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)

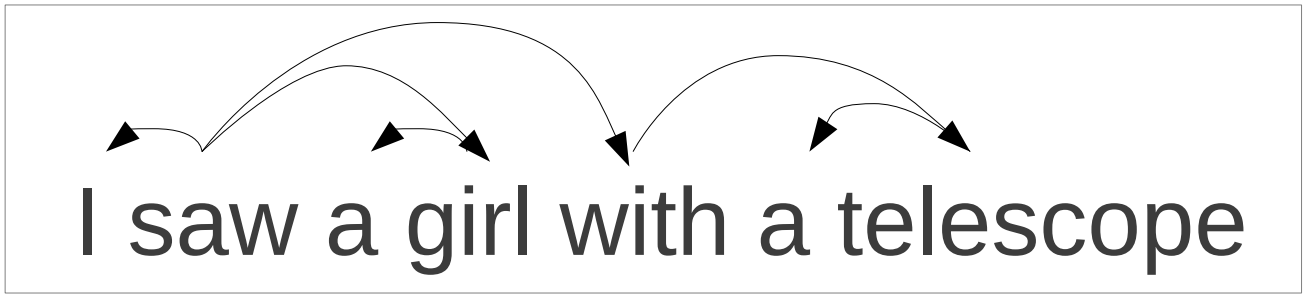
自然言語は曖昧性だらけ！

I saw a girl with a telescope

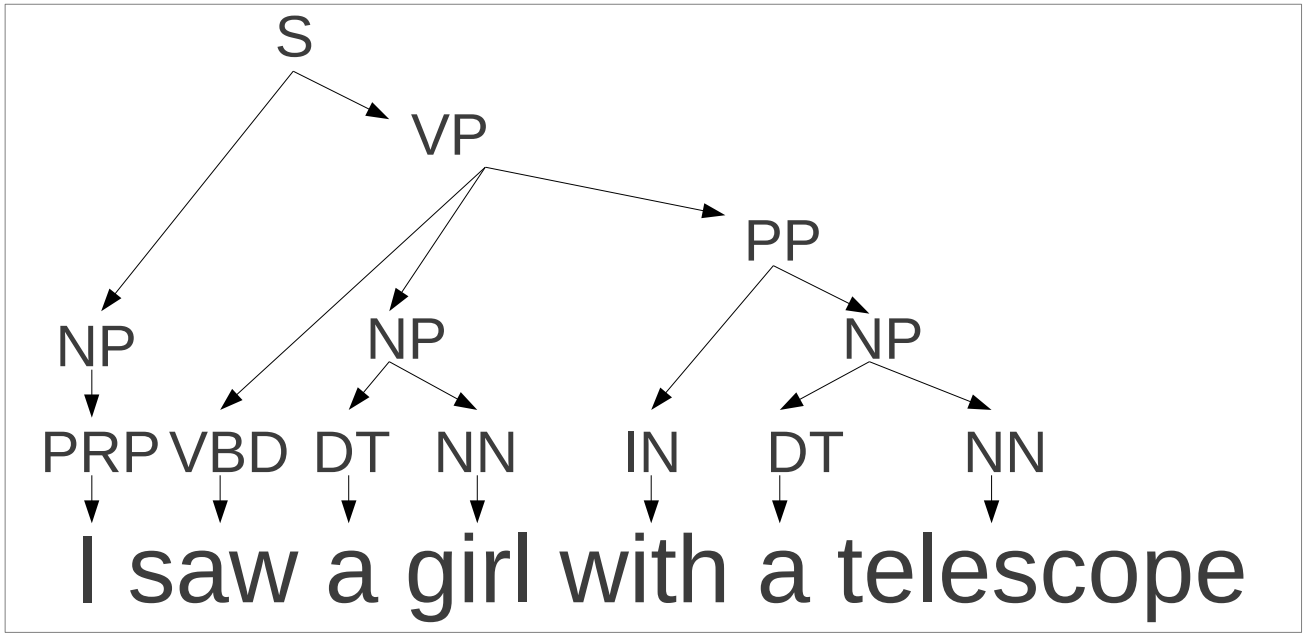
- 構文解析（パージング）は構造的な曖昧性を解消

構文解析の種類

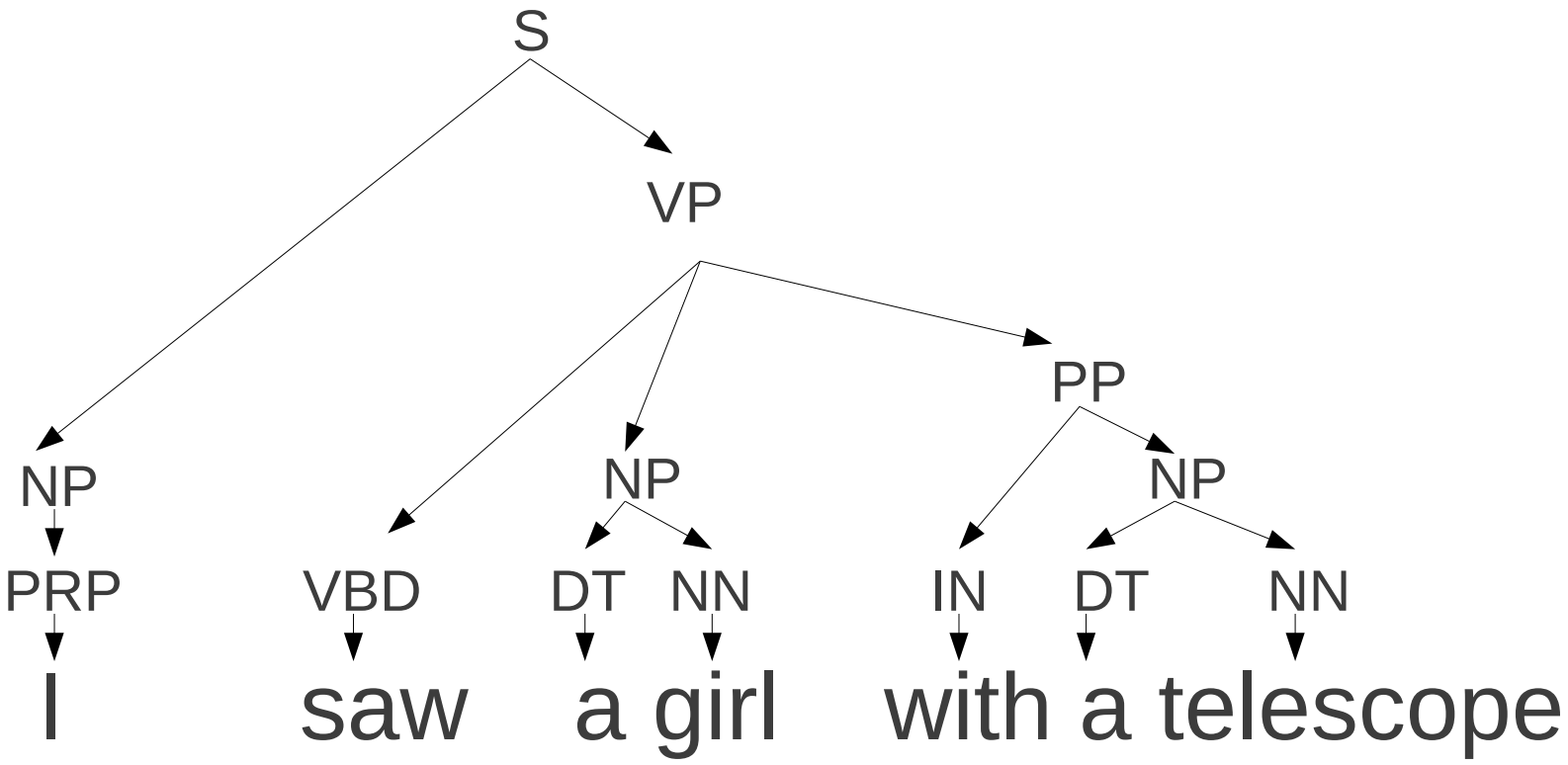
- 係り受け解析 : 単語と単語のつながりを重視



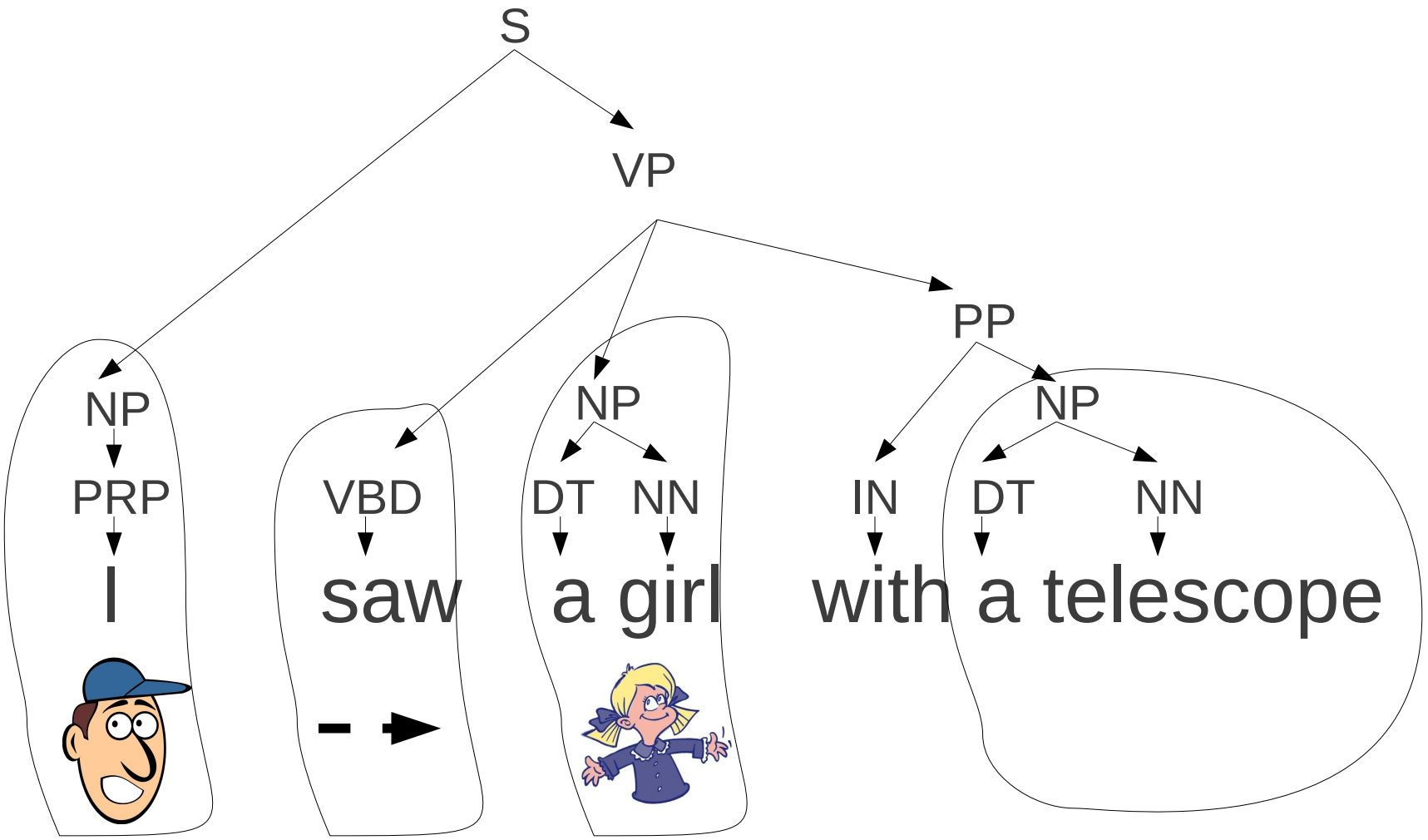
- 句構造解析 : 句とその再帰的な構造を重視



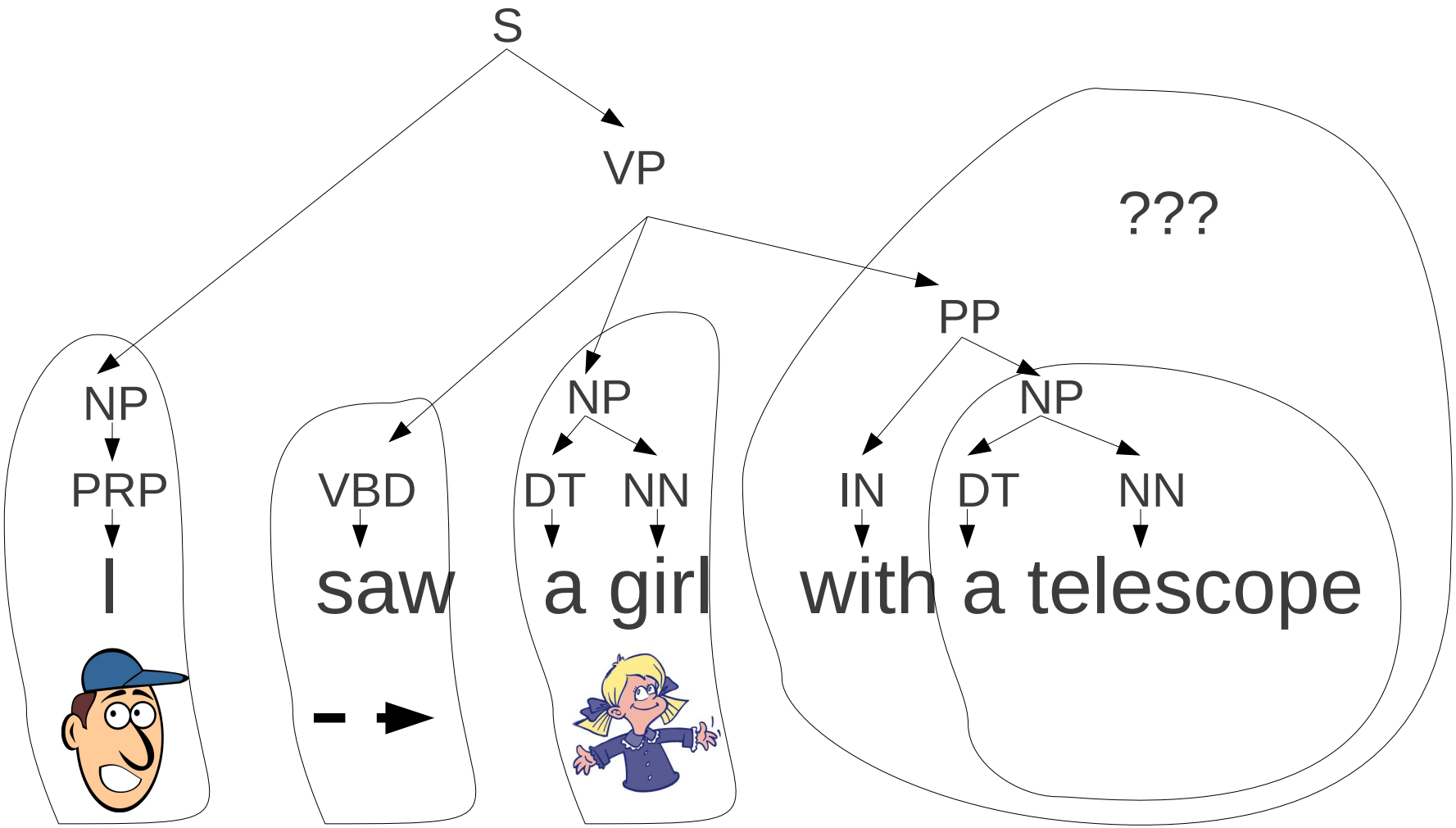
句の再帰的な構造


S
VP
PP
NP
NP
NP
PRP
VBD
DT
NN
IN
DT
NN
I
saw
a girl
with a telescope

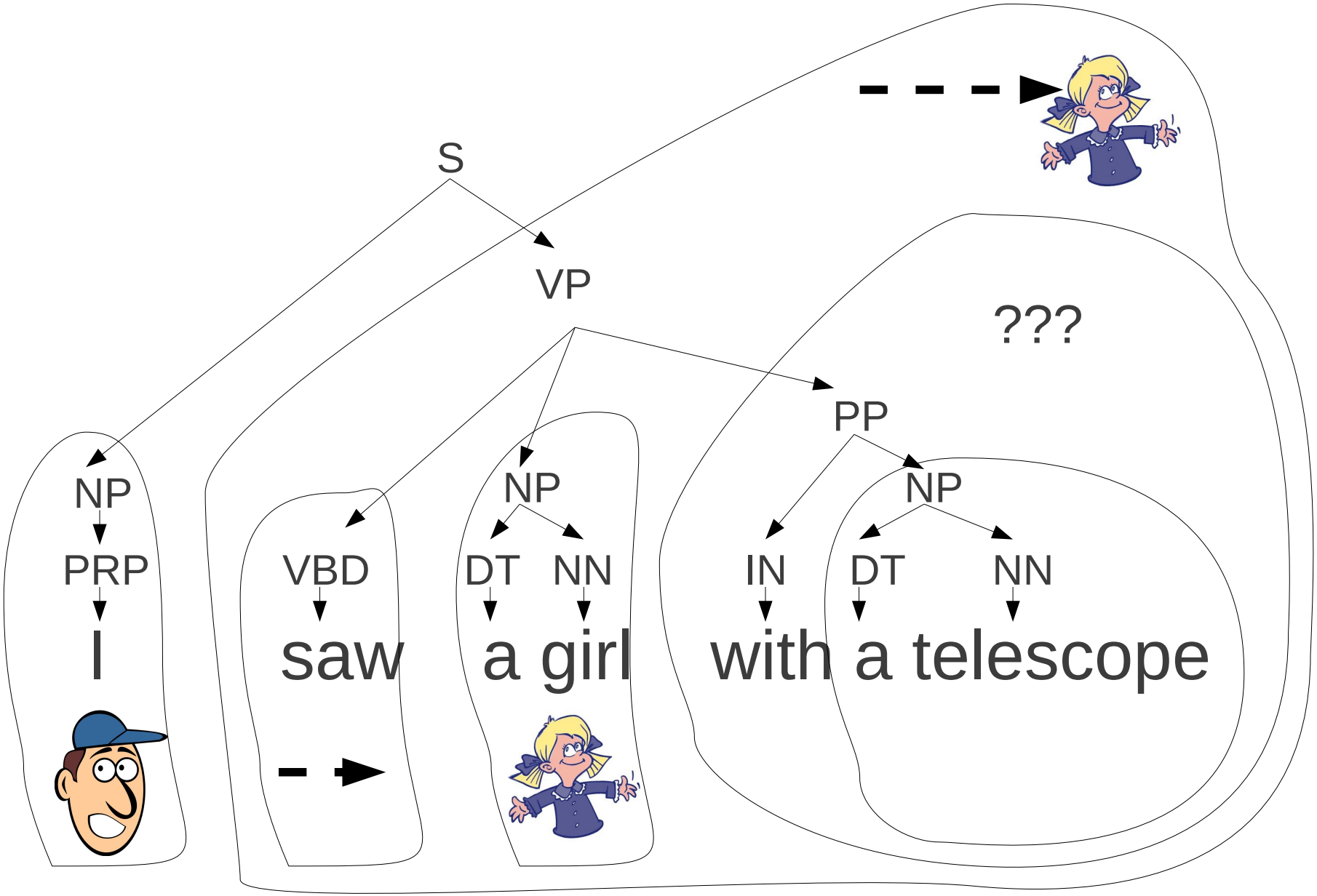
句の再帰的な構造


S
VP
PP
NP
PRP
I
VBD
saw
NP
DT
NN
a girl
IN
with
NP
DT
NN
a telescope

