Ling 131A
Introduction to NLP with Python

Trees, Grammars and Parsers

Marc Verhagen, Fall 2018

Today

- Grades for assignment 3 are posted
- Quiz 2
- Virtual environments
- Constituents and Trees
- Penn Treebank
- Grammars and Parsing

Quiz 2

- All class notes starting with WordNet
- NLTK Ch 3: 3.4 – 3.7
- NLTK Ch 5: 5.1-5.2, 5.4-5.7
- NLTK Ch 6: 6.1.1-6.1.5, 6.3-6.5
- NLTK Ch 8: 8.1-8.5
- Questions:
  - Python class, regular expressions, WordNet, decision trees or bayes, taggers, classifiers, vectors, evaluation, trees, grammars, parsers

Virtual Environments

- Dependency hell
- Third party packages installed at a few spots


```
>>> import site
>>> site.getsitepackages()
['/usr/local/Cellar/python/3.6.5/Frameworks
/Python.framework/Versions/3.6/lib/python3.6
/site-packages',
'/Library/Python/3.6/site-packages']
>>>
```


- But you can only have one version of a package
- What if two programs require different versions?

Virtual Environments

- Create an isolated environment for Python projects
  - each project has its own dependencies, regardless of what dependencies other projects have
  - https://docs.python.org/3/library/venv.html


```
$ python3 -m venv /virtual-environments/nltk
$ source /virtual-environments/nltk/bin/activate
$ which python
/virtual-environments/nltk/bin/python
```

Beyond Bigrams

- Bigrams work well for modeling part of speech tags
- But not so much for sentences
  - *"The worst part and clumsy looking for whoever heard light."*
  - Perfectly okay when seen as a sequence of tags within an bigram model
- Using a grammar as a model is better here

Grammar Model

- Here is one rule from a grammar
- If `X1` and `X2` are both phrases of grammatical category C, then the phrase `[X1 and X2]` is also of category C
- This is violated by

  `[The worst part]`^{NP} `and [clumsy looking]`^{AP}

Constituents

- A group of words that form a single unit in a hierarchical structure
- Substitution
  - A constituent of one type can typically be replaced by another one of the same type
    - [The little bear] [sleeps]
    - [The bear] [sleeps]
    - [She] [sleeps]
  - The meaning changes, but the sentence is syntactically very similar

Constituent structure

the	little	bear	saw	the	fine	fat	trout	in	the	brook
the	bear		saw	the	trout			in	it	
He			saw	it				there		
He			ran					there		
He			ran							

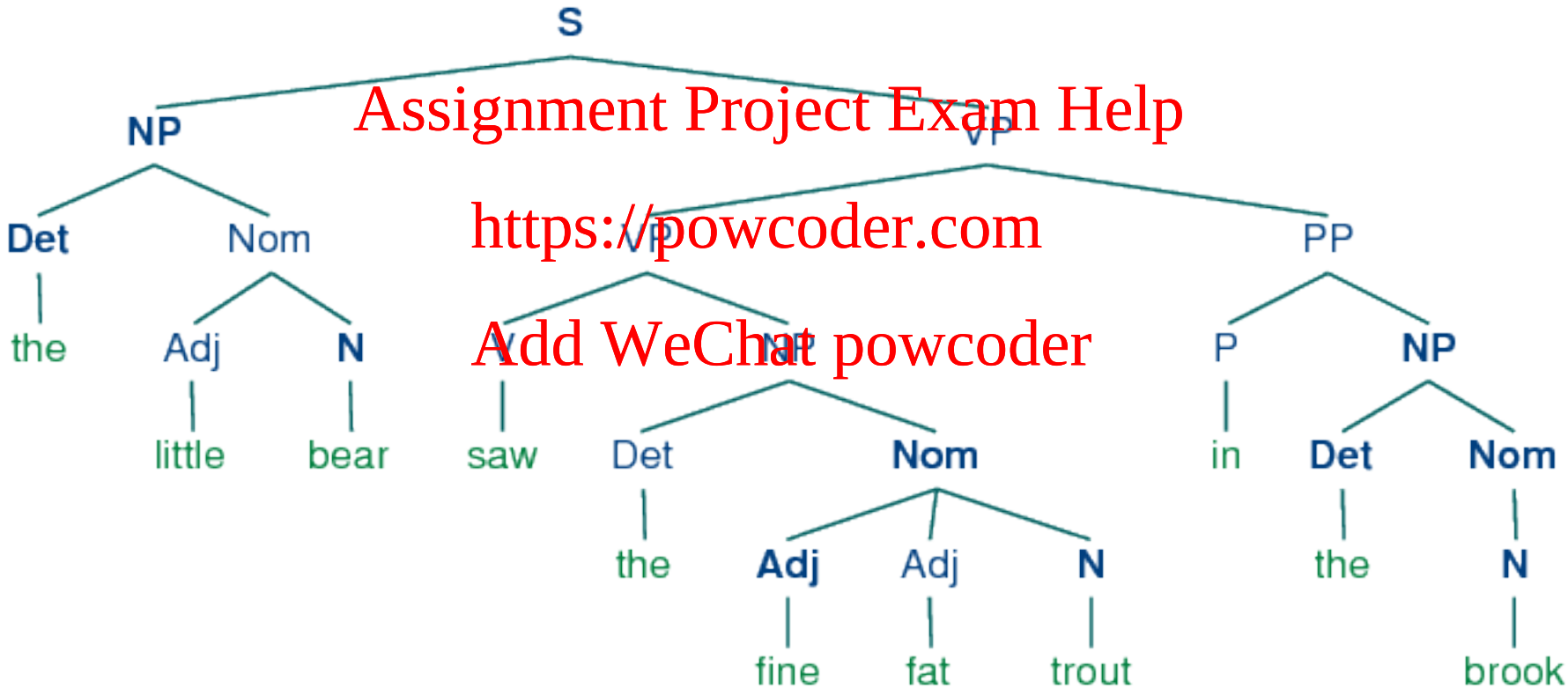
Syntactic categories

Symbol	Meaning	Example
S	sentence	the man walked
NP	noun phrase	a dog
VP	verb phrase	saw a park
PP	prepositional phrase	with a telescope
Det	determiner	the
N	noun	dog
V	verb	walked
P	preposition	in

Constituent structure

Det the	Adj little	N bear	V saw	Det the	Adj fine	Adj fat	N trout	P in	Det the	N brook
Det the	Nom bear		V saw	Det the	Nom trout			P in	NP it	
NP He			V saw	NP it				PP there		
NP He			VP ran					PP there		
NP He			VP ran							

Constituent structure tree



Penn Treebank

- Phrase structure annotation in the generative tradition
- The most influential treebank in NLP.
  - Google scholar citation: 6019 (Marcus et al 1993)
- PTB I (Marcus et al 1993)
  - Context-free backbone
  - Skeletal structures
  - Limited empty elements
  - No argument/adjunct distinction

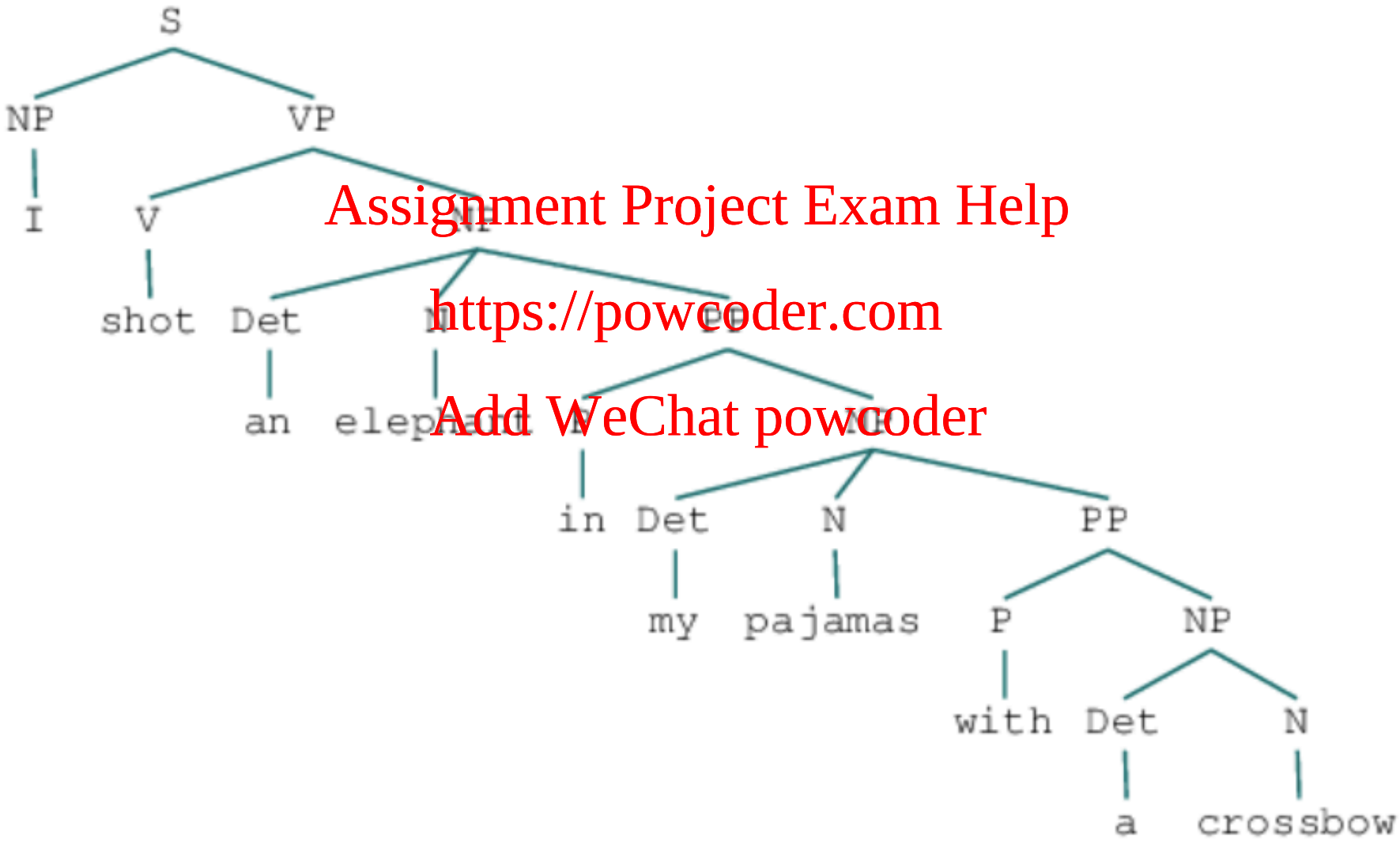
Penn Treebank

- PTB II (Marcus et al 1994)
  - Added function tags to mark up grammatical roles (thus argument/adjunct distinction, though not structurally)
  - Enriched the set of empty elements
  - One million words of 1989 Wall Street Journal material annotated in Treebank-2 style.
  - Tools for processing Treebank data, including "tgrep," a tree-searching and manipulation package
    - usability of tgrep is limited, you may find the software to be difficult or impossible to port

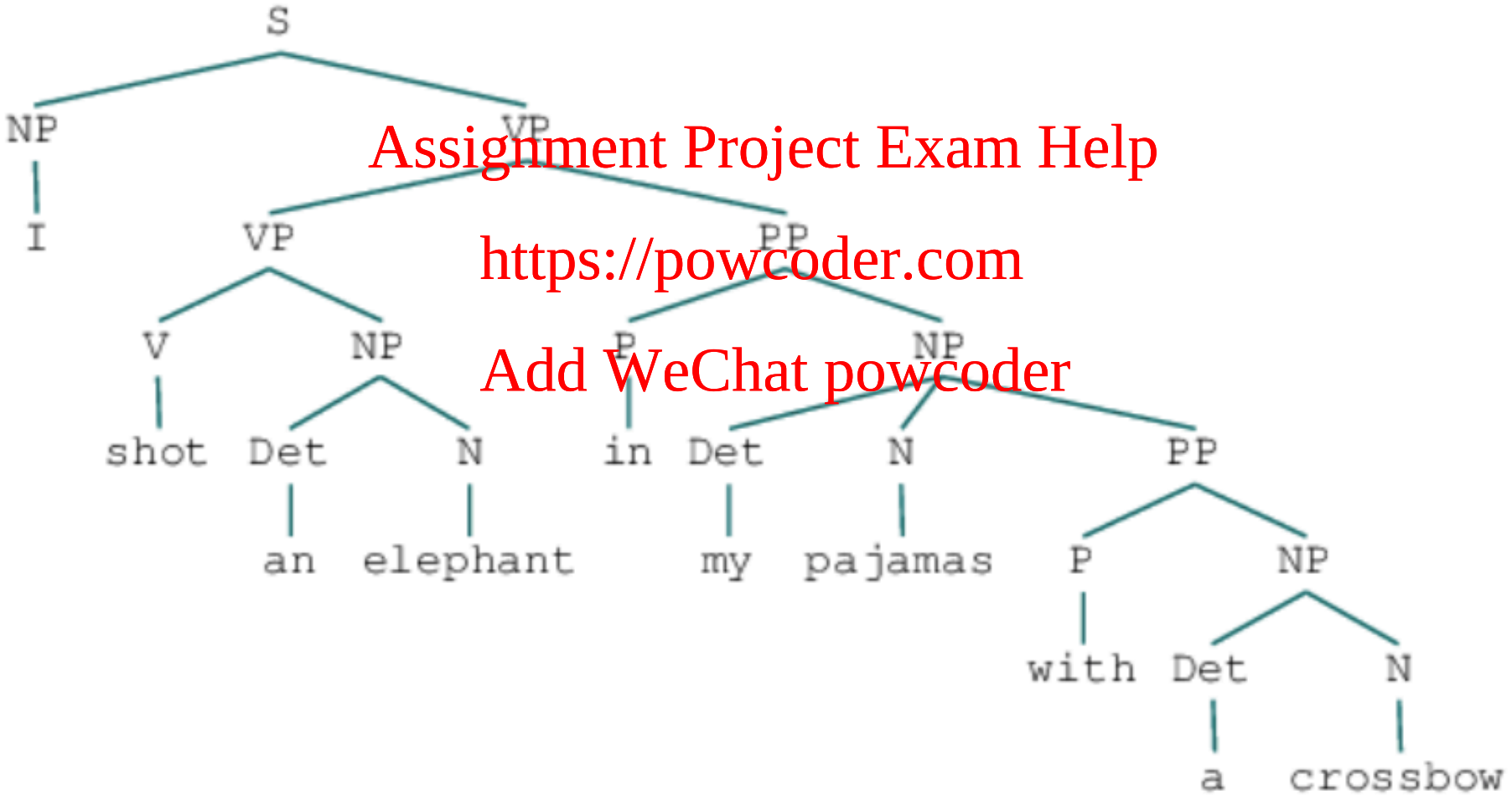
Penn Treebank

- PTB III
  - Added the Brown corpus
- Penn Treebank in NLTK
  - A fragment of Penn Treebank II
  - You can expand this yourself
  - Added functionality in the nltk.Tree class

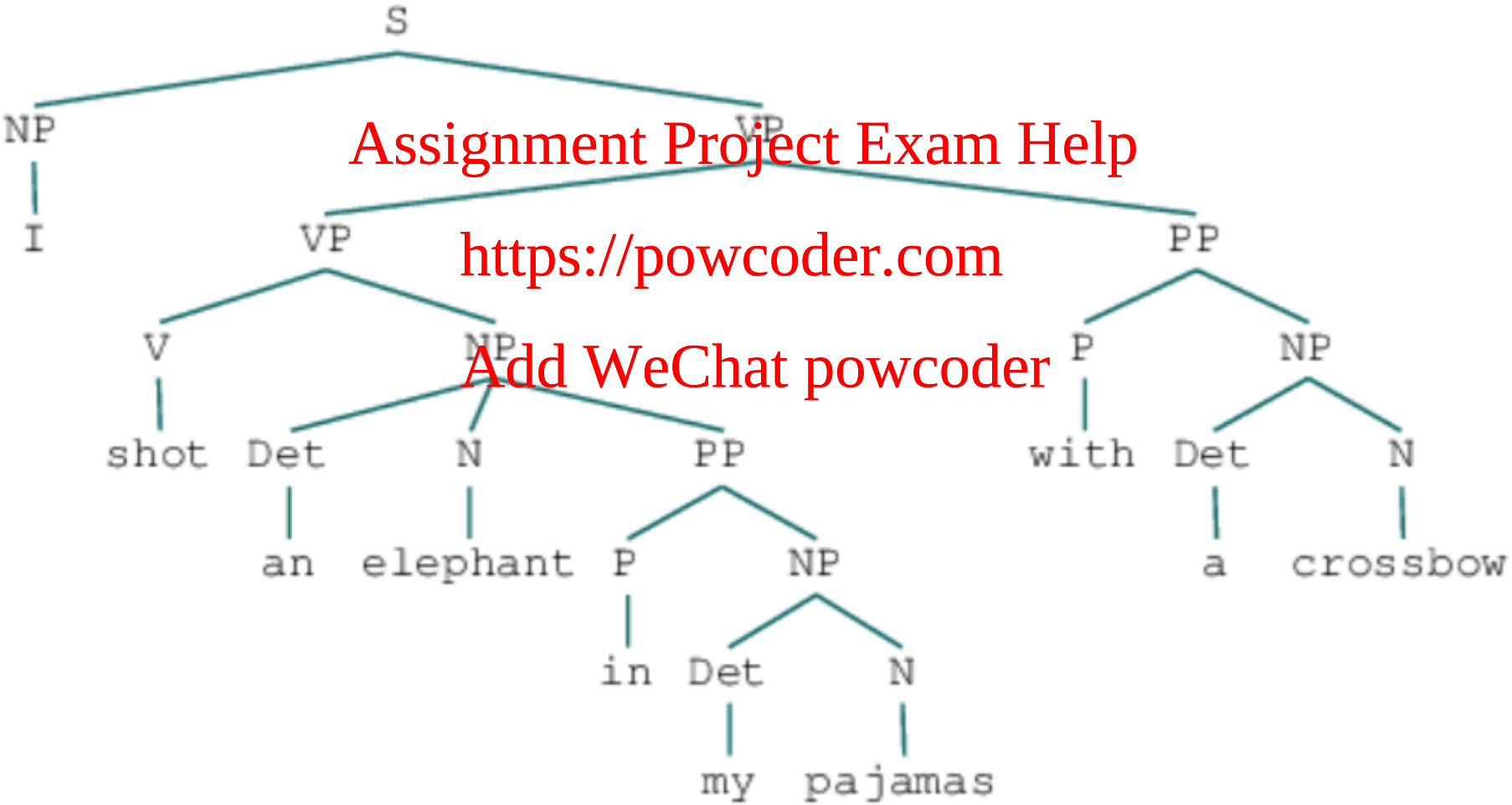
Ambiguity



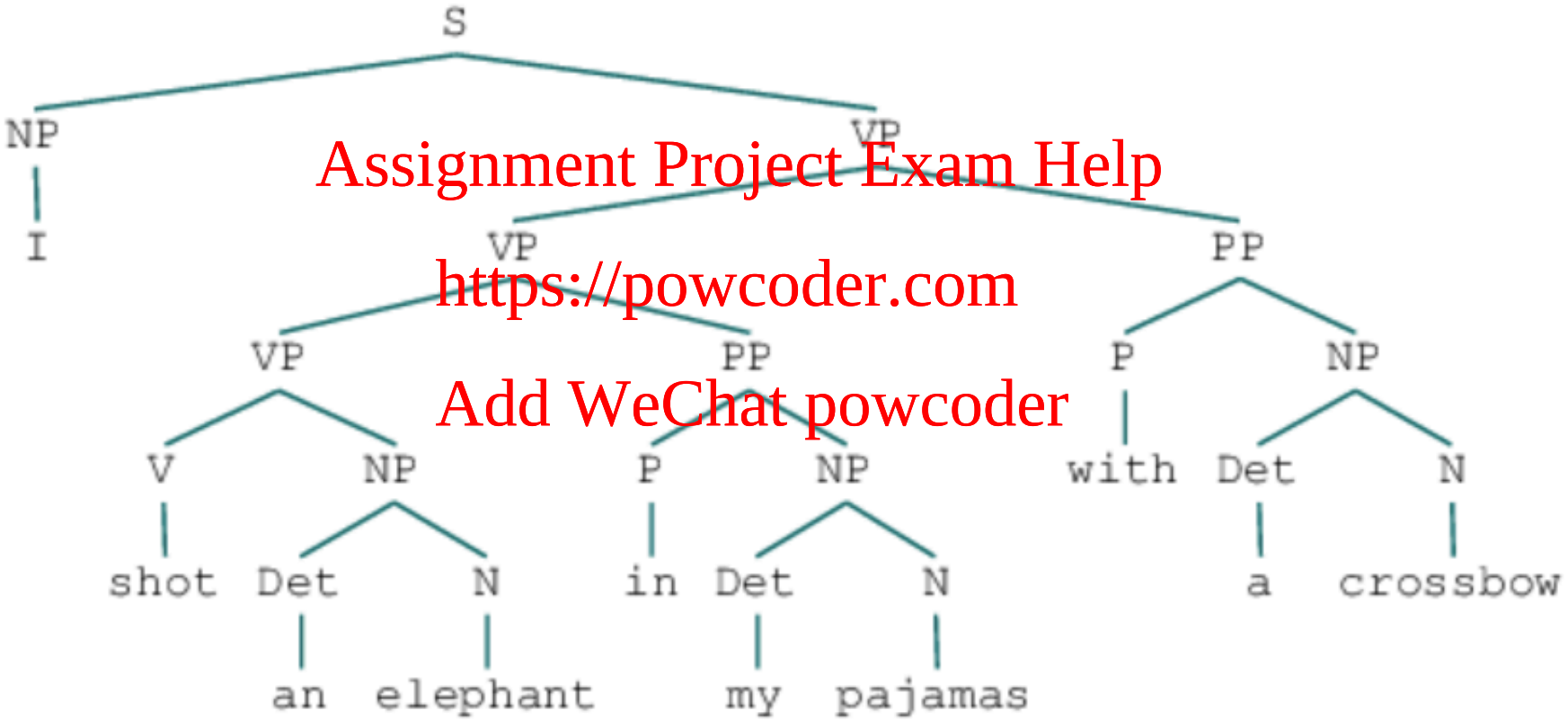
Ambiguity



Ambiguity



Ambiguity



Tree as a mathematical object

- A tree is a set of connected labeled nodes, each reachable by a unique path from a distinguished root node.
- Parent, child, siblings

