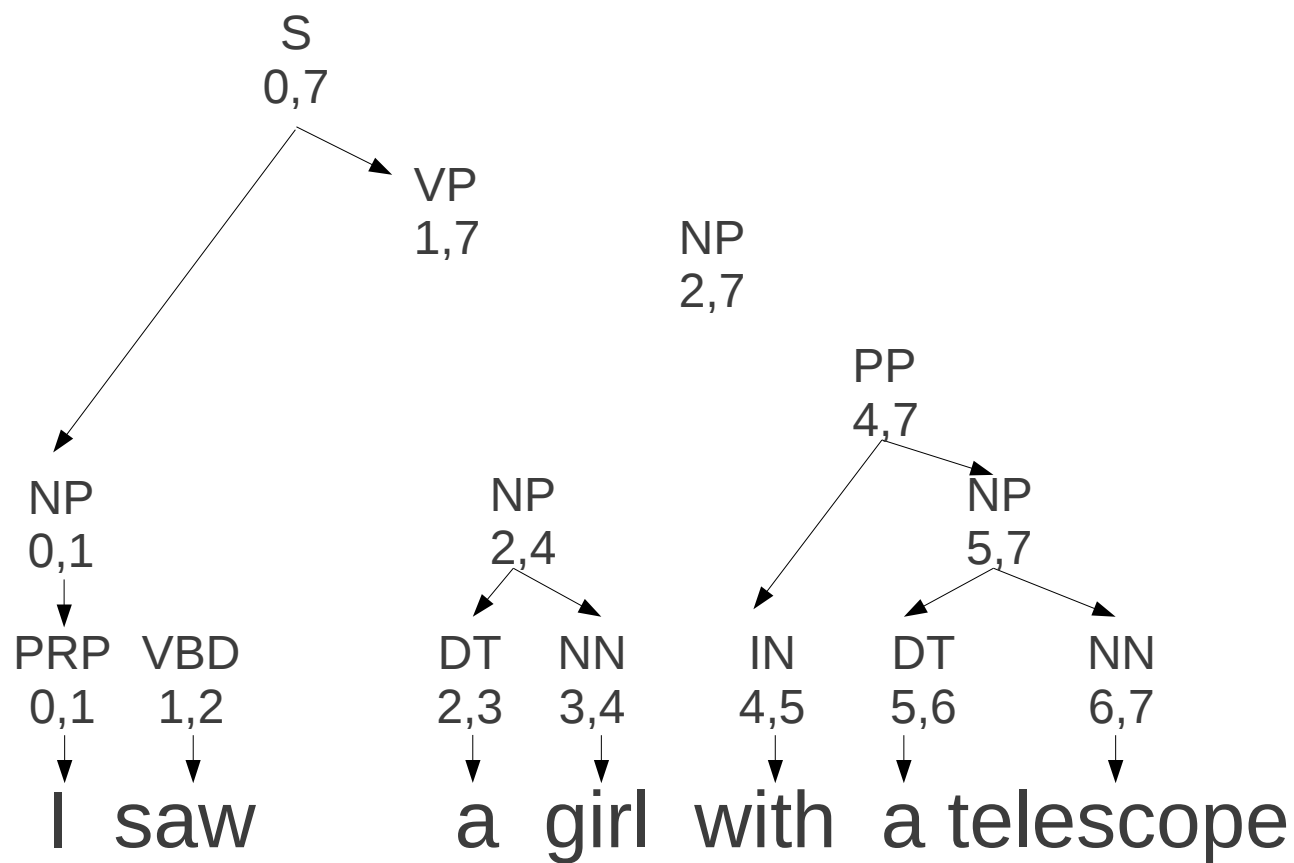
超グラフとは？

- その大半は同じ



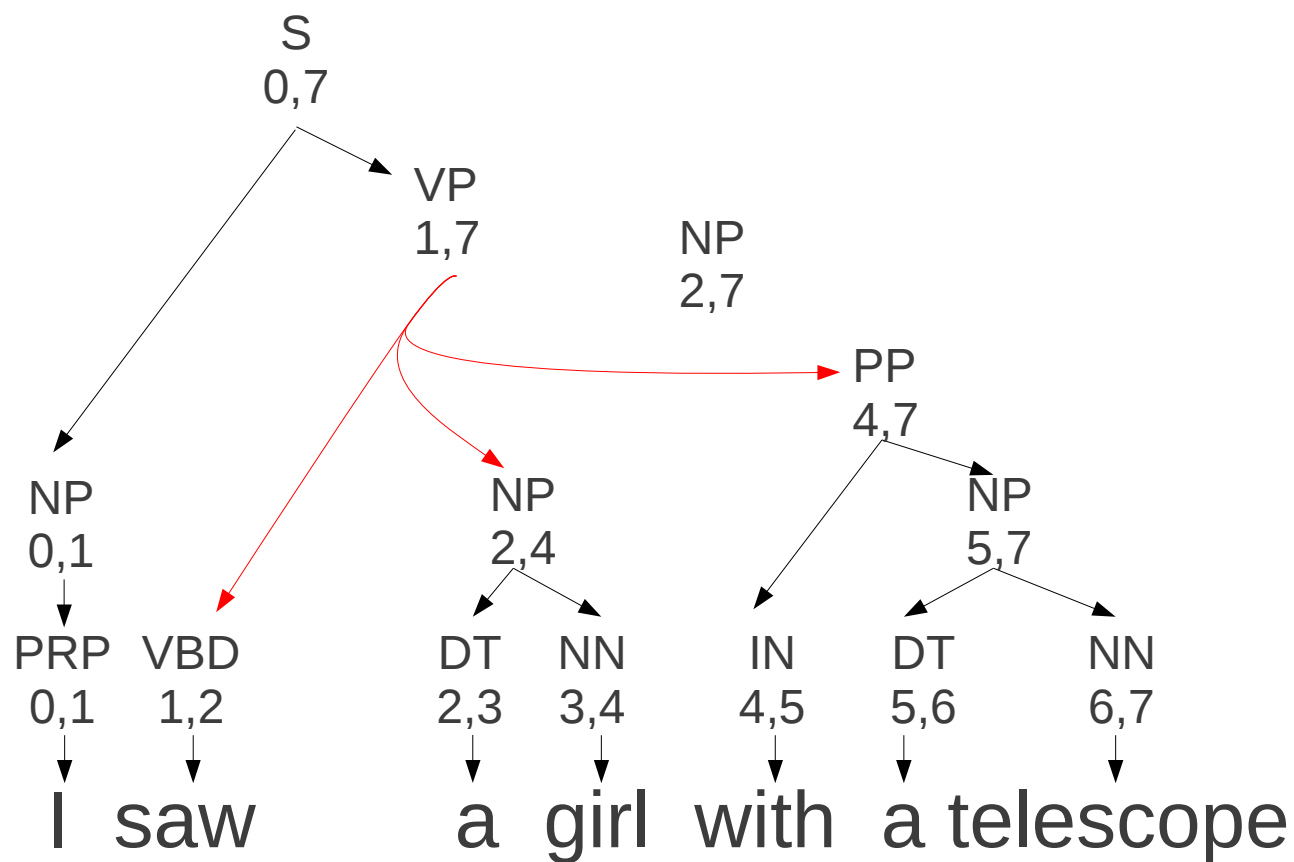
超グラフとは？

- 両方に現れるエッジだけを残すと：



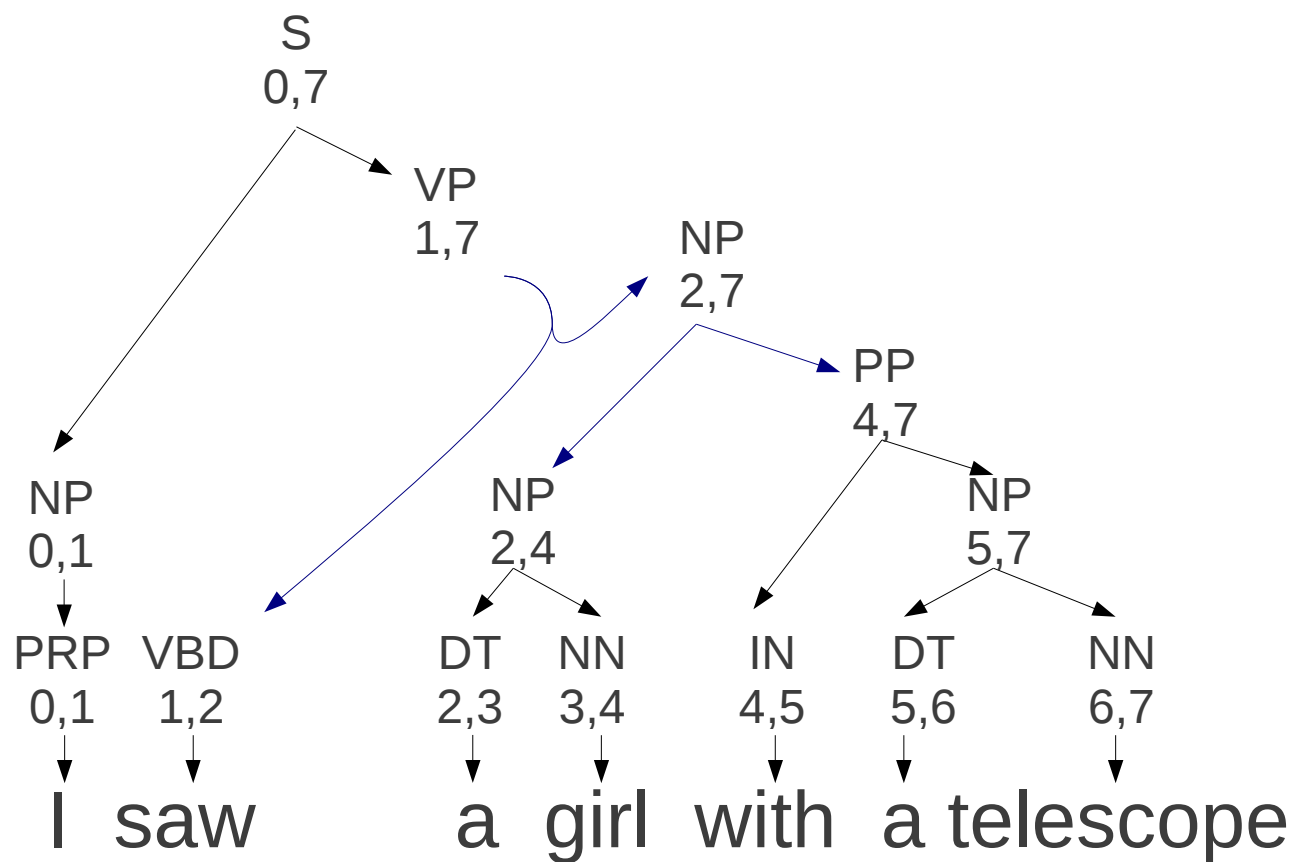
超グラフとは？

- **1 番目**の構文木のみが存在するエッジを追加：



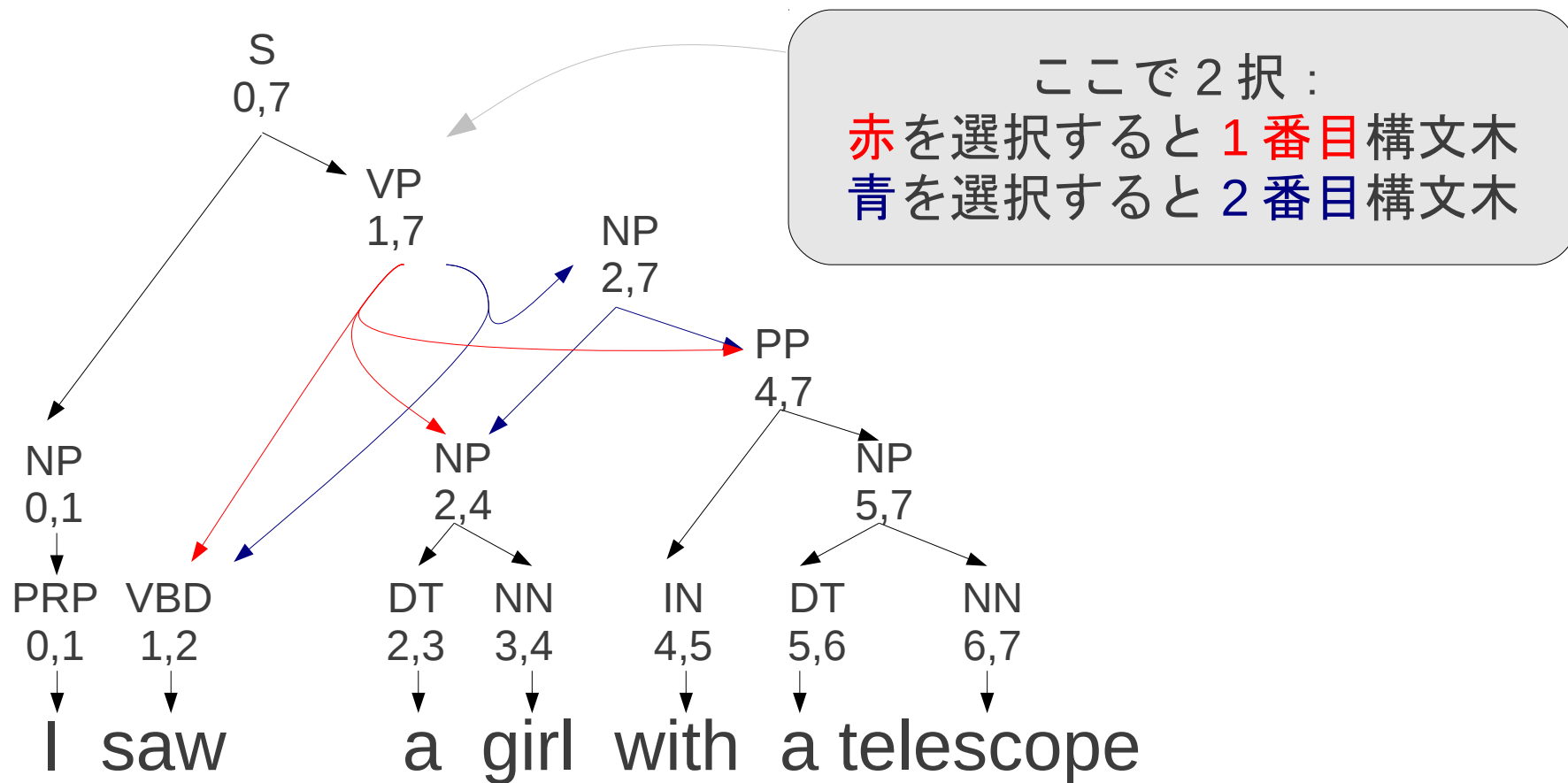
超グラフとは？

- 2 番目の構文木のみが存在するエッジを追加：