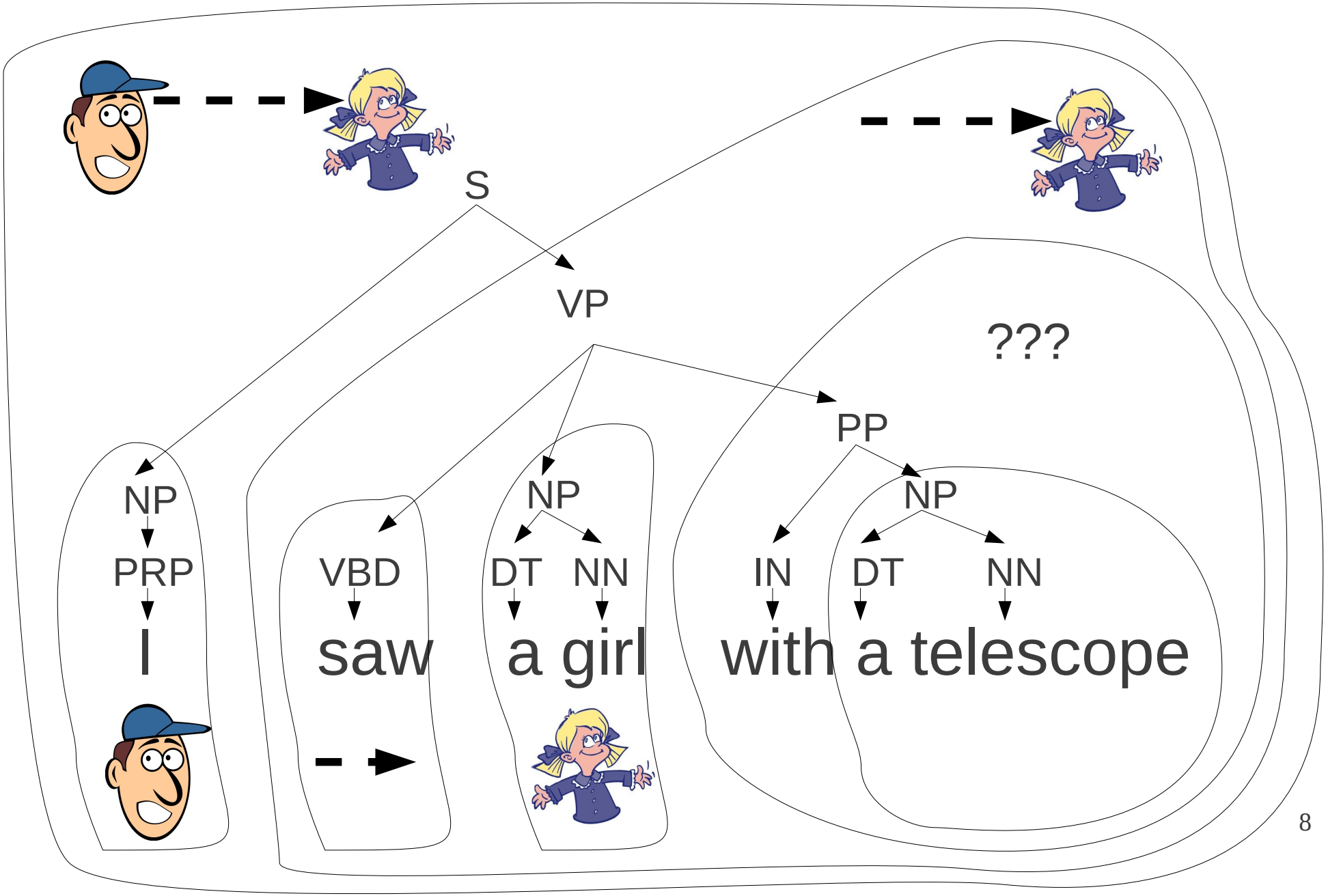
句の再帰的な構造


S
VP
???
PP
NP
PRP
I
VBD
saw
NP
DT
NN
a girl
IN
with
NP
DT
NN
a telescope

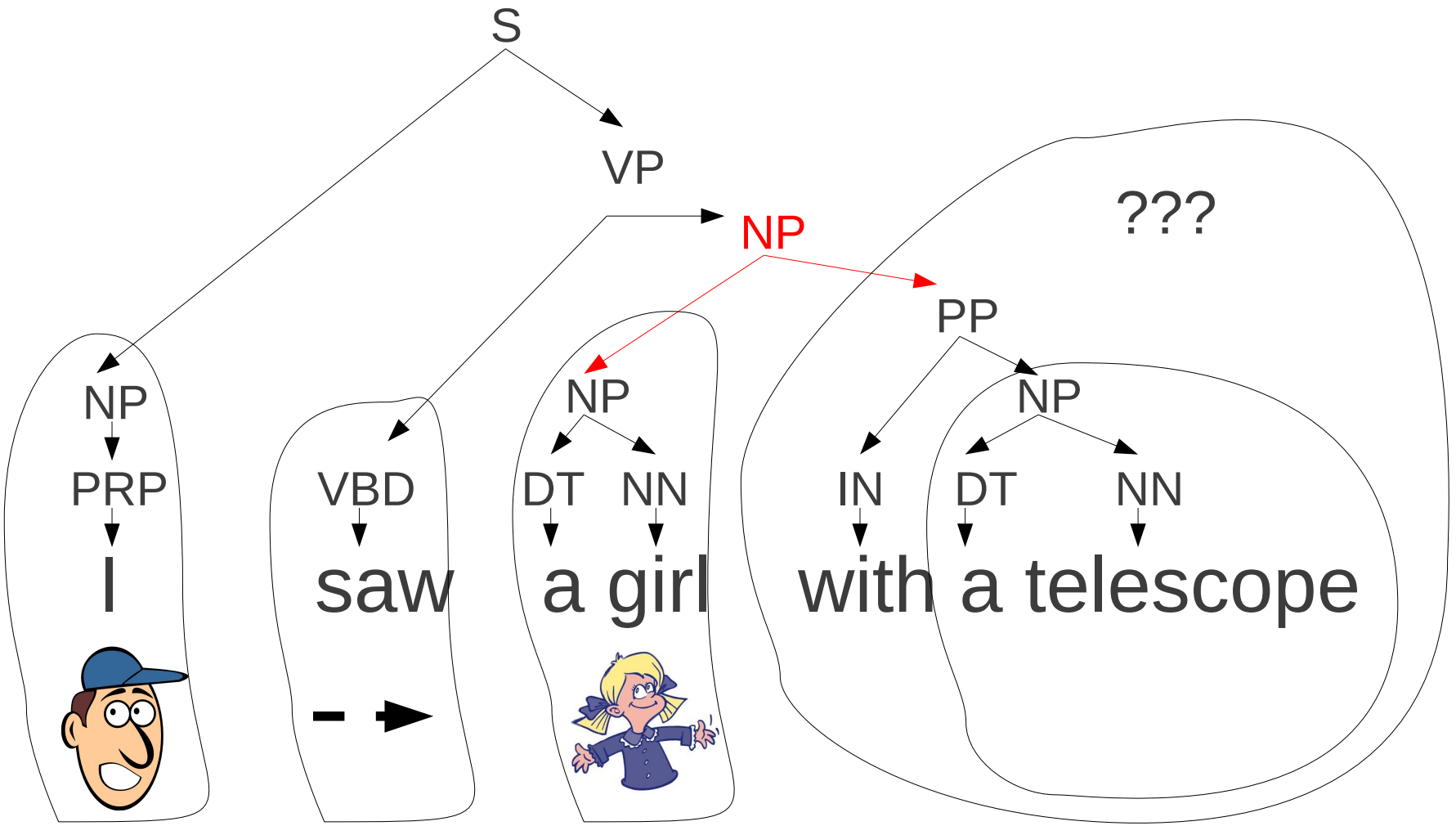
句の再帰的な構造


S
VP
???
PP
NP
PRP
I
VBD
saw
NP
DT
NN
a girl
IN
with
NP
DT
NN
a telescope

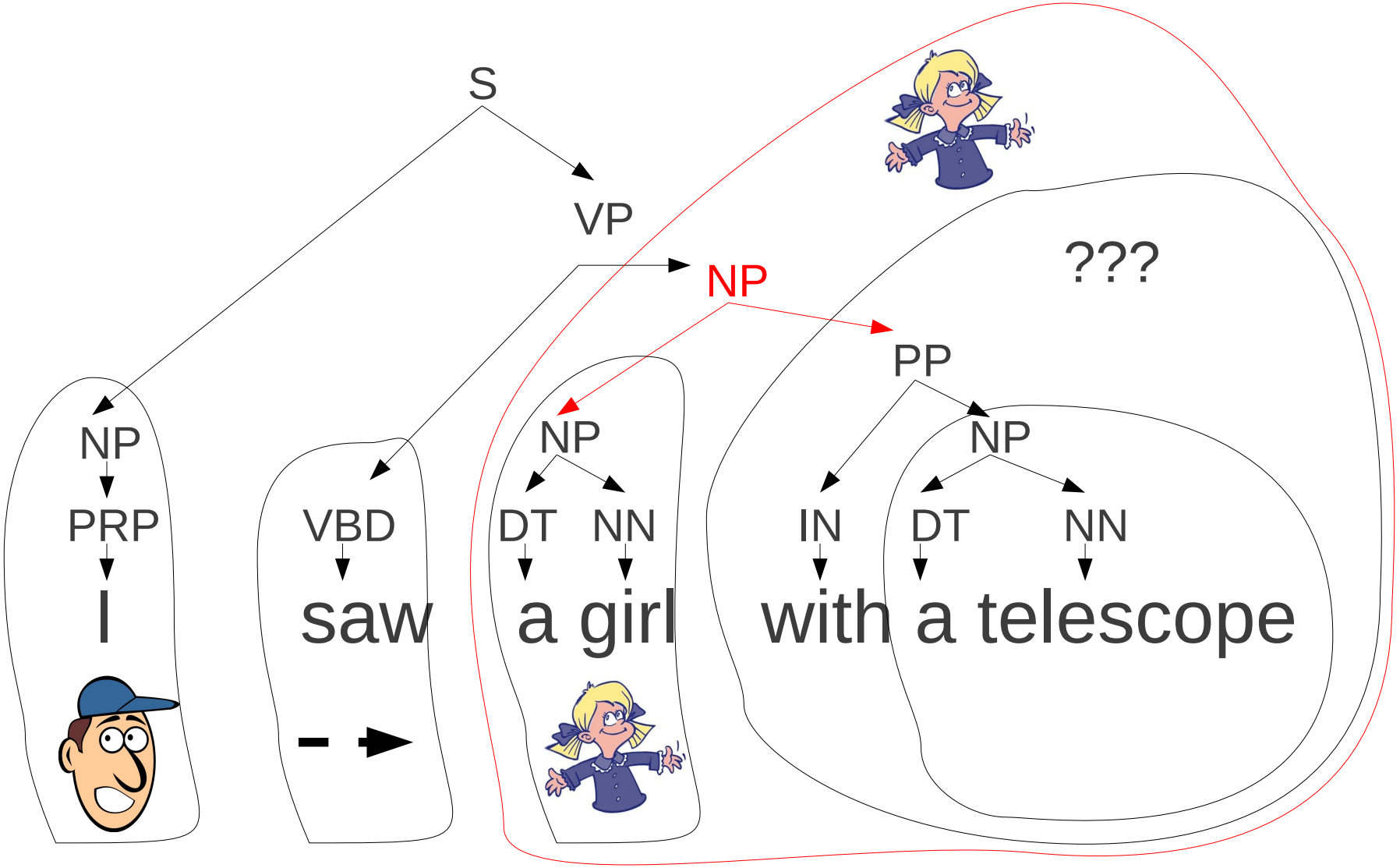
句の再帰的な構造


S
VP
???
PP
NP
PRP
I
VBD
saw
NP
DT
NN
a girl
IN
with
NP
DT
NN
a telescope

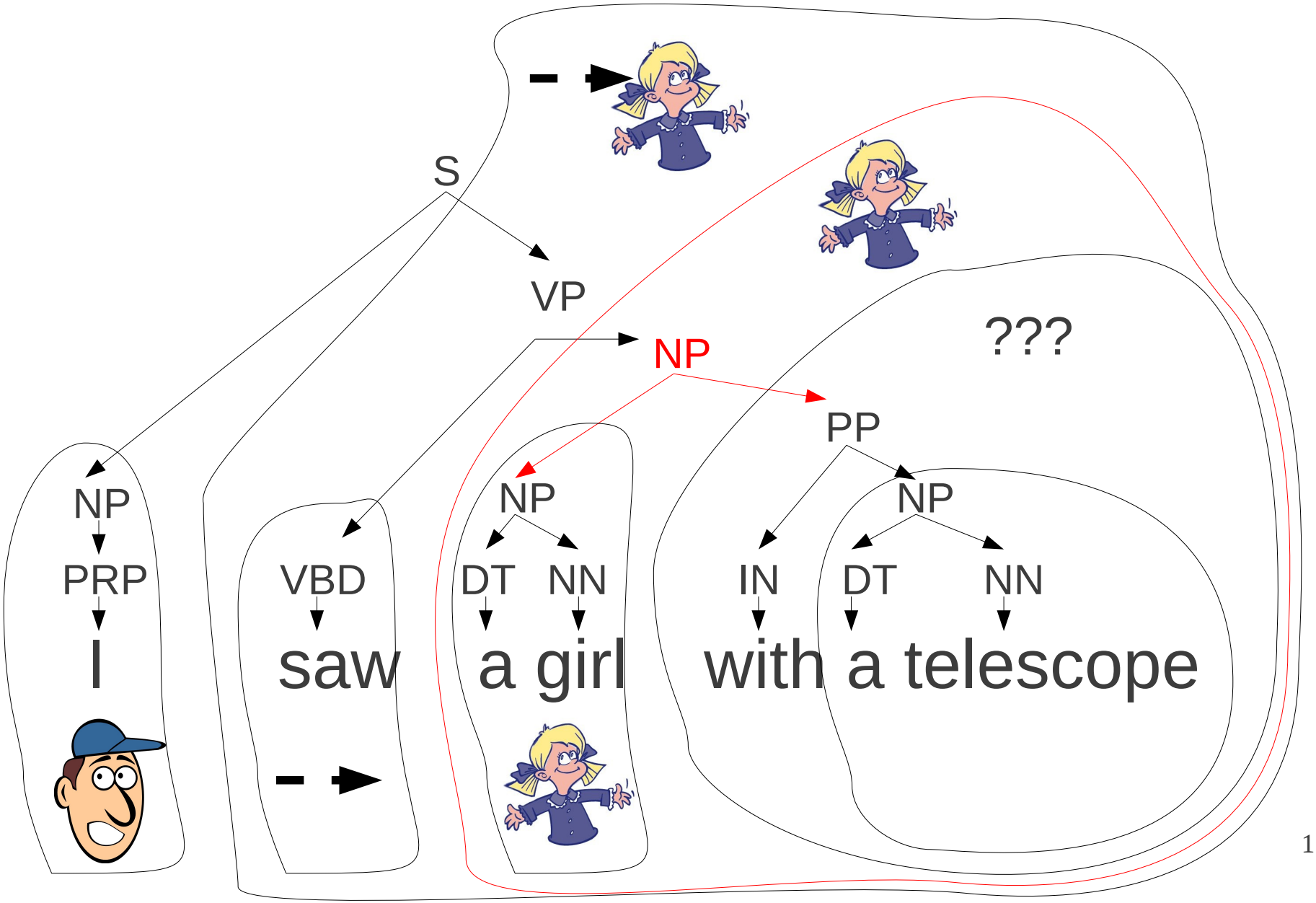
違う構造→違う解釈


S
VP
NP
???
PP
NP
NP
NP
PRP
VBD
DT
NN
IN
DT
NN
I
saw
a girl
with a telescope

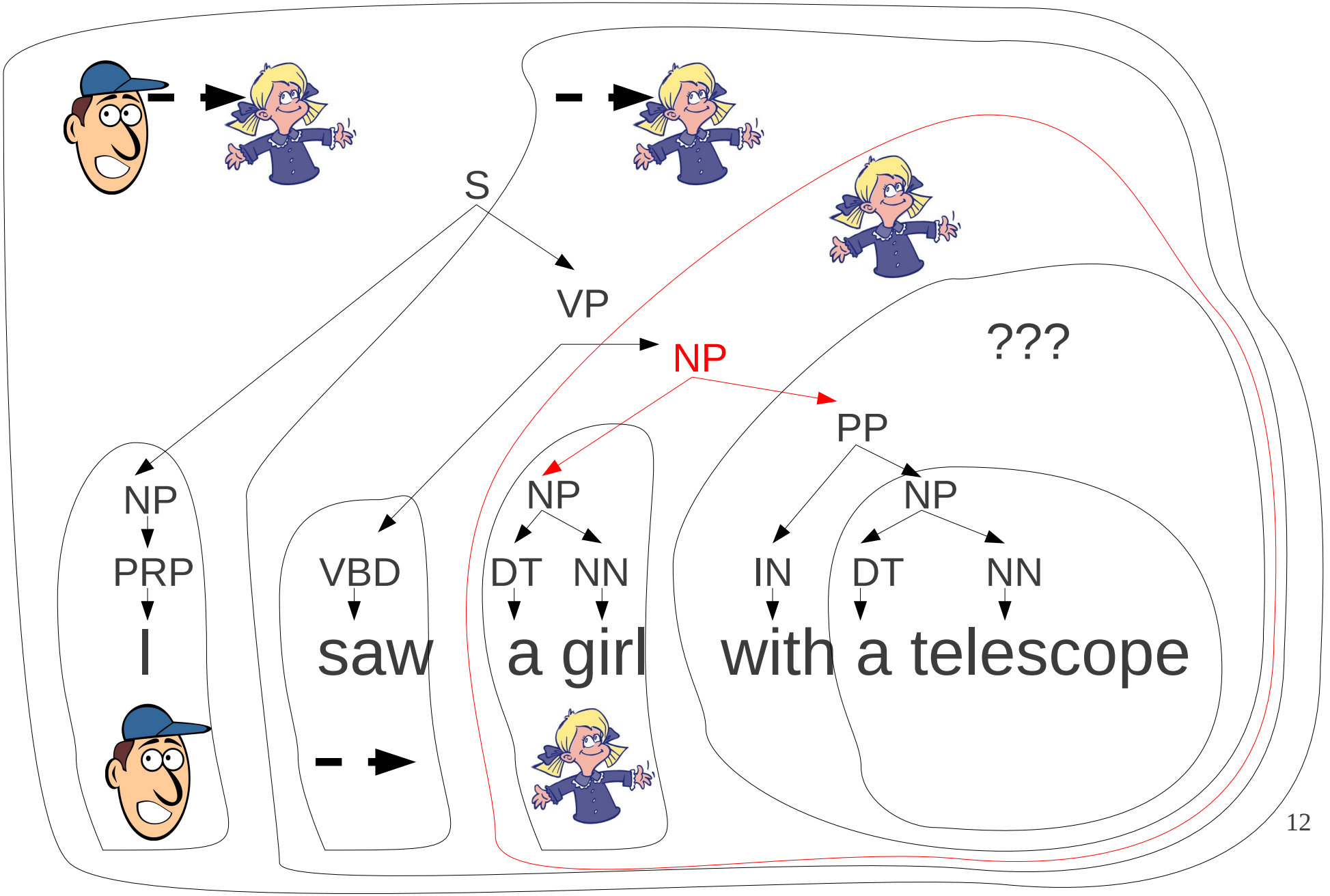
違う構造→違う解釈


S
VP
NP
PP
NP
PRP
VBD
NP
IN
DT
NN
DT
NN
I
saw
a girl
with
a telescope
???