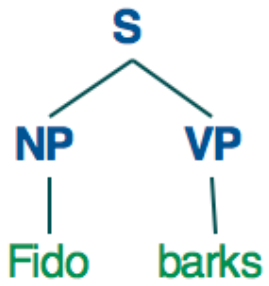
Getting the trees

- Tree class with initialization methods
- Tree reader
  - turn a parse from a string representation into a tree representation
  - Shift reduce parsing
    - A form of automatic parsing
    - turn a list of tokens into a tree representation

Constructing a tree with NLTK

- nltk.Tree()
  - Can be used to build small trees


```
>>> t = nltk.Tree('S',
...               [nltk.Tree('NP', ['Fido']),
...                nltk.Tree('VP', ['barks'])])
>>> t
(S (NP Fido) (VP barks))
>>> t.draw()
```

Constructing a tree with NLTK

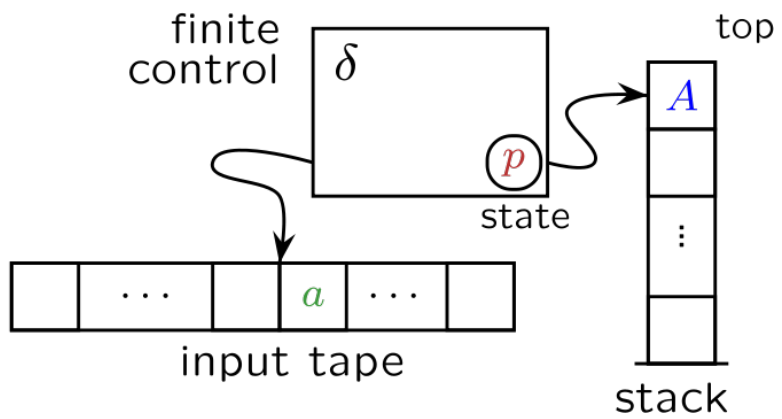
- nltk.Tree.fromstring()
  - To parse strings into trees
  - Can be used to read in manually constructed treebanks and turn the trees into an nltk.Tree object
  - A more practical way for getting trees

nltk.Tree.fromstring()

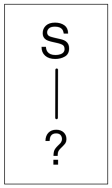
(S (NP (DT the) (N cat)) (VP (V sleeps)))

Q: How does this work under the hood?

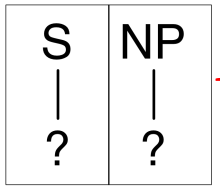
A: By using a stack



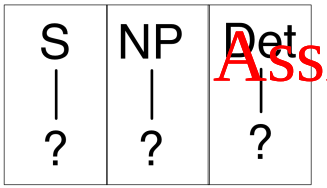
(S (NP (DT the) (N cat)) (VP (V sleeps)))



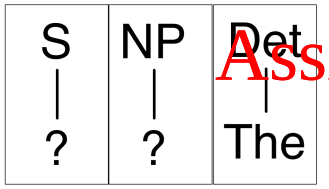
(S (NP (DT the) (N cat)) (VP (V sleeps)))



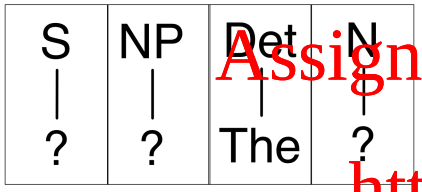
(S (NP (DT the) (N cat)) (VP (V sleeps)))



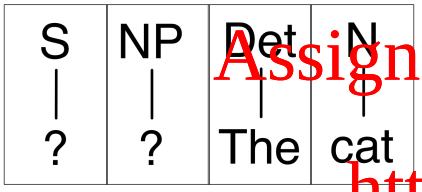
(S (NP (DT the) (N cat)) (VP (V sleeps)))



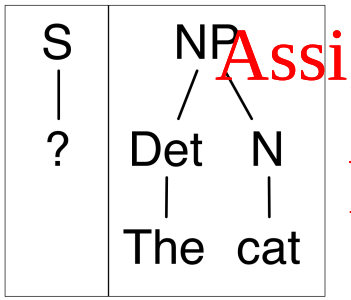
(S (NP (DT the) (N cat)) (VP (V sleeps)))



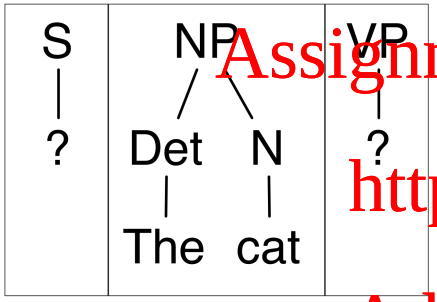
(S (NP (DT the) (N cat)) (VP (V sleeps)))



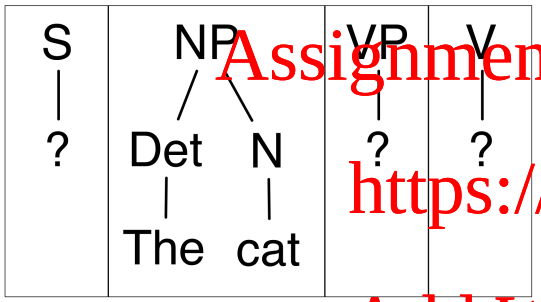
(S (NP (DT the) (N cat)) (VP (V sleeps)))



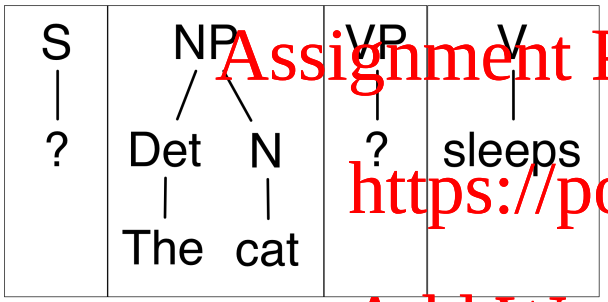
(S (NP (DT the) (N cat)) (VP (V sleeps)))



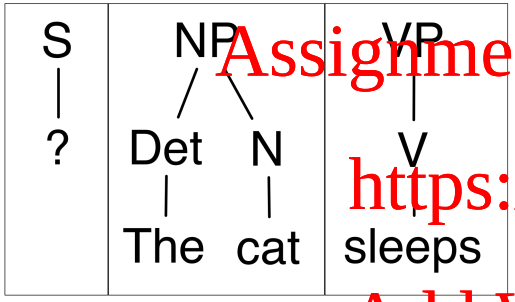
(S (NP (DT the) (N cat)) (VP (V sleeps)))



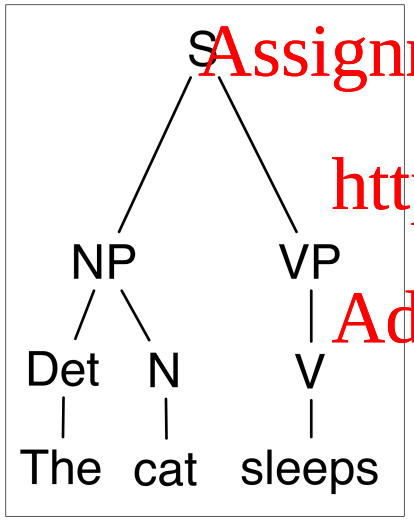
(S (NP (DT the) (N cat)) (VP (V sleeps)))



(S (NP (DT the) (N cat)) (VP (V sleeps)))



(S (NP (DT the) (N cat)) (VP (V sleeps)))



Grammars

- With regular expressions we had a regular grammar
- Now we are looking at the next level: the context free grammar
  - One category on the left-hand-side of the rule
  - One or more categories on the right-hand-side
  - Not allowed
    - S --> ε
    - NP VP --> Det N VP

Epsilon stands for the empty string

Context-free grammar

S -> NP VP
VP -> V NP | V NP PP
PP -> P NP
V -> "saw" | "ate" | "walked"
NP -> "John" | "Mary" | "Bob" | Det N | Det N PP
Det -> "a" | "an" | "the" | "my"
N -> "man" | "dog" | "cat" | "telescope" | "park"
P -> "in" | "on" | "by" | "with"

Recursion in syntactic structure

S -> NP VP
NP -> Det Nom | PropN
Nom -> Adj Nom | N
VP -> V Adj | V NP | V S | V NP PP
PP -> P NP
PropN -> 'Buster' | 'Chatterer' | 'Joe'
Det -> 'the' | 'a'
N -> 'bear' | 'squirrel' | 'tree' | 'fish' | 'log' Adj -> 'angry' | 'frightened' | 'little' | 'tall'
V -> 'chased' | 'saw' | 'said' | 'thought' | 'was' | 'put' P -> 'on'

Parsing with CFG

- A parser processes input sentences according to the productions of a grammar
- It produces one or more constituent structures that conform to the grammar
- Grammar
  - a declarative specification of well-formedness
- Parser
  - a procedural interpretation of the grammar that searches the space of trees licensed by a grammar

Parsing with CFG

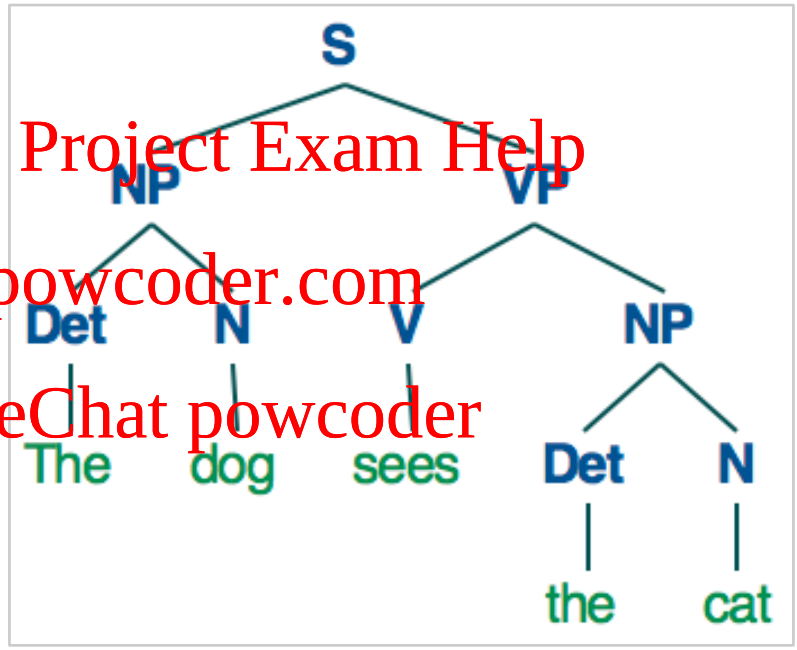
- Top-down parsing
  - Predicts/generates sentences by starting at the top with the goal of creating a category of type S
  - Driven by the grammar
- Bottom-up parsing
  - Build a parse tree from the ground up
  - Driven by the input sentence

Recursive descent parsing

- Top-down parsing
- Recursively breaks a high-level goal into several lower-level subgoals
- Expand, and match
- Maintains a list of nodes in a hypothetic tree that needs to be expanded (called frontier)
- Use back-tracking if you get into a dead end

The dog sees the cat

```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```



```
S → NP VP          Det → "the"
NP → Det N         N → "cat"
VP → V             N → "dog"
VP → V NP          V → "sees"
```

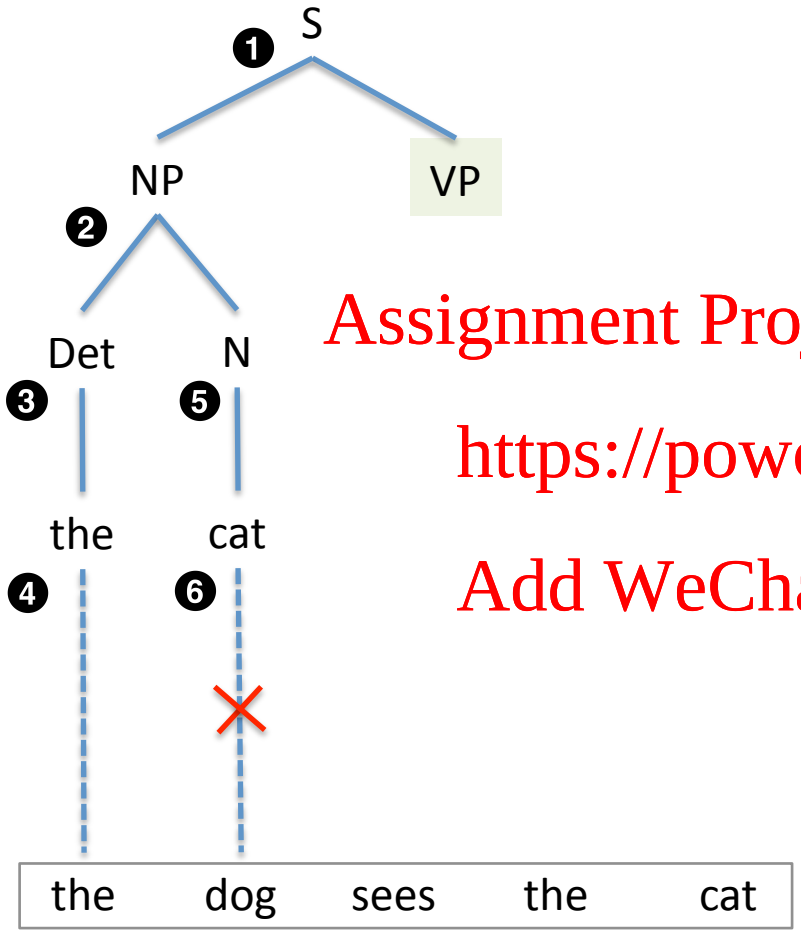
❶ S

❶ We start off with the start symbol of the grammar. The frontier of the tree is the set of symbols that you can still expand, nodes in the frontier have a light green background

❷ And we have our input

❷

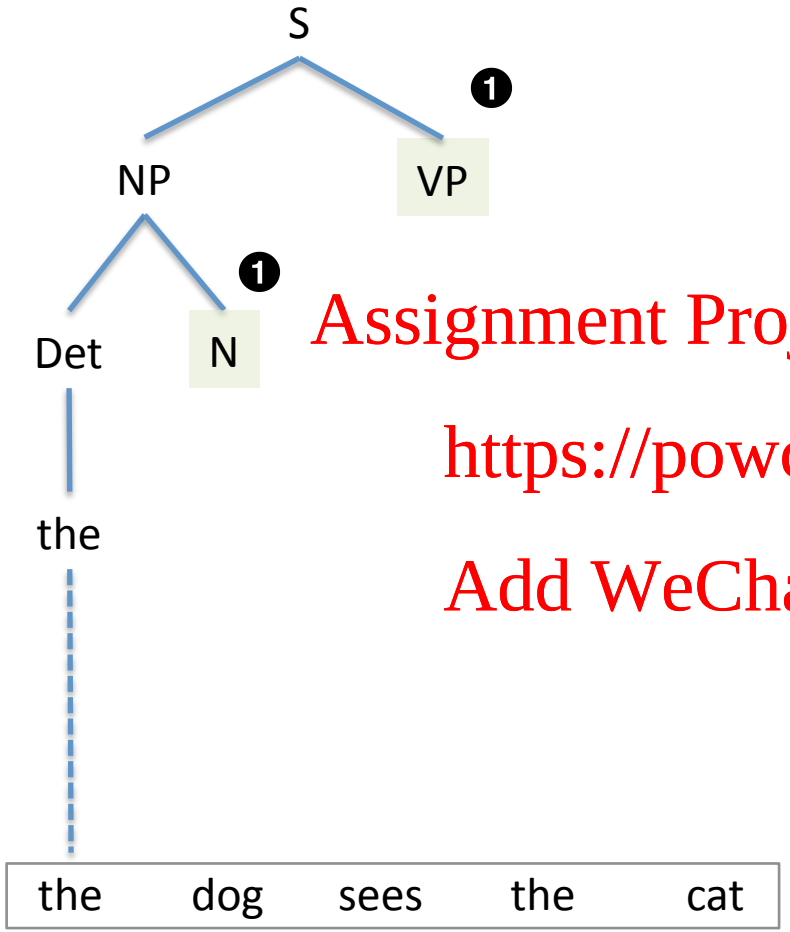
the	dog	sees	the	cat




```
S → NP VP          Det → "the"
NP → Det N         N → "cat"
VP → V             N → "dog"
VP → V NP          V → "sees"
```

1. Apply S → NP VP
2. Apply NP → Det N
3. Apply Det → "the"
4. Confirm "the" is in input
5. Apply N → "cat"
6. Wrong: "cat" is not in input

Now we backtrack. That is, we undo the last step and see whether there was an alternative, if not, we keep undoing steps until we do find an alternative. If we backtrack all the way to S without finding alternatives then the parser fails.



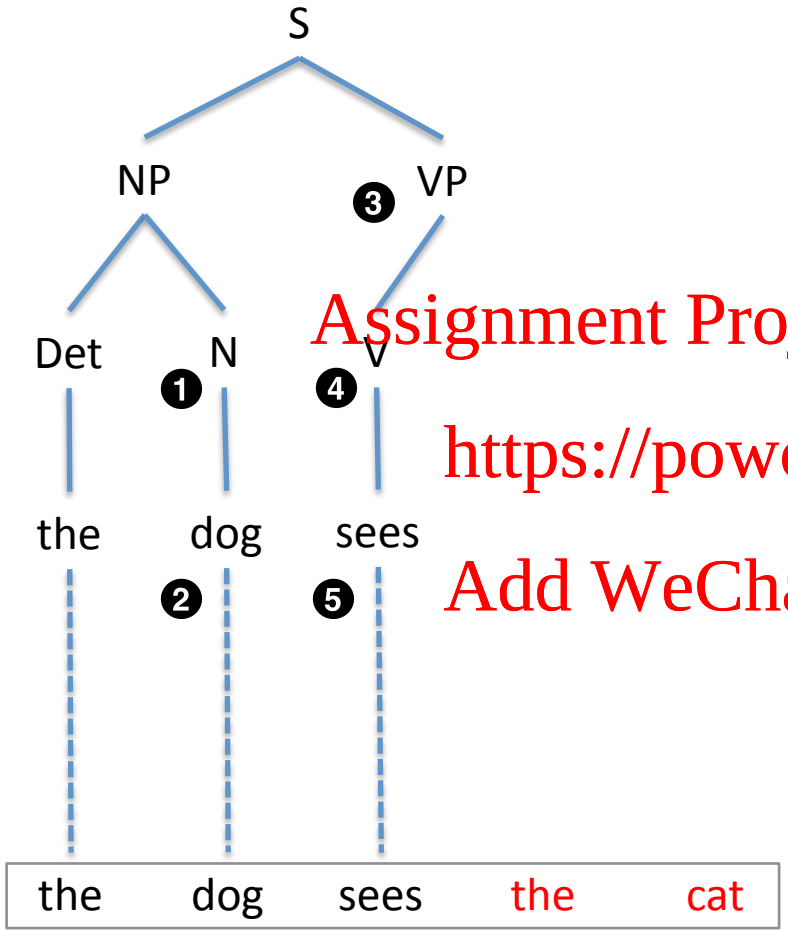