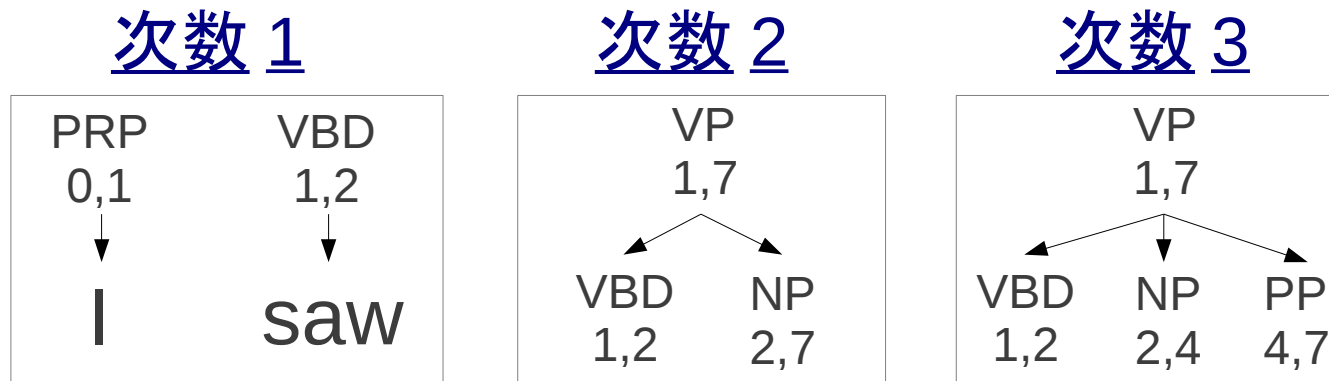
超グラフとは？

- 両方の構文木のみが存在するエッジを追加：



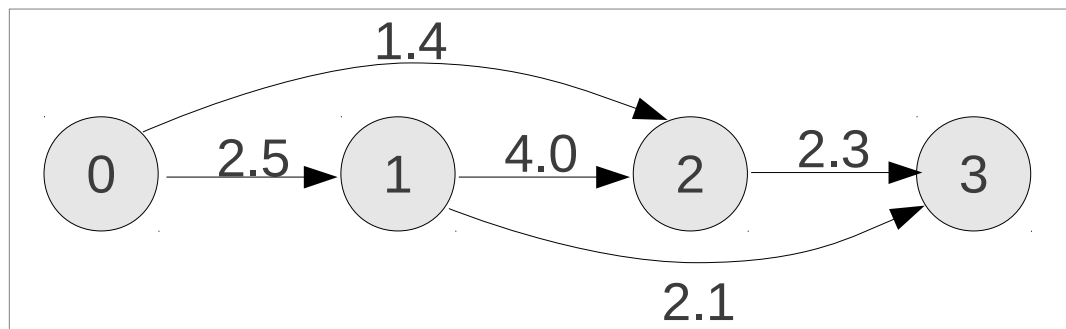
なぜ「超」グラフ？

- エッジの「**次数**」は子の数



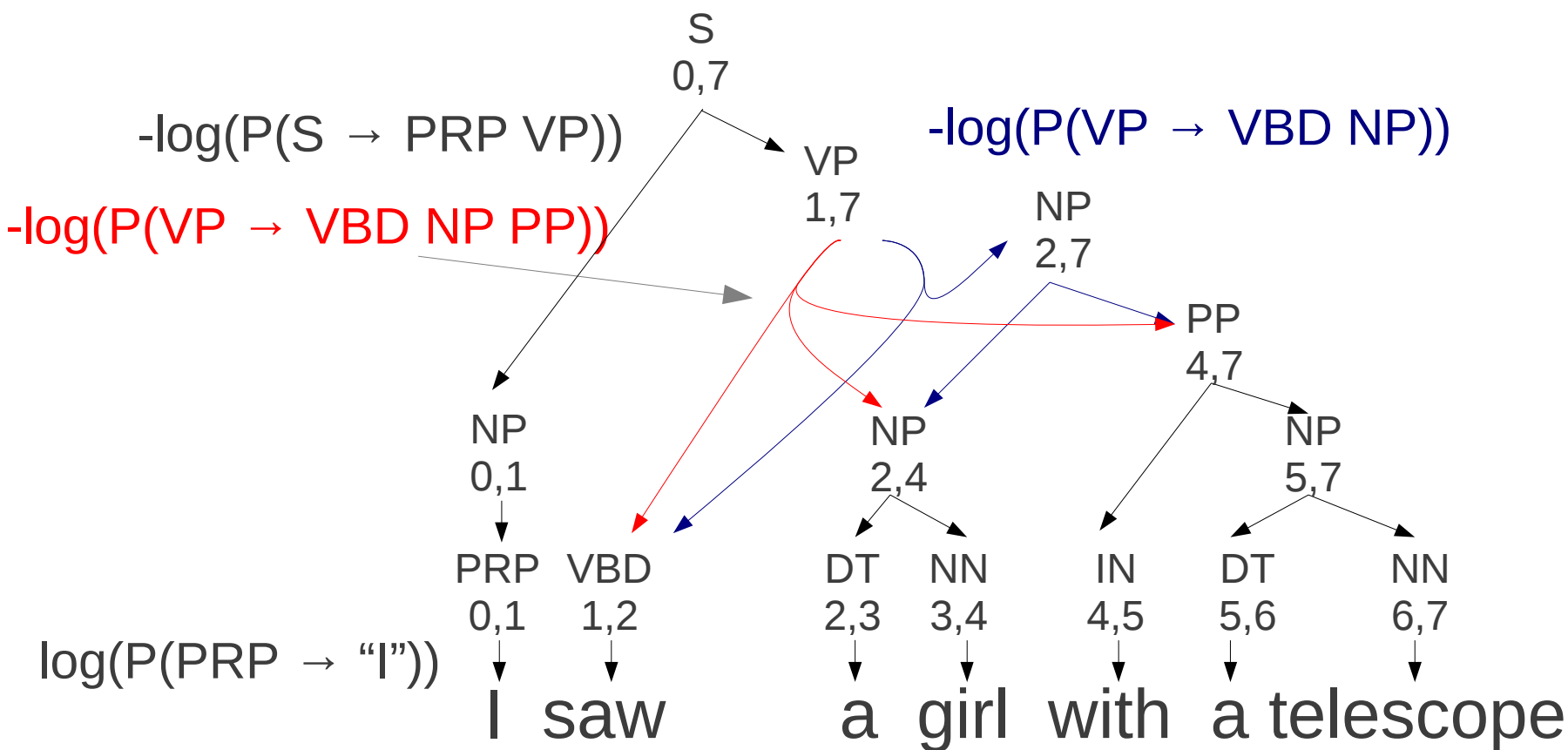
- 超グラフの次数は**エッジの次数の最大値**
- グラフは次数 1 の超グラフ！**

例 →



重み付き超グラフ

- グラフと同じく :
 - 超グラフのエッジに重みを付与
 - 負の対数確率 (ビタビアルゴリズムと同等の理由)