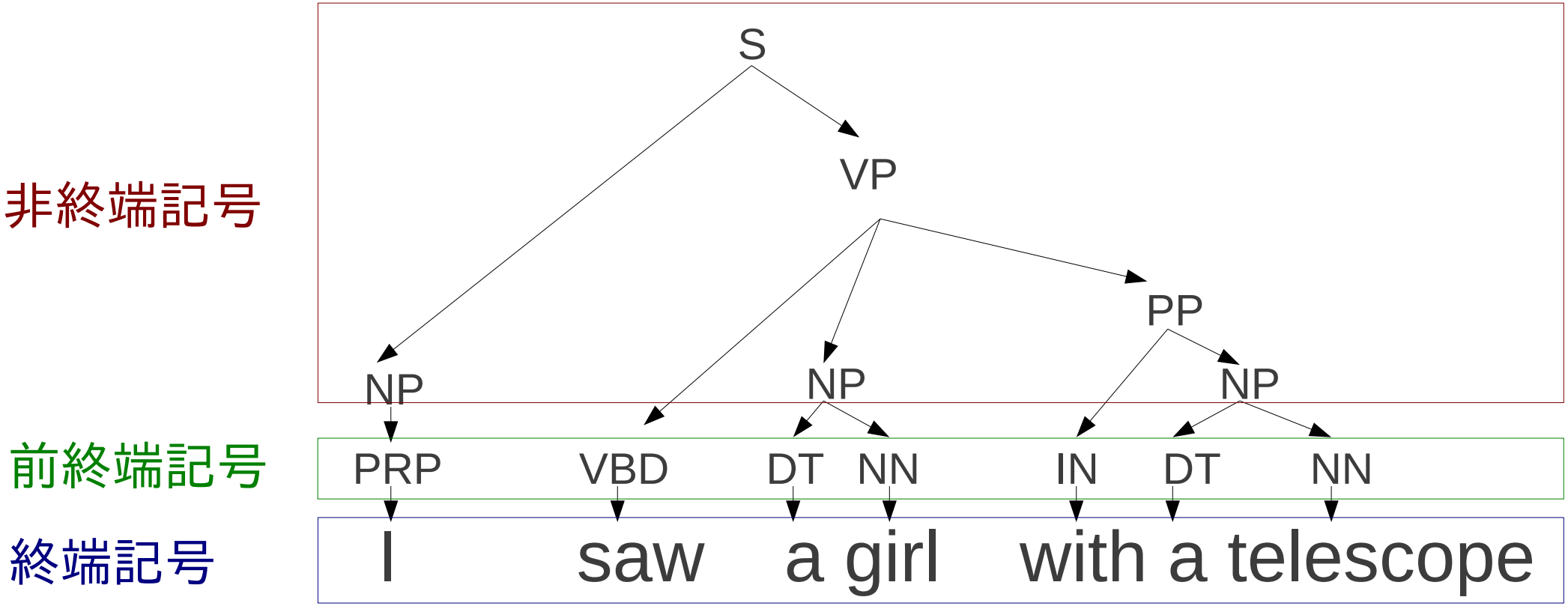
違う構造→違う解釈


S
VP
NP
PP
???
NP
PRP
I
VBD
saw
NP
DT
NN
a girl
IN
with
NP
DT
NN
a telescope

違う構造→違う解釈

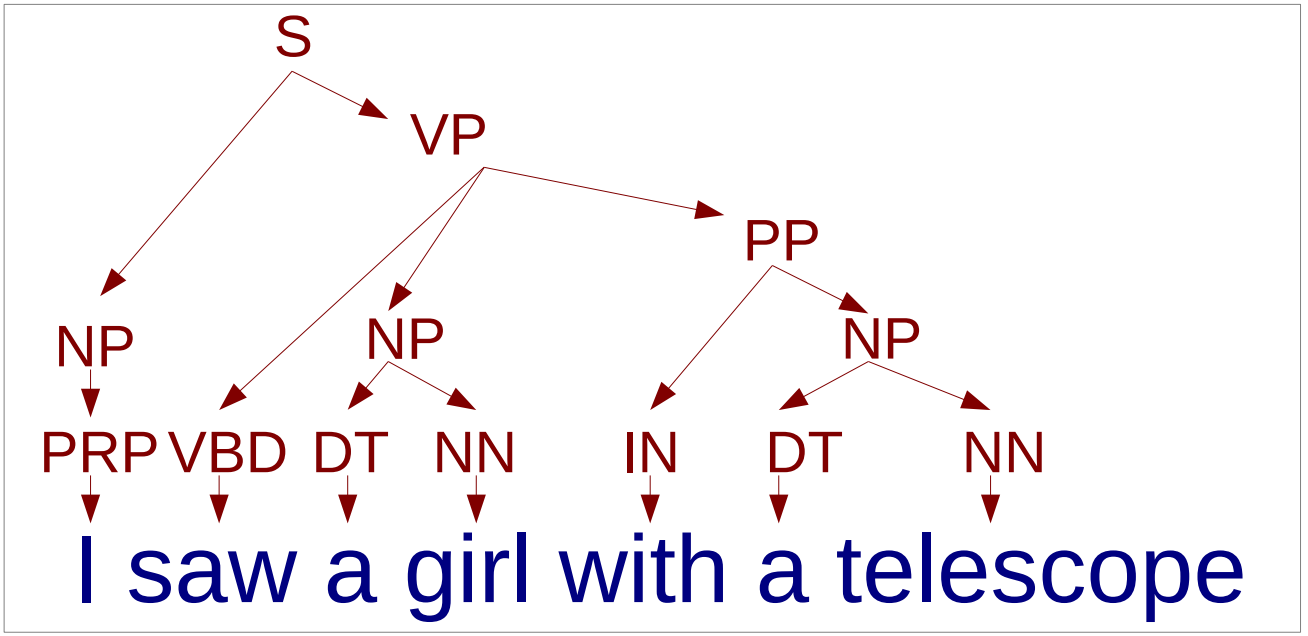

S
VP
NP
???
PP
NP
PRP
I
VBD
saw
NP
DT
NN
a girl
IN
with
NP
DT
NN
a telescope

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

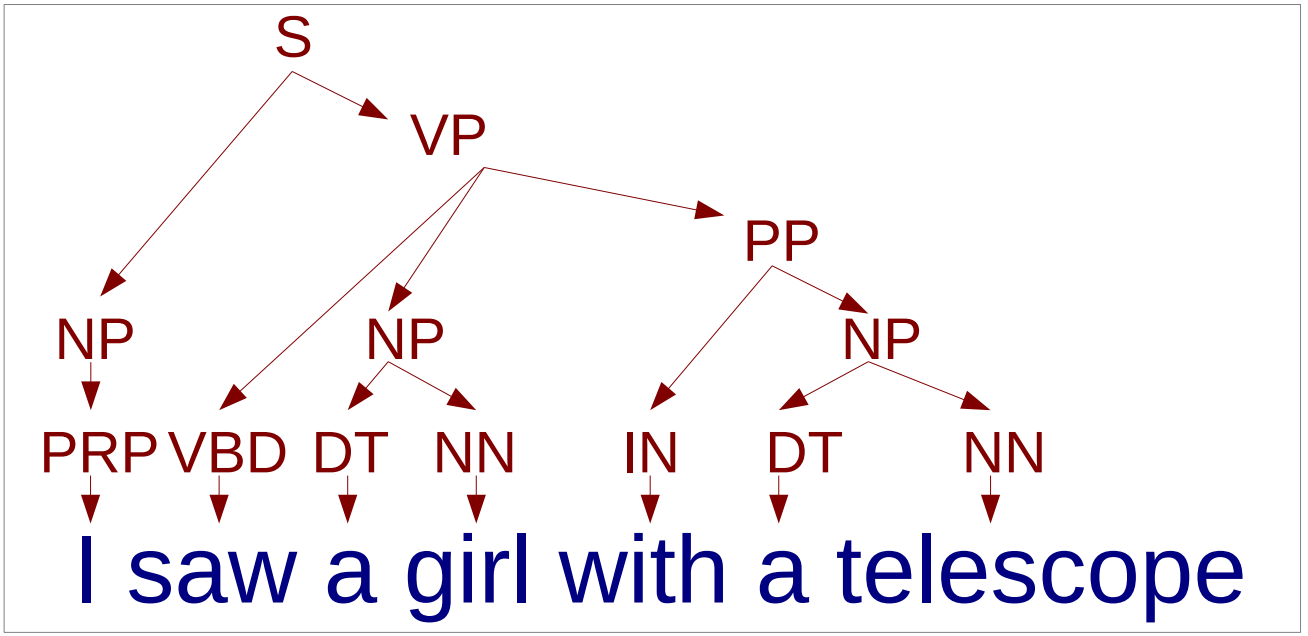
- 文 X が与えられ、構文木 Y を予測



- 「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



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

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

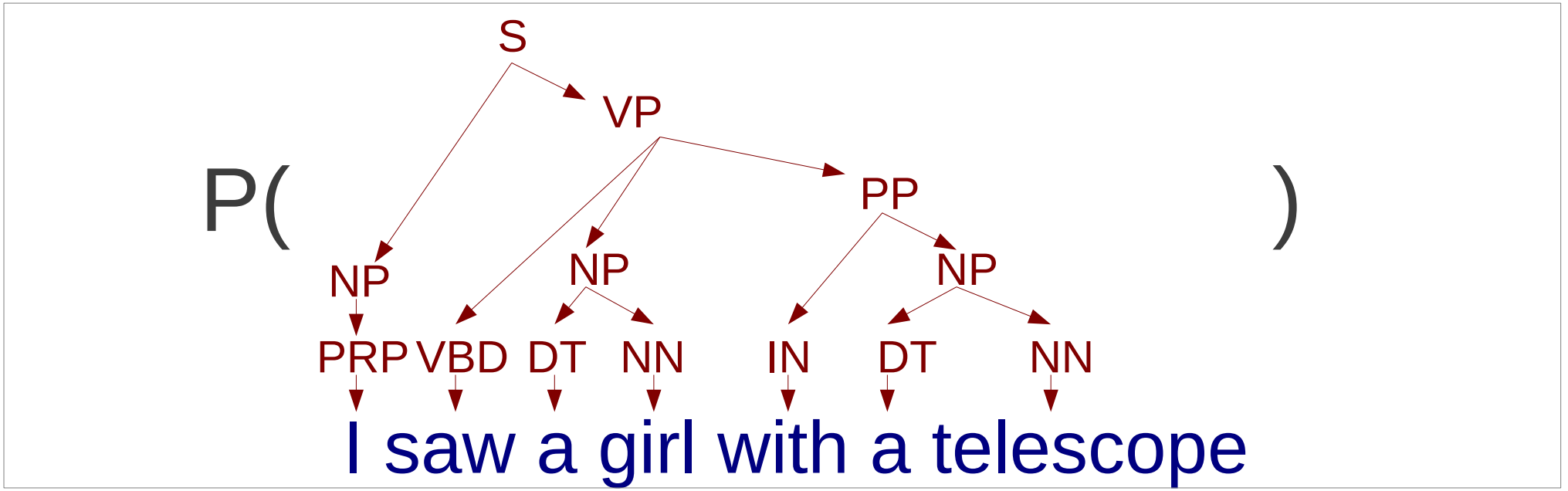
$$P(Y, X)$$

- 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

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

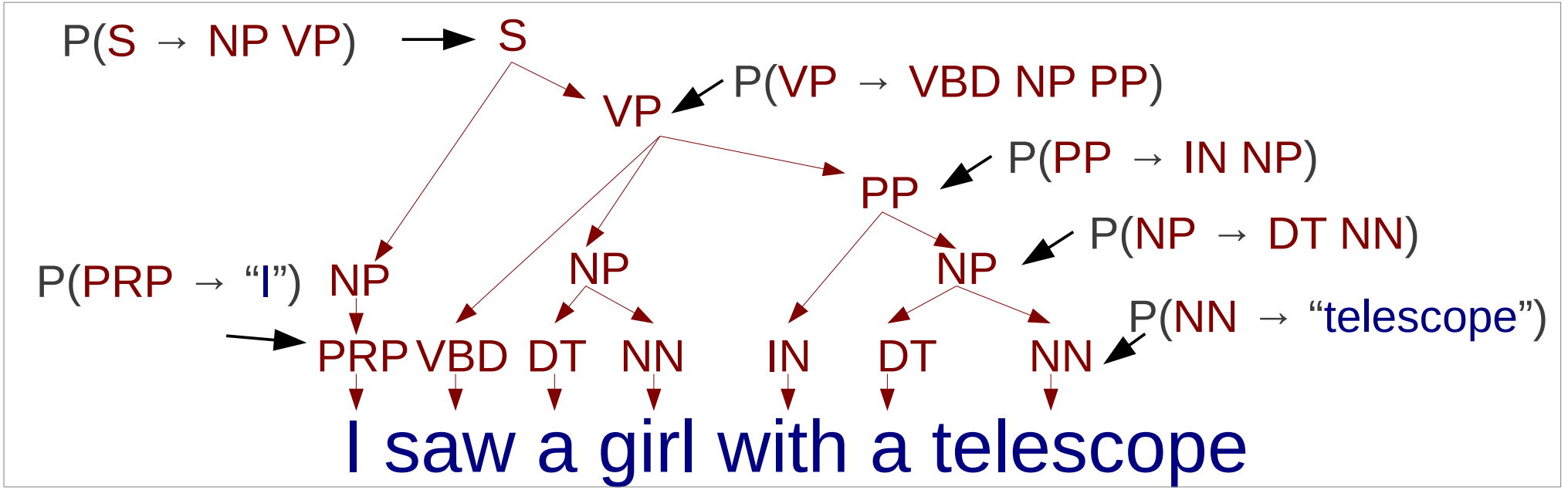
確率的文脈自由文法 (PCFG)

- 構文木の同時確率をどう定義するか？



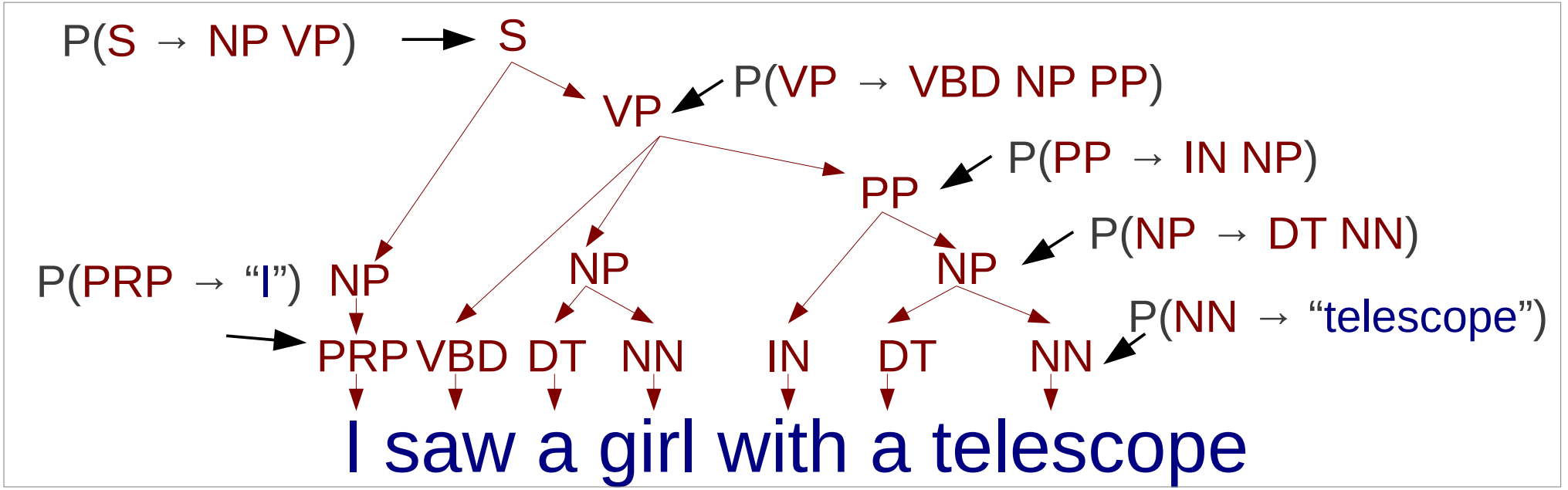
確率的文脈自由文法 (PCFG)

- 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

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”)