```
S → NP VP          Det → "the"
NP → Det N         N → "cat"
VP → V             N → "dog"  ❷
VP → V NP          V → "sees"
```

❶ We undid step 5 from the last slide and the frontier is now N and VP

❷ And we have an alternative which is to use N → "dog" instead of N → "cat"

Note that this backtracking would not have been needed if the parser had just been a wee little bit smarter, for example by grouping the rules for pre-terminals (N, V and Det) and peeking ahead at the input.

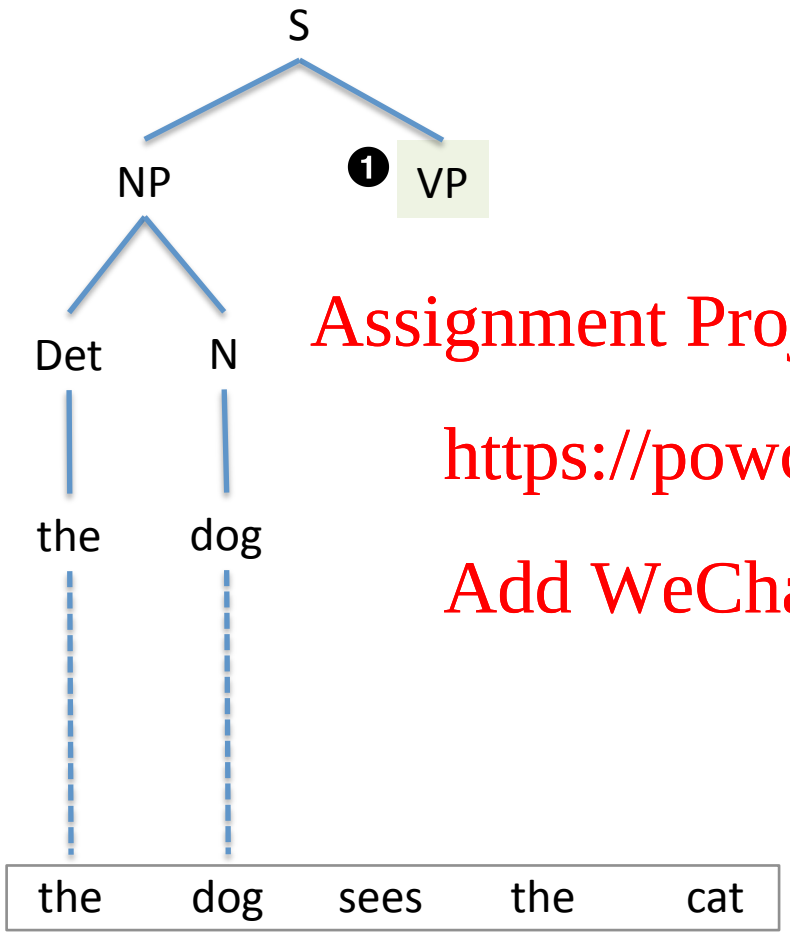
```
S → NP VP          Det → "the"
NP → Det N         N → "cat"
VP → V             N → "dog"
VP → V NP          V → "sees"
```



1. Apply N → "dog"
2. Confirm that "dog" is in the input
3. Apply VP → V
4. Apply V → "sees"
5. Confirm that "sees" is in the input

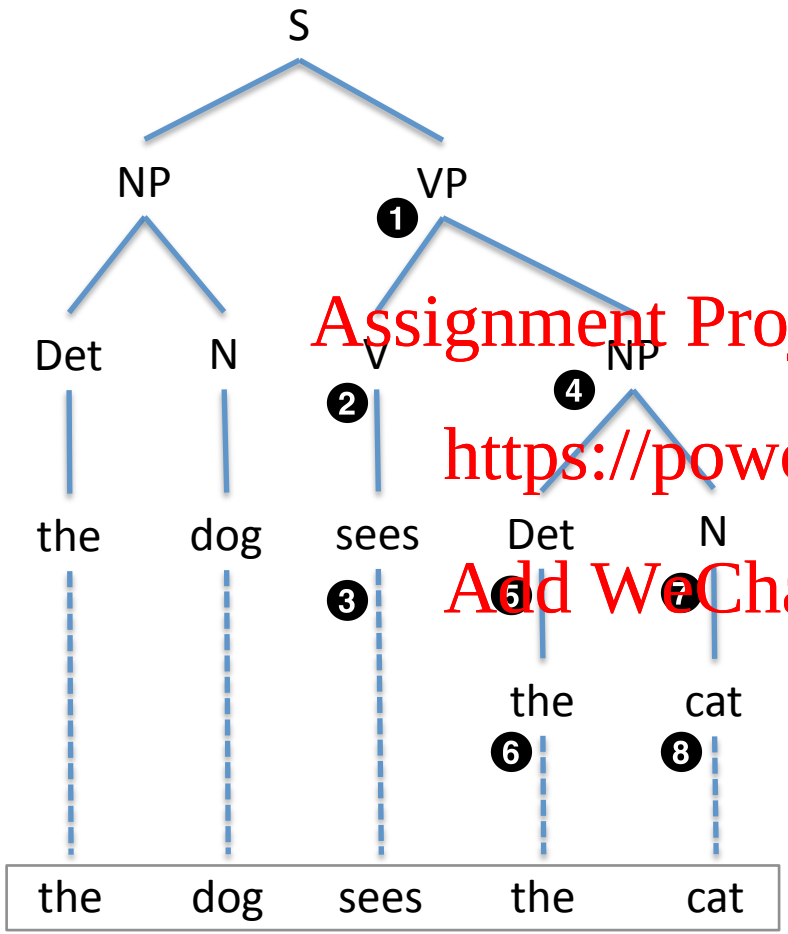
But now we are stuck. There are no nodes on the frontier anymore, but we have not covered all input. So we backtrack again.

```
S → NP VP          Det → "the"
NP → Det N         N → "cat"
VP → V             N → "dog"
VP → V NP ❷        V → "sees"
```



❶ We undid step 3 from the
❷ previous slide and the frontier is
❸ now the VP.
❹ And we have an alternative which is to use VP → V NP instead of VP → V.

```
S → NP VP          Det → "the"
NP → Det N         N → "cat"
VP → V             N → "dog"
VP → V NP          V → "sees"
```



1. Apply VP → V NP
2. Apply V –> "Sees"
3. Check for "sees" in the input
4. Apply NP → Det N
5. Apply Det → "the"
6. Check for "the" in the input
7. Apply N → "cat"
8. Check for "cat" in the input

And the parse was successful.

Recursive Descent Parser Demo

```
import nltk
nltk.app.rdparser_app.app()
```

Weakness of recursive descent parsing

- Left-recursive productions like NP → NP PP leads to an infinite loop
- The parser wastes a lot of time considering structures or words that do not correspond to the input structure
- Backtracking may discard reusable structures

So nobody uses this

Bottom-up parsing

- Family of parsing method where the process is driven by the input text
- Generally more efficient

Bottom-up parsing

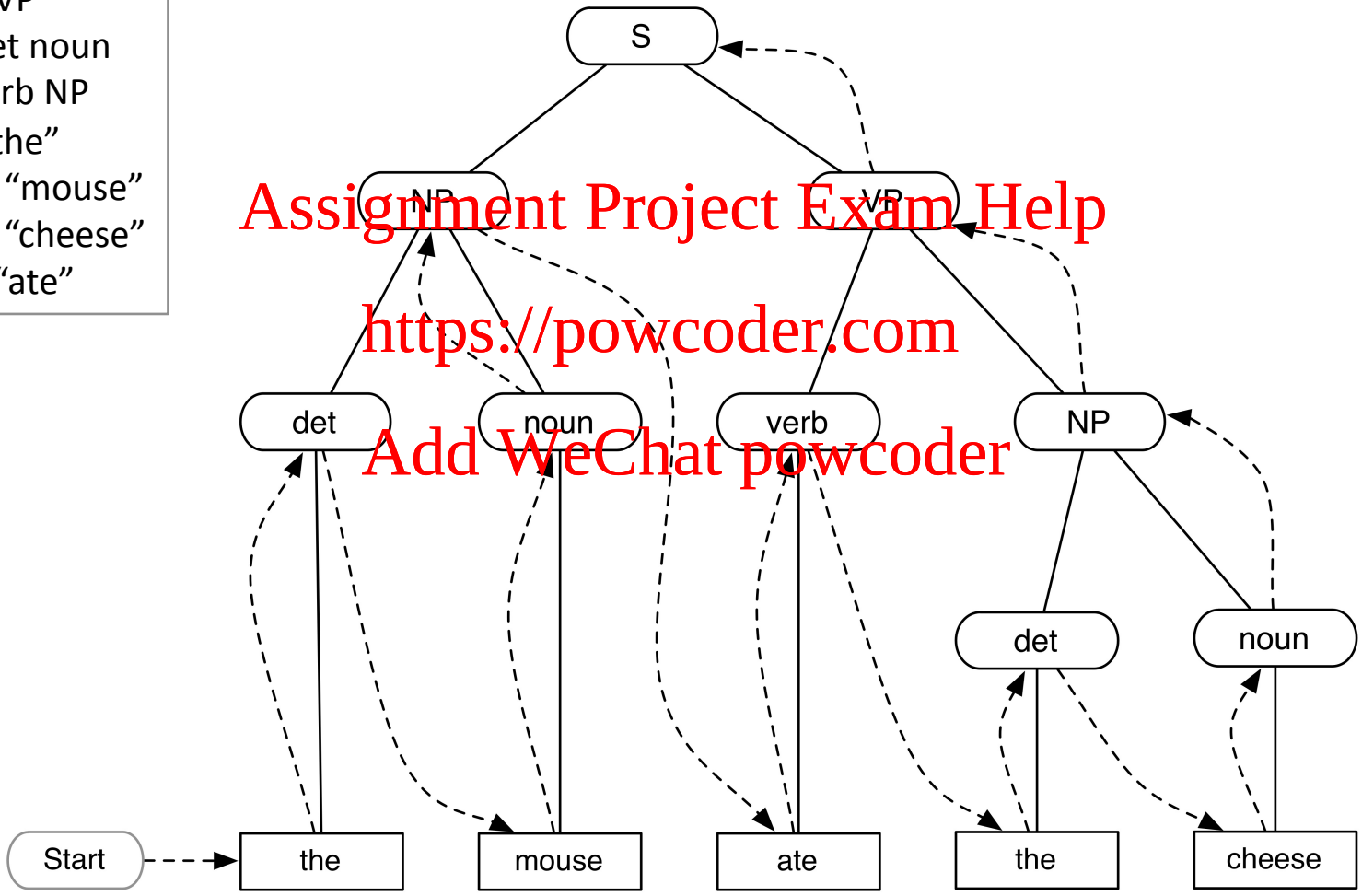
```
S → NP VP
NP → det noun
VP → verb NP
det → "the"
noun → "mouse"
noun → "cheese"
verb → "ate"
```

Input:

The mouse ate the cheese

Bottom-up parsing

S → NP VP
NP → det noun
VP → verb NP
det → “the”
noun → “mouse”
noun → “cheese”
verb → “ate”



Shift Reduce Parser

- A simple bottom-up parsing strategy
- Start with the input sentence (a list of words) and an empty stack
- Repeatedly push the next word onto the stack
- But… if the top n items on the stack match the right hand-side of a CFG rule, then they are popped off the stack, and replaced with the single item on the left-hand side of the rule

Shift Reduce Parser

- Shift: pushing to the stack
- Reduce: moving m elements from the stack and replace them with n elements (n ≤ m)
  - Reduce is guided by a grammar and can only be applied to the top items of the stack (a limitation of the stack data structure)

Shift Reduce Parser

- The parser finishes when all the input is consumed
  - success if there is a single S node on the stack
  - failure if there is more than one item on the stack

Shift-reduce example

Grammar:

```
S    →  NP VP
NP   →  Det N
VP   →  V
VP   →  V NP
Det  →  "the"
N    →  "cat"
N    →  "dog"
V    →  "sees"
```

Input:

"The dog sees the cat"

Stack	Remaining text
	the dog sees the cat

We start with an empty stack and the full input sentence. Our only option at the start is to shift, that is, we add an element from the remaining text to the stack. Elements involved in the next action (words, rules and stack elements) are printed in blue.

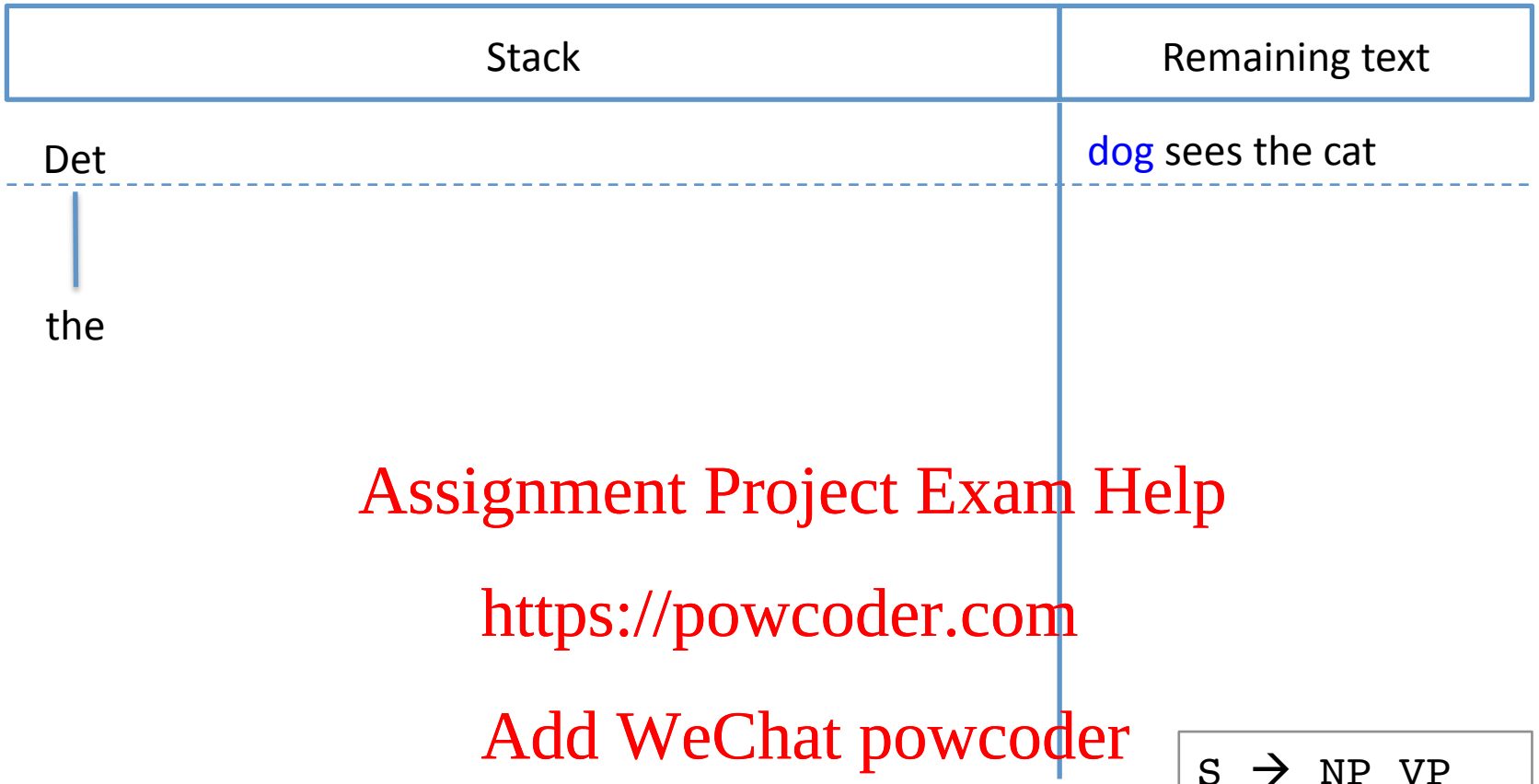
```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```

Stack	Remaining text
the	dog sees the cat

Shifted "the" onto the stack.

The rule was to shift unless you could reduce. We can now reduce because "the" on the stack matches the right-hand side (rhs) of a rule so we will do that.

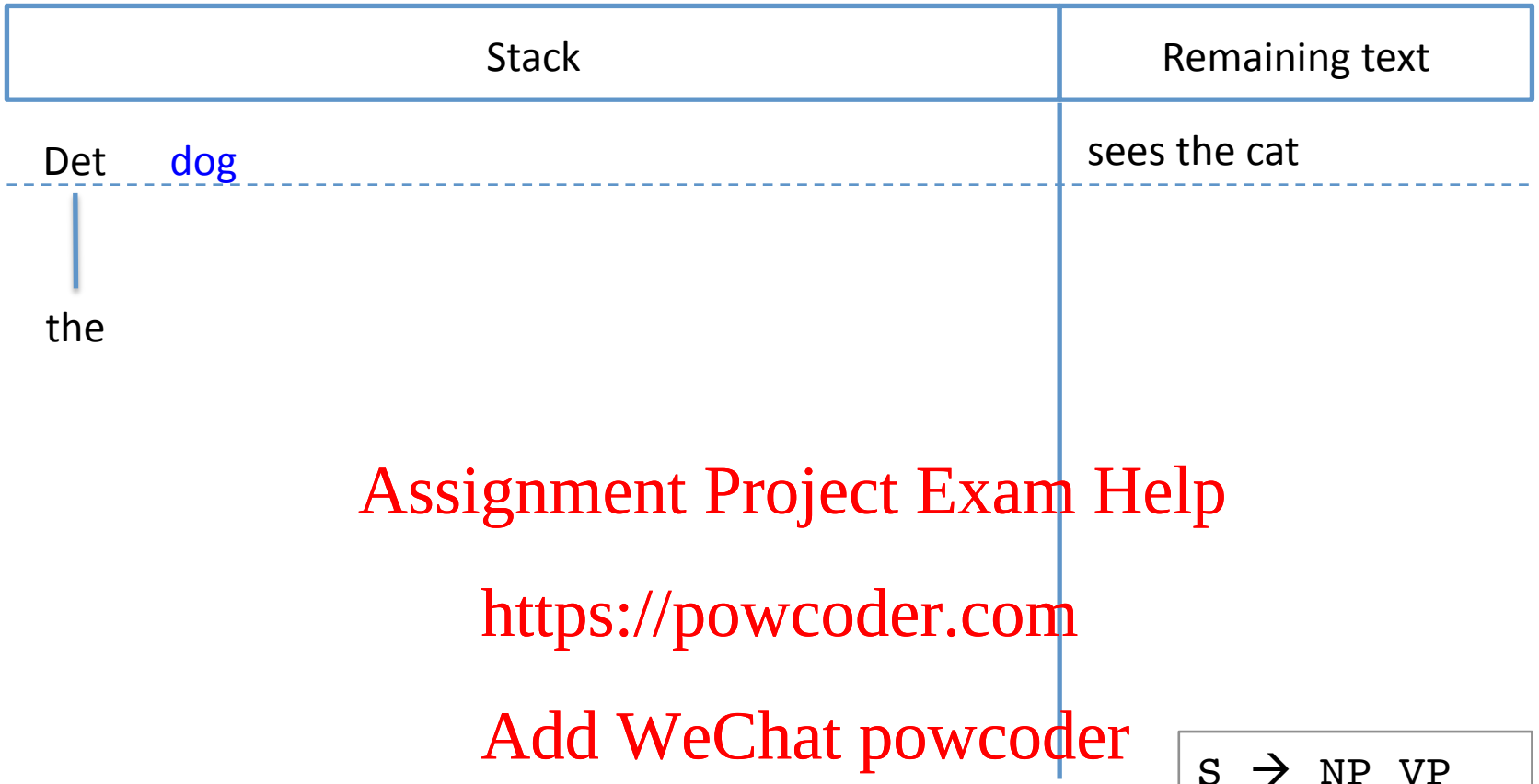
```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```



Reduced by removing "the" from the stack and adding the mini tree using the rule Det → "the".

Next up: shift

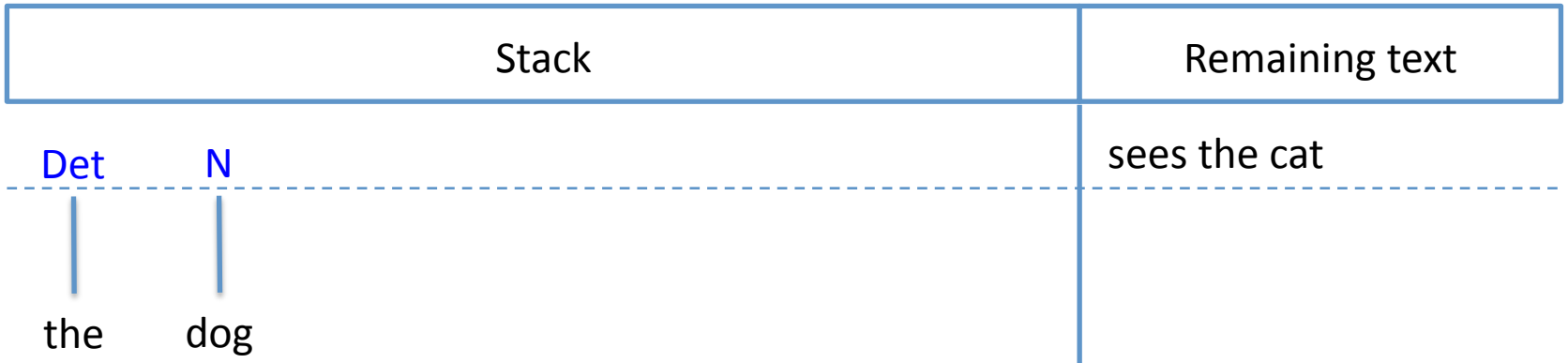
```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```



Added "dog" to the stack.