超グラフの探索法

- 構文解析 = 超グラフの最もスコアの小さい木を探索

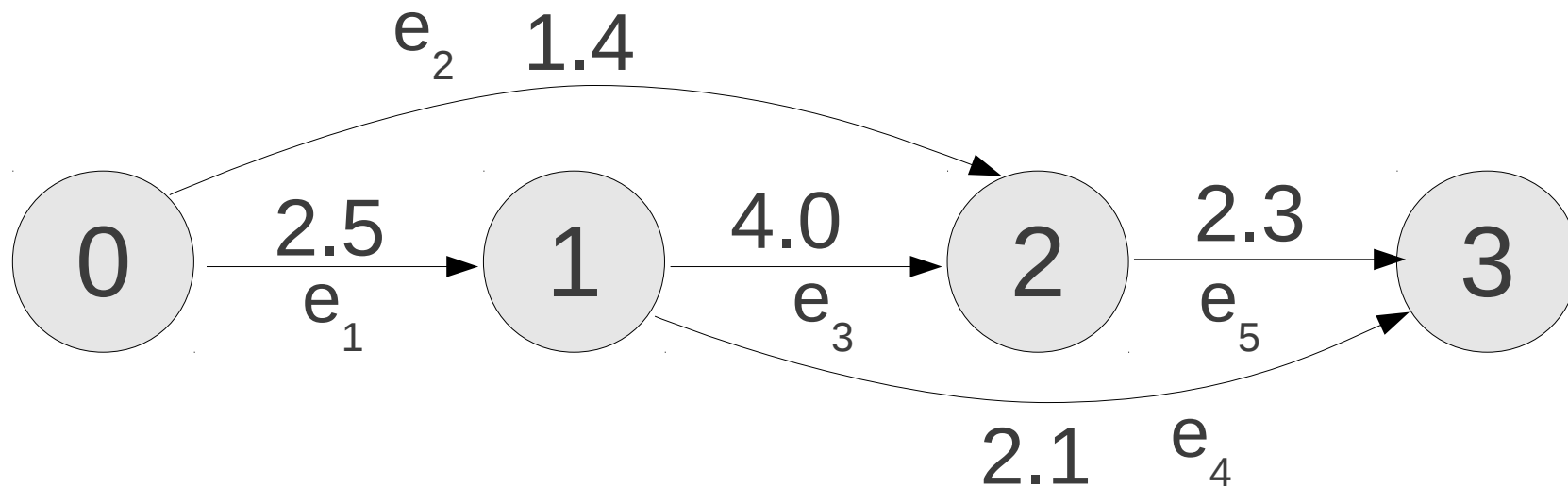
超グラフの探索法

- 構文解析 = 超グラフの最もスコアの小さい木を探索
- グラフではビタビアルゴリズムを利用
 - 前向きステップ: 各ノードまでの最短経路を計算
 - 後ろ向き: 最短経路を復元

超グラフの探索法

- 構文解析 = 超グラフの最もスコアの小さい木を探索
- グラフではビタビアルゴリズムを利用
 - 前向きステップ: 各ノードまでの最短経路を計算
 - 後ろ向き: 最短経路を復元
- 超グラフもほとんど同等のアルゴリズム
 - 内ステップ: 各ノードの最小部分木のスコアを計算
 - 外ステップ: スコア最小の木を復元

復習：ビタビアルゴリズム



$best_score[0] = 0$

for each *node* in the graph (昇順)

$best_score[node] = \infty$

for each incoming edge of *node*

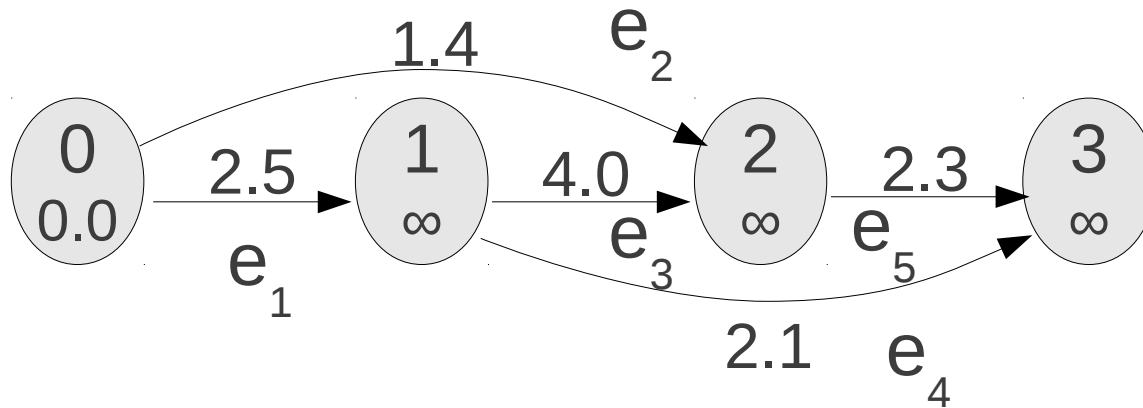
$score = best_score[edge.prev_node] + edge.score$

if $score < best_score[node]$

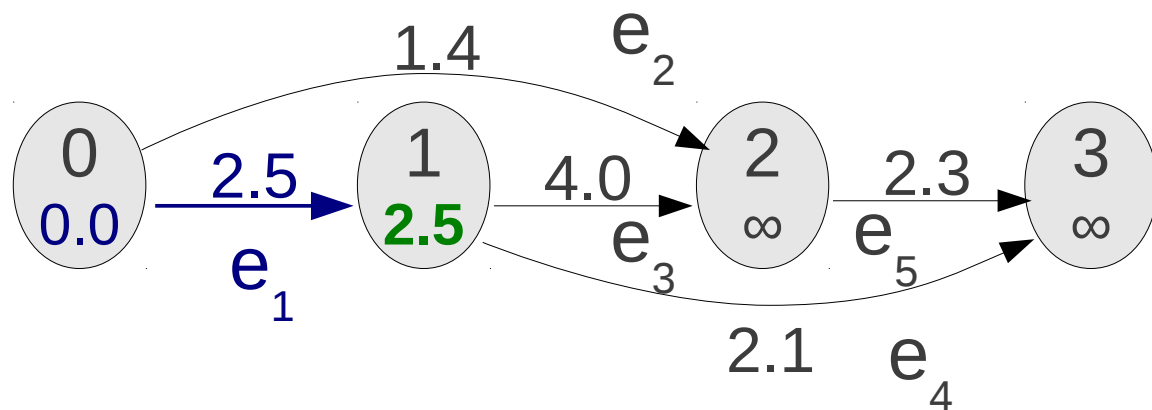
$best_score[node] = score$

$best_edge[node] = edge$

例：

初期化： $\text{best_score}[0] = 0$

例：

初期化：

$$\text{best_score}[0] = 0$$

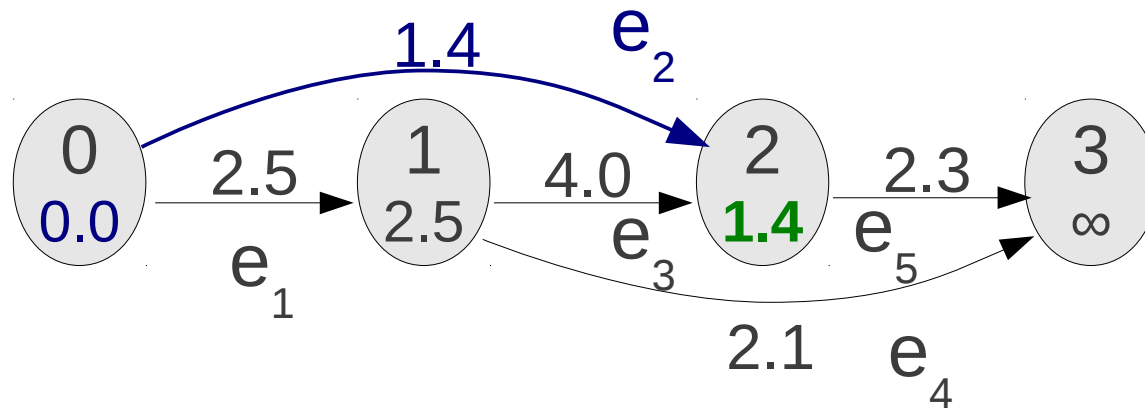
 e_1 を計算：

$$\text{score} = 0 + 2.5 = 2.5 (< \infty)$$

$$\text{best_score}[1] = 2.5$$

$$\text{best_edge}[1] = e_1$$

例：

初期化：

$$\text{best_score}[0] = 0$$

 e_1 を計算：

$$\text{score} = 0 + 2.5 = 2.5 (< \infty)$$

$$\text{best_score}[1] = 2.5$$

$$\text{best_edge}[1] = e_1$$

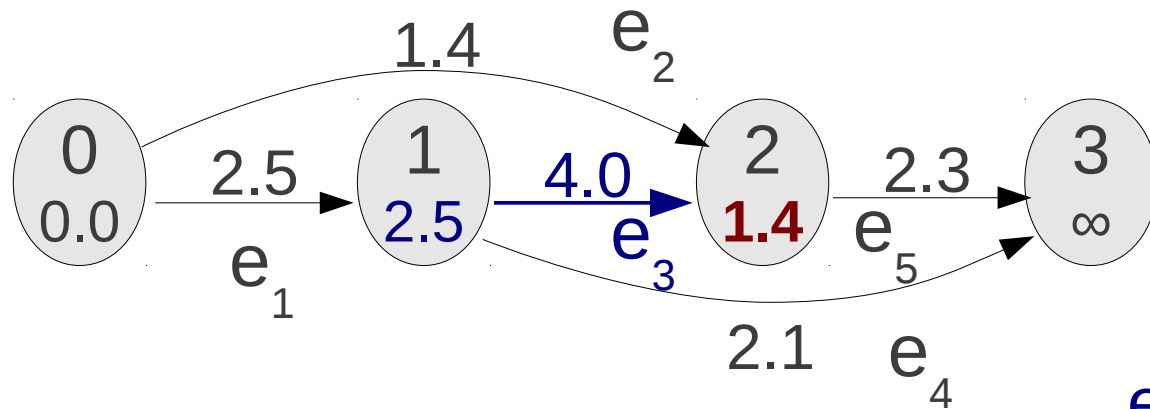
 e_2 を計算：

$$\text{score} = 0 + 1.4 = 1.4 (< \infty)$$

$$\text{best_score}[2] = 1.4$$

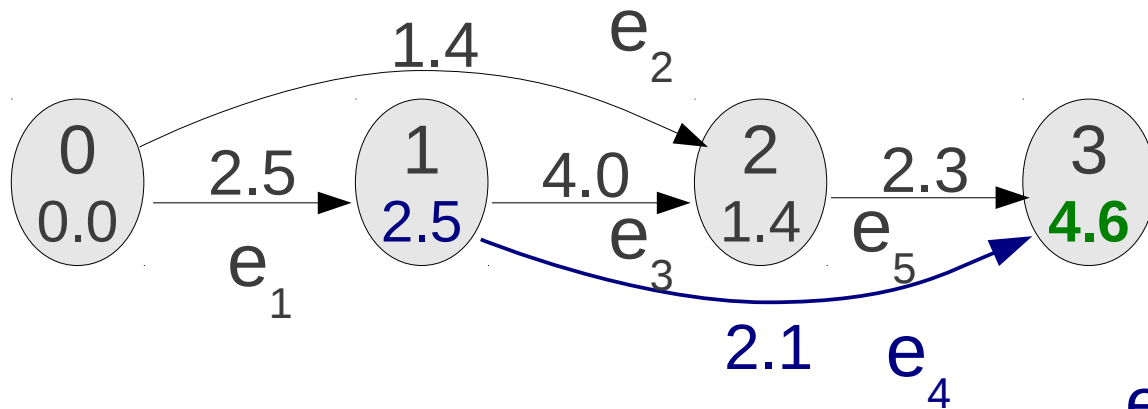
$$\text{best_edge}[2] = e_2$$

例：

初期化： $\text{best_score}[0] = 0$ e_1 を計算： $\text{score} = 0 + 2.5 = 2.5 (< \infty)$ $\text{best_score}[1] = 2.5$ $\text{best_edge}[1] = e_1$ e_2 を計算： $\text{score} = 0 + 1.4 = 1.4 (< \infty)$ $\text{best_score}[2] = 1.4$ $\text{best_edge}[2] = e_2$ e_3 を計算： $\text{score} = 2.5 + 4.0 = 6.5 (> 1.4)$