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

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

- ビタビアルゴリズムは利用可能か？

確率的構文解析

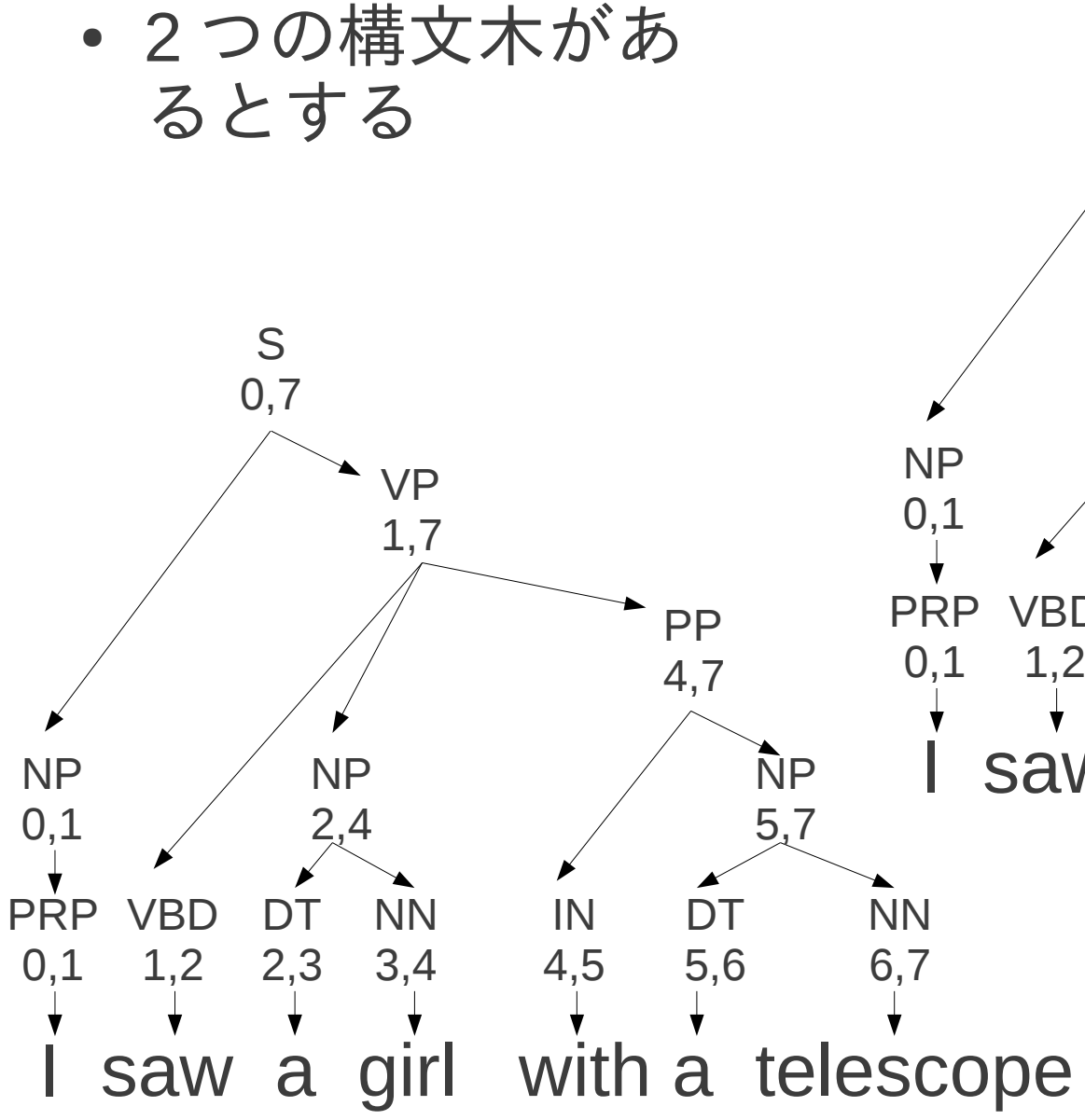
- 構文解析は確率が最大の構文木を探索すること

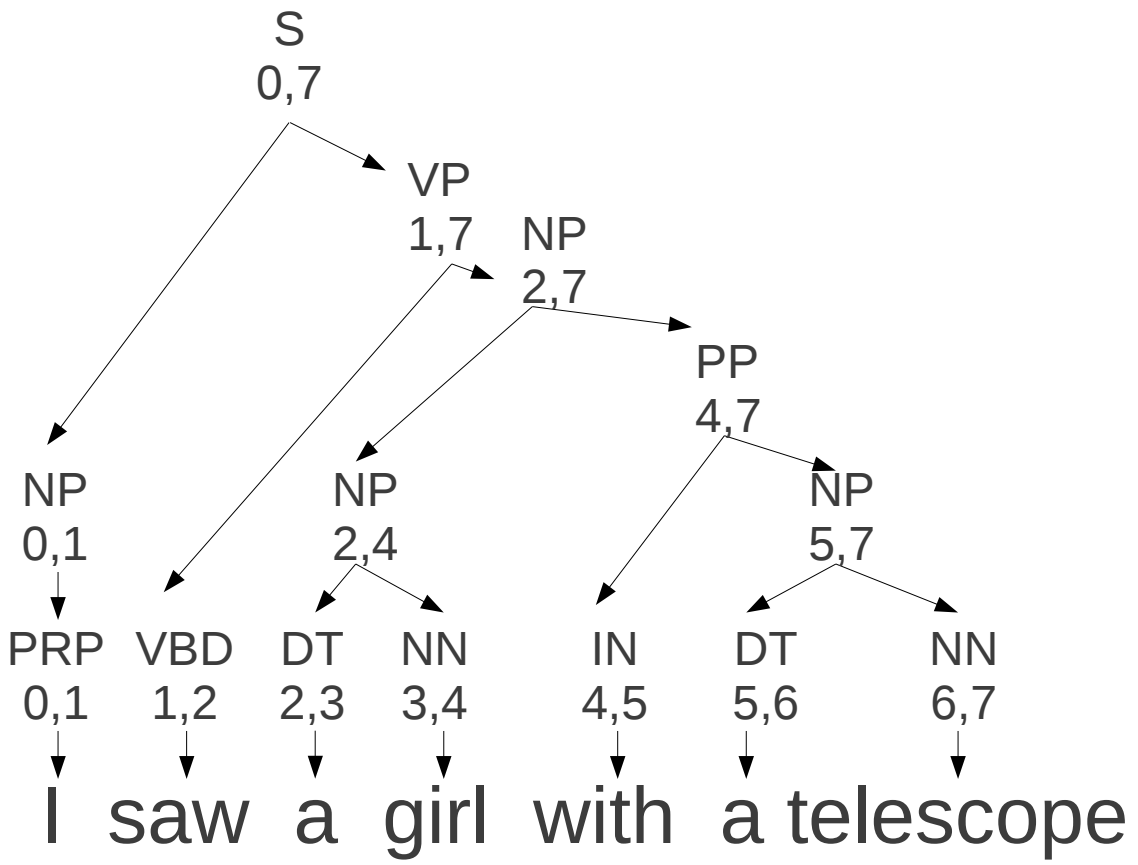
$$\underset{Y}{\operatorname{argmax}}\, P\left(Y, X\right)$$