Next up: reduce because "dog" is on the rhs of a rule.

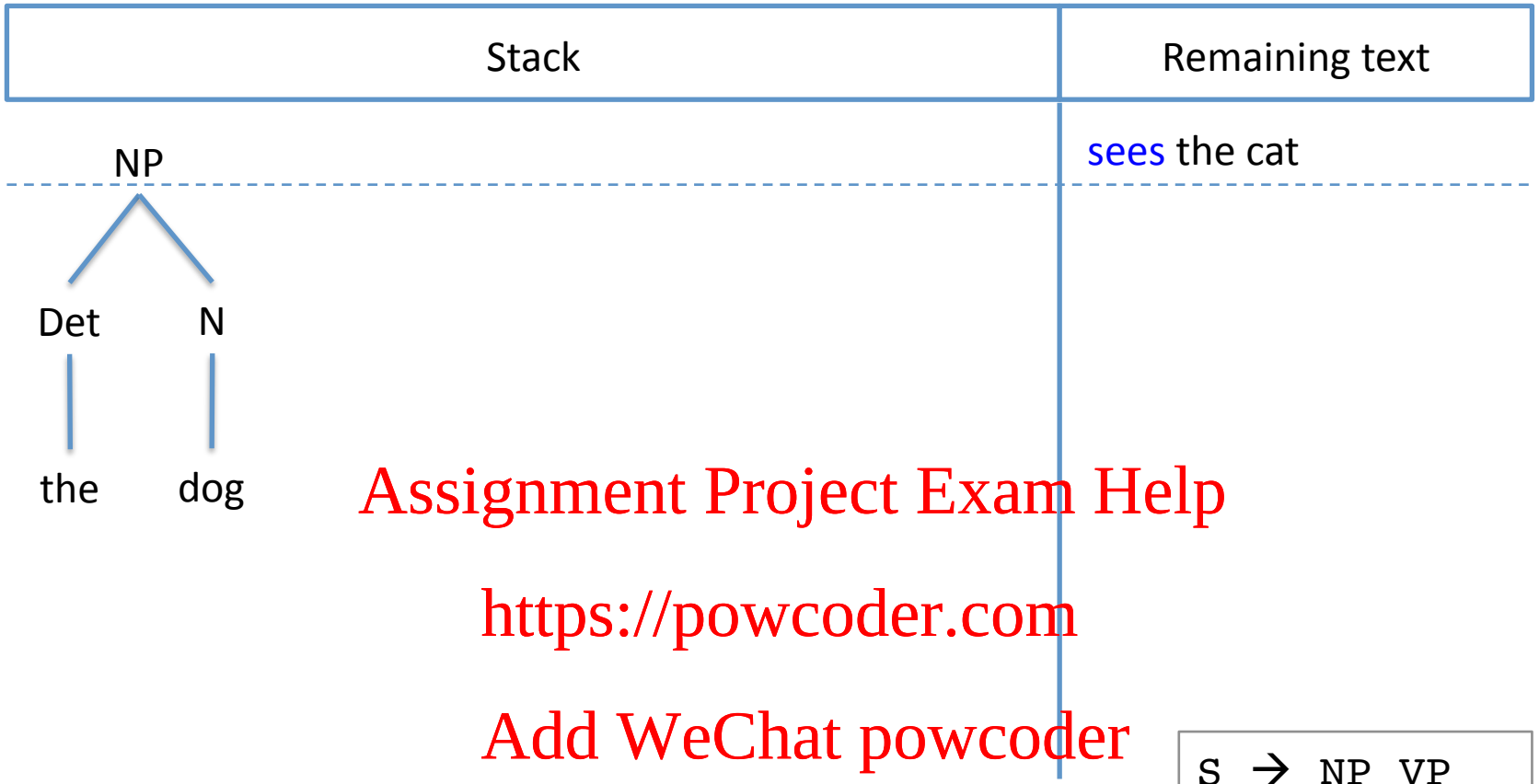
```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```



Removed "dog" from the stack and added another mini tree.

Next up: Det and N are the rhs of a rule, so we can do another reduce

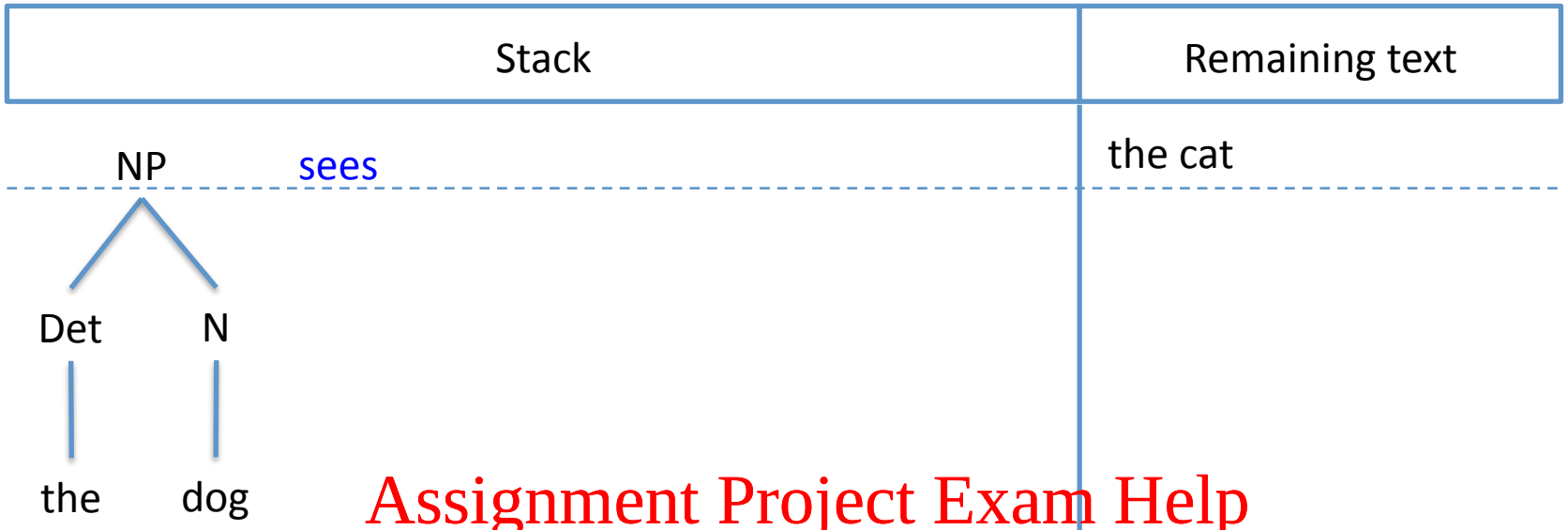
```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```



Two top elements of the stack were removed, and one was added, driven by the rule NP → Det N.

Next up: shift

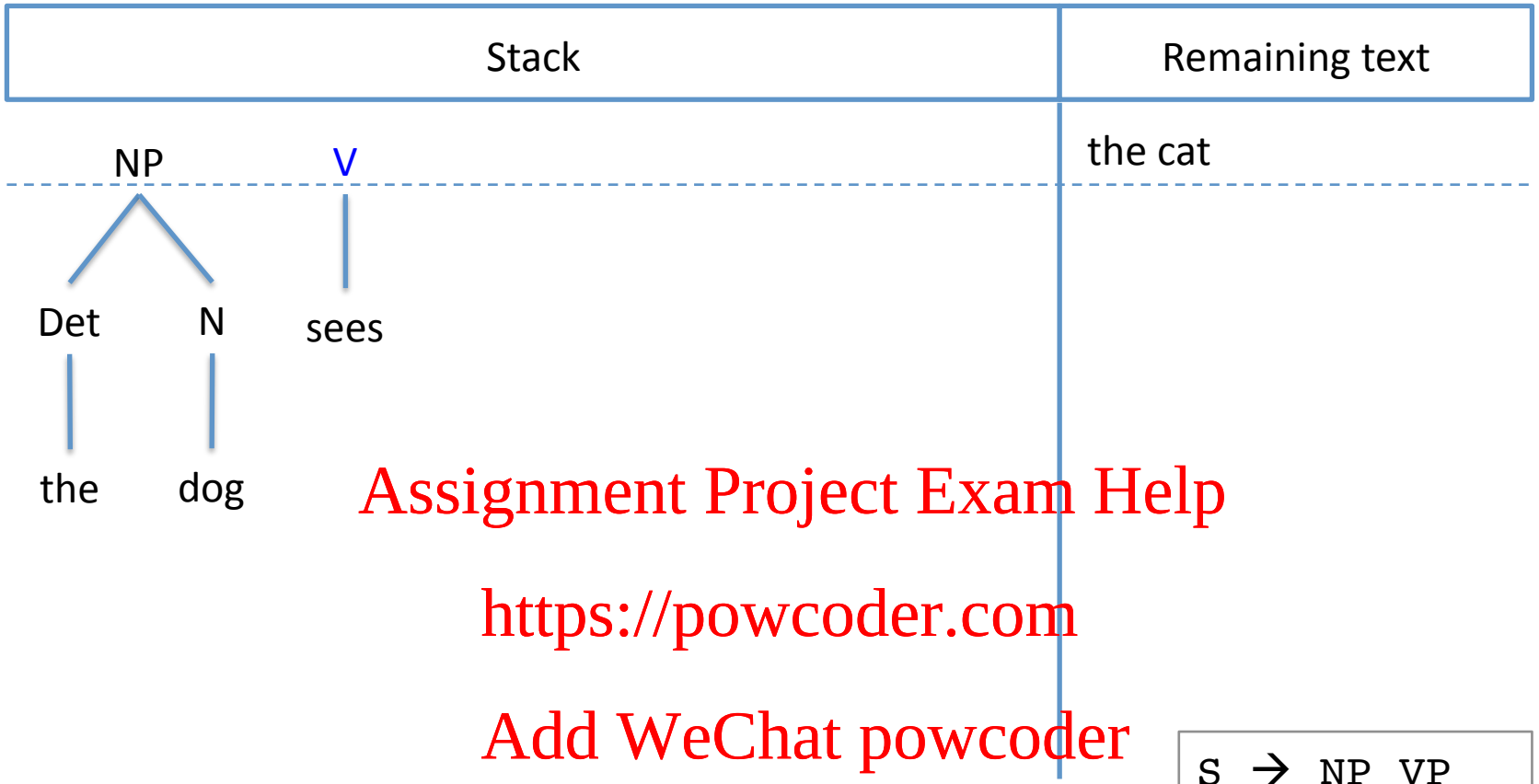
```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```



Shifted "sees" onto the stack.

Next up: we can do another reduce and since we still prefer those over shifts we will go ahead.

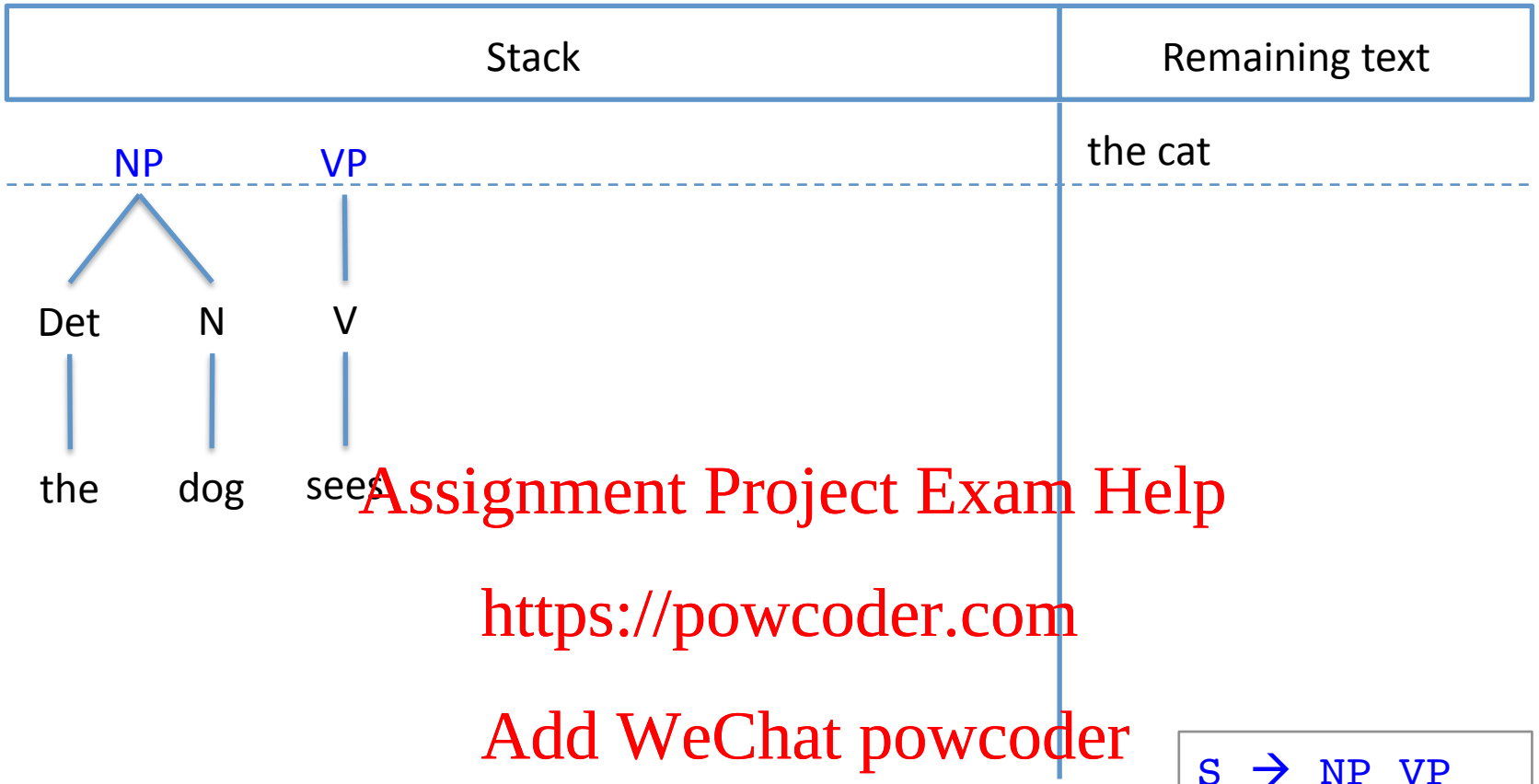
```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```



Replaced "sees" on the stack with a mini tree.

Next up: we can do another reduce.

```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```

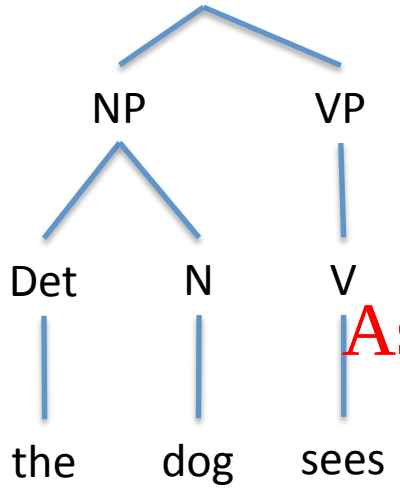


Removed V("sees") from the stack and replaced it with VP(V("sees").

Next up: yet another reduce this time driven by the rule S → NP VP.

```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```

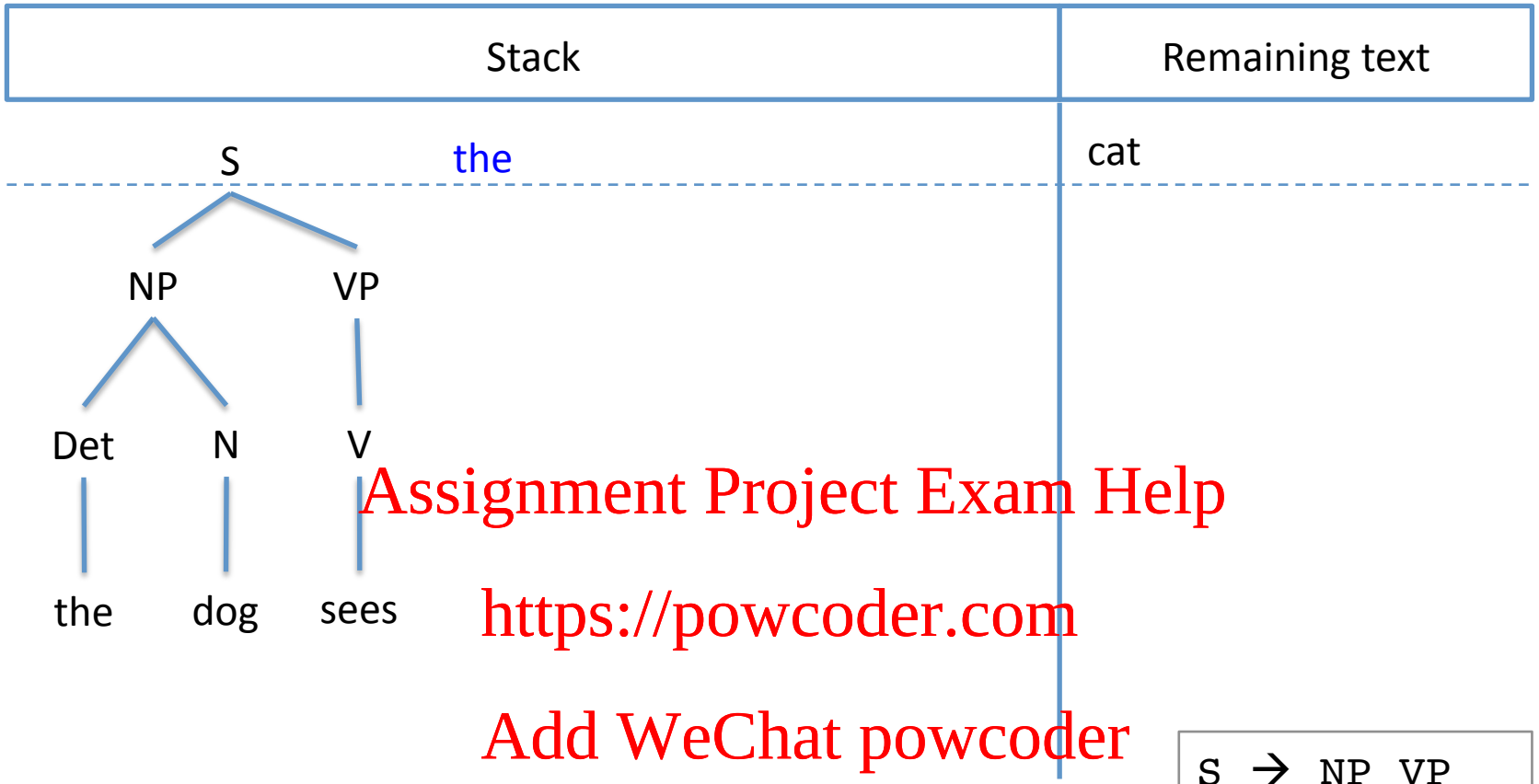
Stack	Remaining text
S	the cat



We took two elements from the stack and replaced it by one.