変更なし！

例：



初期化：

$$\text{best_score}[0] = 0$$
 e_1 を計算：
$$\text{score} = 0 + 2.5 = 2.5 (< \infty)$$

$$\text{best_score}[1] = 2.5$$

$$\text{best_edge}[1] = e_1$$
 e_2 を計算：
$$\text{score} = 0 + 1.4 = 1.4 (< \infty)$$

$$\text{best_score}[2] = 1.4$$

$$\text{best_edge}[2] = e_2$$
 e_3 を計算：
$$\text{score} = 2.5 + 4.0 = 6.5 (> 1.4)$$

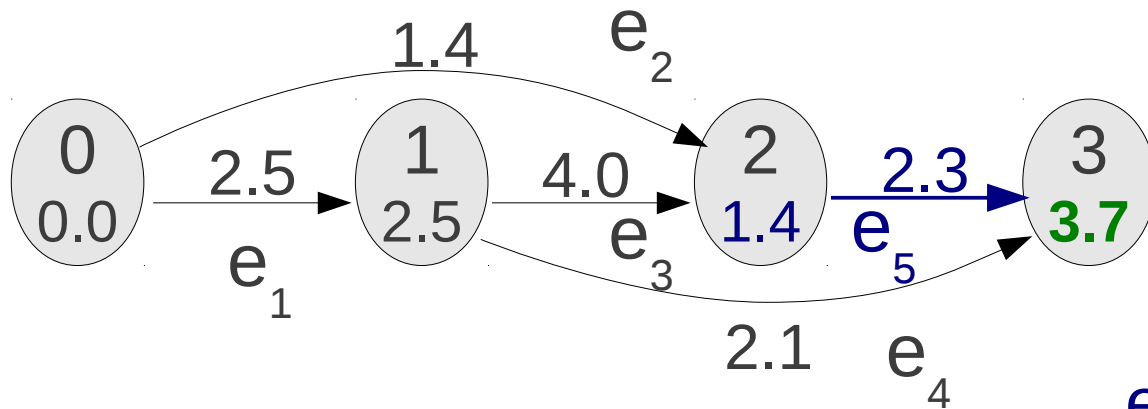
変更なし！

 e_4 を計算：
$$\text{score} = 2.5 + 2.1 = 4.6 (< \infty)$$

$$\text{best_score}[3] = 4.6$$

$$\text{best_edge}[3] = e_4$$

例：



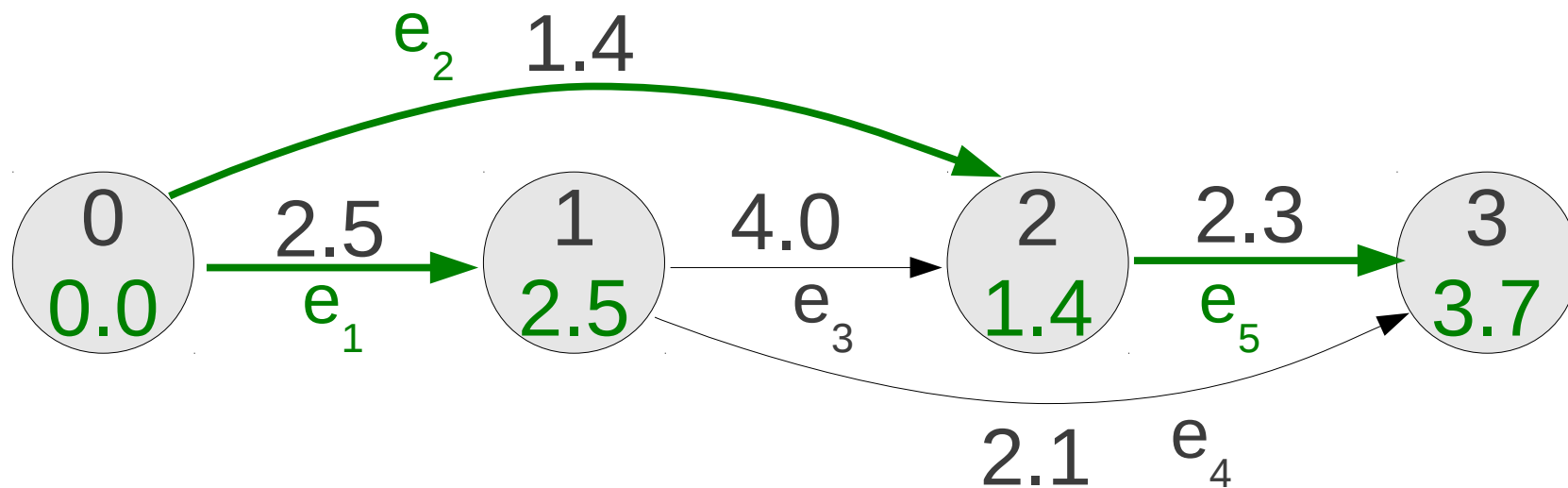
初期化：

 $\text{best_score}[0] = 0$ e_1 を計算：score = $0 + 2.5 = 2.5$ ($< \infty$) $\text{best_score}[1] = 2.5$ $\text{best_edge}[1] = e_1$ e_2 を計算：score = $0 + 1.4 = 1.4$ ($< \infty$) $\text{best_score}[2] = 1.4$ $\text{best_edge}[2] = e_2$ e_3 を計算：score = $2.5 + 4.0 = 6.5$ (> 1.4)

変更なし！

 e_4 を計算：score = $2.5 + 2.1 = 4.6$ ($< \infty$) ~~$\text{best_score}[3] = 4.6$~~ ~~$\text{best_edge}[3] = e_4$~~ e_5 を計算：score = $1.4 + 2.3 = 3.7$ (< 4.6) $\text{best_score}[3] = 3.7$ $\text{best_edge}[3] = e_5$

前向きステップの結果：

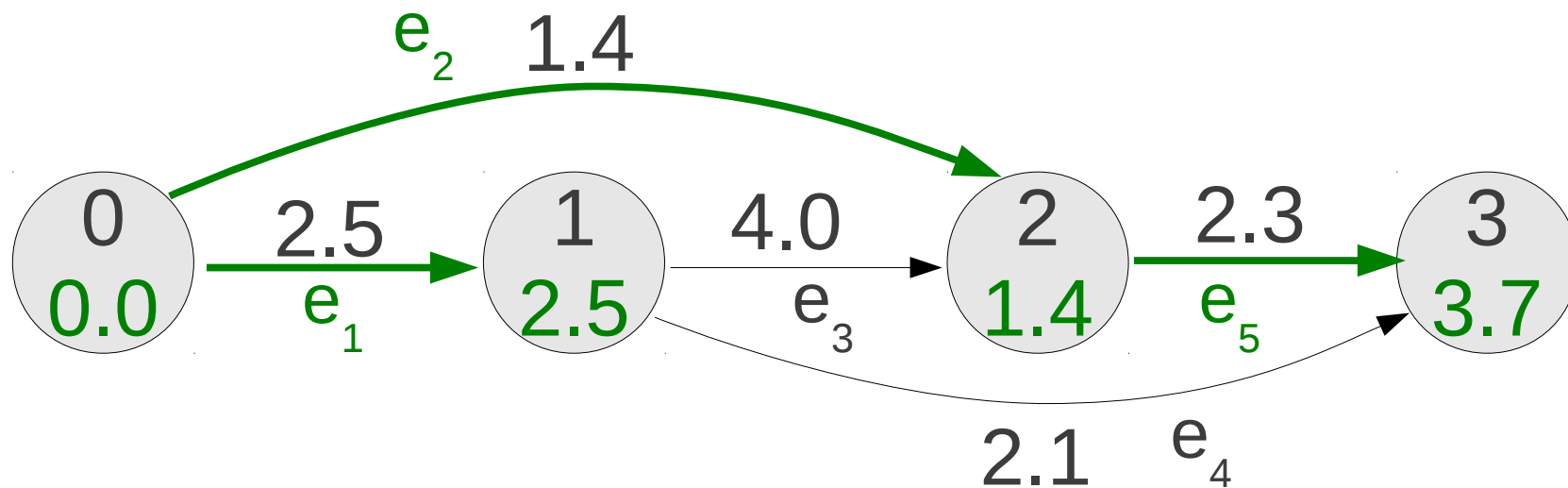


$best_score = (0.0, 2.5, 1.4, 3.7)$

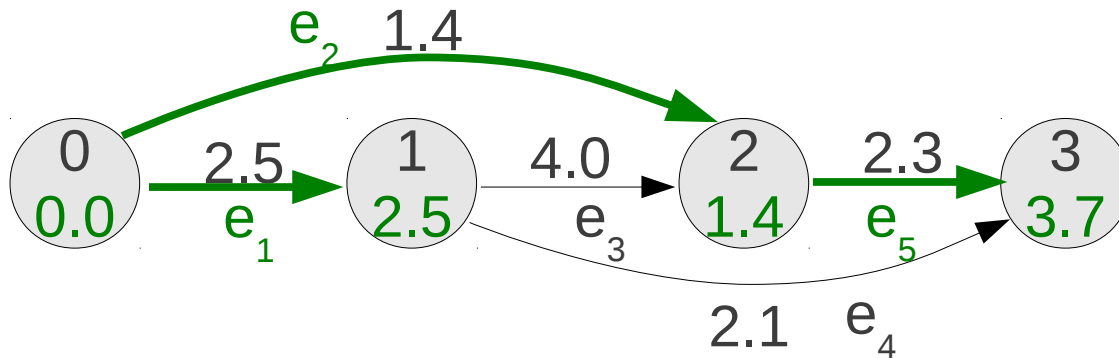
$best_edge = (NULL, e_1, e_2, e_5)$

後ろ向きステップ

後ろ向きステップのアルゴリズム



```
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
```

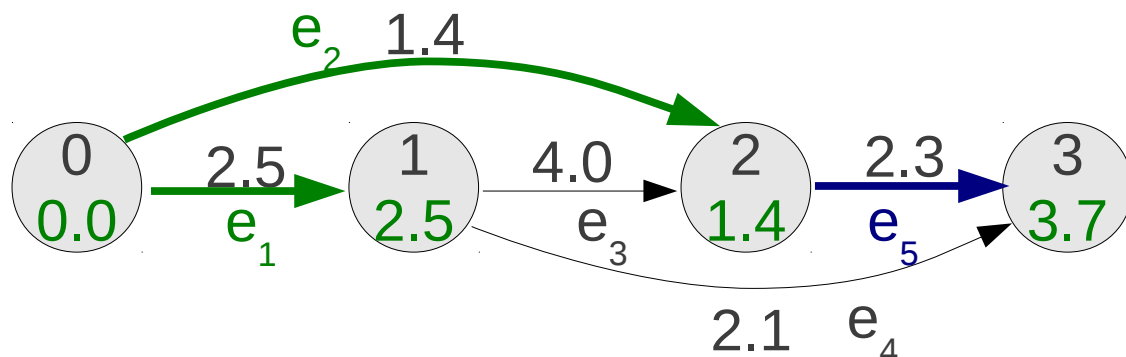


初期化:

best_path = []

next_edge = best_edge[3] = e₅

後ろ向きステップの例



初期化:

$best_path = []$

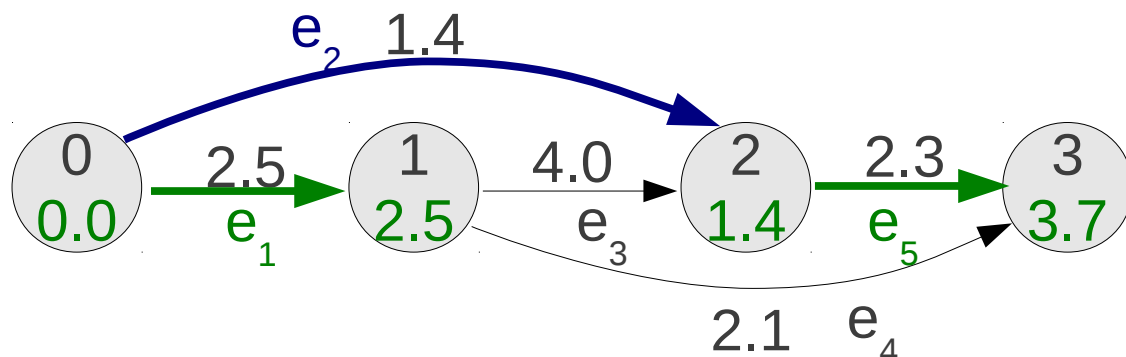
$next_edge = best_edge[3] = e_5$

e_5 を計算:

$best_path = [e_5]$

$next_edge = best_edge[2] = e_2$

後ろ向きステップの例



初期化 :

$best_path = []$

$next_edge = best_edge[3] = e_5$

e_5 を計算 :

$best_path = [e_5]$

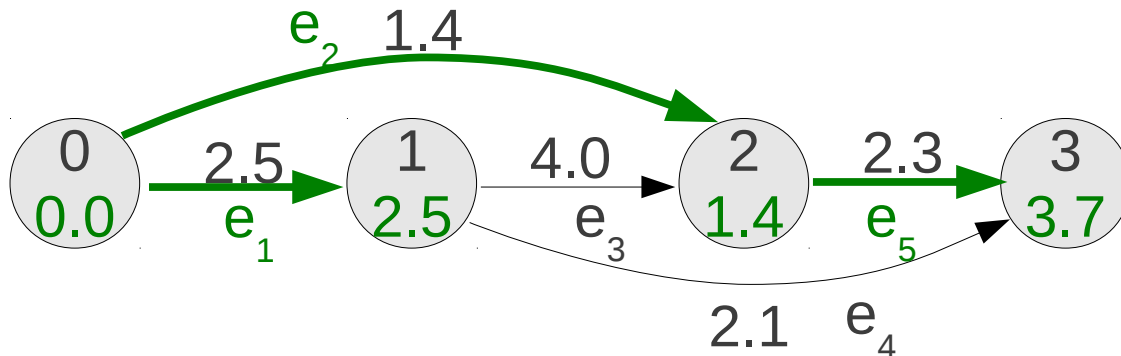
$next_edge = best_edge[2] = e_2$

e_2 を計算 :

$best_path = [e_5, e_2]$

$next_edge = best_edge[0] = NULL$

後ろ向きステップの例



初期化:

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

e_5 を計算:

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

e_5 を計算:

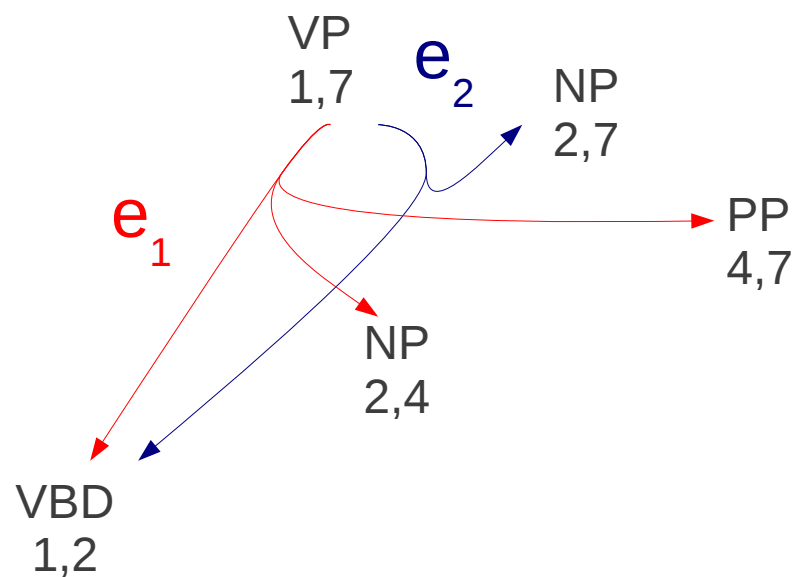
$best_path = [e_5, e_2]$
 $next_edge = best_edge[0] = NULL$

逆順に並べ替え:

$best_path = [e_2, e_5]$

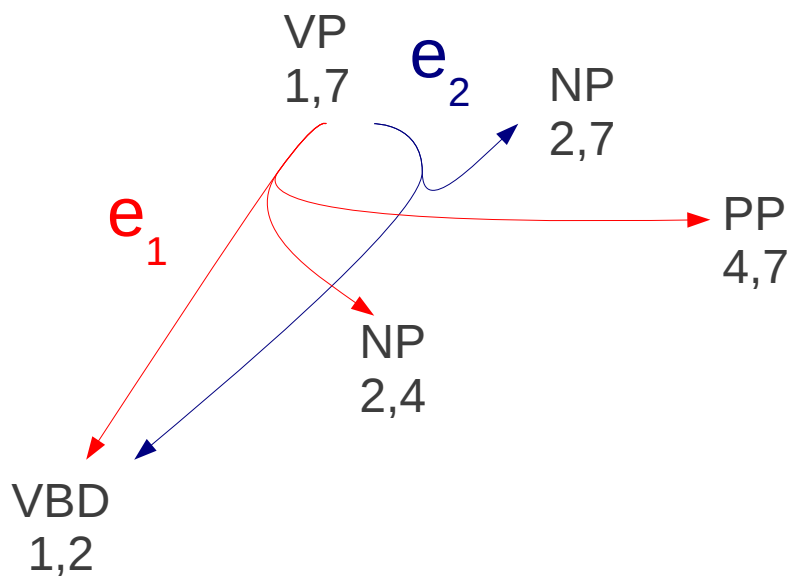
ノードの内ステップ

- VP1,7 の最小スコアを計算



ノードの内ステップ

- VP1,7 の最小スコアを計算



$$\text{score}(e_1) =$$

$$-\log(P(\text{VP} \rightarrow \text{VBD NP PP})) +$$

$$\text{best_score}[\text{VBD}1,2] +$$

$$\text{best_score}[\text{NP}2,4] +$$

$$\text{best_score}[\text{NP}2,7]$$

$$\text{score}(e_2) =$$

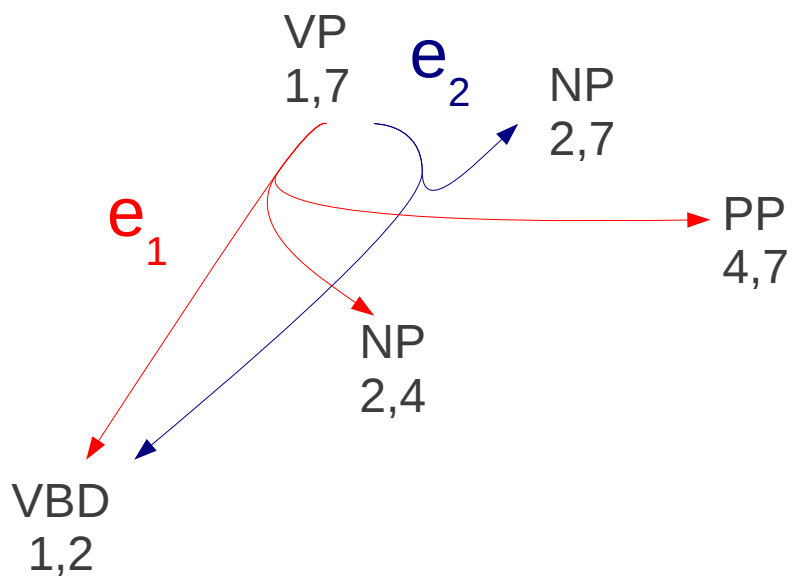
$$-\log(P(\text{VP} \rightarrow \text{VBD NP})) +$$

$$\text{best_score}[\text{VBD}1,2] +$$

$$\text{best_score}[\text{VBD}2,7]$$

ノードの内ステップ

- VP1,7 の最小スコアを計算



$$\text{score}(e_1) =$$

$$-\log(P(\text{VP} \rightarrow \text{VBD NP PP})) +$$

$$\text{best_score}[\text{VBD}1,2] +$$

$$\text{best_score}[\text{NP}2,4] +$$

$$\text{best_score}[\text{NP}2,7]$$

$$\text{score}(e_2) =$$

$$-\log(P(\text{VP} \rightarrow \text{VBD NP})) +$$

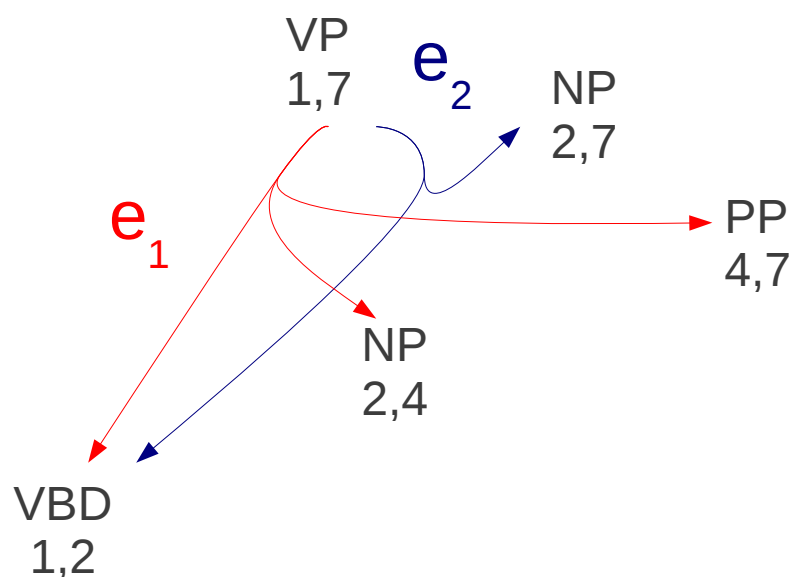
$$\text{best_score}[\text{VBD}1,2] +$$

$$\text{best_score}[\text{VBD}2,7]$$

$$\text{best_edge}[\text{VB}1,7] = \underset{e_1, e_2}{\text{argmin}} \text{score}$$

ノードの内ステップ

- VP1,7 の最小スコアを計算



$$\text{score}(e_1) =$$

$$-\log(P(\text{VP} \rightarrow \text{VBD NP PP})) +$$

$$\text{best_score}[\text{VBD}1,2] +$$

$$\text{best_score}[\text{NP}2,4] +$$

$$\text{best_score}[\text{NP}2,7]$$

$$\text{score}(e_2) =$$

$$-\log(P(\text{VP} \rightarrow \text{VBD NP})) +$$

$$\text{best_score}[\text{VBD}1,2] +$$

$$\text{best_score}[\text{VBD}2,7]$$

$$\text{best_edge}[\text{VB}1,7] = \underset{e_1, e_2}{\text{argmin}} \text{score}$$

$$\text{best_score}[\text{VB}1,7] =$$

$$\text{score}(\text{best_edge}[\text{VB}1,7])$$

文法からの超グラフ構築

- 超グラフは解けるが、構文解析で与えられるのは

文法

$P(S \rightarrow NP VP) = 0.8$
 $P(S \rightarrow PRP VP) = 0.2$
 $P(VP \rightarrow VBD NP PP) = 0.6$
 $P(VP \rightarrow VBD NP) = 0.4$
 $P(NP \rightarrow DT NN) = 0.5$
 $P(NP \rightarrow NN) = 0.5$
 $P(PRP \rightarrow "I") = 0.4$
 $P(VBD \rightarrow "saw") = 0.05$
 $P(DT \rightarrow "a") = 0.6$
...