- 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？

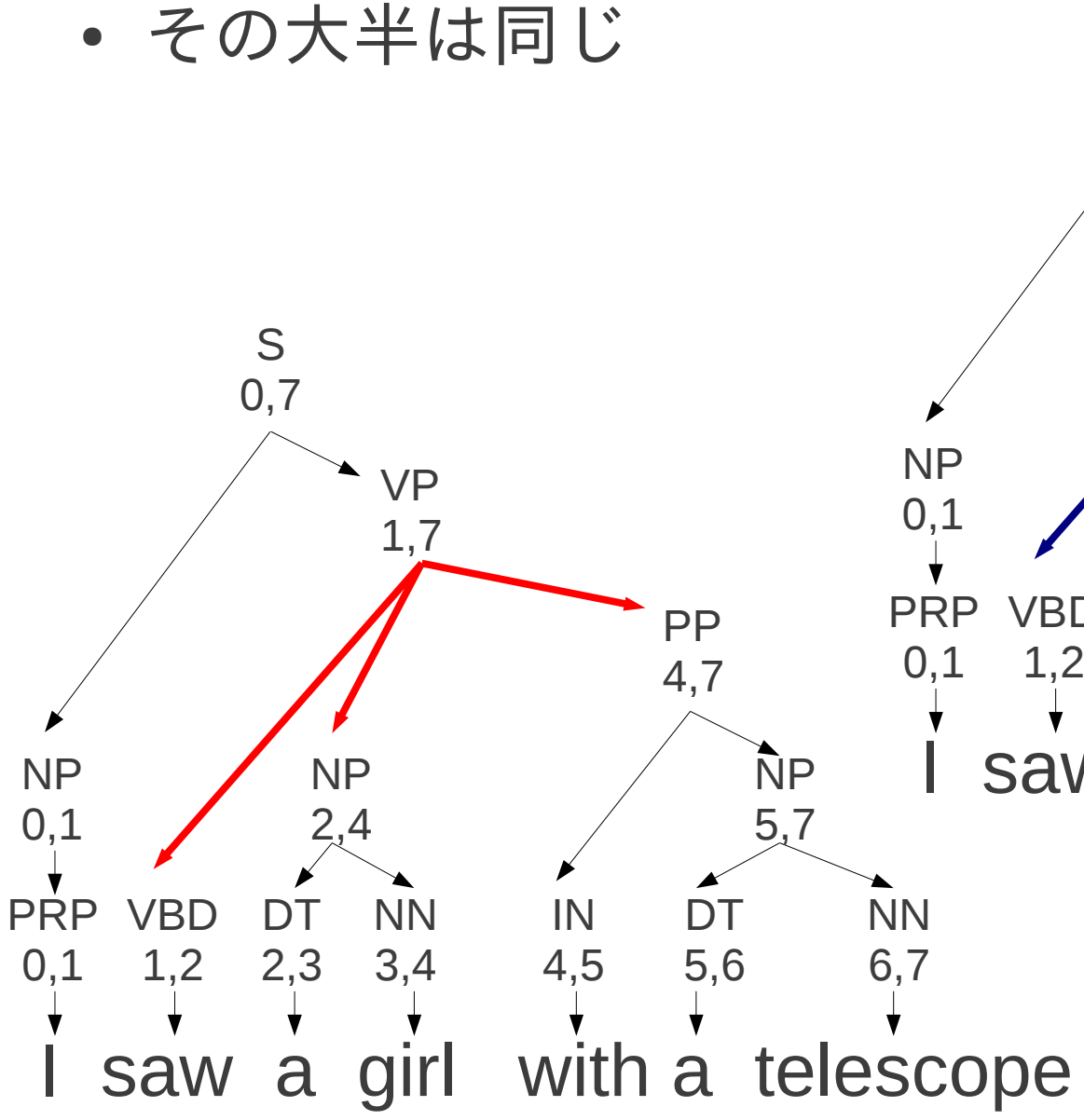
- 2 つの構文木があるとする

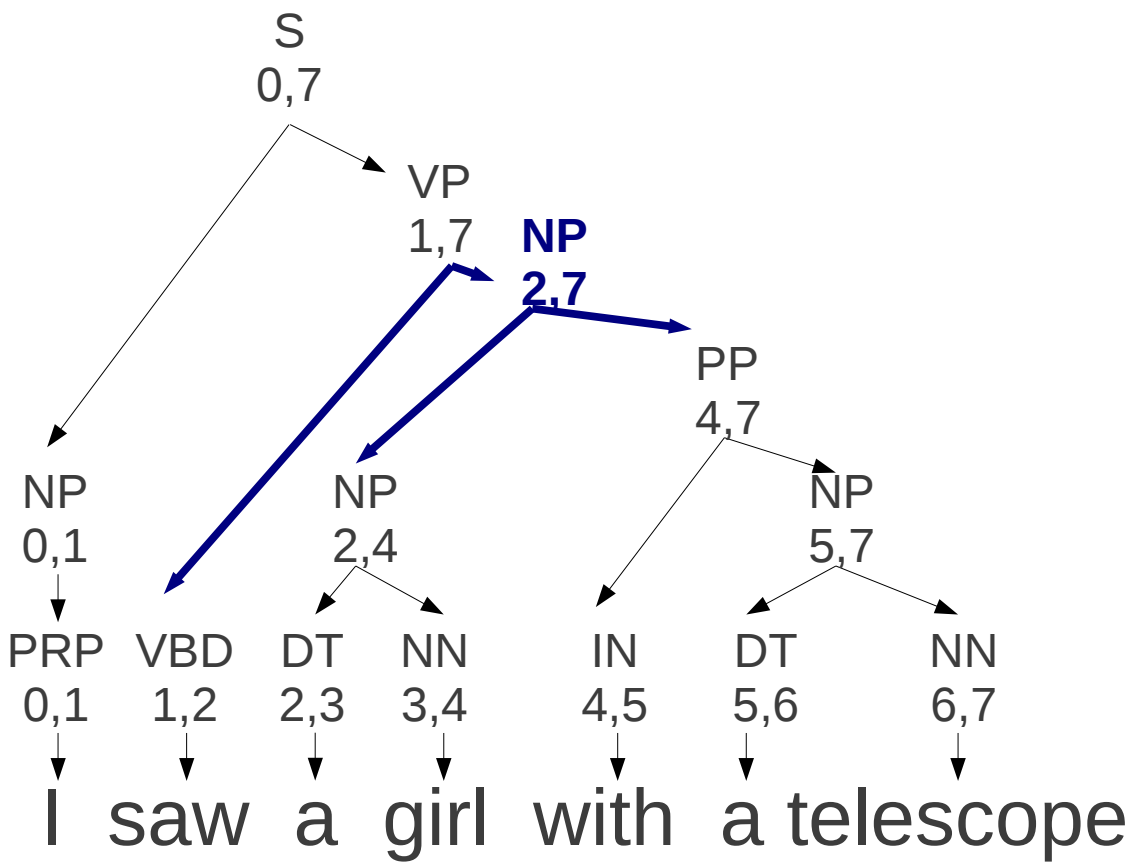




超グラフとは？

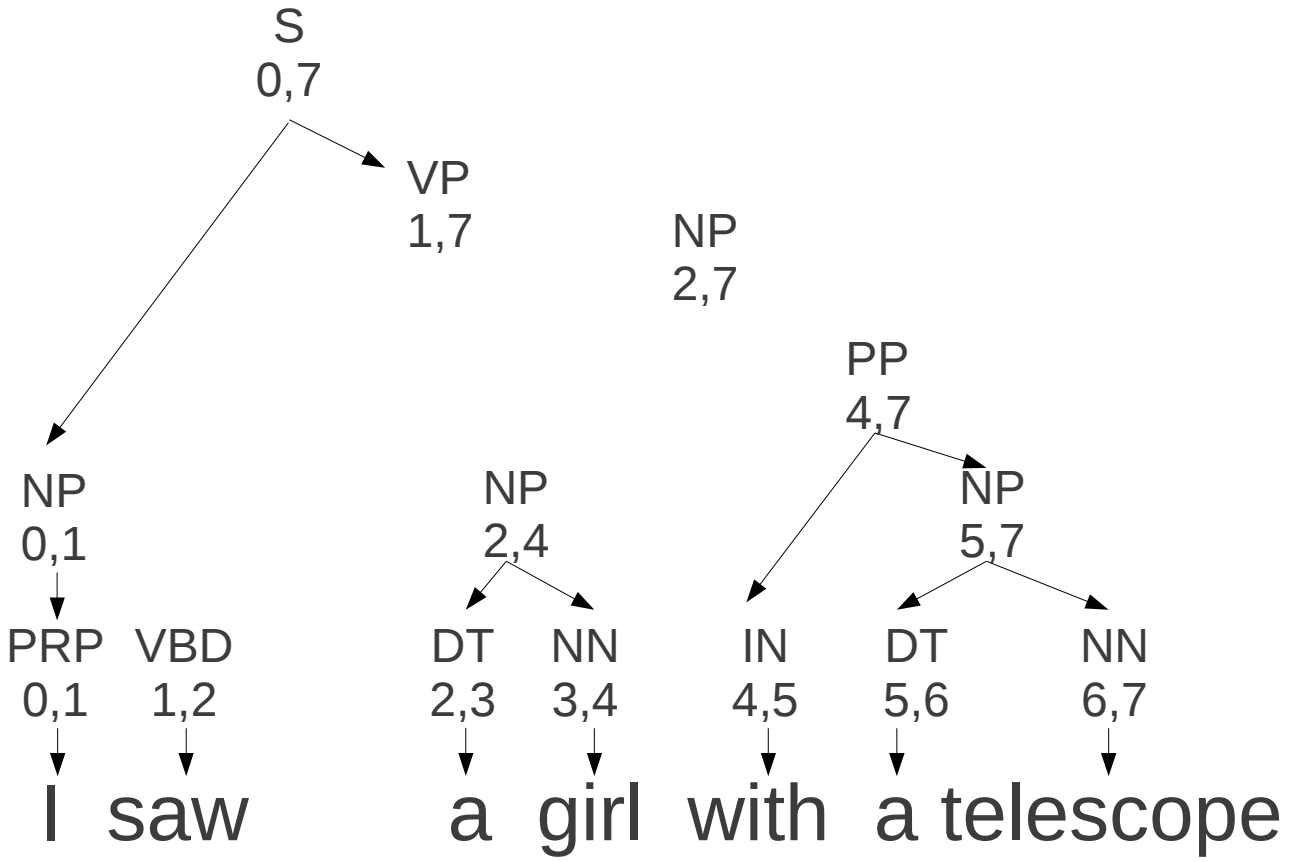
- その大半は同じ





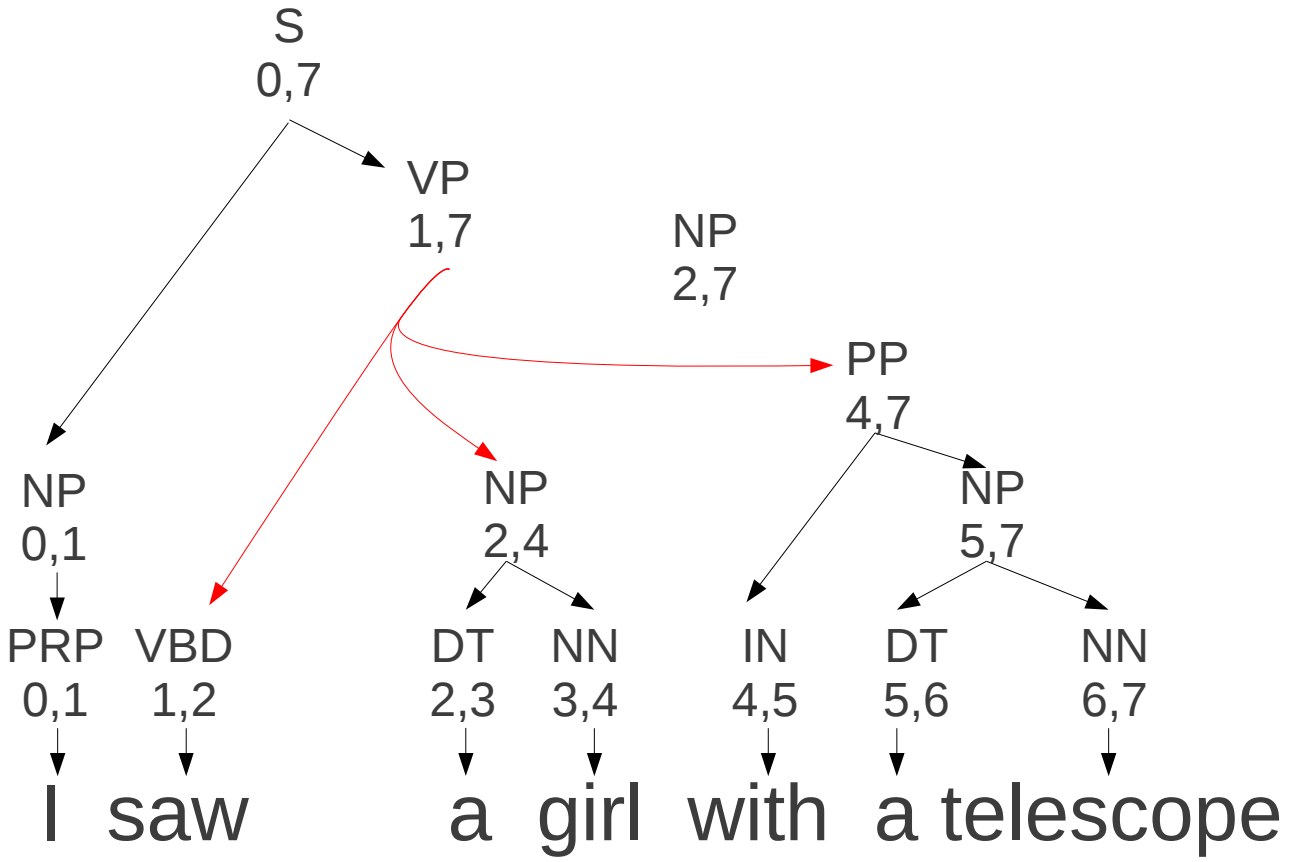
超グラフとは？

- 両方に現れるエッジだけを残すと：



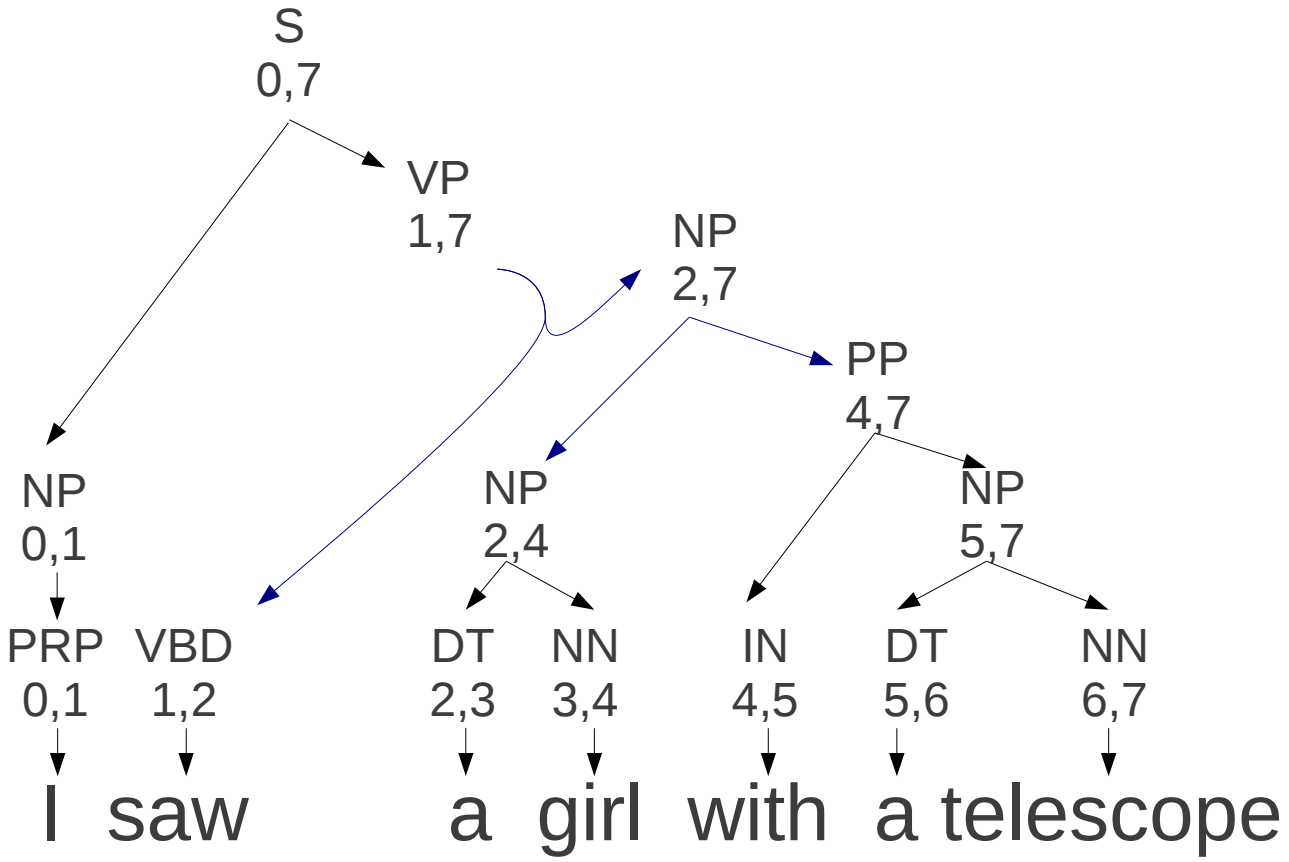
超グラフとは？

- 1 番目の構文木のみに存在するエッジを追加：



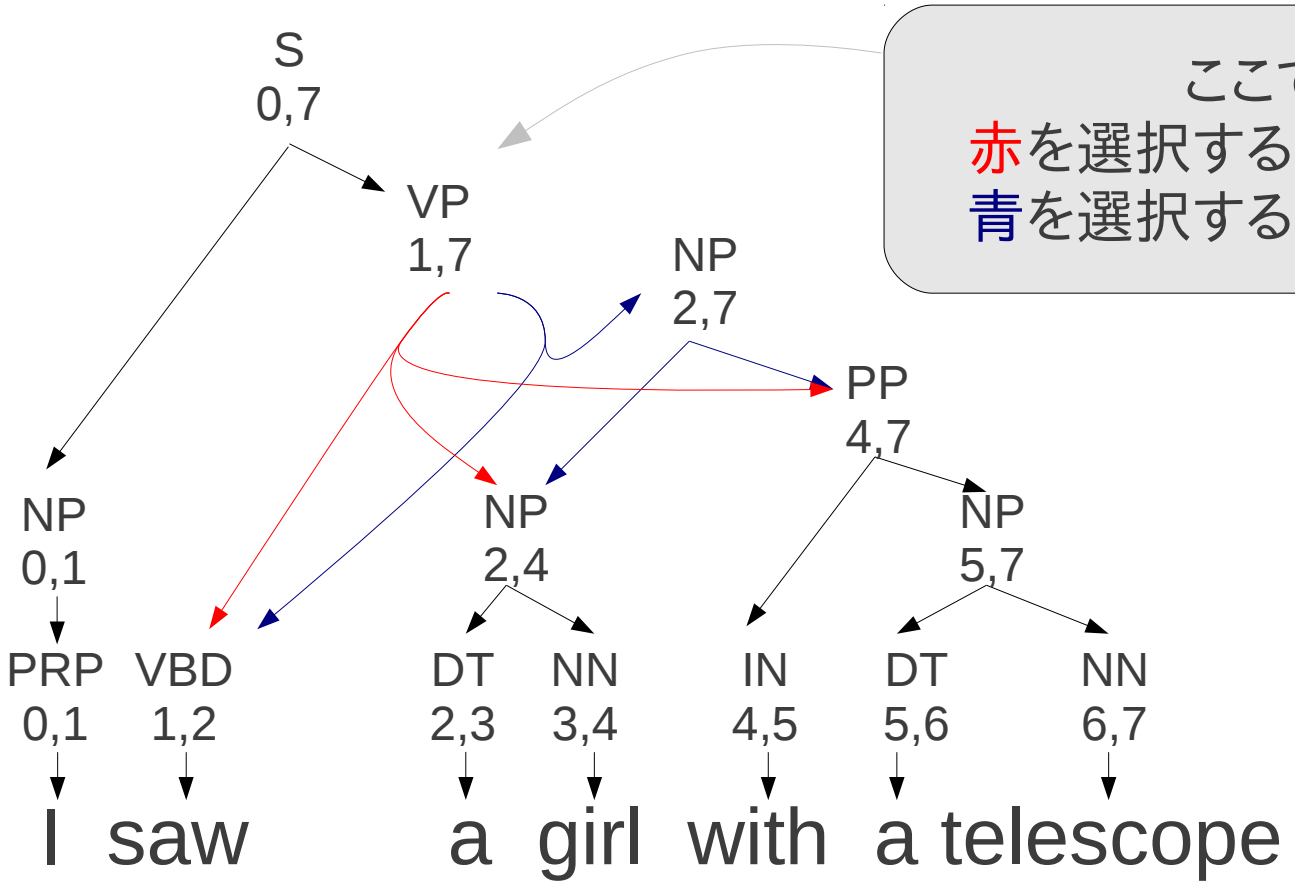
超グラフとは？

- 2 番目の構文木のみに存在するエッジを追加：



超グラフとは？

- 両方の構文木のみに存在するエッジを追加：

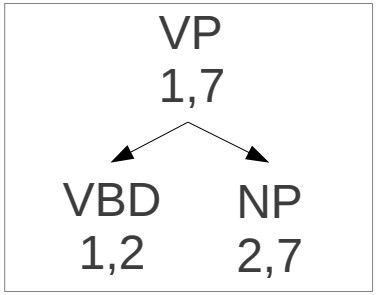


なぜ「超」グラフ？

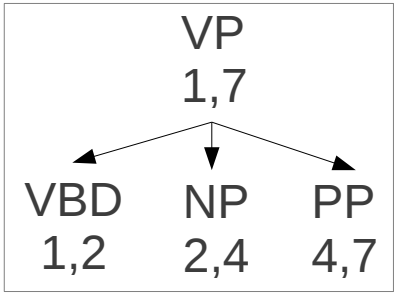
- エッジの「次数」は子の数

次数 1

次数 2

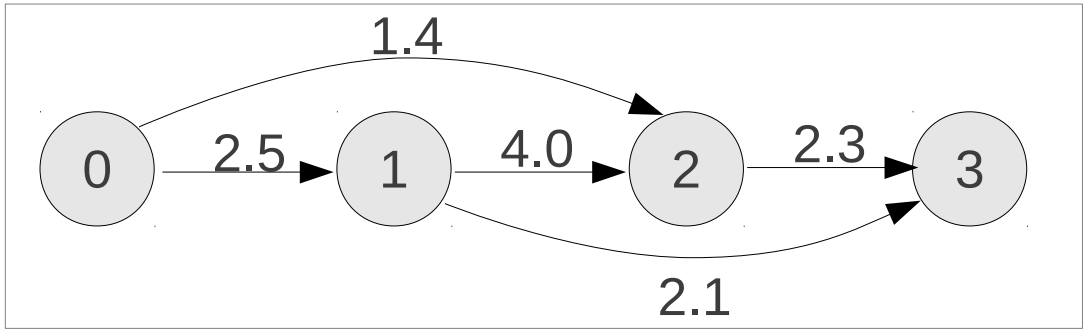


次数 3



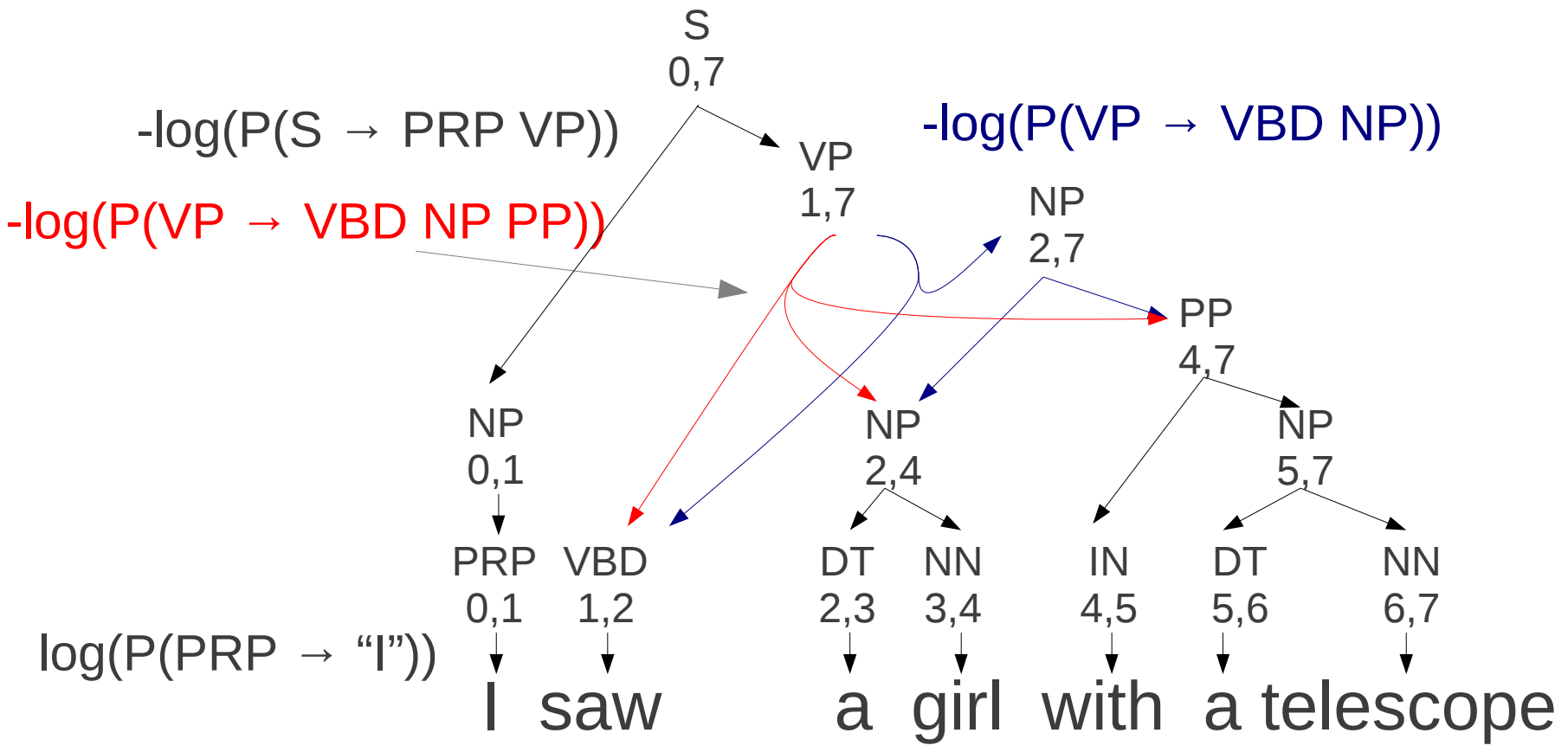
- 超グラフの次数はエッジの次数の最大値
- グラフは次数 1 の超グラフ！

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

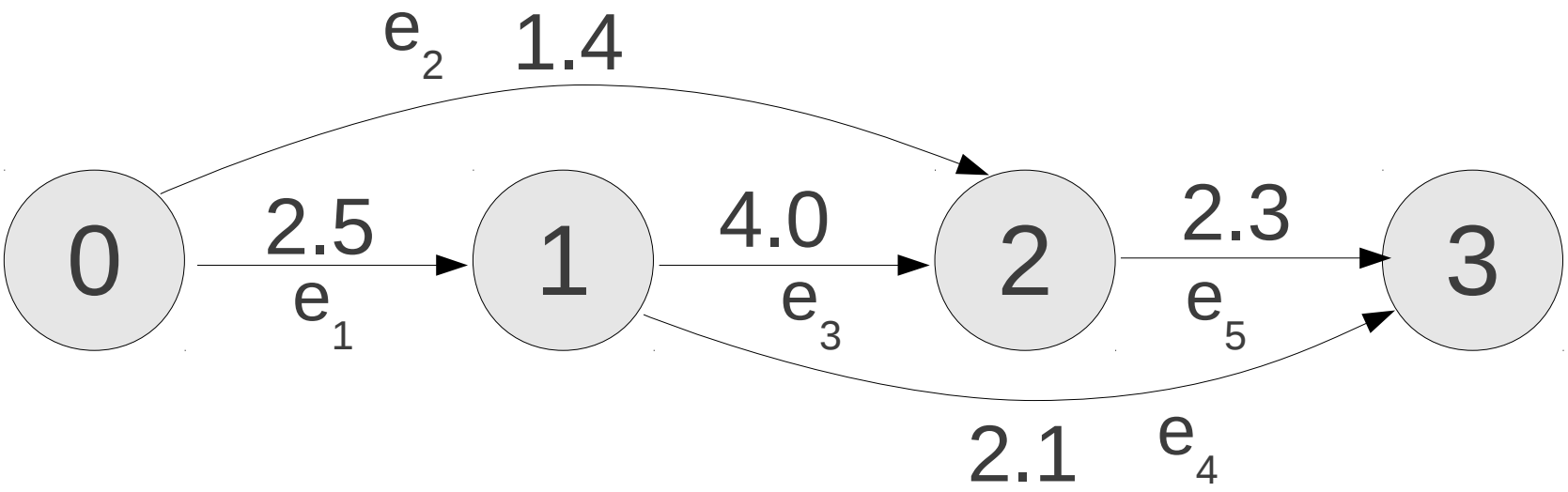
超グラフの探索法

- 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

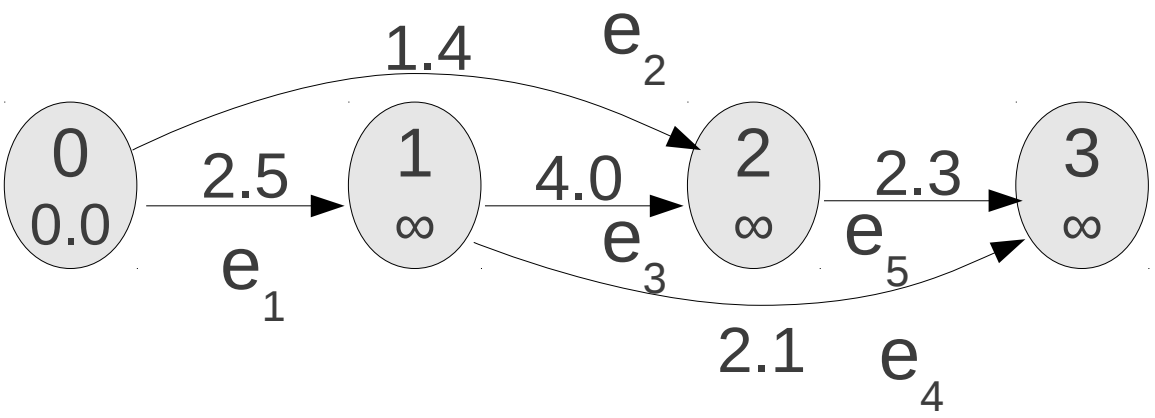
- 構文解析＝超グラフの最もスコアの小さい木を探索
- グラフではビタビアルゴリズムを利用
  - 前向きステップ: 各ノードまでの最短経路を計算
  - 後ろ向き: 最短経路を復元
- 超グラフもほとんど同等のアルゴリズム
  - 内ステップ: 各ノードの最小部分木のスコアを計算
  - 外ステップ: スコア最小の木を復元