Next up: a shift

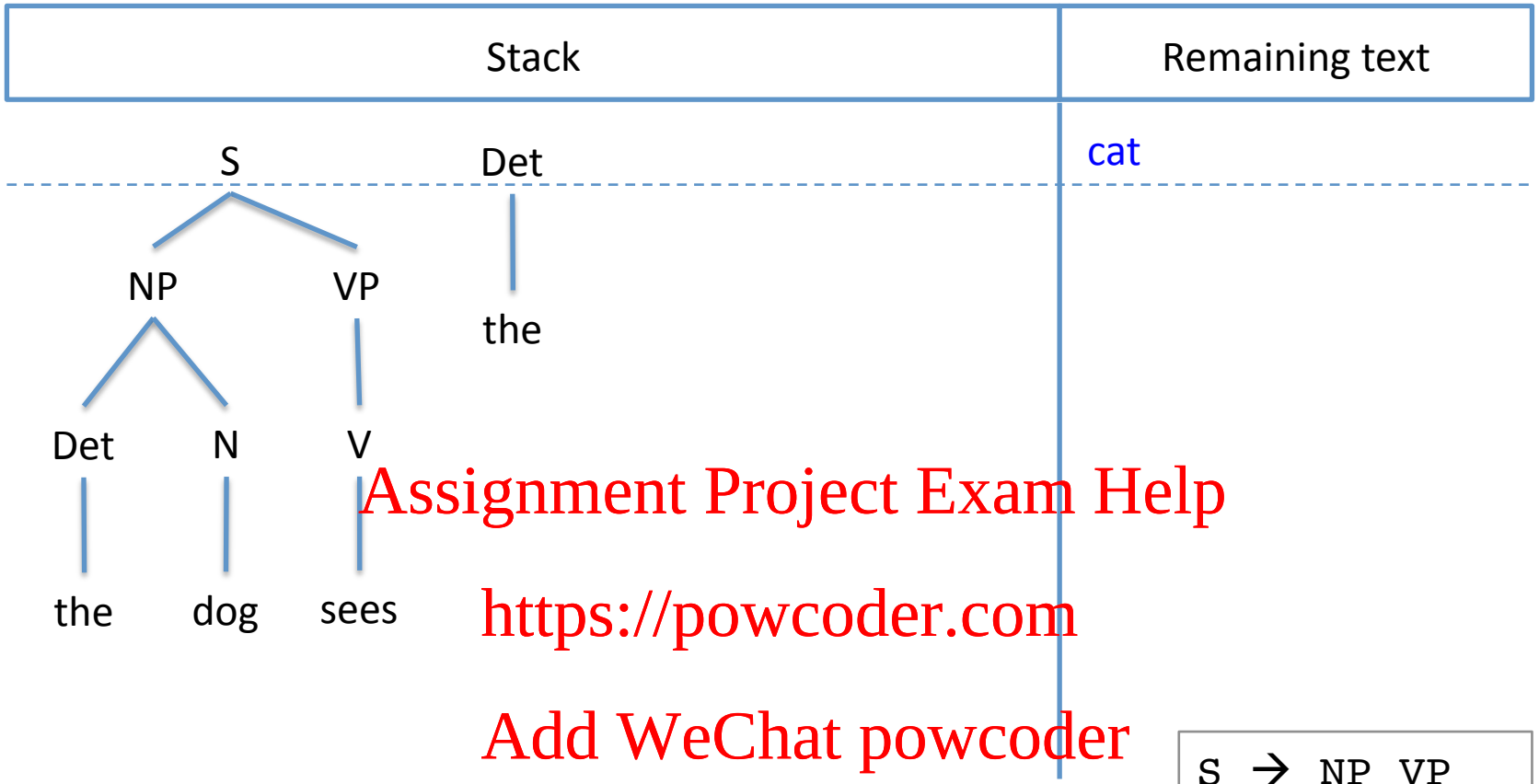
```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```



We shifted "the" onto the stack.

Next up: a reduce

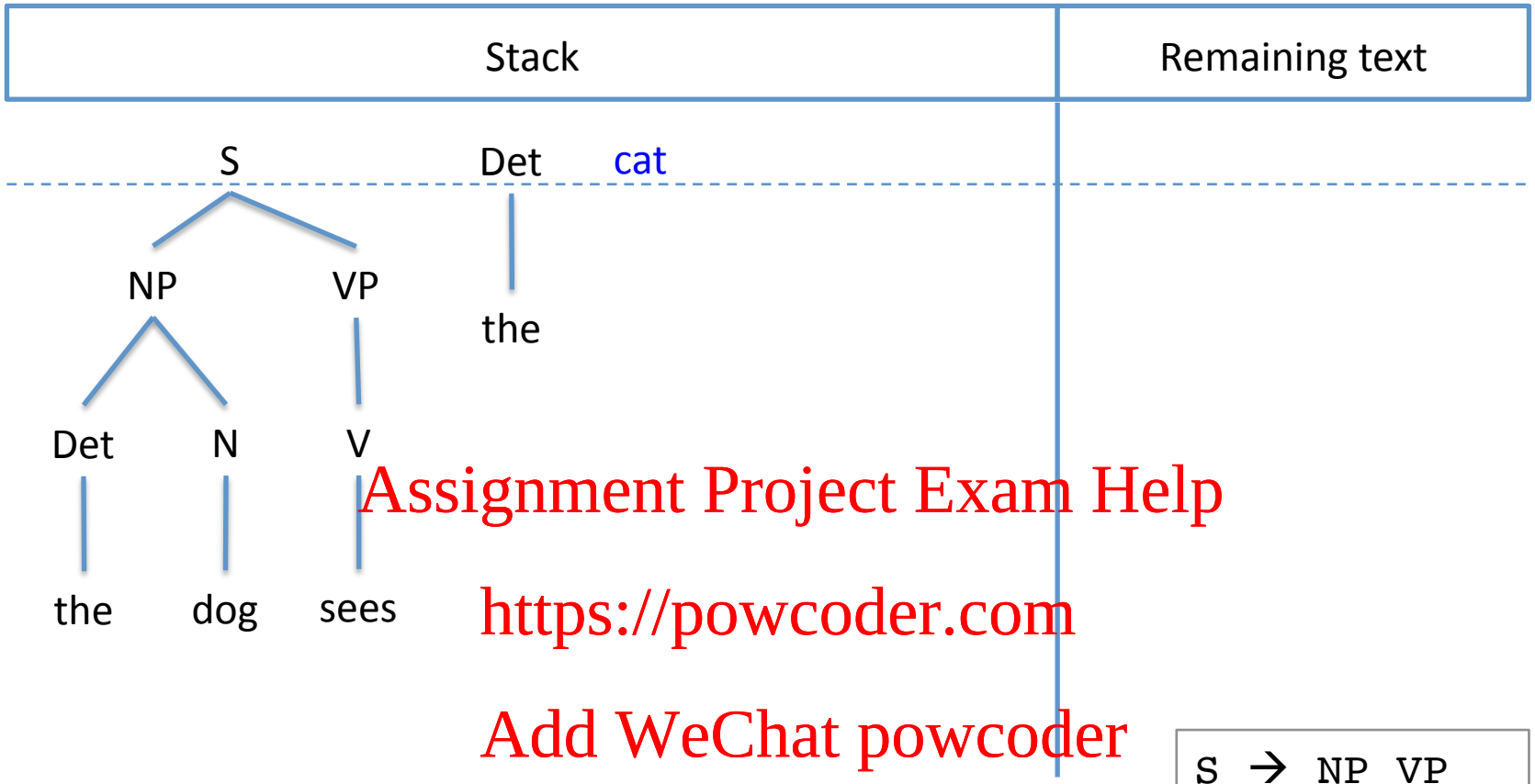
```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```



We took "the" from the stack and replaced it by a mini tree.

Next up: a shift

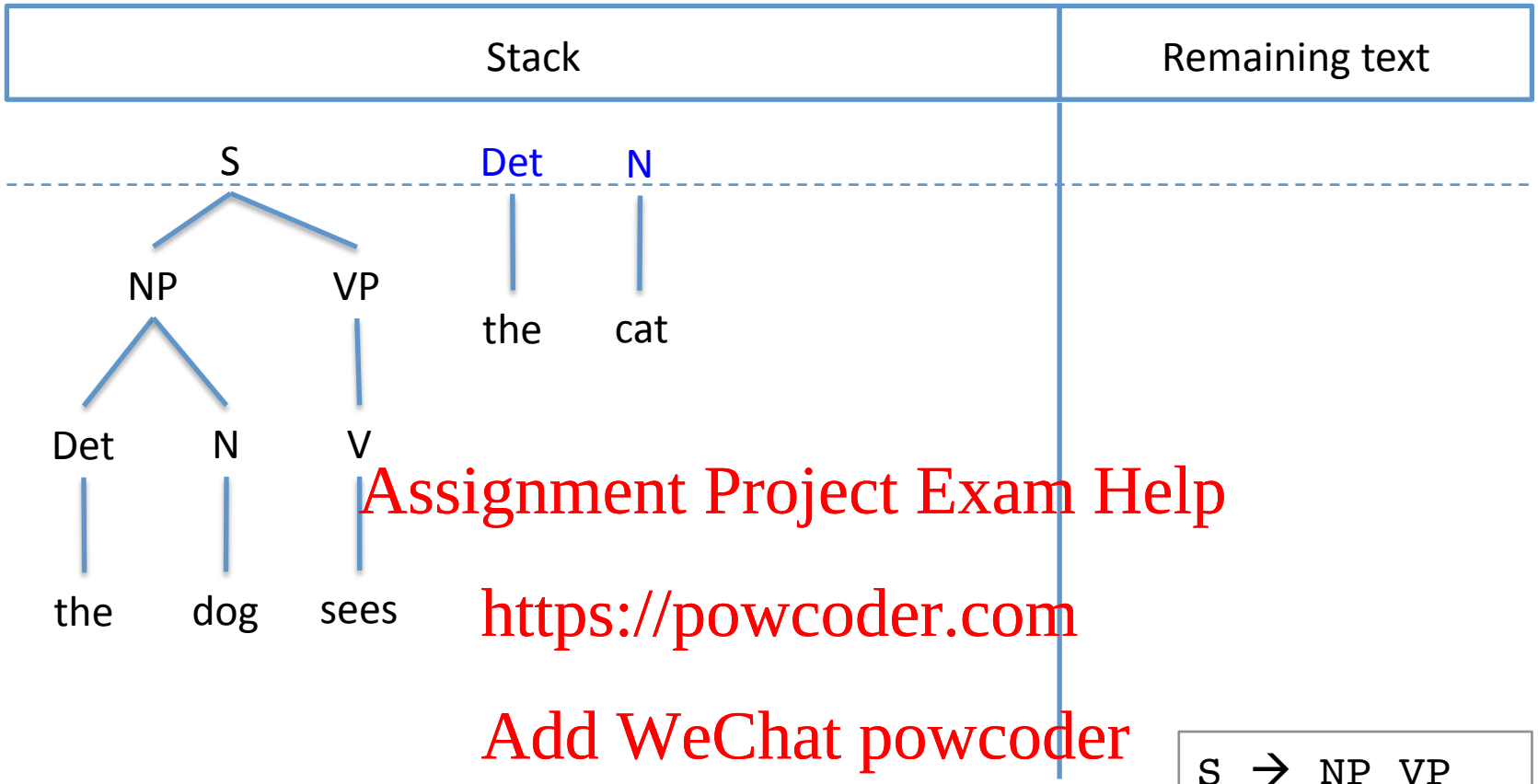
```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```



We added "cat" to the stack.

Next up: a reduce

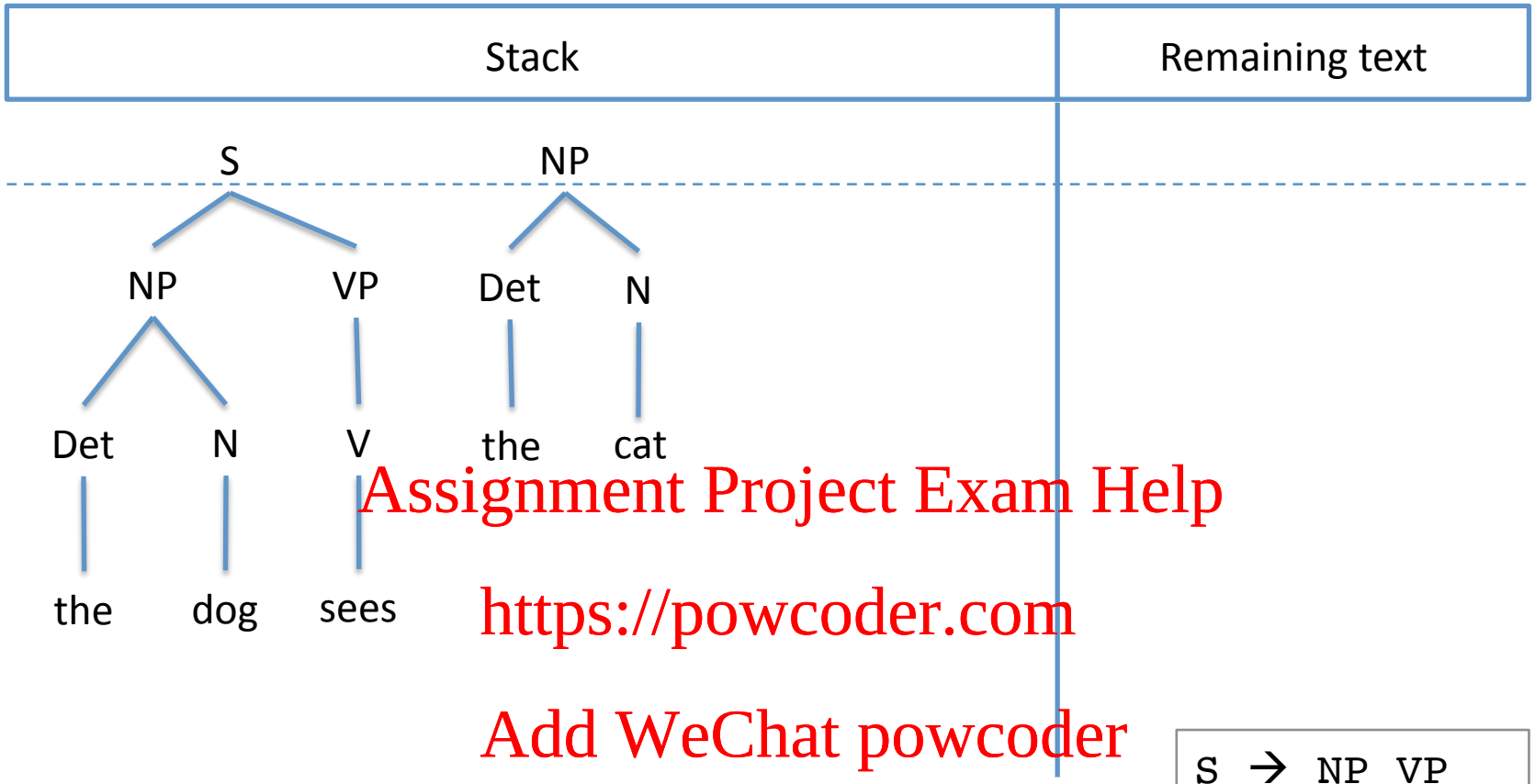
```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```



We removed "cat" from the stack and added a mini tree.

Next up: a reduce

```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```




```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```

We replaced two elements from the stack and replaced them with one NP.

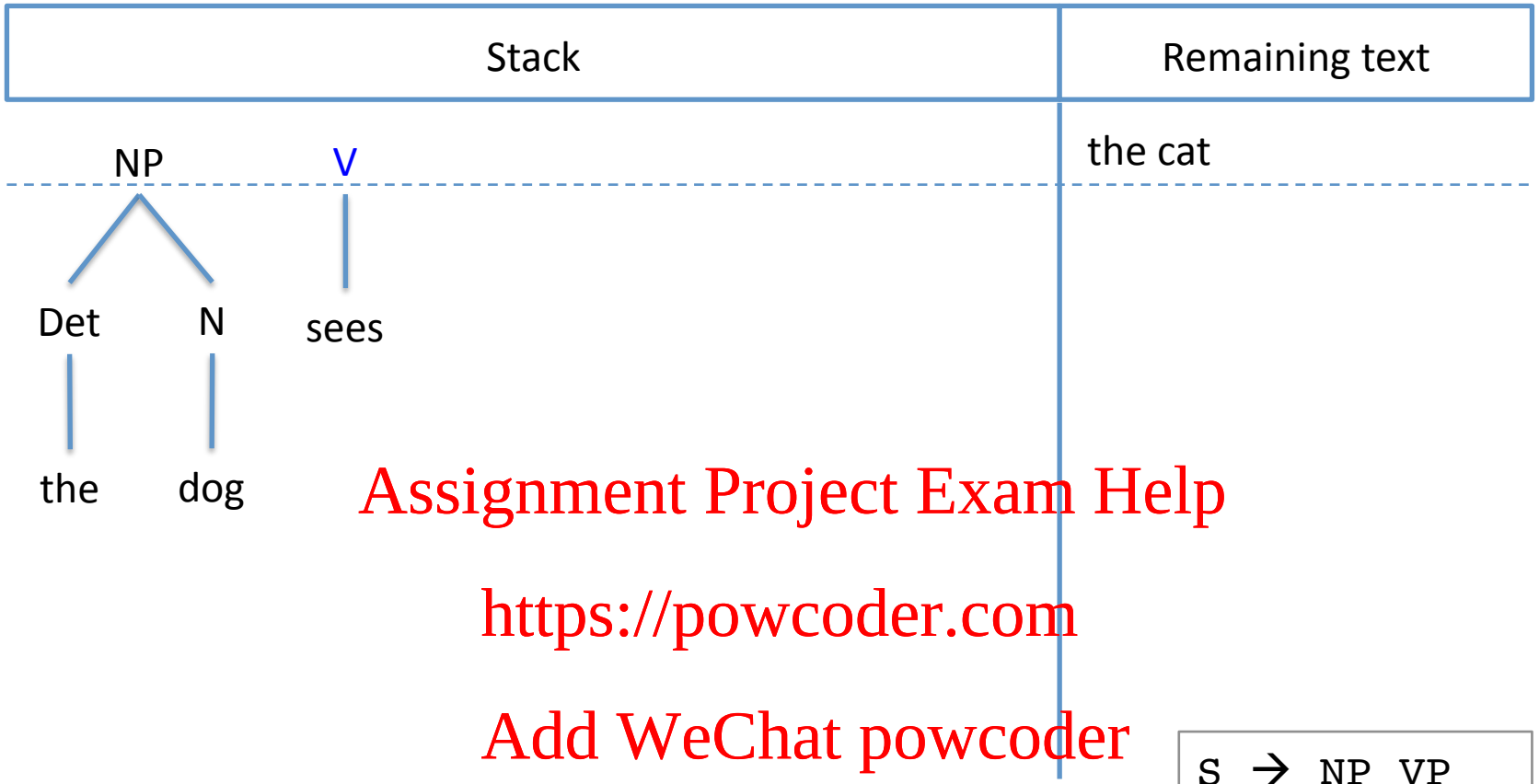
Now there is nothing that we can do, and even though we consumed all input, the parse failed because we should have only one element on the stack at the end.