文

I saw a girl with a telescope

- 解くための超グラフをどうやって構築するか？

CKY アルゴリズム

- CKY(Cocke-Kasami-Younger) アルゴリズムは文法に基づいてハイパーグラフを構築して解く
- 文法はチョムスキー標準形 (CNF)
 - ルールの右側は非終端記号 2 つもしくはは終端記号 1 つ

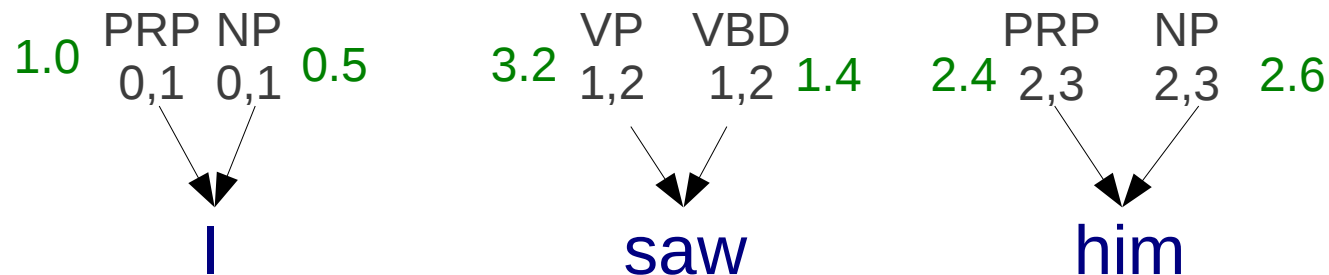
<u>OK</u>	<u>OK</u>	<u>Not OK!</u>
S → NP VP	PRP → "I"	VP → VBD NP PP
S → PRP VP	VBD → "saw"	NP → NN
VP → VBD NP	DT → "a"	NP → PRP

- この条件を満たさないルールは変更可能

VP → VBD NP PP	➔	VP → VBD NP_PP
		NP_PP → NP PP
NP → PRP + PRP → "I"	➔	NP → "I"