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*] = ∞
    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*

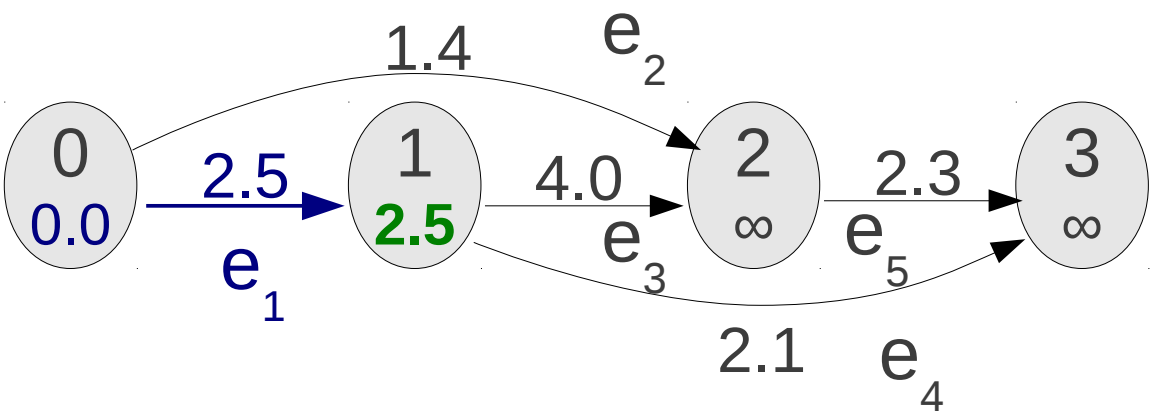
例 :



初期化 :

best_score[0] = 0

例 :



初期化 :

best_score[0] = 0

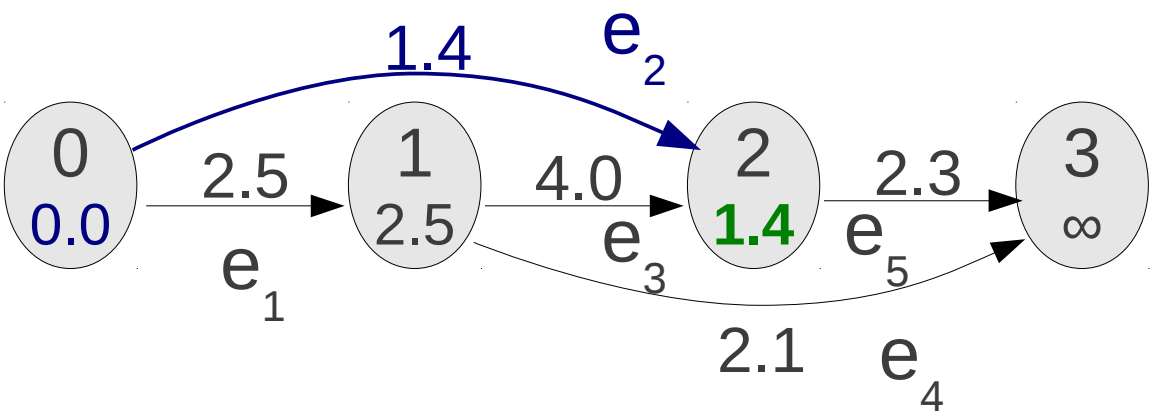
e_1 を計算 :

score = 0 + 2.5 = 2.5 (< ∞)

best_score[1] = 2.5

best_edge[1] = e_1

例 :



初期化 :

best_score[0] = 0

e_1 を計算 :

score = 0 + 2.5 = 2.5 (< ∞)

best_score[1] = 2.5

best_edge[1] = e_1

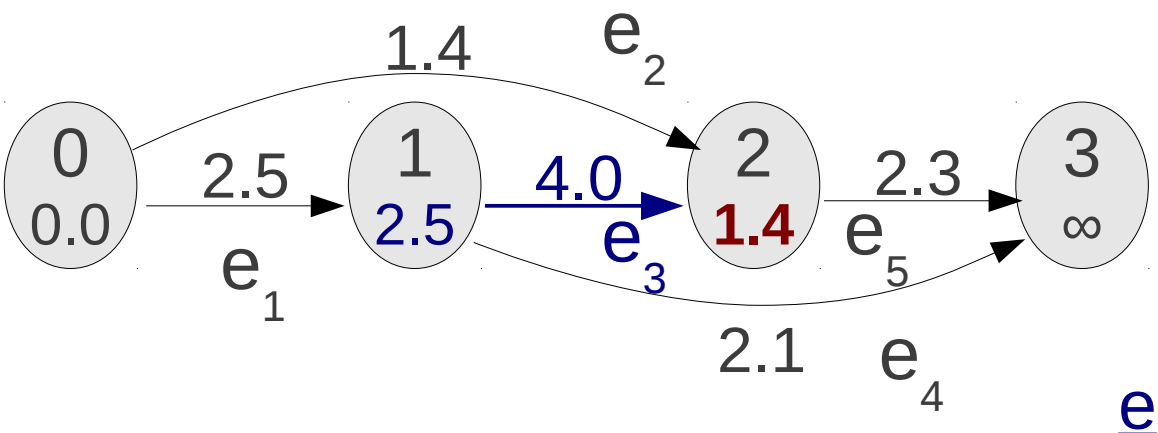
e_2 を計算 :

score = 0 + 1.4 = 1.4 (< ∞)

best_score[2] = 1.4

best_edge[2] = e_2

例 :



初期化 :

best_score[0] = 0

e_1 を計算 :

score = 0 + 2.5 = 2.5 (< ∞)

best_score[1] = 2.5

best_edge[1] = e_1

e_2 を計算 :

score = 0 + 1.4 = 1.4 (< ∞)

best_score[2] = 1.4

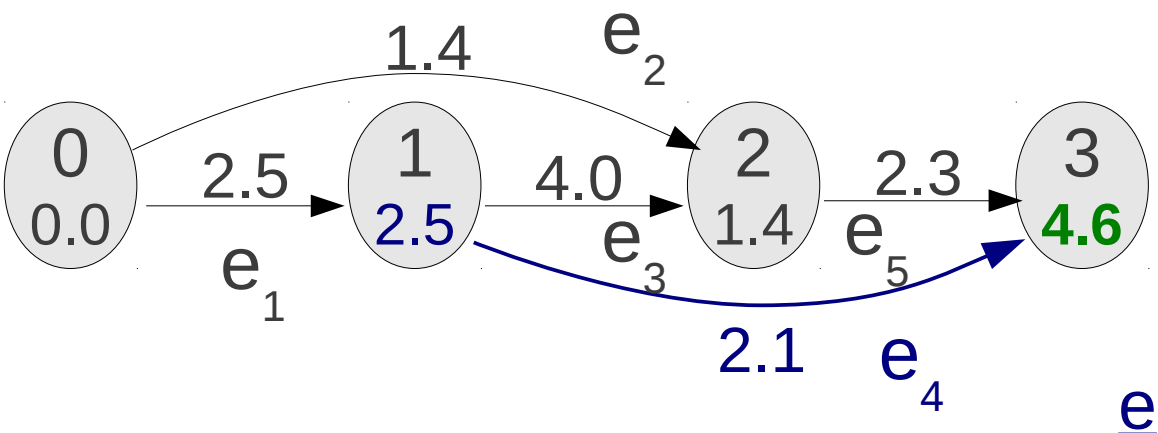
best_edge[2] = e_2

e_3 を計算 :

score = 2.5 + 4.0 = 6.5 (> 1.4)

変更なし !

例 :



初期化 :

best_score[0] = 0

e_1 を計算 :

score = 0 + 2.5 = 2.5 (< ∞)

best_score[1] = 2.5

best_edge[1] = e_1

e_2 を計算 :

score = 0 + 1.4 = 1.4 (< ∞)

best_score[2] = 1.4

best_edge[2] = e_2

e_3 を計算 :

score = 2.5 + 4.0 = 6.5 (> 1.4)

変更なし !

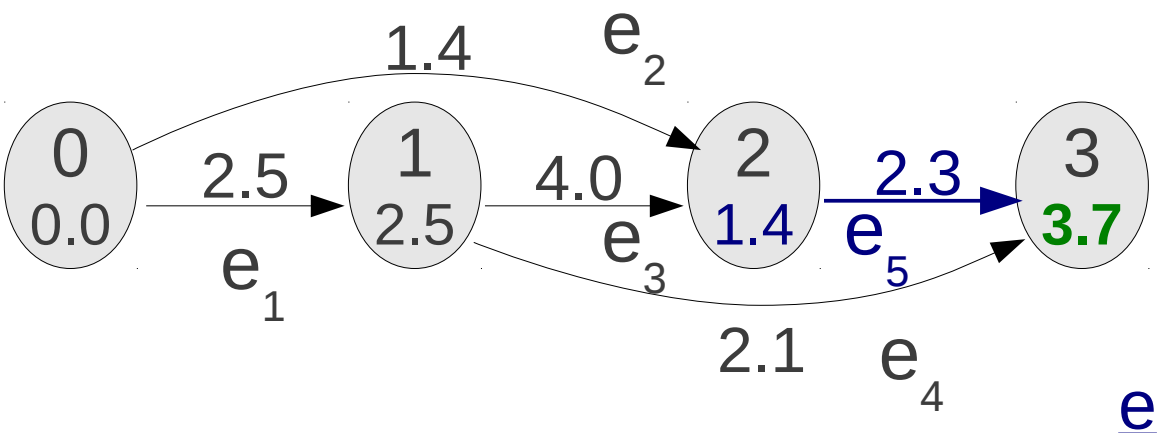
e_4 を計算 :

score = 2.5 + 2.1 = 4.6 (< ∞)

best_score[3] = 4.6

best_edge[3] = e_4

例 :



初期化 :

best_score[0] = 0

e_1 を計算 :

score = 0 + 2.5 = 2.5 (< ∞)

best_score[1] = 2.5

best_edge[1] = e_1

e_2 を計算 :

score = 0 + 1.4 = 1.4 (< ∞)

best_score[2] = 1.4

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 (< ∞)

best_score[3] = 4.6

best_edge[3] = e_4

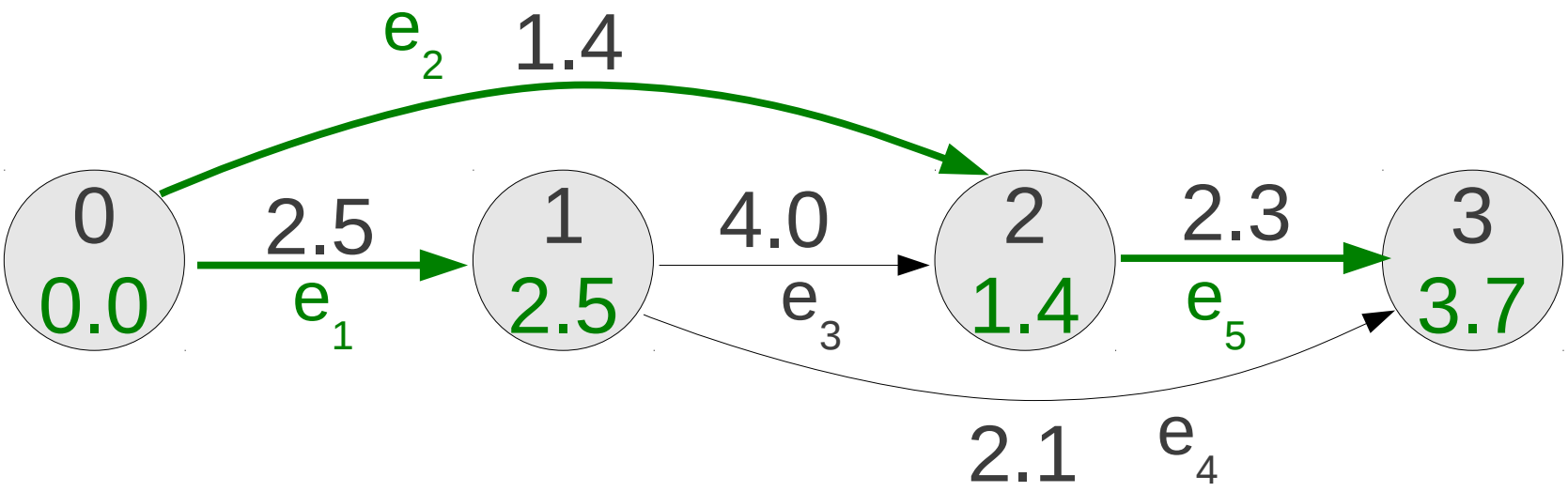
e_5 を計算 :

score = 1.4 + 2.3 = 3.7 (< 4.6)

best_score[3] = 3.7

best_edge[3] = e_5

前向きステップの結果：

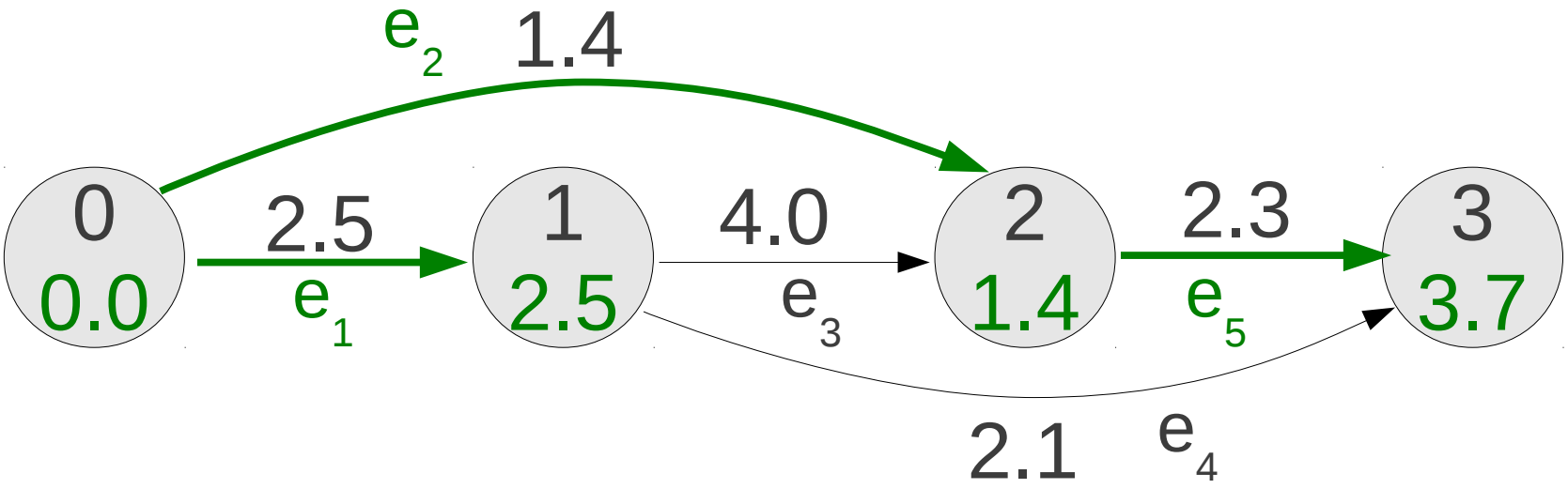


best_score = (0.0, 2.5, 1.4, 3.7)

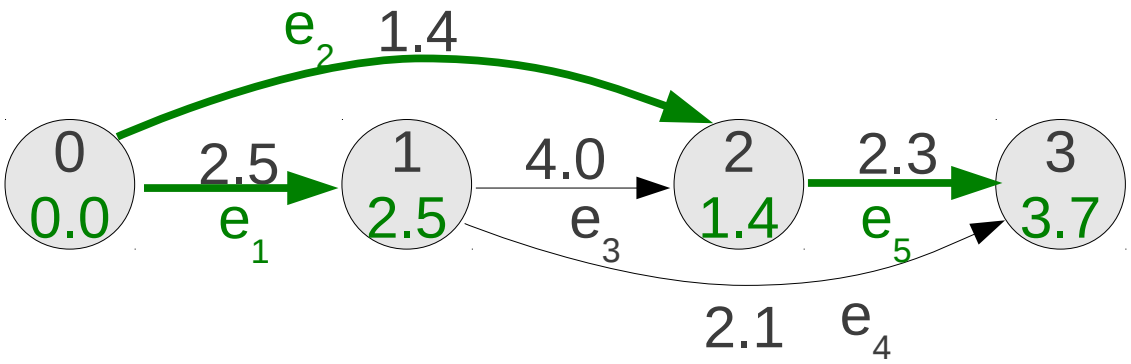
best_edge = (NULL, e_1, e_2, e_5)

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

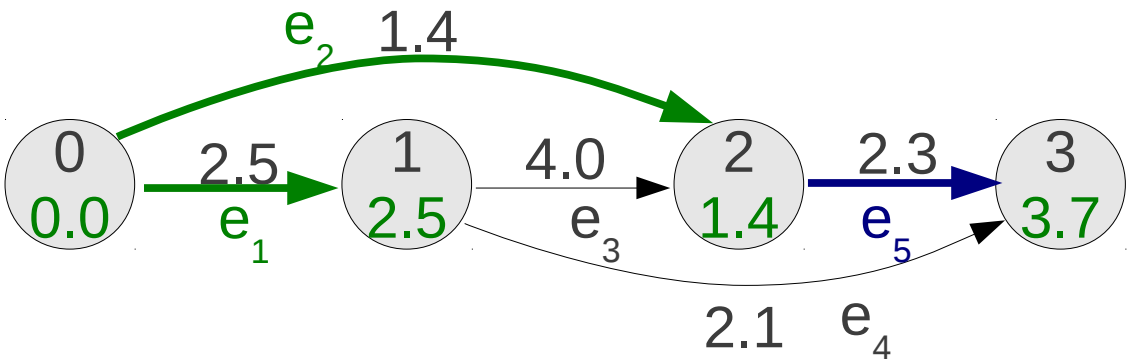


初期化 :

best_path = []

next_edge = best_edge[3] = e_5

後ろ向きステップの例



初期化 :

best_path = []

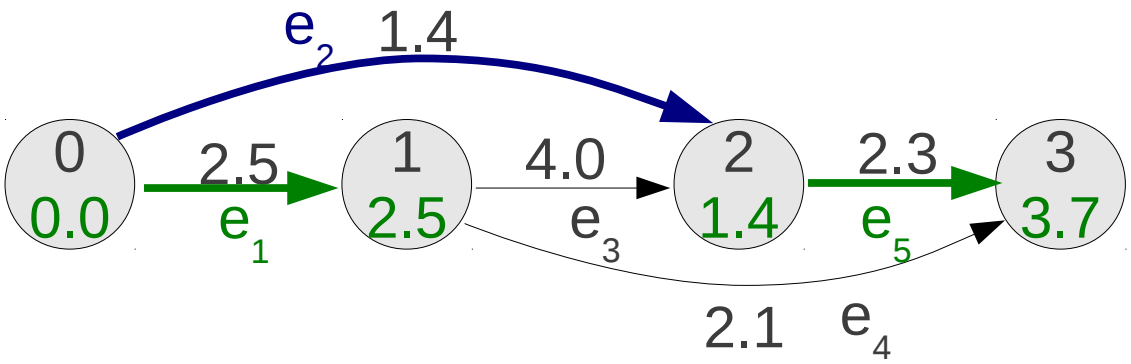
next_edge = best_edge[3] = e_5

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]