Shift or reduce, that is the question

- When it is possible to either shift or reduce, what should the parser do?
- Which reduce to use if there are more reduce options?
- The answer to these questions can mean success or failure for the parser

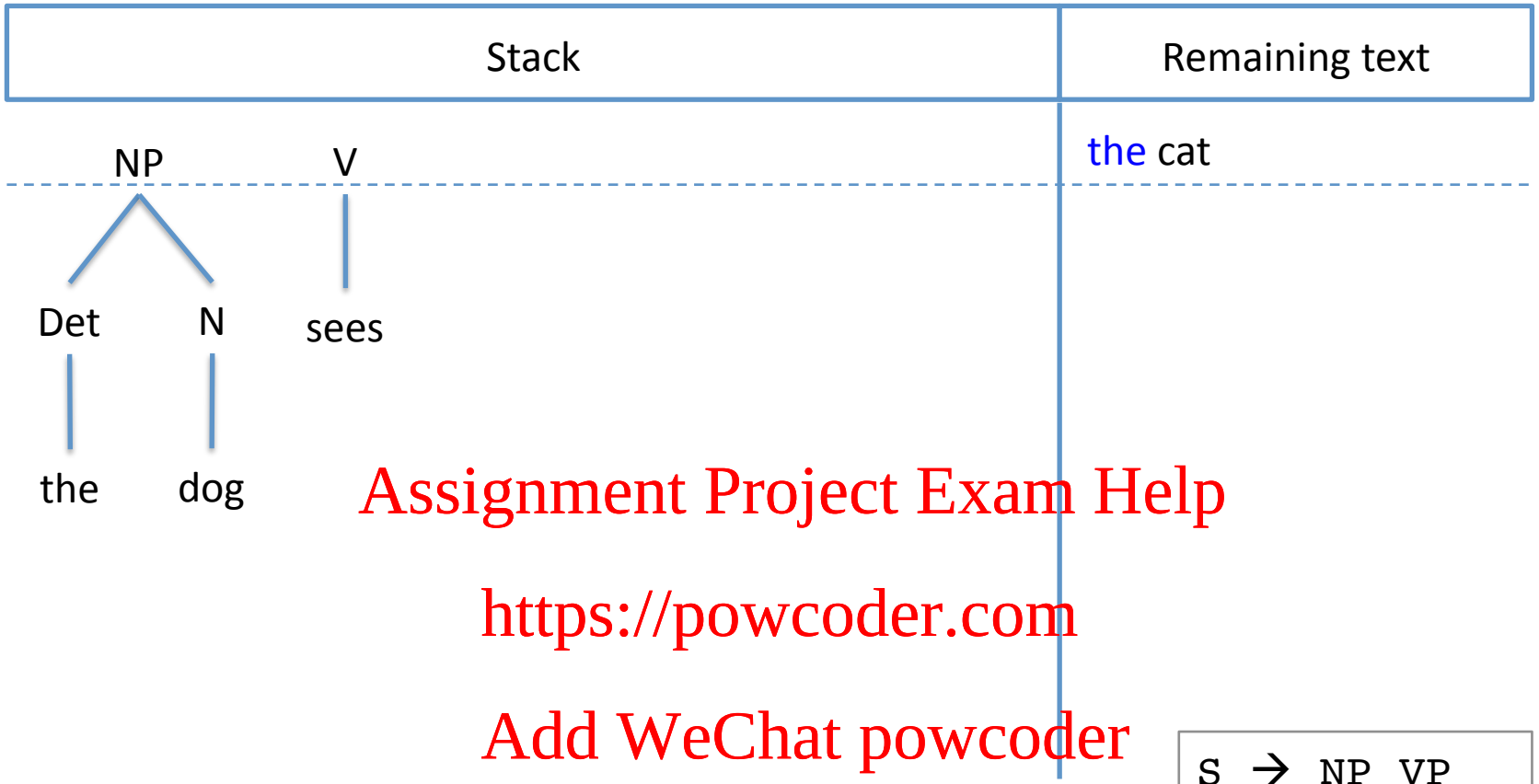
Shift or reduce, that is the question

- Possible answers:
  - Use heuristics
    - Reduce has priority over shift;
    - Use the reduce that covers the most items
  - Assigning a probability to each action, and the action with the highest probability wins
  - Use backtracking to try out all choices if needed



Here is where we took the wrong turn the last time. We were going to do a reduce since those were favored over shifts, but what if we change our rules. (Note that there is a sleight of hand here since I am now changing the rules in the middle of a parse, bad, but it is for explanatory purposes).

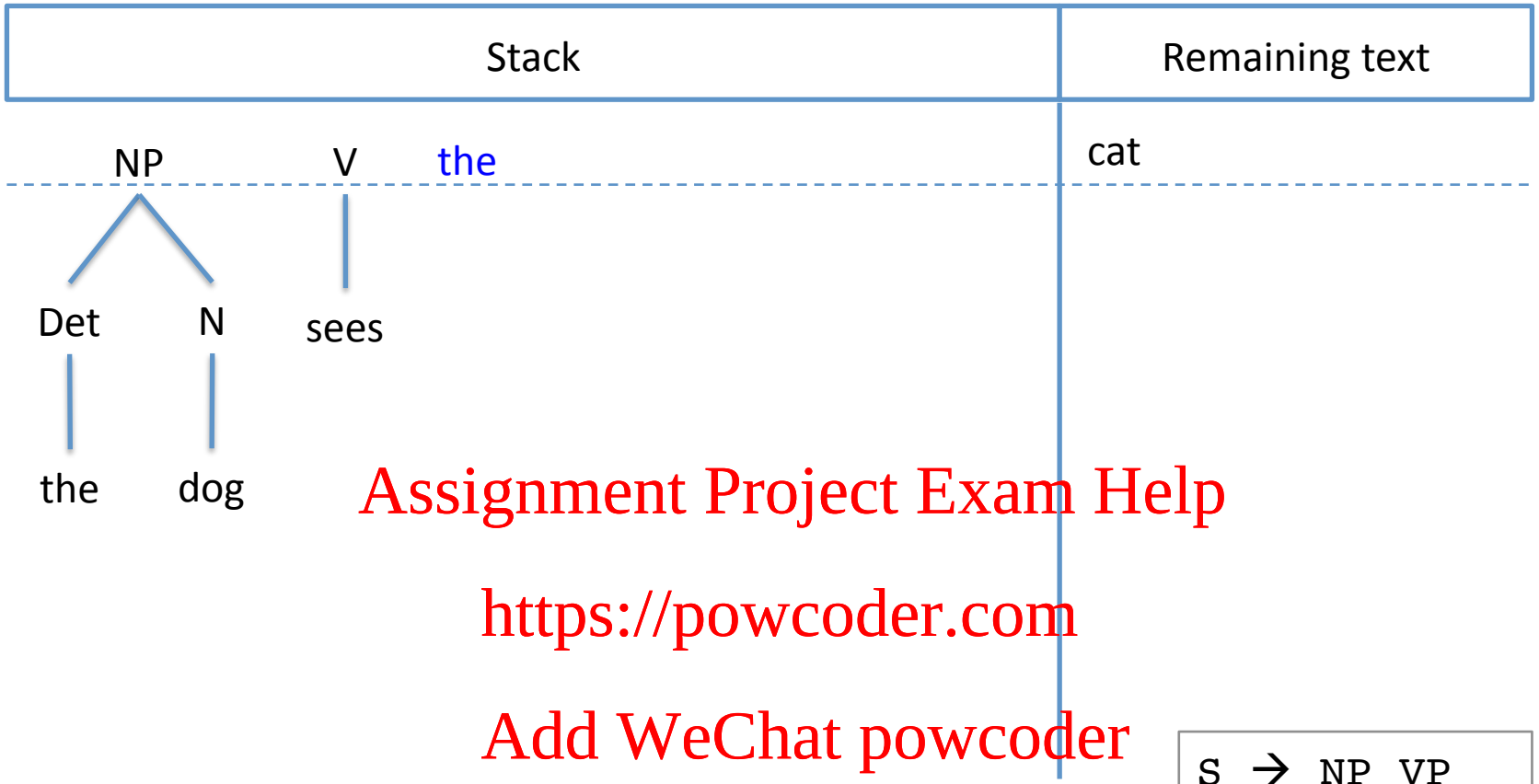
```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```




```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```

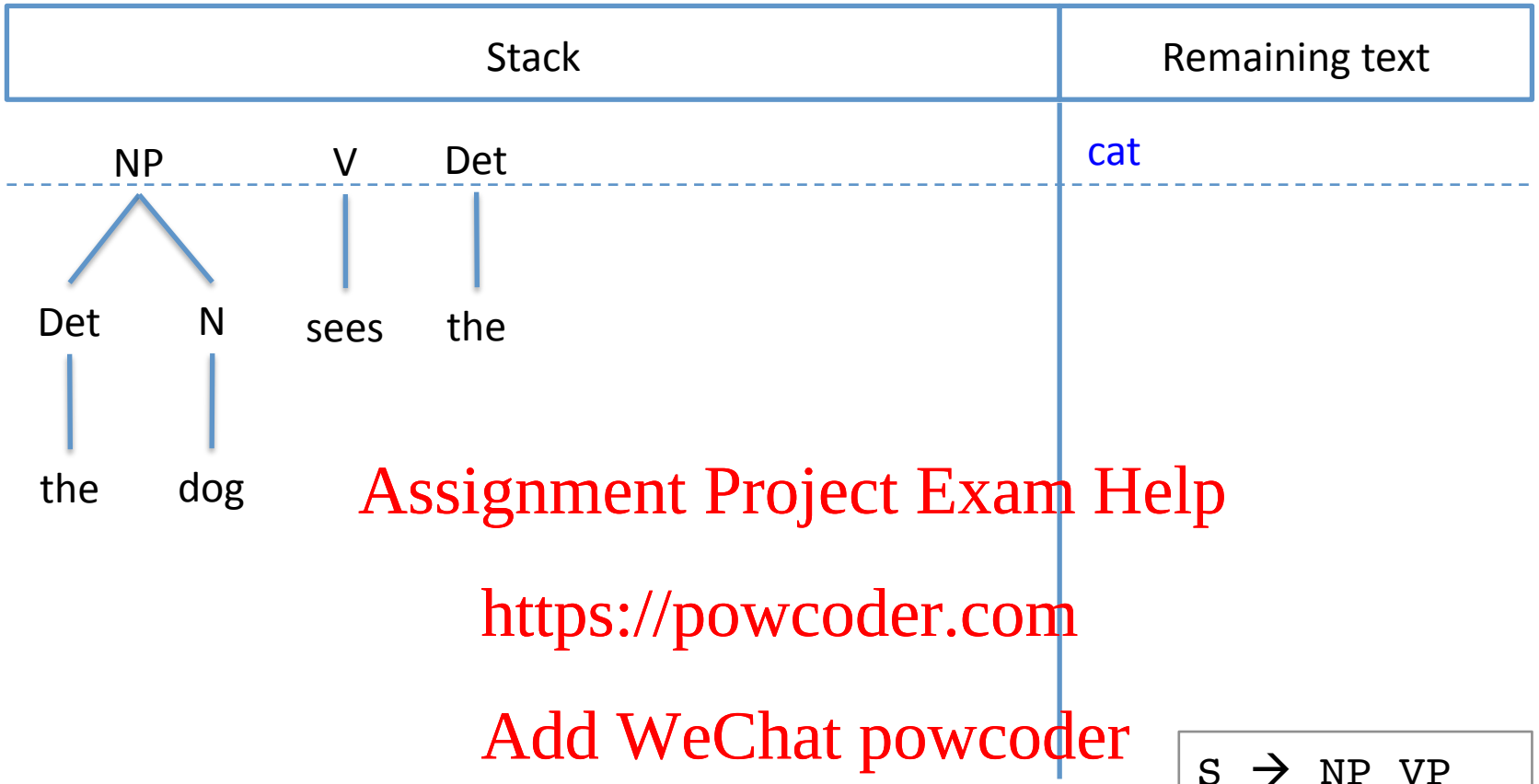
Let's say we now use the following heuristics. First, if available, use a reduce with a terminal rule. Second, if no terminal rules is available use a shift. Third, if there is no shift use the reduce with the rule with the longest right-hand side.

So here we shift.



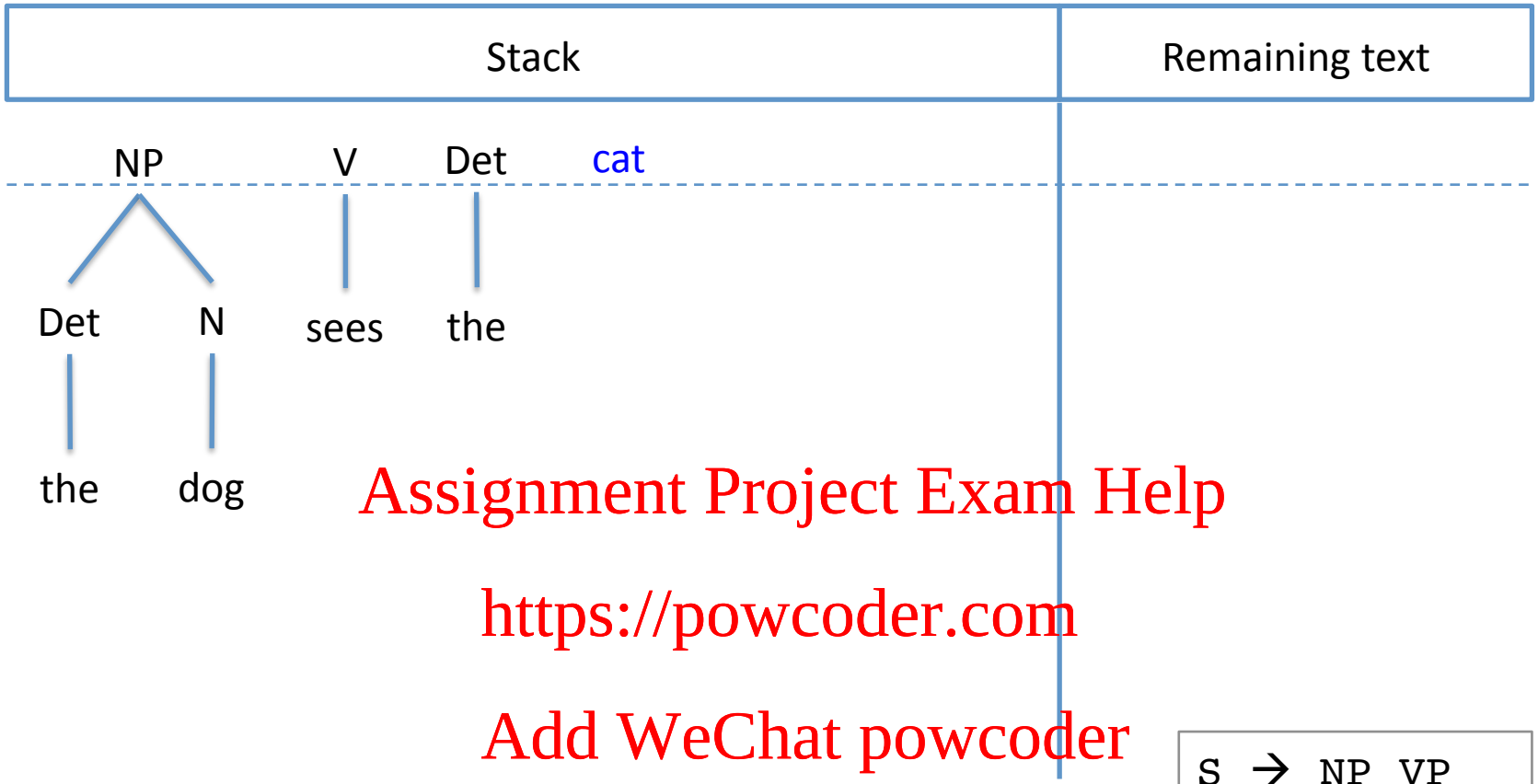
Now we can use a reduce with a terminal rule.

```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```



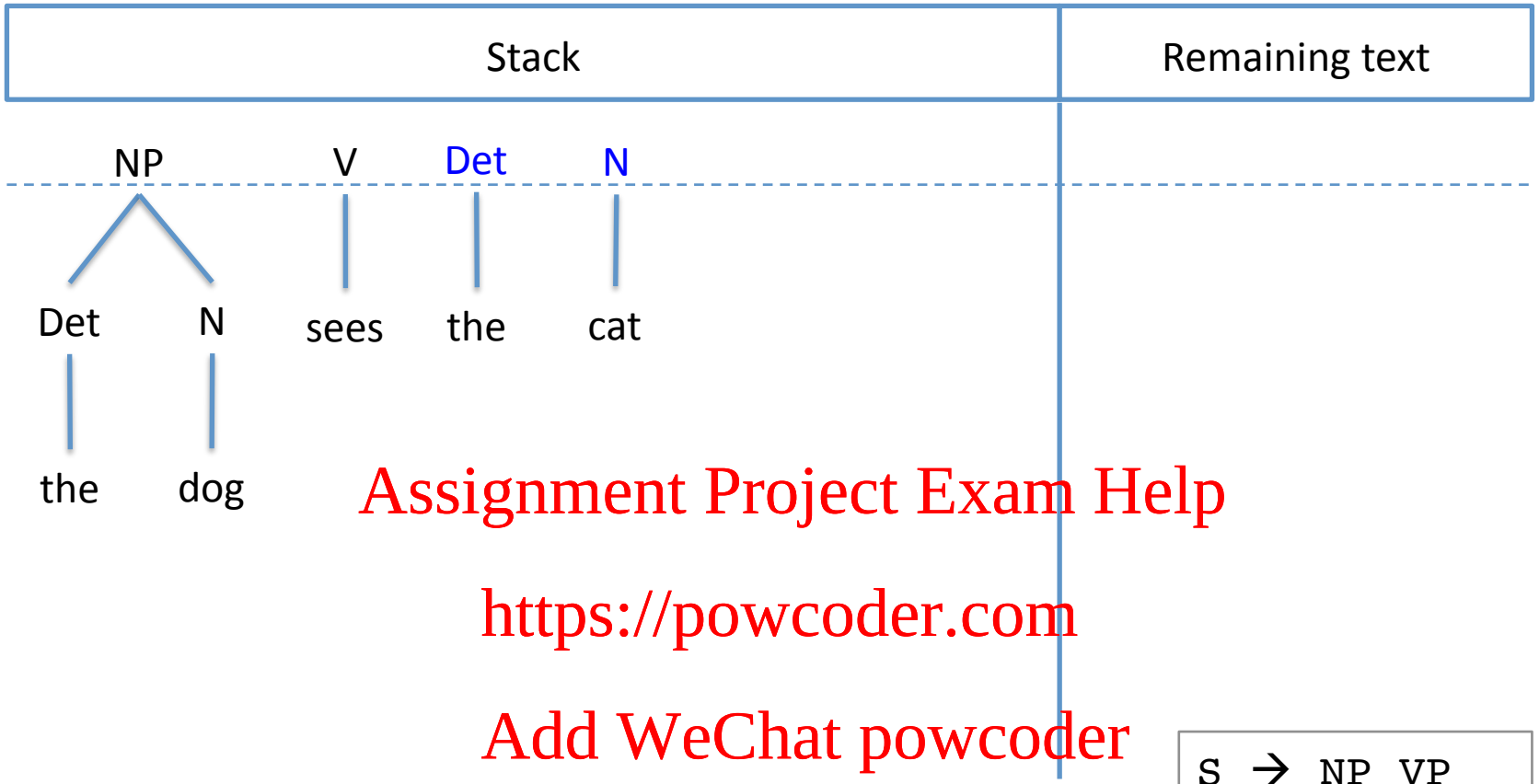
Here we shift again.