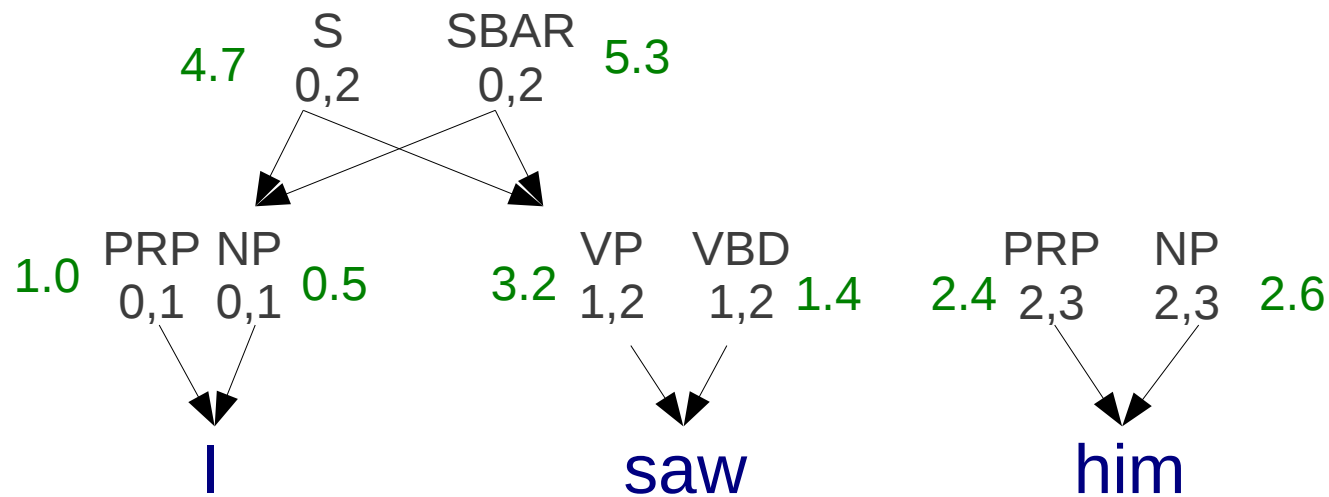
CKY アルゴリズム

- まずは終端記号のルールをスコア付きで展開



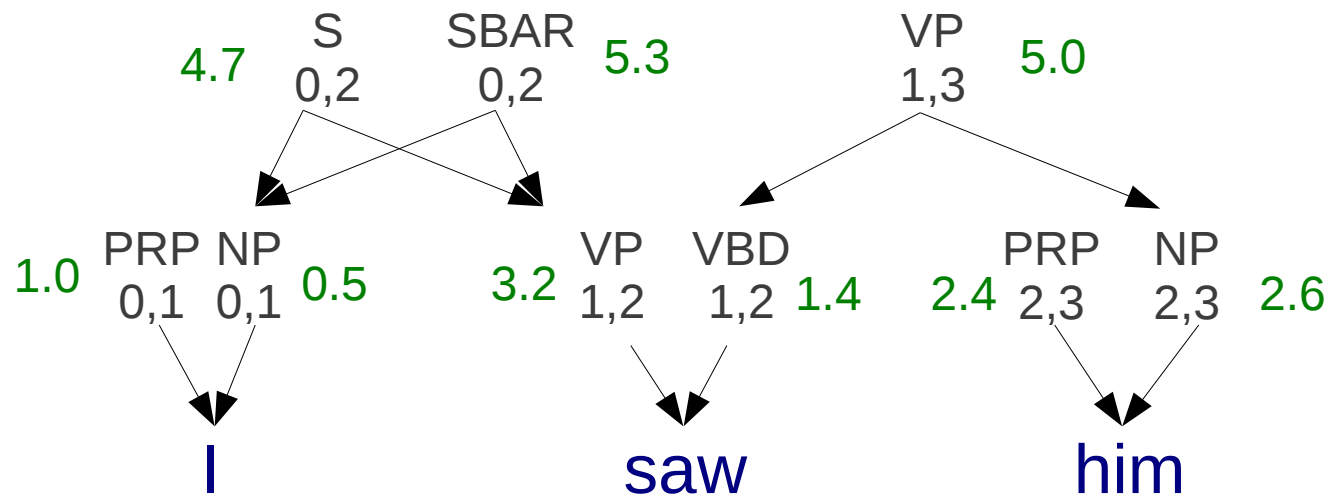
CKY アルゴリズム

- 0,2 のノードを全て展開



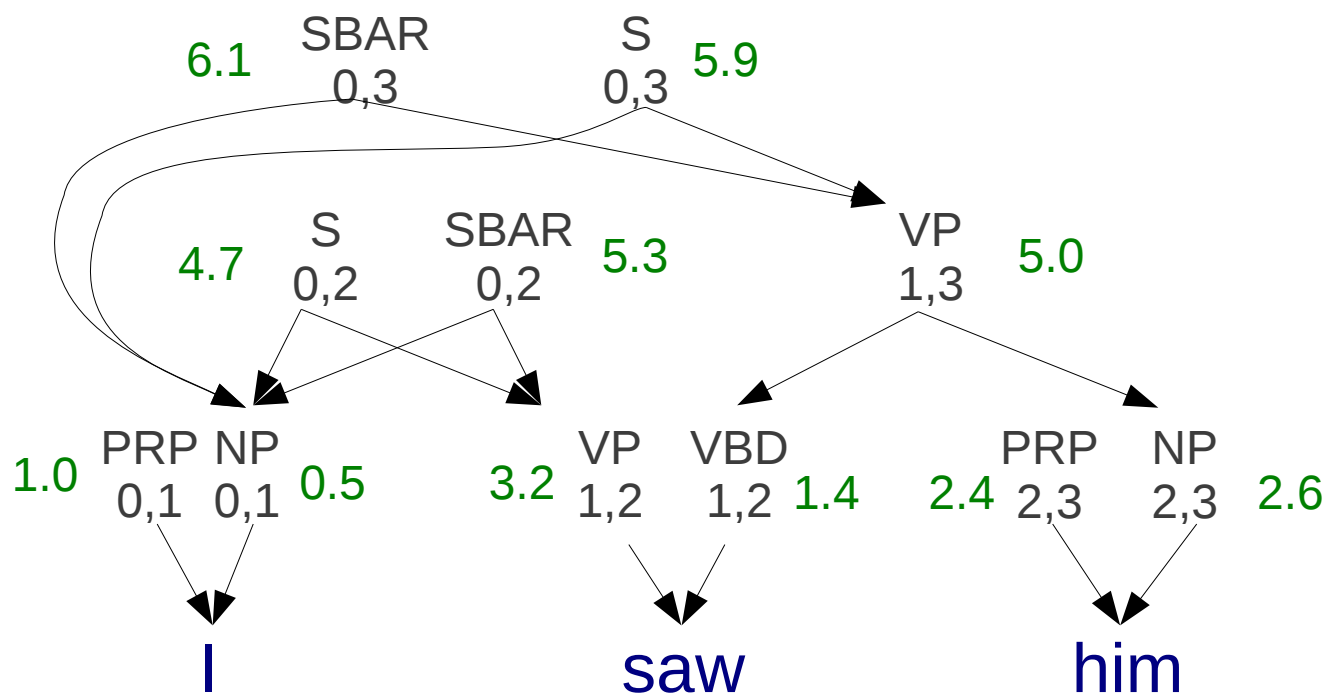
CKY アルゴリズム

- 1,3 のノードを全て展開



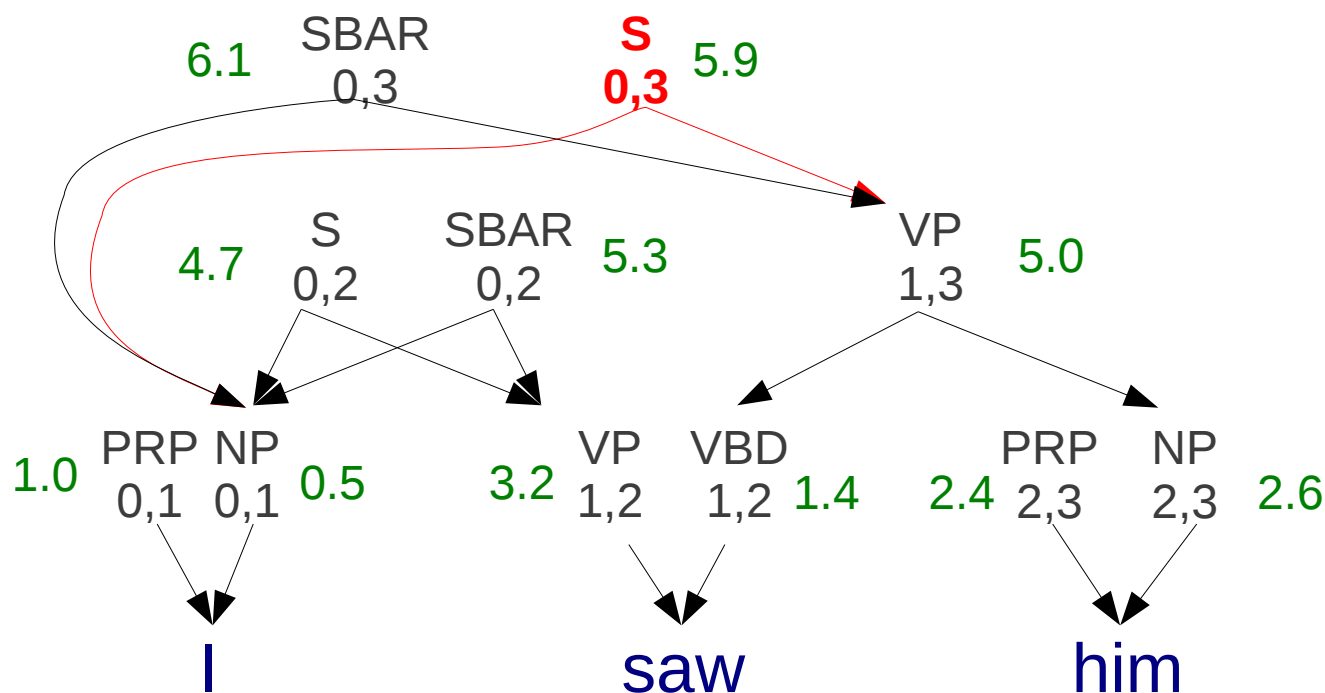
CKY アルゴリズム

- 0,3 のノードを全て展開



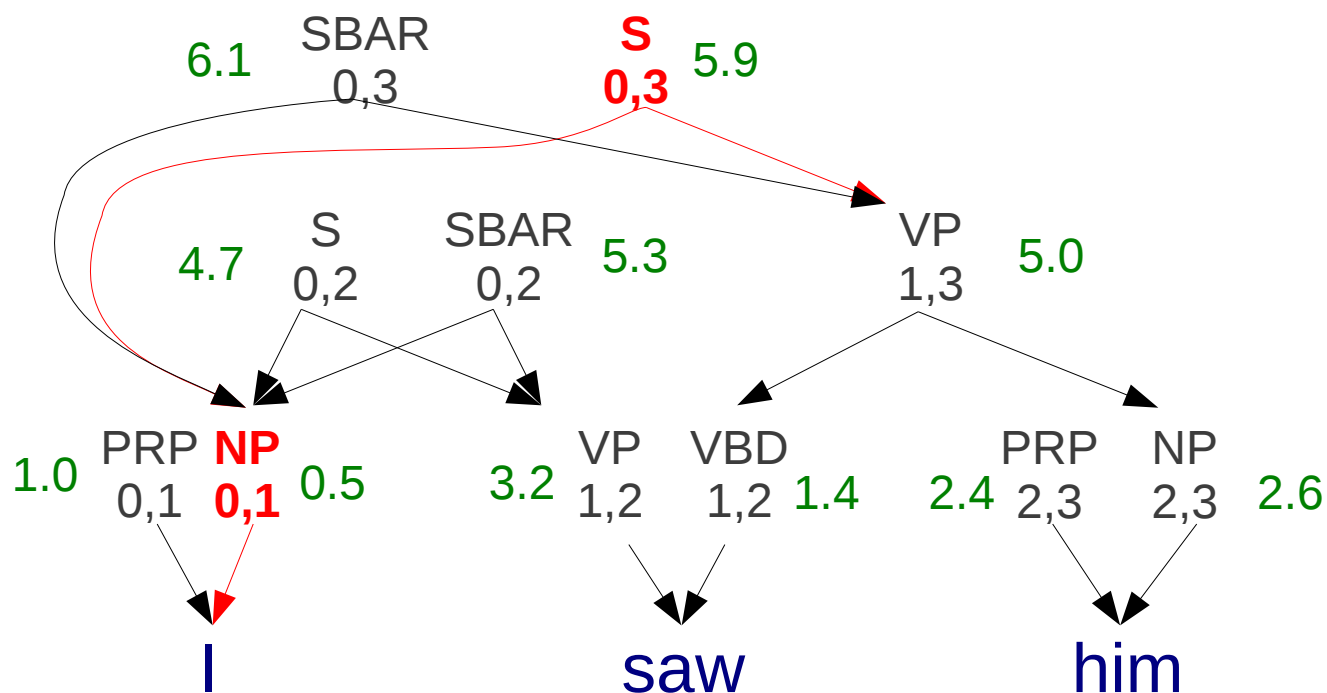
CKY アルゴリズム

- 文を全てカバーする「S」ノードを見つけて、エッジを展開



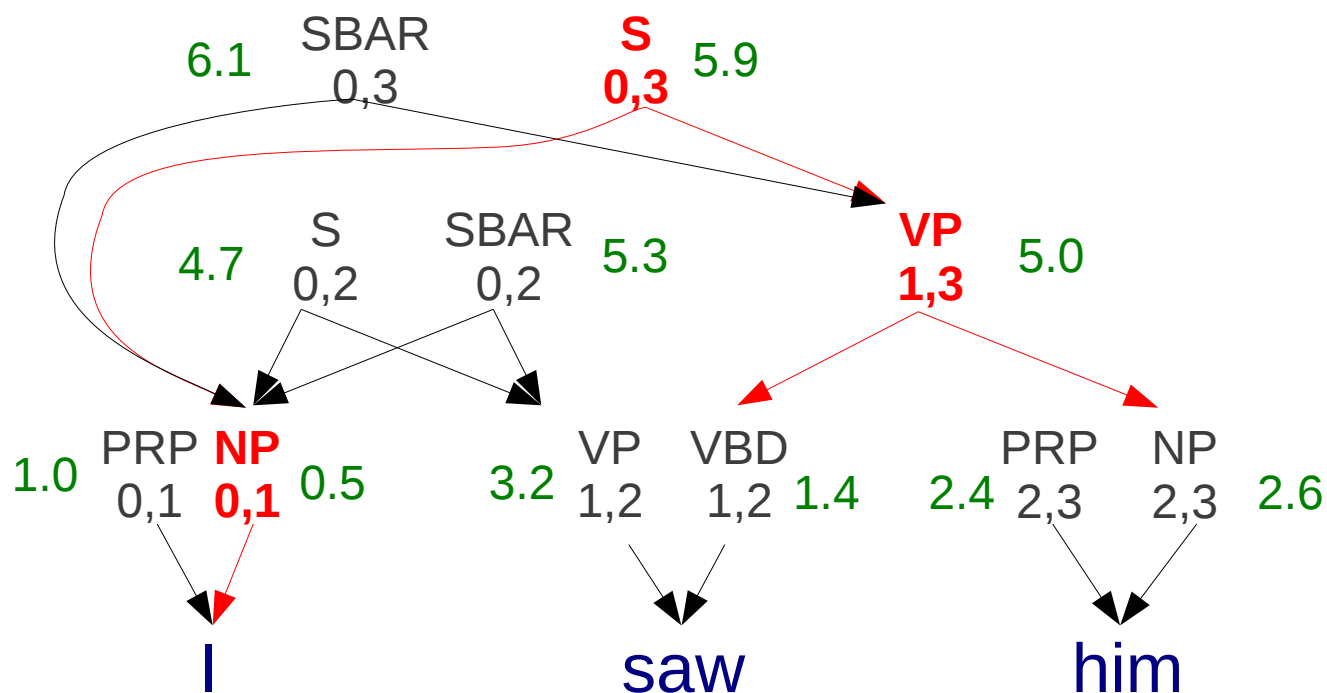
CKY アルゴリズム

- 左の子、右の子を再帰的に展開



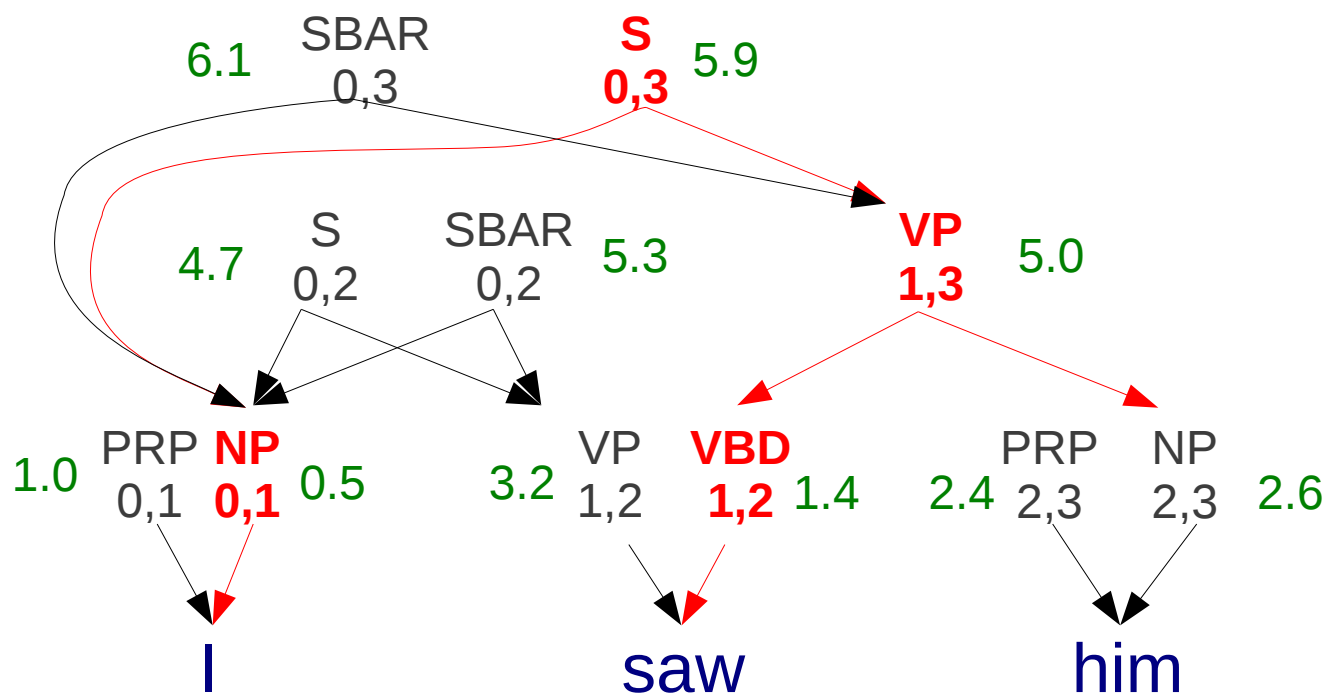
CKY アルゴリズム

- 左の子、右の子を再帰的に展開



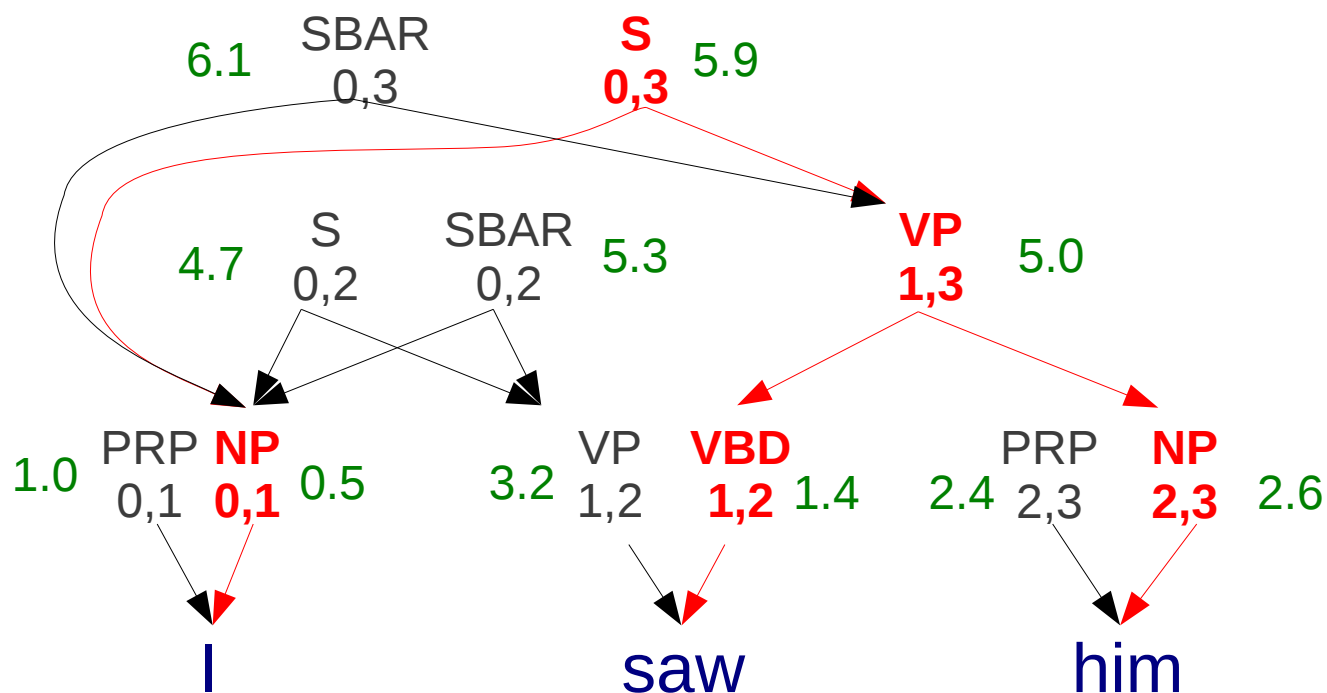
CKY アルゴリズム

- 左の子、右の子を再帰的に展開



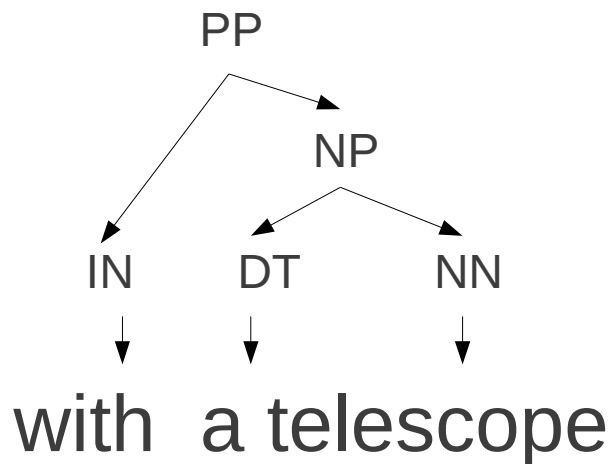
CKY アルゴリズム

- 左の子、右の子を再帰的に展開



構文木の出力

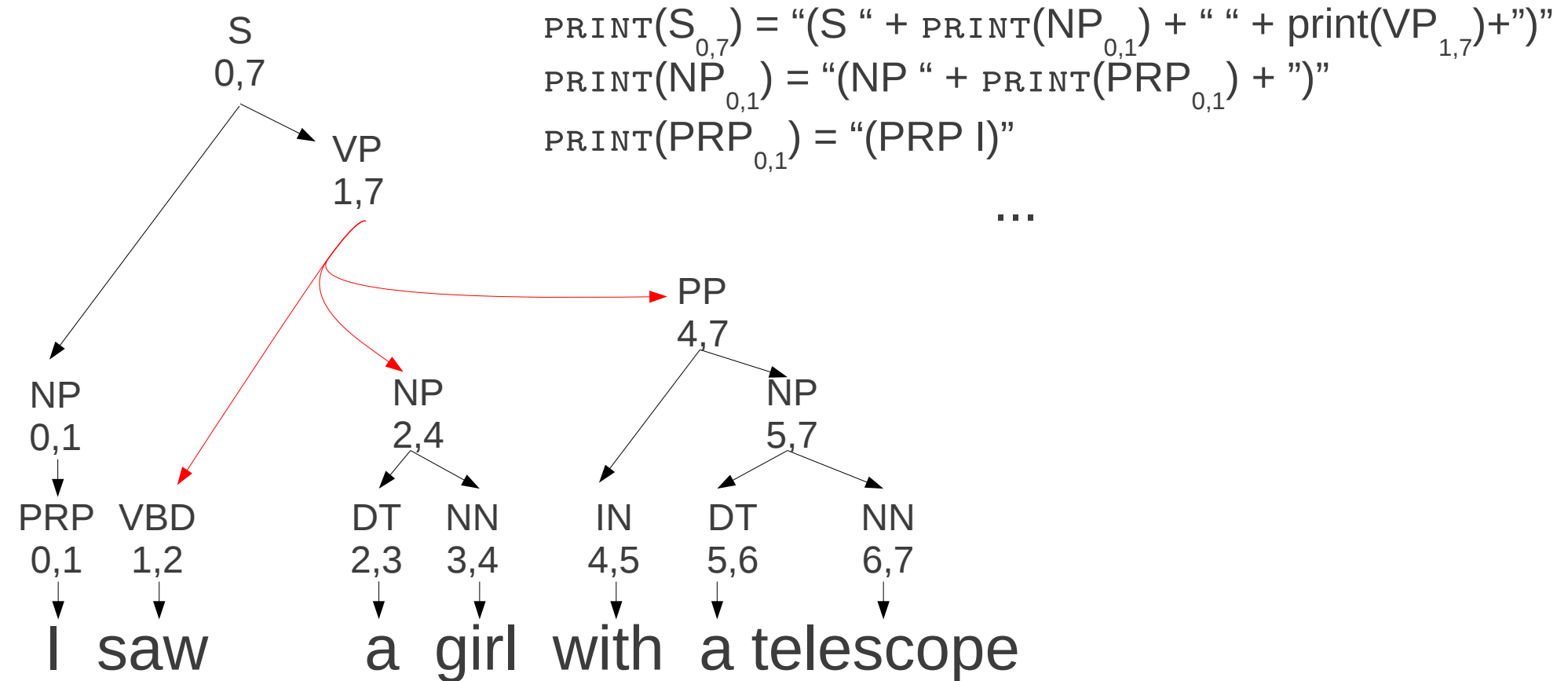
- 構文木の出力：「Penn Treebank 形式」 (S 式)



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

構文木の出力

- 再帰を使うと簡単に出力できる：



擬似コード

CKY 擬似コード：文法の読み込み

```
# “lhs \t rhs \t prob \n” 形式の文法を読み込む
make list nonterm # (左, 右 1, 右 2, 確率) の非終端記号
make map preterm # pre[右] = [(左, 確率) ...] 形式のマッピング
for rule in grammar_file
    split rule into lhs, rhs, prob (with “\t”) #  $P(\text{左} \rightarrow \text{右}) = \text{確率}$ 
    split rhs into rhs_symbols (with “ ”)
    if length(rhs) == 1: # 前終端記号
        add (lhs, log(prob)) to preterm[rhs]
    else: # 非終端記号
        add (lhs, rhs[0], rhs[1], log(prob)) to nonterm
```

CKY 擬似コード：前終端記号を追加

split line into words

make map best_score # 引数 = $\text{sym}_{i,j}$ 値 = 最大対数確率

make map best_edge # 引数 = $\text{sym}_{i,j}$ 値 = $(\text{lsym}_{i,k}, \text{rsym}_{k,j})$