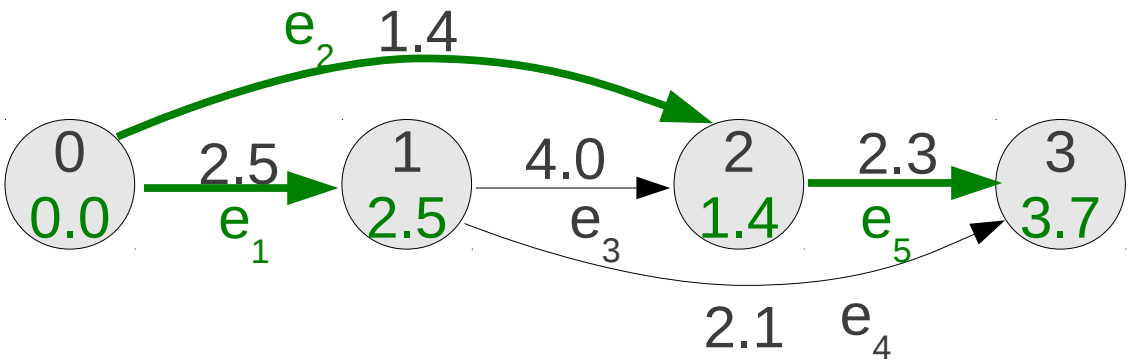
next_edge = best_edge[2] = e_2

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]

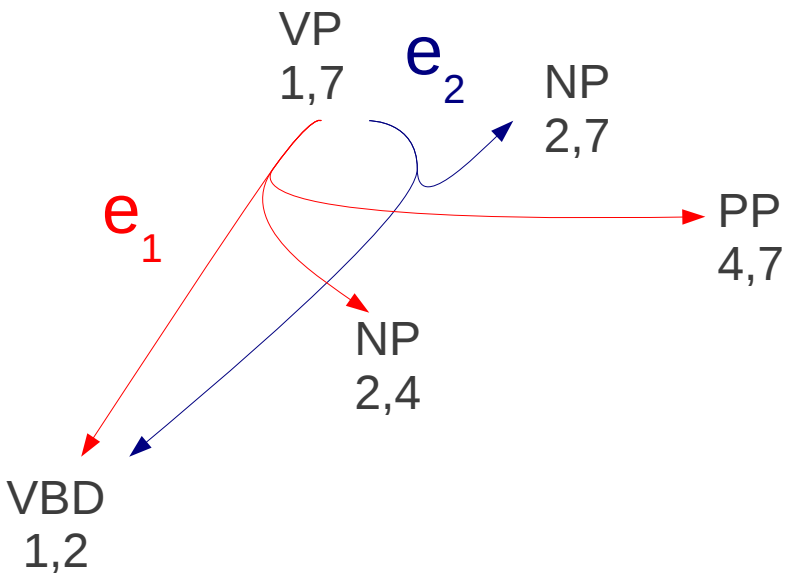
next_edge = best_edge[0] = NULL

逆順に並べ替え :

best_path = [e_2, e_5]

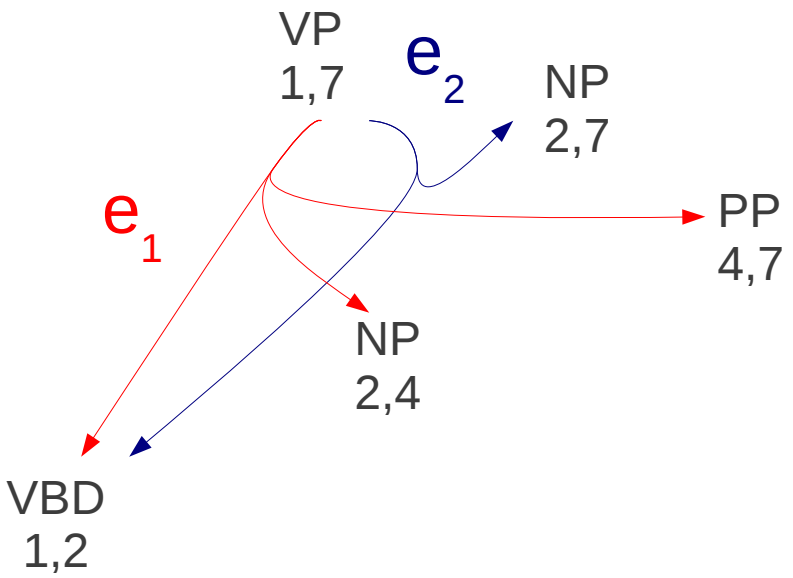
ノードの内ステップ

- 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

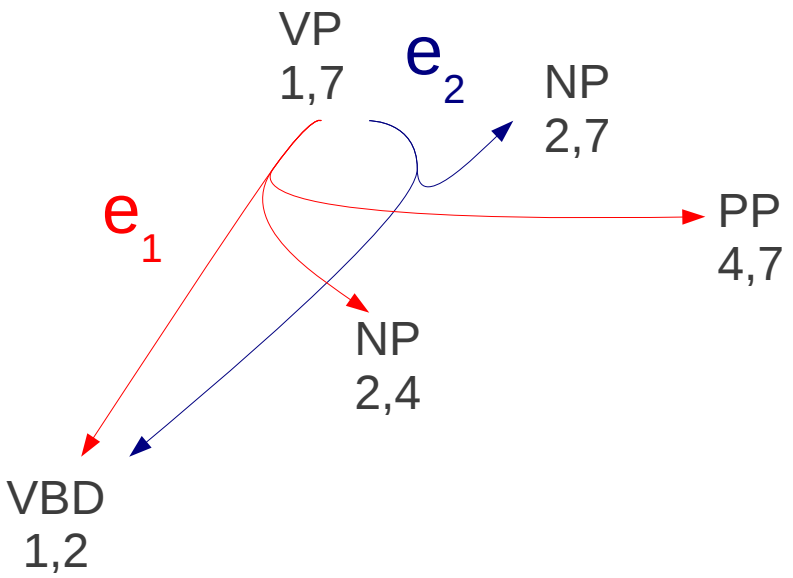


score(e_1) =
  -log(P(VP → VBD NP PP)) +
  best_score[VBD1,2] +
  best_score[NP2,4] +
  best_score[NP2,7]

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

ノードの内ステップ

- VP1,7 の最小スコアを計算



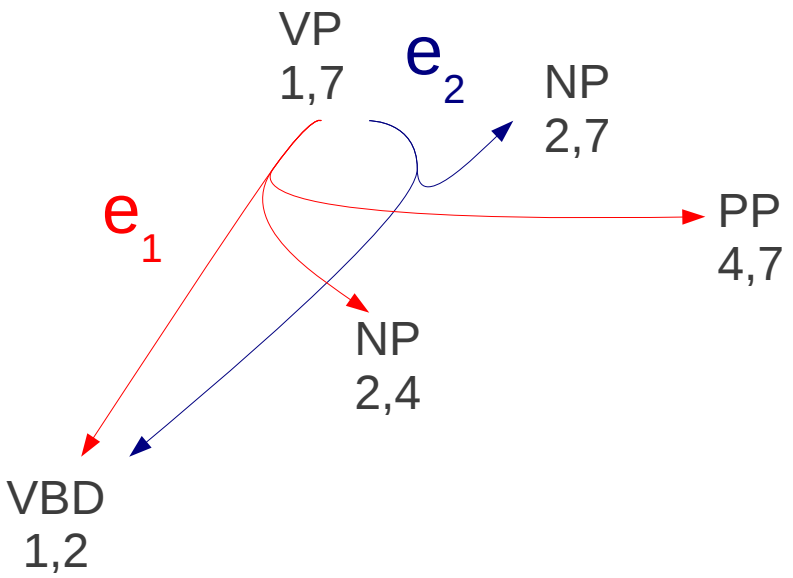
score(e_1) =
  -log(P(VP → VBD NP PP)) +
  best_score[VBD1,2] +
  best_score[NP2,4] +
  best_score[NP2,7]

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

ノードの内ステップ

- VP1,7 の最小スコアを計算



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])

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	文
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

- 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 つもしくは終端記号 1 つ

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

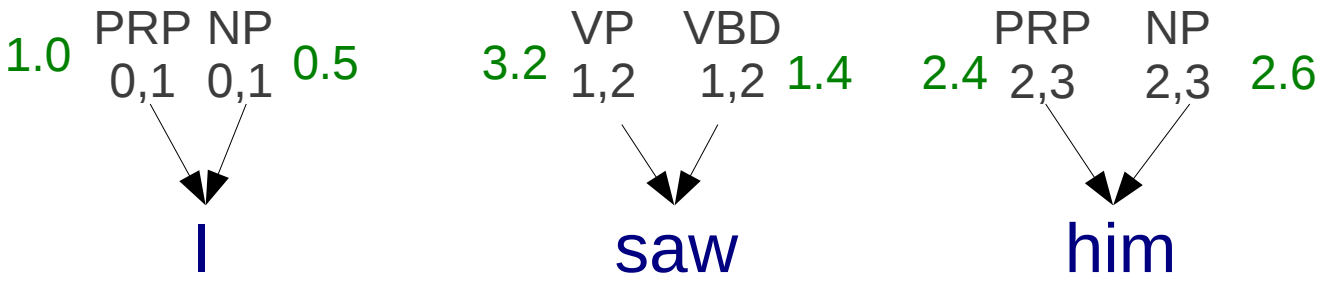
  - この条件を満たさないルールは変更可能

VP → VBD NP PP ➡ VP → VBD NP_PP
NP_PP → NP PP

NP → PRP + PRP → “I” ➡ NP → “I”