```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```



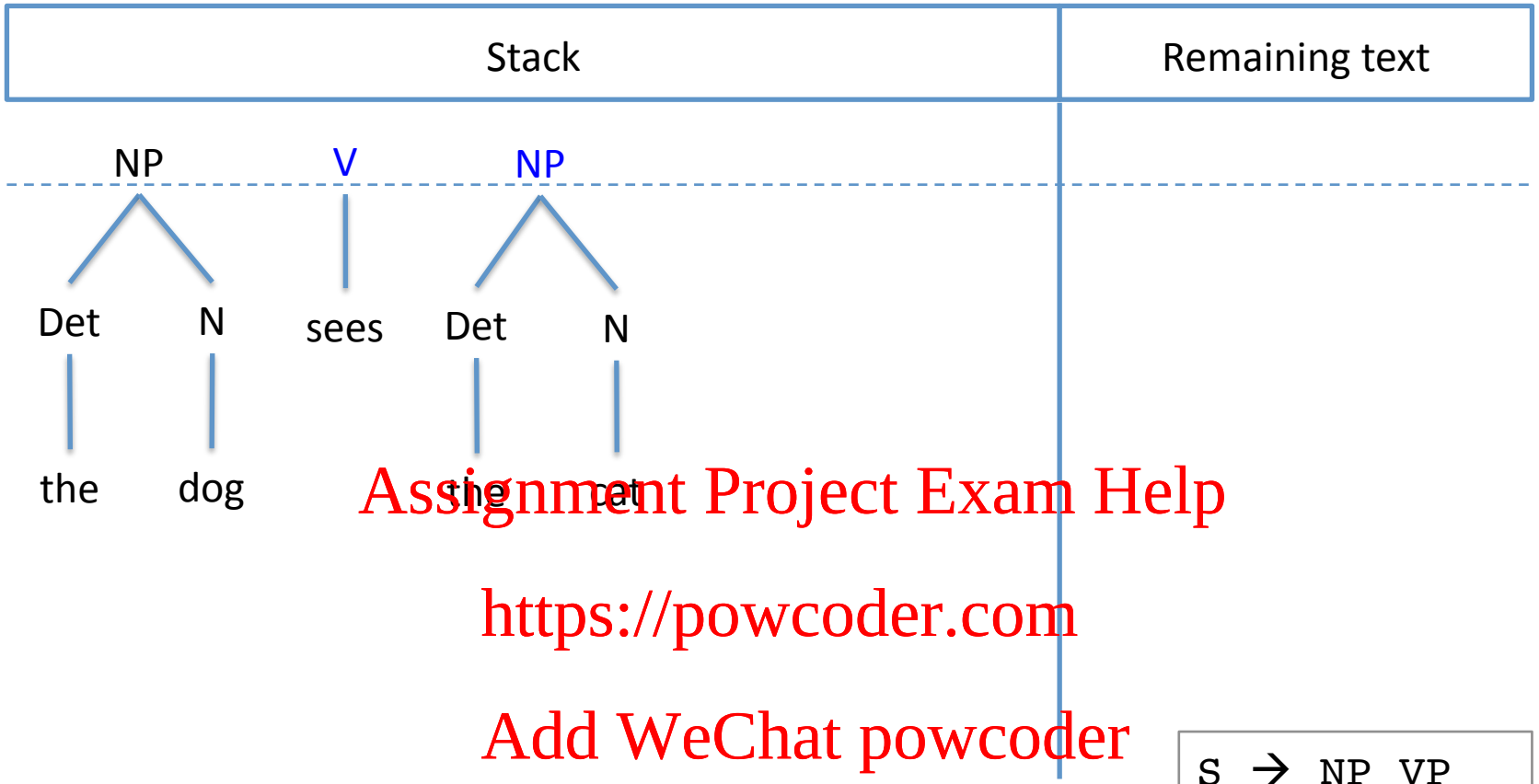
And here we reduce with a terminal rule.

```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```



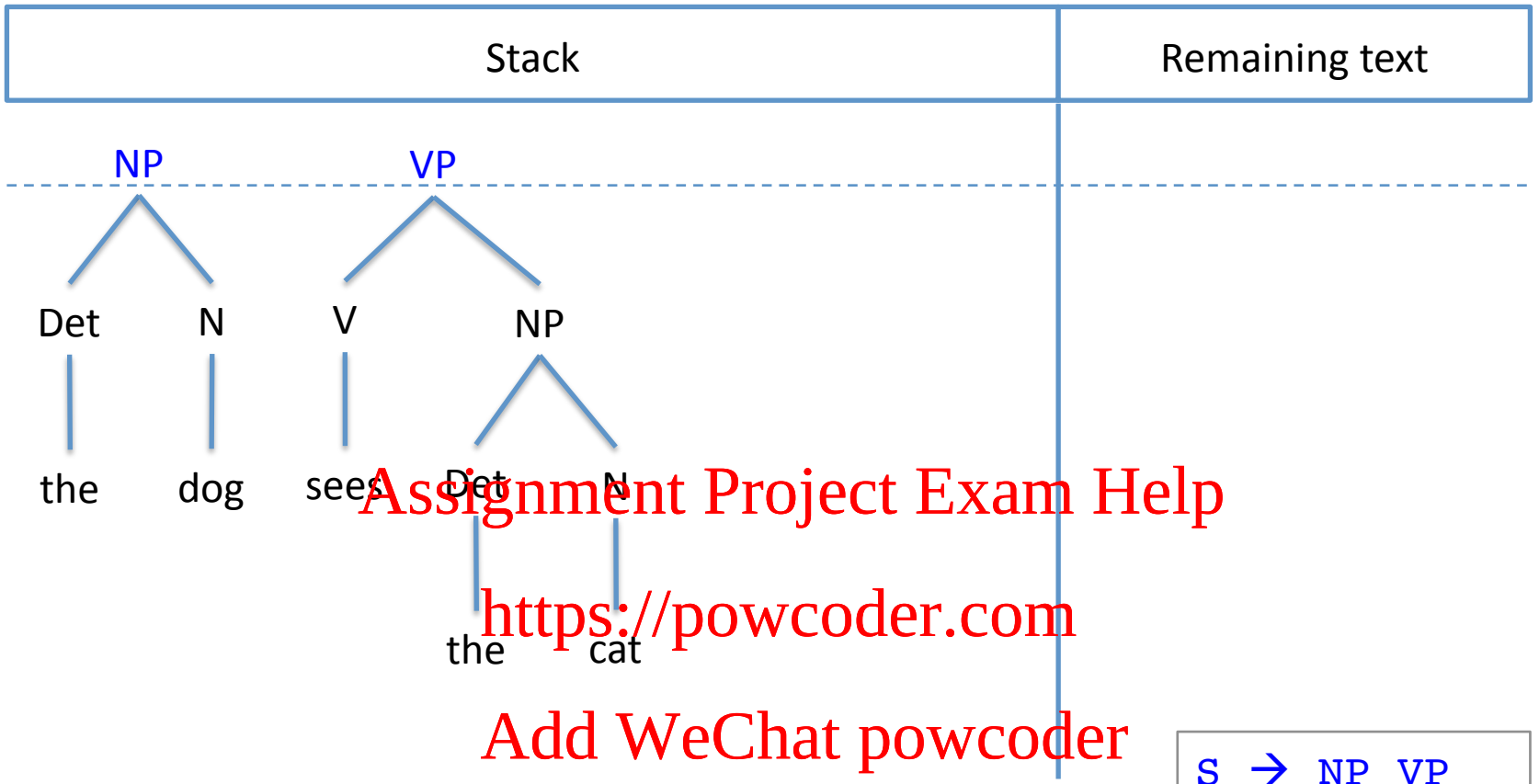
We now do a reduce with the longest available rule, which is NP → Det N.

```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```



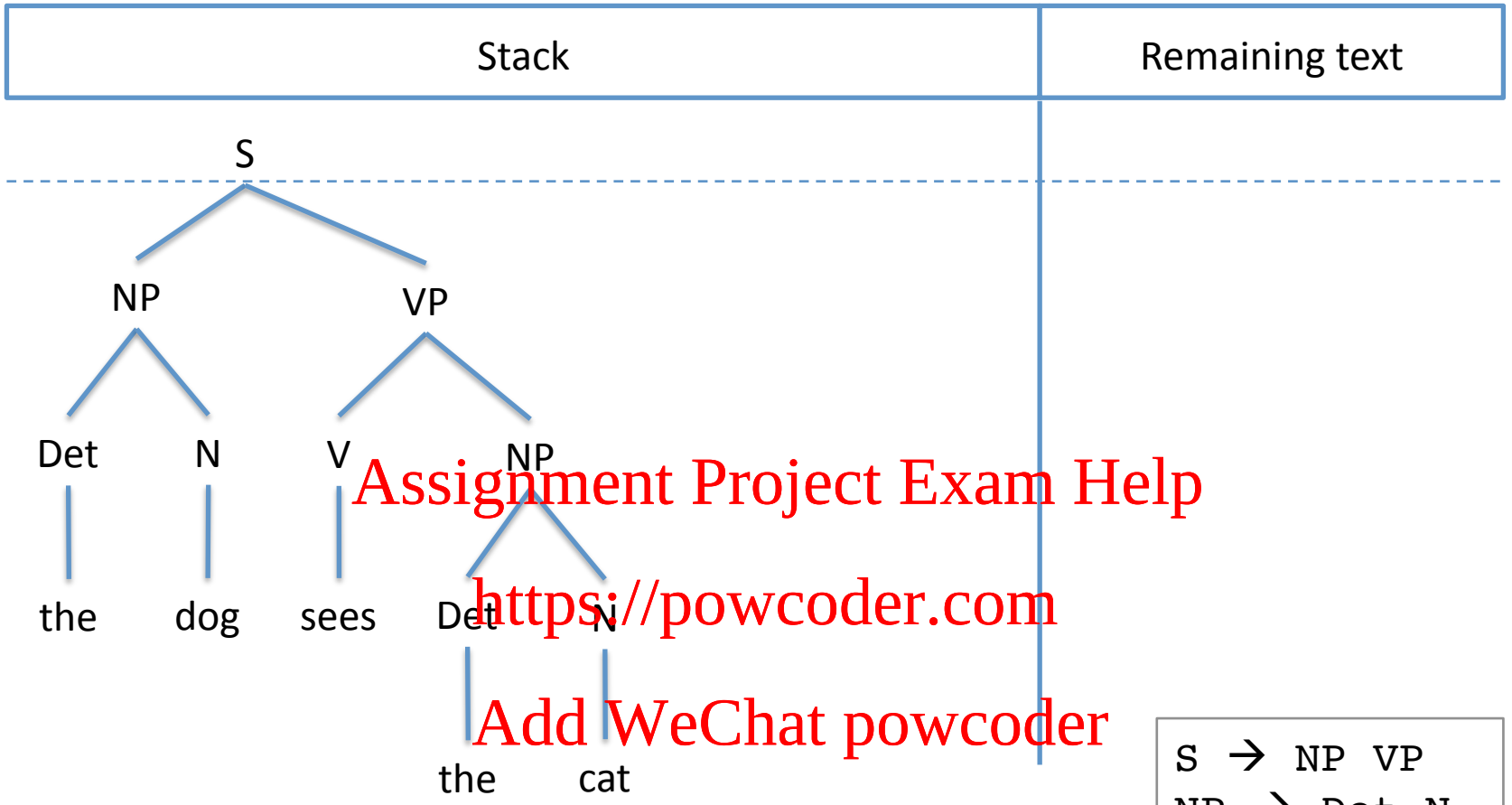

```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```

And again…



And again…

```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```




```
S → NP VP
NP → Det N
VP → V
VP → V NP
Det → "the"
N → "cat"
N → "dog"
V → "sees"
```

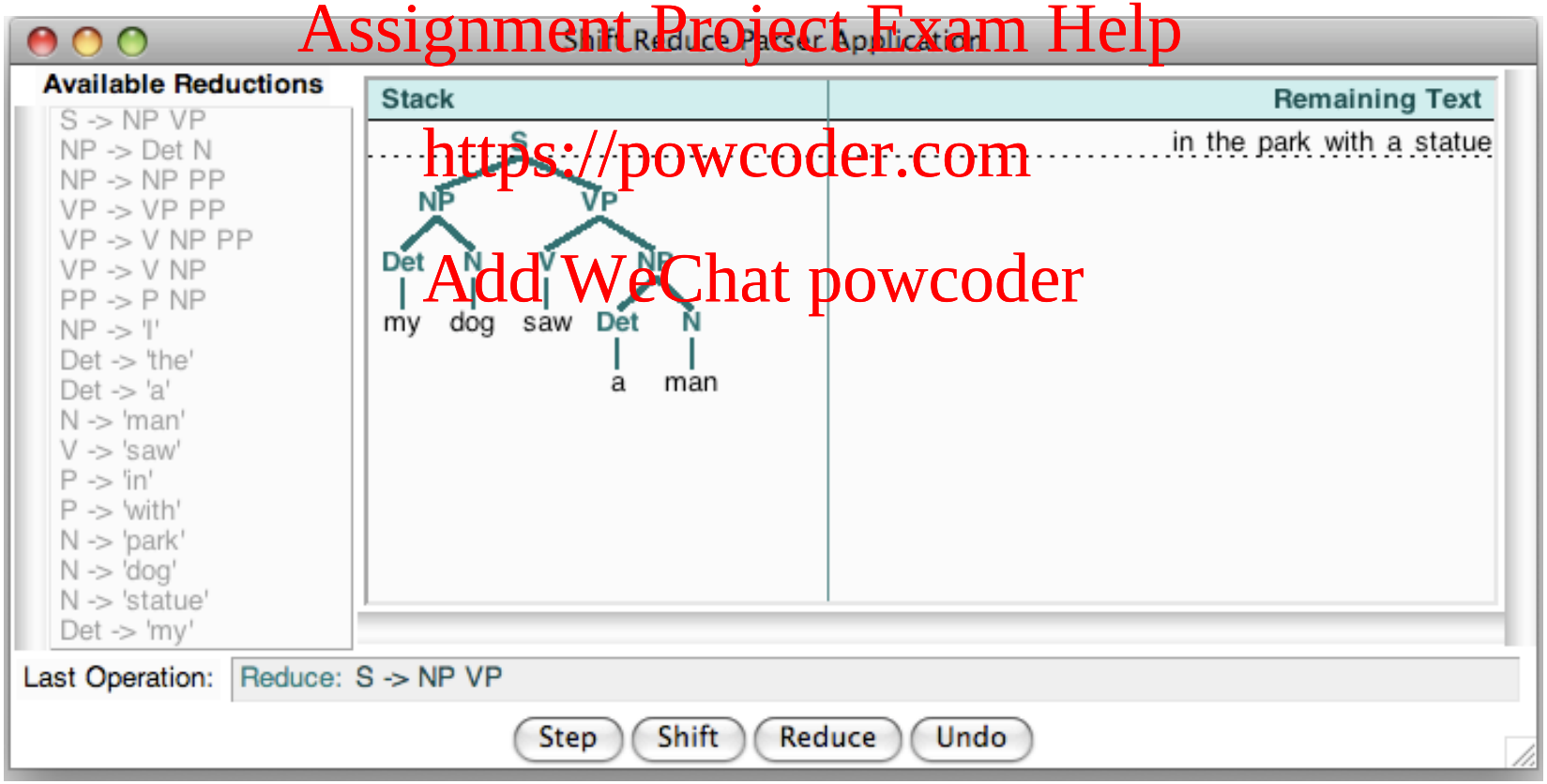
And now we succeeded.

Shift Reduce: pros and cons

- A shift-reduce parser only builds structure that corresponds to the words in the input
- Generally more efficient than top-down
- Problem:
  - If the parser takes the wrong turn, it may not find a parse even if there is one (incompleteness), and it is hard to come up with good heuristics.
  - So backtracking is needed and excessive ambiguity may cause trouble.

Shift-Reduce parser demo

```
import nltk
nltk.app.srparser_app.app()
```

Shift reduce parsing

1. Initial state

Stack	Remaining Text
	the dog saw a man in the park

2. After one shift

Stack	Remaining Text
the	dog saw a man in the park

3. After reduce shift reduce

Stack	Remaining Text
Det N the dog	saw a man in the park

4. After recognizing the second NP

Stack	Remaining Text
NP V NP in Det N saw Det N the dog a man	the park

5. After building a complex NP

Stack	Remaining Text
NP V NP Det N saw NP PP the dog Det N P NP a man in Det N the park	

6. Built a complete parse tree

Stack	Remaining Text
S NP VP Det N V NP the dog saw NP PP Det N P NP a man in Det N the park	

Chart Parser

- Efficient and complete
- An example of Dynamic Programming as applied to parsing

Dynamic Programming

- Solve a problem by breaking it into smaller sub problems
- Two requirements:
  - Optimal substructure: optimal solution can be created from optimal solutions of sub problems
  - Overlapping sub problems: same sub problem is solved over and over again
- Related to Divide and Conquer and Recursion

Dynamic Programming

- Avoid computing a sub problem repeatedly by storing them in a lookup table, thus reducing waste
- It is a technique that is widely used in NLP to solve problems of this kind

Chart Parsing

- Dynamic programming applied to syntactic parsing
- A constituent like *the mouse* will be built only once and stored in a table
- A Well-Formed Substring Table (WFST) can be used to store the intermediate results
- The following is an overly simple example of using an WFST

WFST Example

Input:

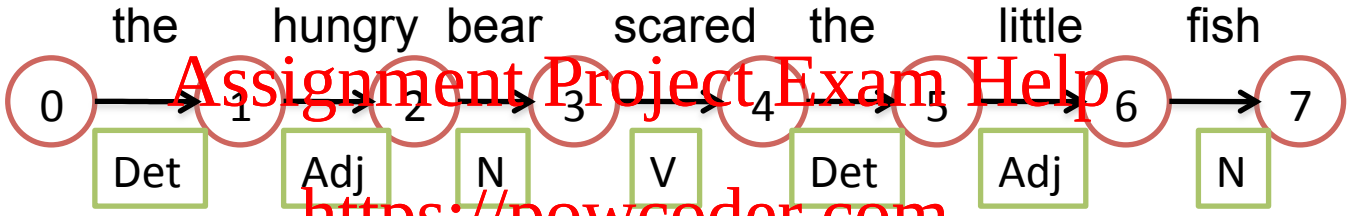
"the hungry bear scared the little fish"

Grammar:


```
S → NP VP
NP → Det Nom
Nom → Adj N
VP → V NP

Det → "the"
Adj → "hungry" | "little"
N → "bear" | "fish"
V → "scared"
```

WFST

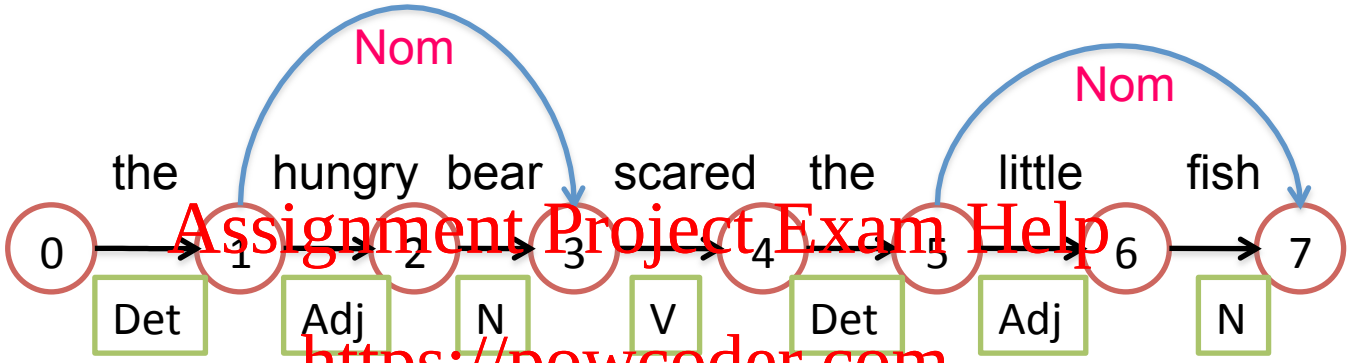


end

Initialization

start

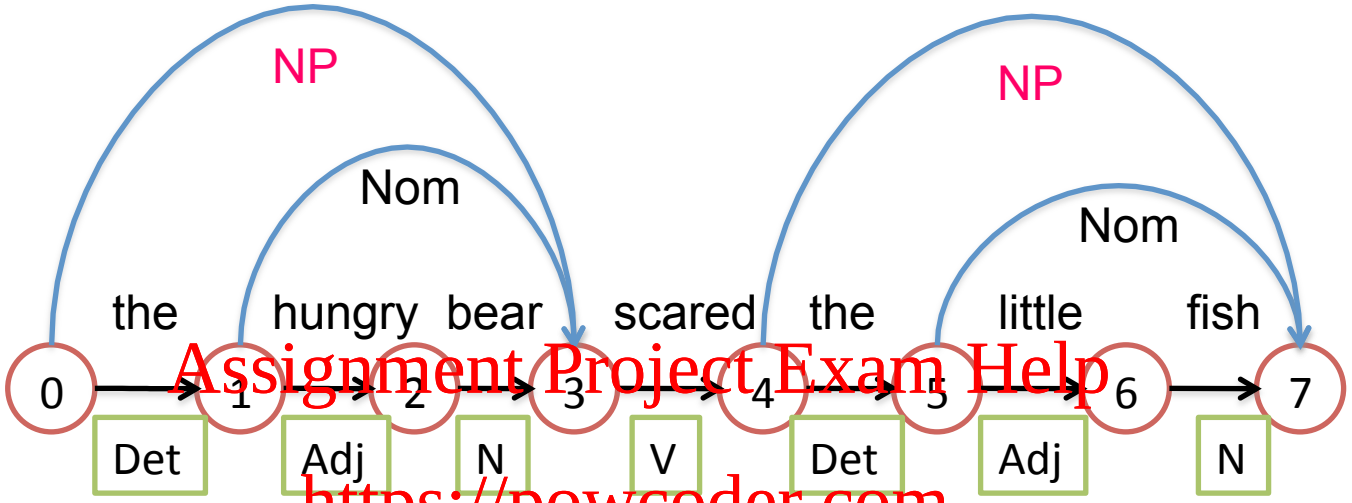
wi	1	2	3	4	5	6	7
0	Det						
1		Adj					
2			N				
3				V			
4					Det		
5						Adj	
6							N