前終端記号を追加

for i **in** $0 .. \text{length}(\text{words})-1$:

for lhs, \log_prob **in** preterm **where** $P(lhs \rightarrow \text{words}[i]) > 0$:

$\text{best_score}[\text{lhs}_{i,i+1}] = [\log_prob]$

CKY 擬似コード： 非終端記号の組み合わせ

```
for j in 2 .. length(words): # j はスパンの右側
  for i in j-2 .. 0:         # i はスパンの左側 (右から左へ処理!)
    for k in i+1 .. j-1:     # k は rsym の開始点
      # 各文法ルールを展開 :log(P(sym → lsym rsym)) = logprob
      for sym, lsym, rsym, logprob in nonterm:
        # 両方の子供の確率が 0 より大きい
        if best_score[lsymi,k] > -∞ and best_score[rsymk,j] > -∞:
          # このノード・辺の対数確率を計算
          my_lp = best_score[lsymi,k] + best_score[rsymk,j] + logprob
          # この辺が確率最大のものなら更新
          if my_lp > best_score[symi,j]:
            best_score[symi,j] = my_lp
            best_edge[symi,j] = (lsymi,k, rsymk,j)
```

CKY 擬似コード：木を出力

```
PRINT( $S_{0, \text{length}(\text{words})}$ ) # 文全体を覆う「S」を出力
```

```
subroutine PRINT( $\text{sym}_{i,j}$ ):
```

```
if  $\text{sym}_{i,j}$  exists in best_edge: # 非終端記号
```

```
    return (“+sym+” “  
        + PRINT(best_edge[0]) + “ ” +  
        + PRINT(best_edge[1]) + “)”
```

```
else: # 終端記号
```

```
    return (“+sym+“ ”+words[i]+“)”
```

演習課題

演習課題

- 実装 `cky.py`
- テスト
 - 入力: `test/08-input.txt`
 - 文法: `test/08-grammar.txt`
 - 出力: `test/08-output.txt`
- 実際のデータに対して動かす
 - `data/wiki-en-test.grammar`, `data/wiki-en-short.tok`
- 木を可視化
 - `08-parsing/print-trees.py < wiki-en-test.trees`
 - (NLTK をインストールする必要あり: <http://nltk.org/>)
- チャレンジ 未知語に対処できるように改良

Thank You!