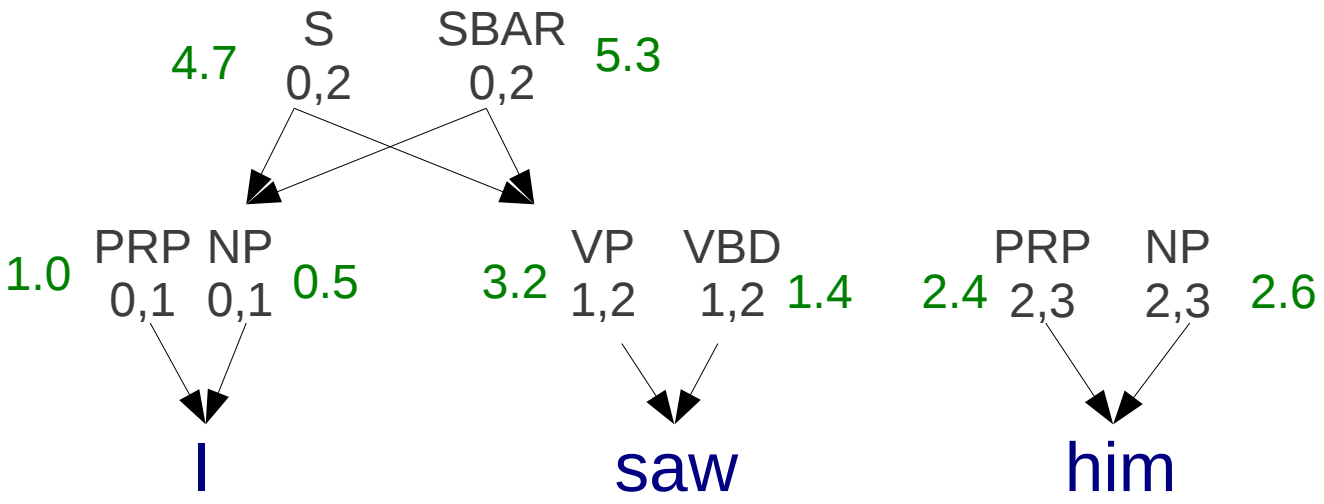
CKY アルゴリズム

- まずは終端記号のルールをスコア付きで展開



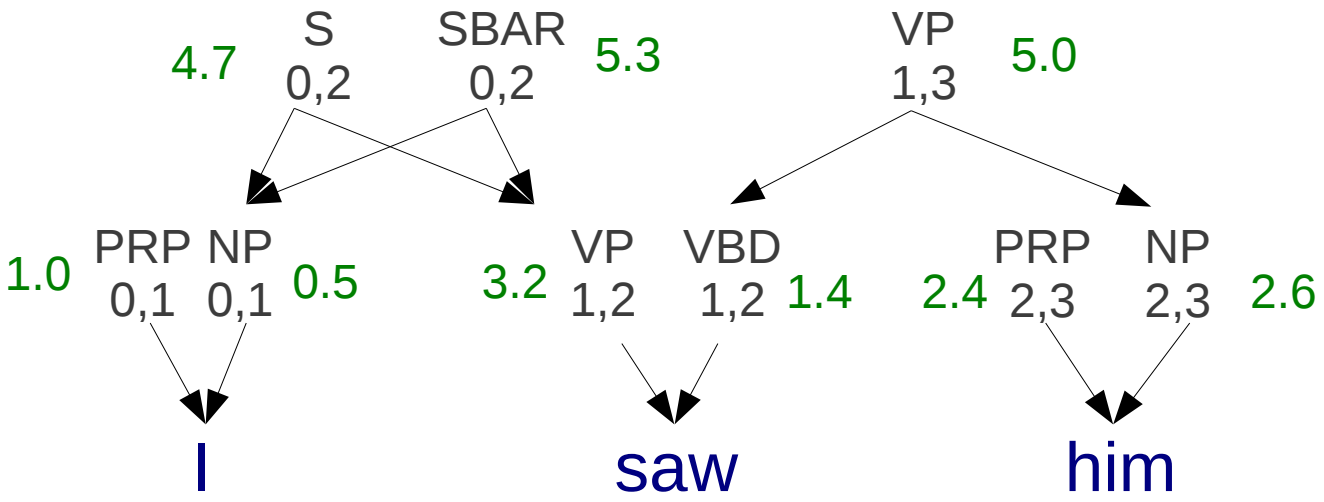
CKY アルゴリズム

- 0,2 のノードを全て展開



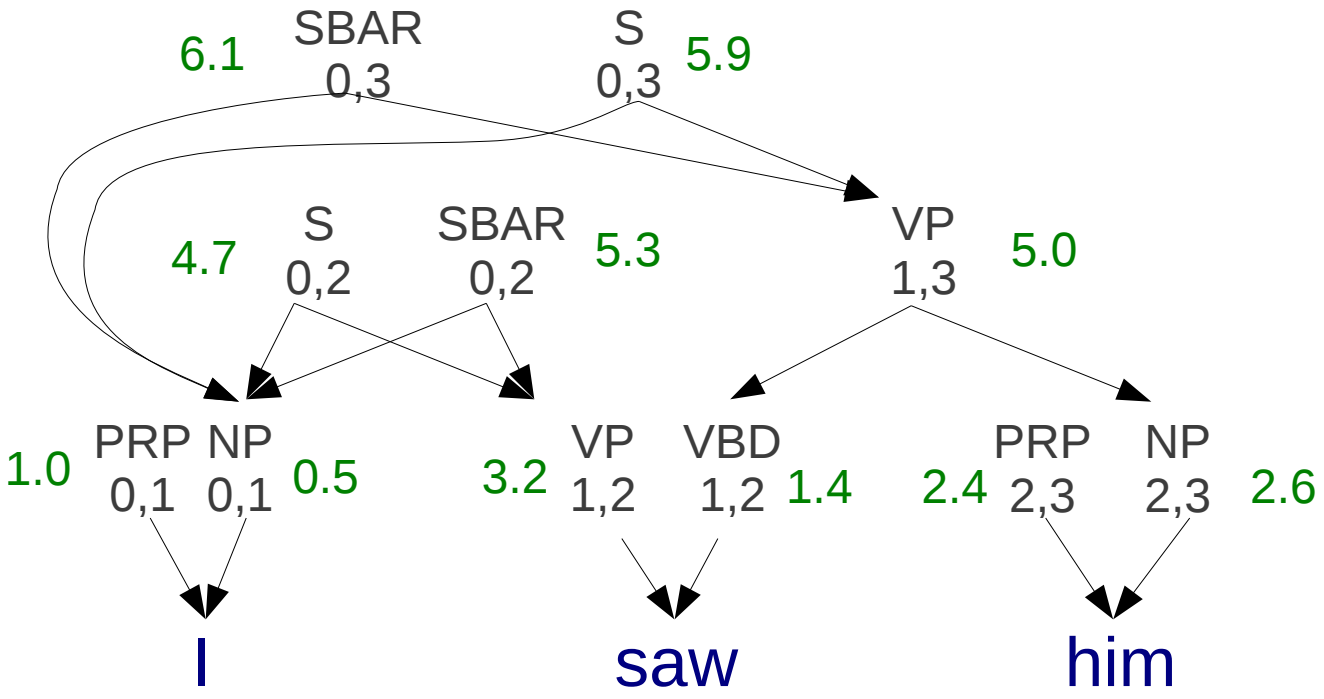
CKY アルゴリズム

- 1,3 のノードを全て展開



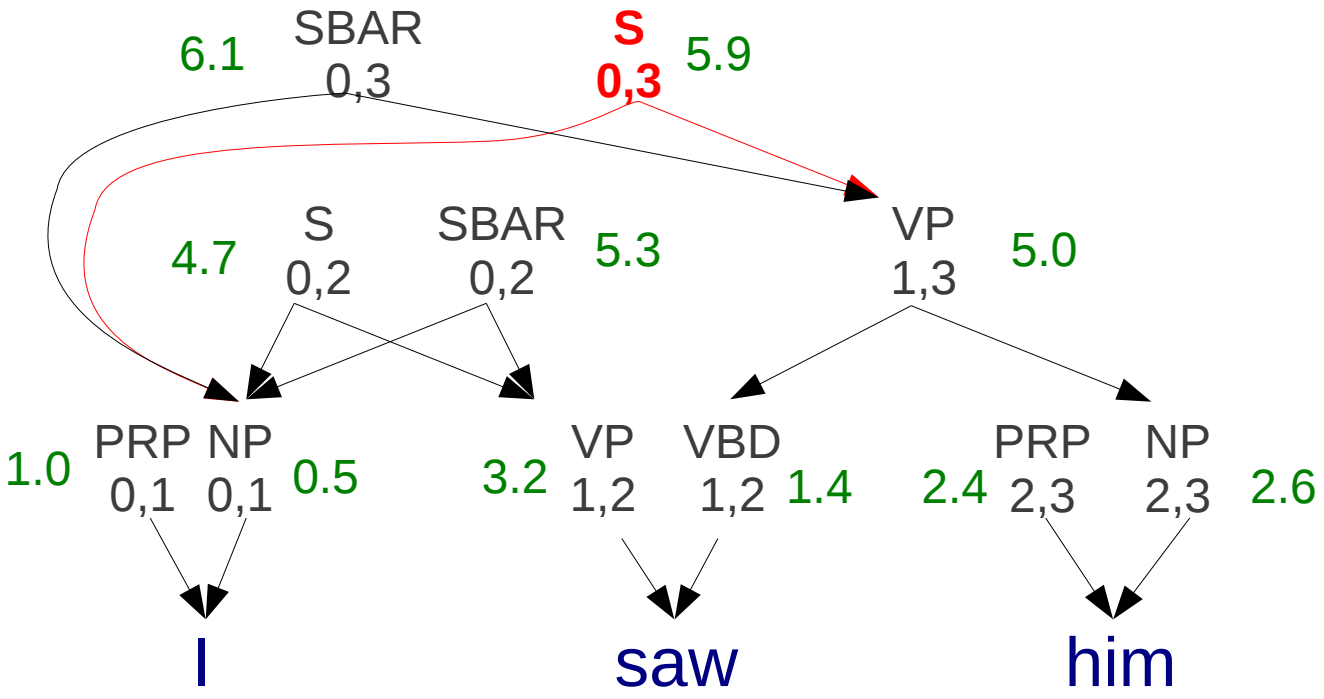
CKY アルゴリズム

- 0,3 のノードを全て展開



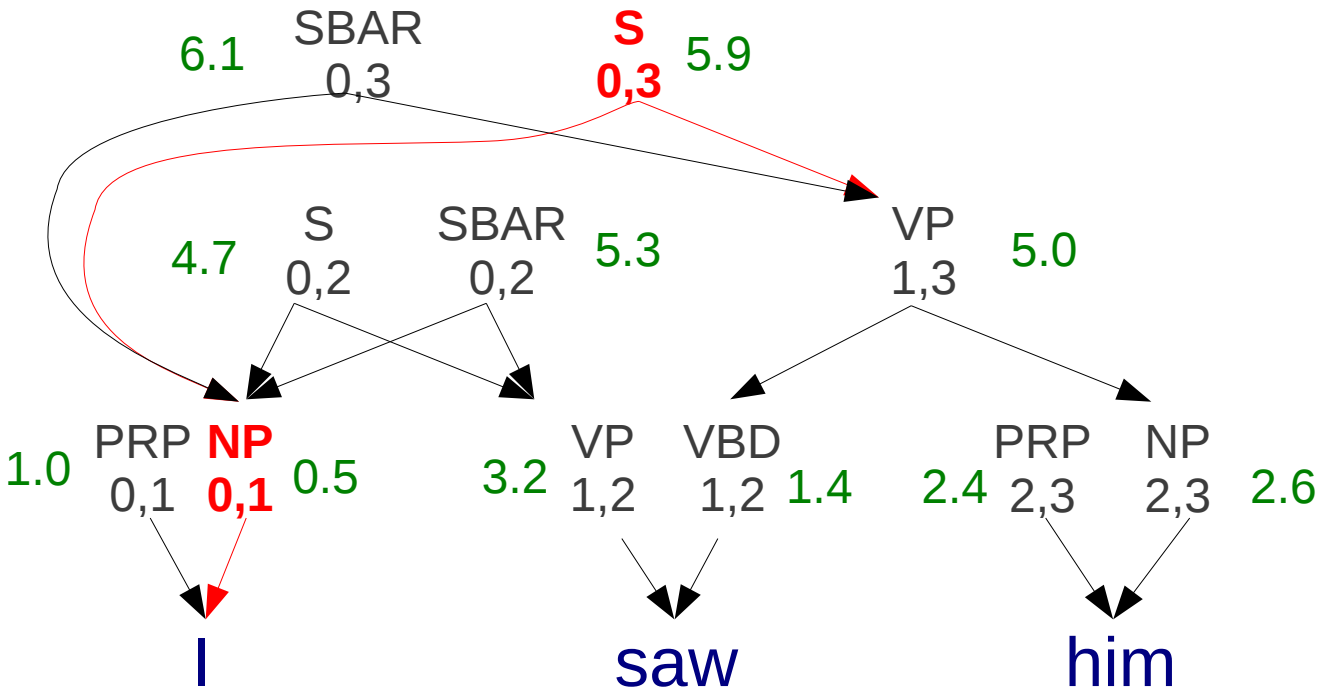
CKY アルゴリズム

- 文を全てカバーする「S」ノードを見つけて、エッジを展開



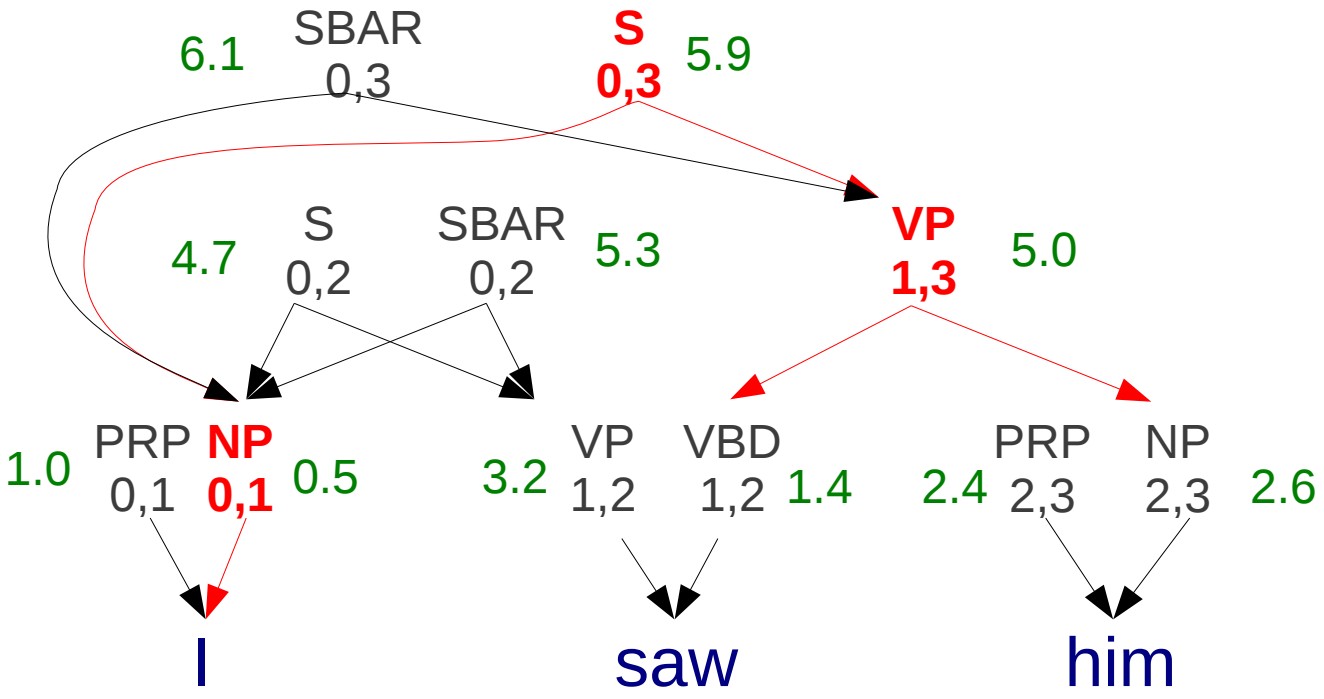
CKY アルゴリズム

- 左の子、右の子を再帰的に展開



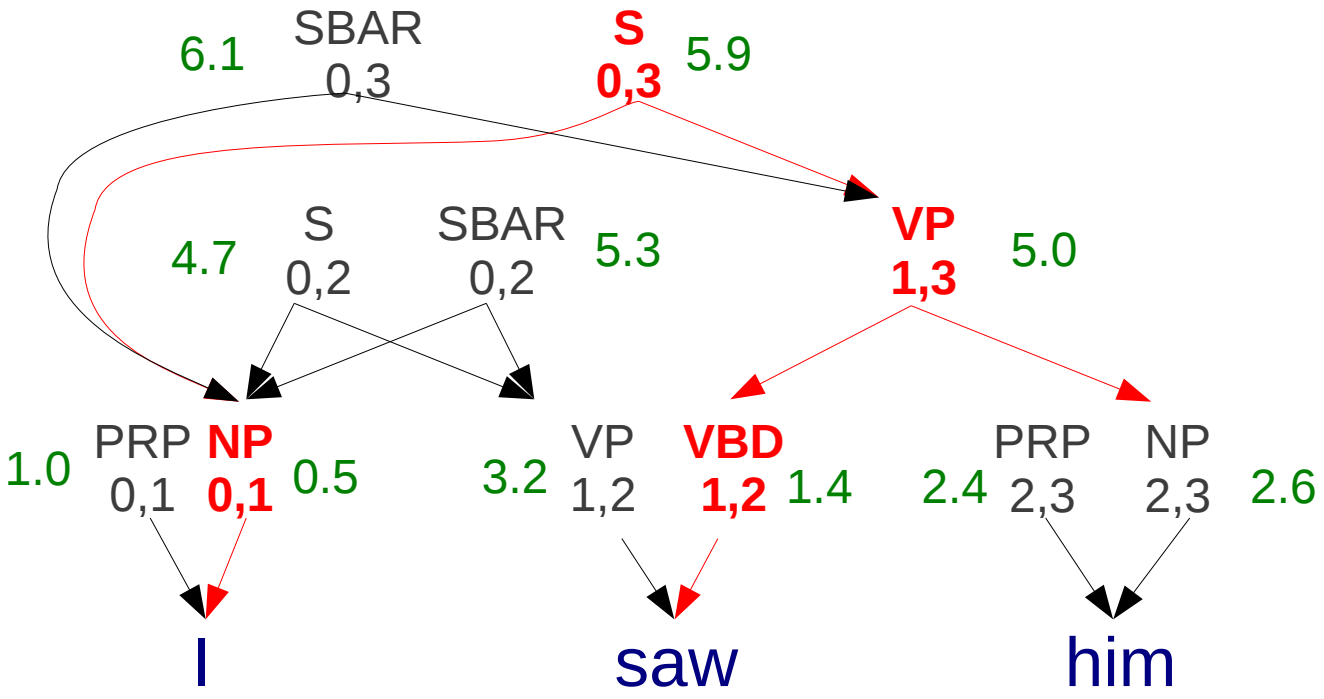
CKY アルゴリズム

- 左の子、右の子を再帰的に展開



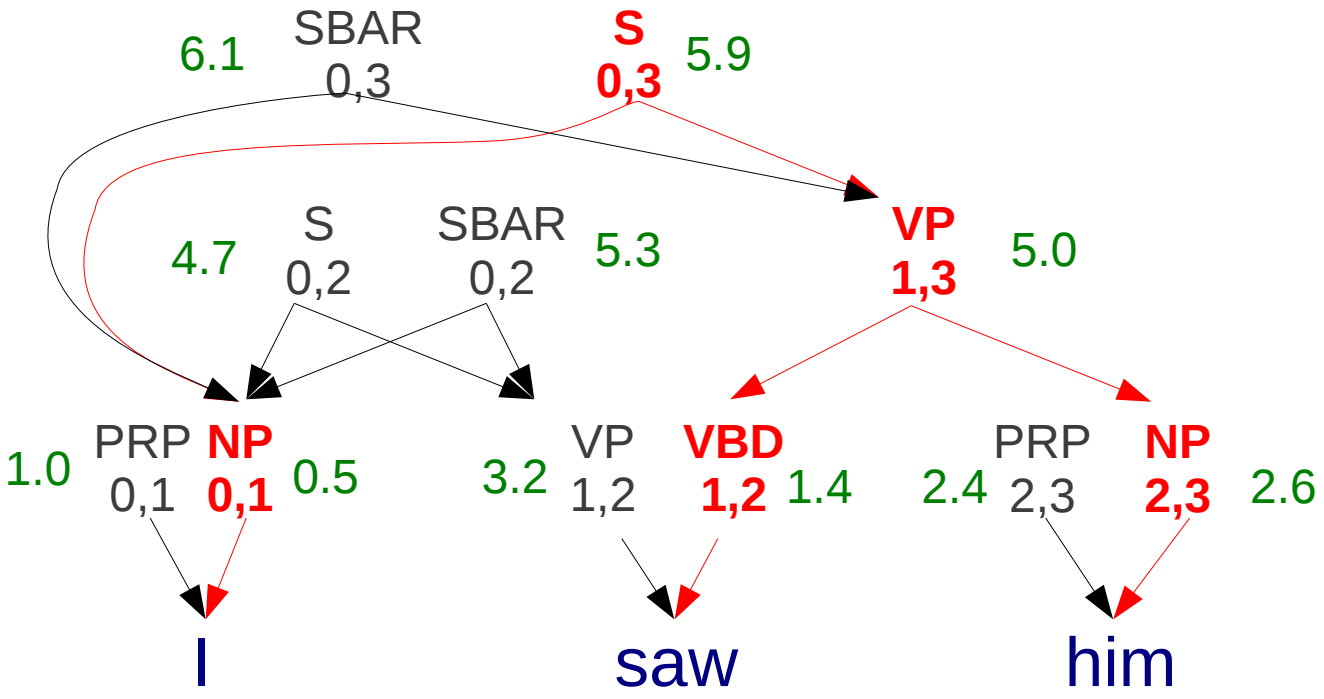
CKY アルゴリズム

- 左の子、右の子を再帰的に展開



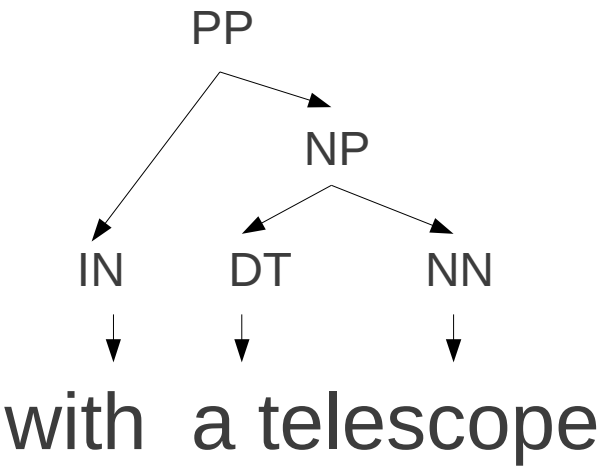
CKY アルゴリズム

- 左の子、右の子を再帰的に展開



構文木の出力

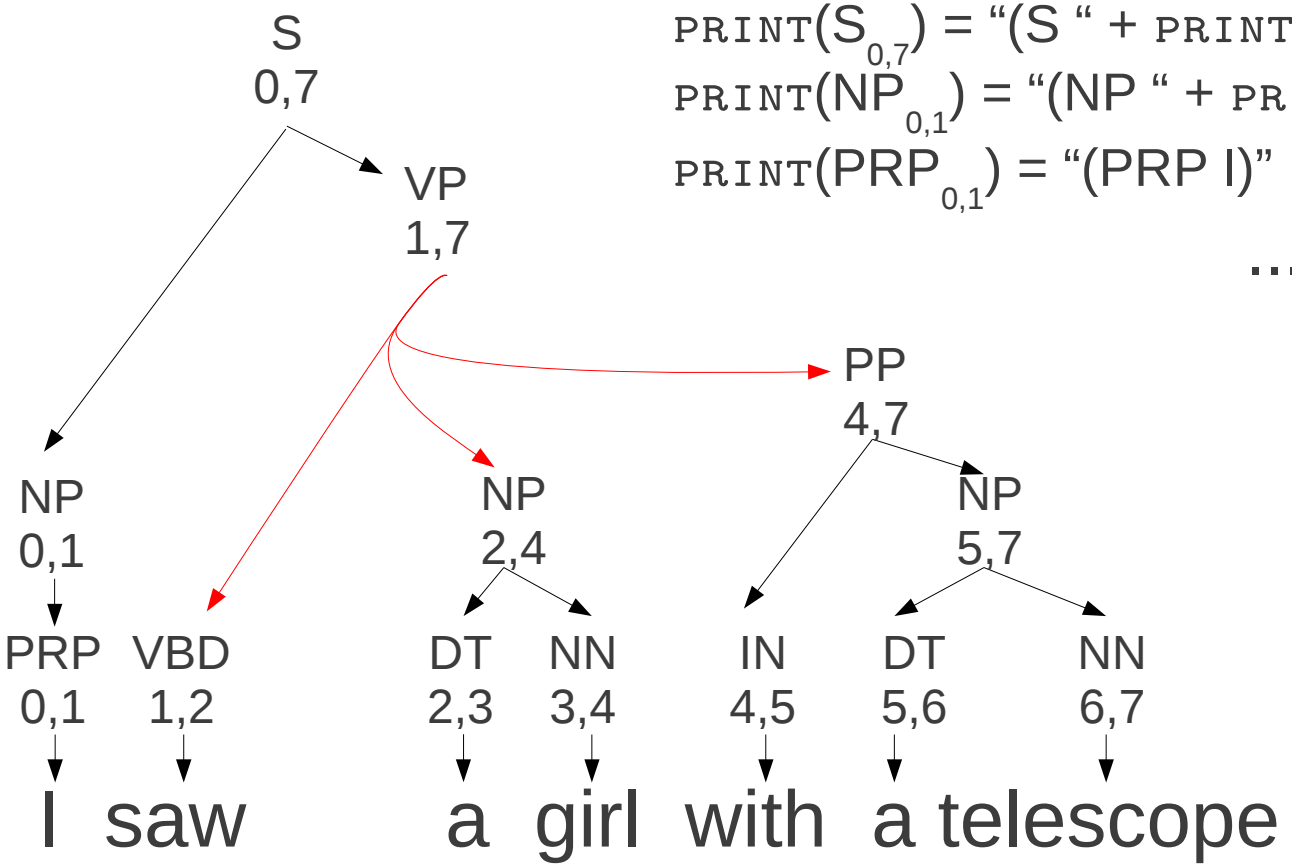
- 構文木の出力：「Penn Treebank 形式」（S 式）




```
(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：



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

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

$\texttt{PRINT}(PRP_{0,1})$ = “(PRP I)”

…

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 ) ...] 形式のマップ
for rule in grammar_file
    split rule into lhs, rhs, prob (with “\t”) # P( 左→右 )= 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 =sym_{i,j}  値 = 最大対数確率
make map best_edge  #  引数 =sym_{i,j}   値 =(lsym_{i,k}, rsym_{k,j})
#  前終端記号を追加
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 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[lsym_i,k] > -∞ and best_score[rsym_k,j] > -∞:
        #  このノード・辺の対数確率を計算
        my_lp = best_score[lsym_i,k] + best_score[rsym_k,j] + logprob
        #  この辺が確率最大のものなら更新
        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 擬似コード : 木を出力

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

```
subroutine PRINT(sym_{i,j}):
  if 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!