Span = 2

Nom -> Adj N

wi st	1	2	3	4	5	6	7
0	Det	-					
1		Adj	Nom				
2			N	-			
3				V	-		
4					Det	-	
5						Adj	Nom
6							N

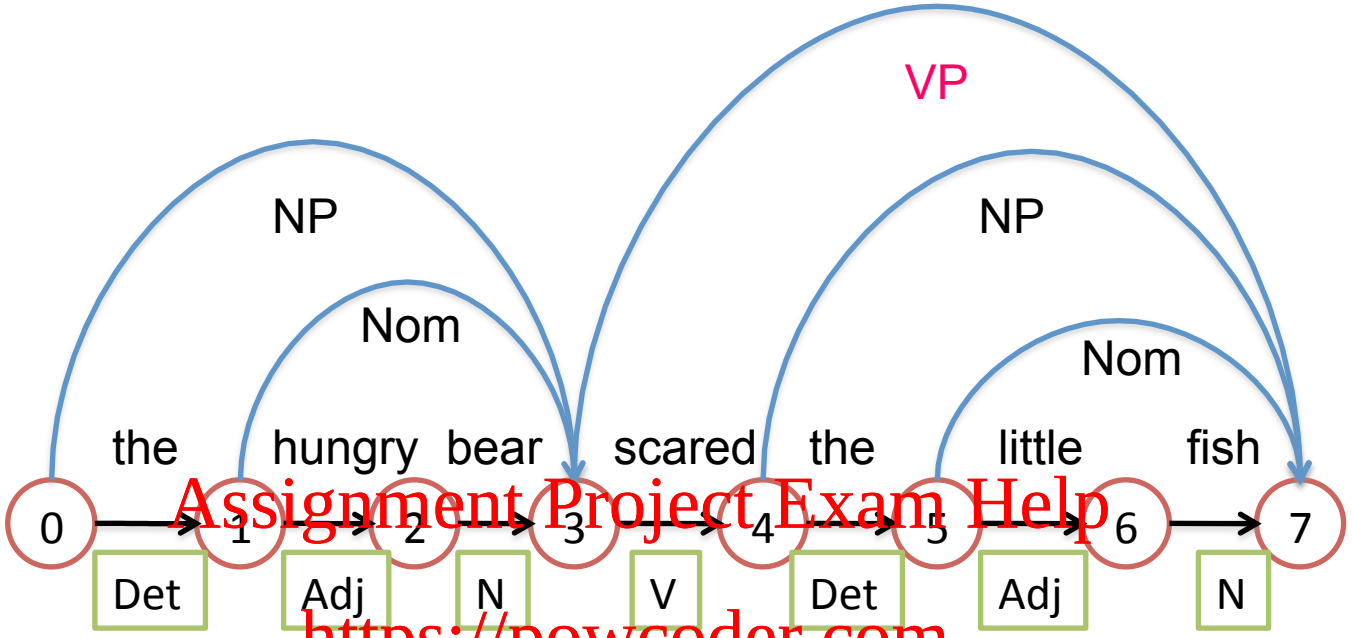


Span = 3

Nom -> Adj N

NP -> Det Nom

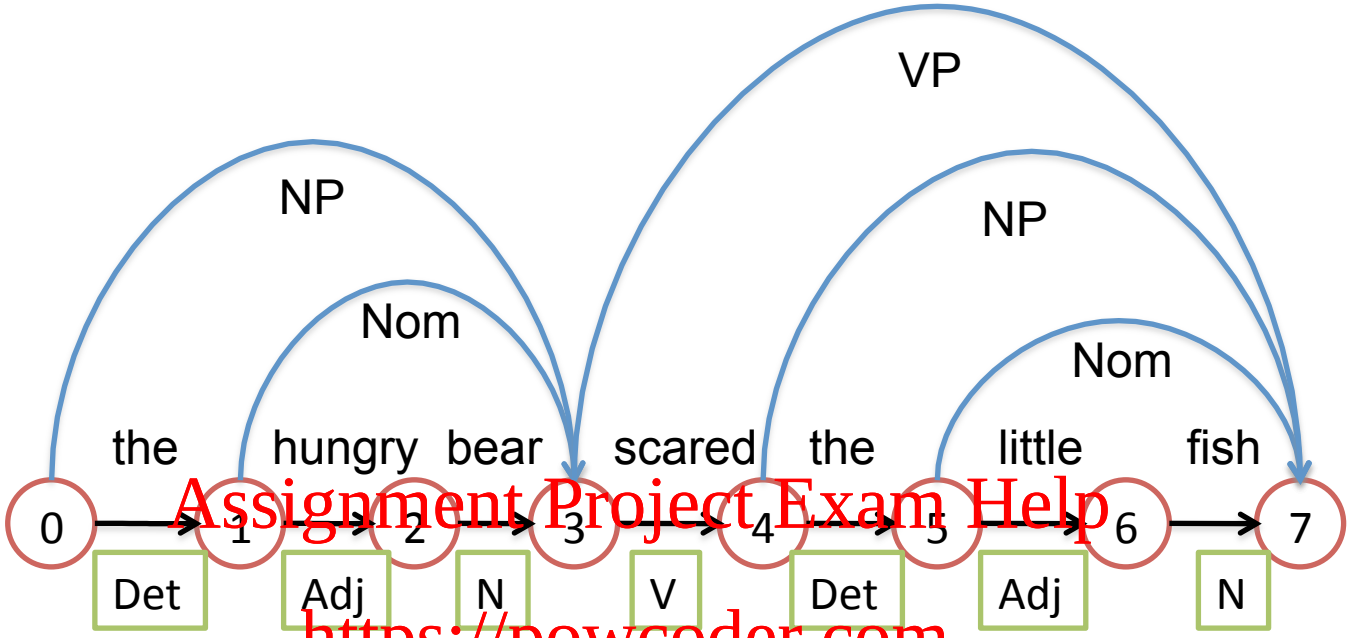
wAst 1	1	2	3	4	5	6	7
0	Det	-	NP				
1		Adj	Nom	-			
2			N	-	-		
3				V	-	-	
4					Det	-	NP
5						Adj	Nom
6							N



Span = 4

Nom -> Adj N
NP -> Det Nom
VP -> V NP

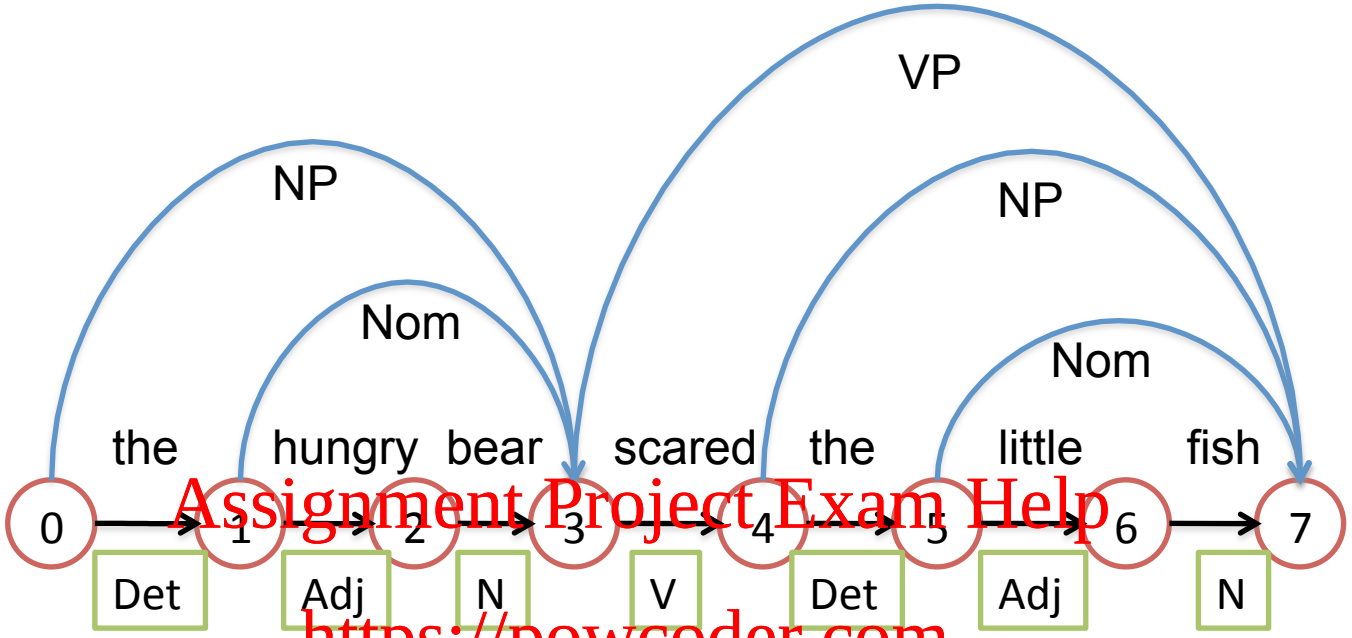
wAst 1	1	2	3	4	5	6	7
0	Det	-	NP	-			
1		Adj	Nom	-	-		
2			N	-	-	-	
3				V	-	-	VP
4					Det	-	NP
5						Adj	Nom
6							N



Span = 5

Nom -> Adj N
NP -> Det Nom
VP -> V NP

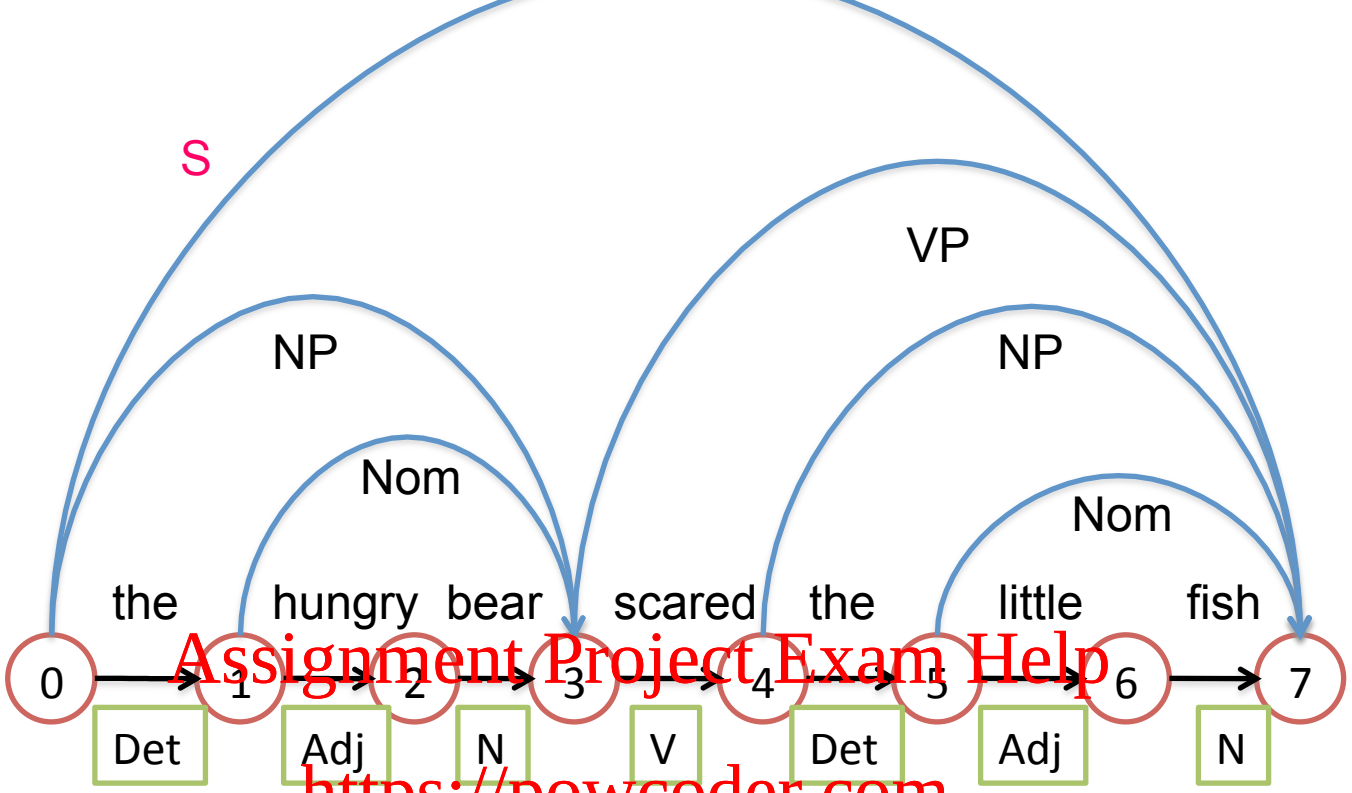
wast	1	2	3	4	5	6	7
0	Det	-	NP	-	-		
1		Adj	Nom	-	-	-	
2			N	-	-	-	-
3				V	-	-	VP
4					Det	-	NP
5						Adj	Nom
6							N



Span = 6

Nom -> Adj N
NP -> Det Nom
VP -> V NP

wfst	1	2	3	4	5	6	7
0	Det	-	NP	-	-	-	
1		Adj	Nom	-	-	-	-
2			N	-	-	-	-
3				V	-	-	VP
4					Det	-	NP
5						Adj	Nom
6							N



Span = 7

Nom -> Adj N
NP -> Det Nom
VP -> V NP
S -> NP VP

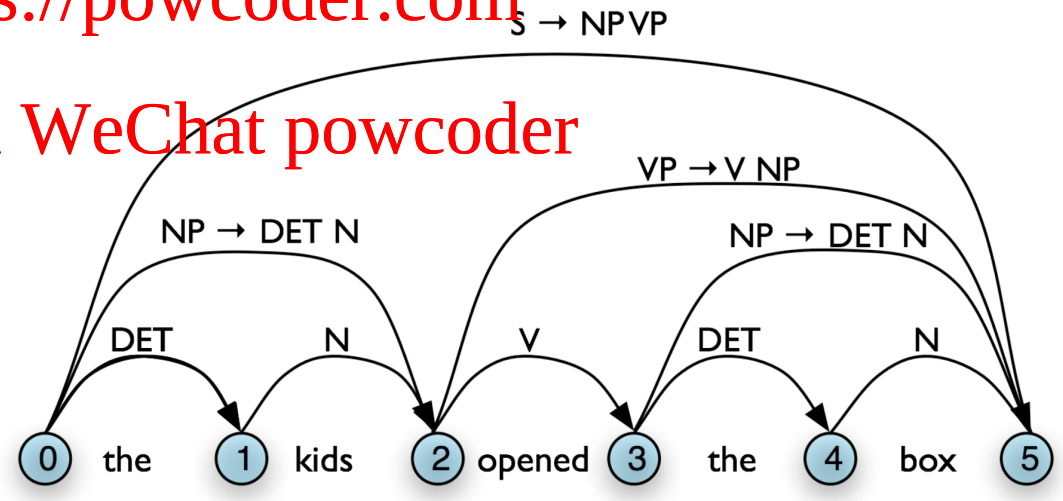
wAst 1	2	3	4	5	6	7
0	Det	-	NP	-	-	-	S
1		Adj	Nom	-	-	-	-
2			N	-	-	-	-
3				V	-	-	VP
4					Det	-	NP
5						Adj	Nom
6							N

Strengths and weaknesses of WFST

- Strength:
  - Much more efficient than the recursive descent parser
  - Does not get stuck in the same way as the shift-reduce parser
- Weaknesses
  - Checks constituents that are not licensed by grammar, potentially wasteful
  - Does not store alternative parses when there is ambiguity (only one category per cell)

Charts

- Graph structure
- More versatile than WFSTs



Alternative parses

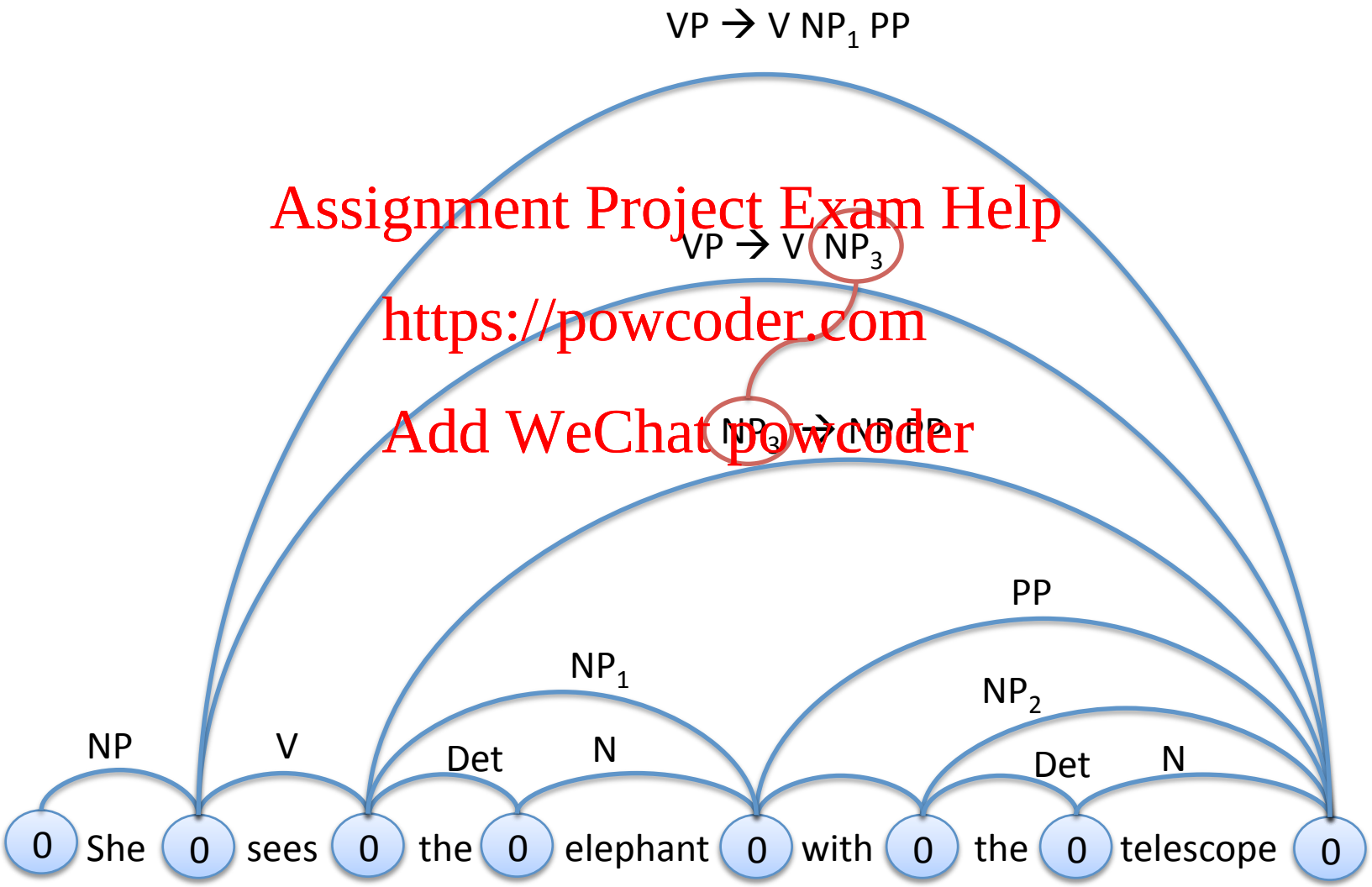


Chart Parsers

- CYK parser
  - Cocke-Younger-Kasami algorithm
  - Bottom-up
- Earley parser
  - Top-down

Earley parser

- Fill the chart in a single left-to-right pass over the input.
- The chart will be size N + 1, where N is the number of words in the input.
- Chart entries are associated with the gaps between the words
  - like slice indexing in Python.
- For each position in the sentence, the chart contains a set of edges representing the partial parse trees generated so far.

Earley parser

- Can deal with all context free grammars
- Operates in $O(n^3)$
- Operations
  - Predict
  - Scan
  - Complete

0 Mary 1 feeds 2 the 3 otter 4 eos 5

	0	1	2	3	4	5
0	X → • S eos S → • NP VP NP → • N NP → • DET N N → • Mary N → • otter DET → • the	N → Mary • NP → N • S → NP • VP			S → NP VP • X → S • eos	**X → S eos •**
1		VP → • V NP V → • feeds	V → feeds • VP → V • NP		VP → V NP •	
2			NP → • N NP → • DET N N → • Mary N → • otter DET → • the	DET → the • NP → DET • N	NP → DET N •	
3				N → • Mary N → • otter	N → otter •	
4					eos → • eos	eos → eos •
5						