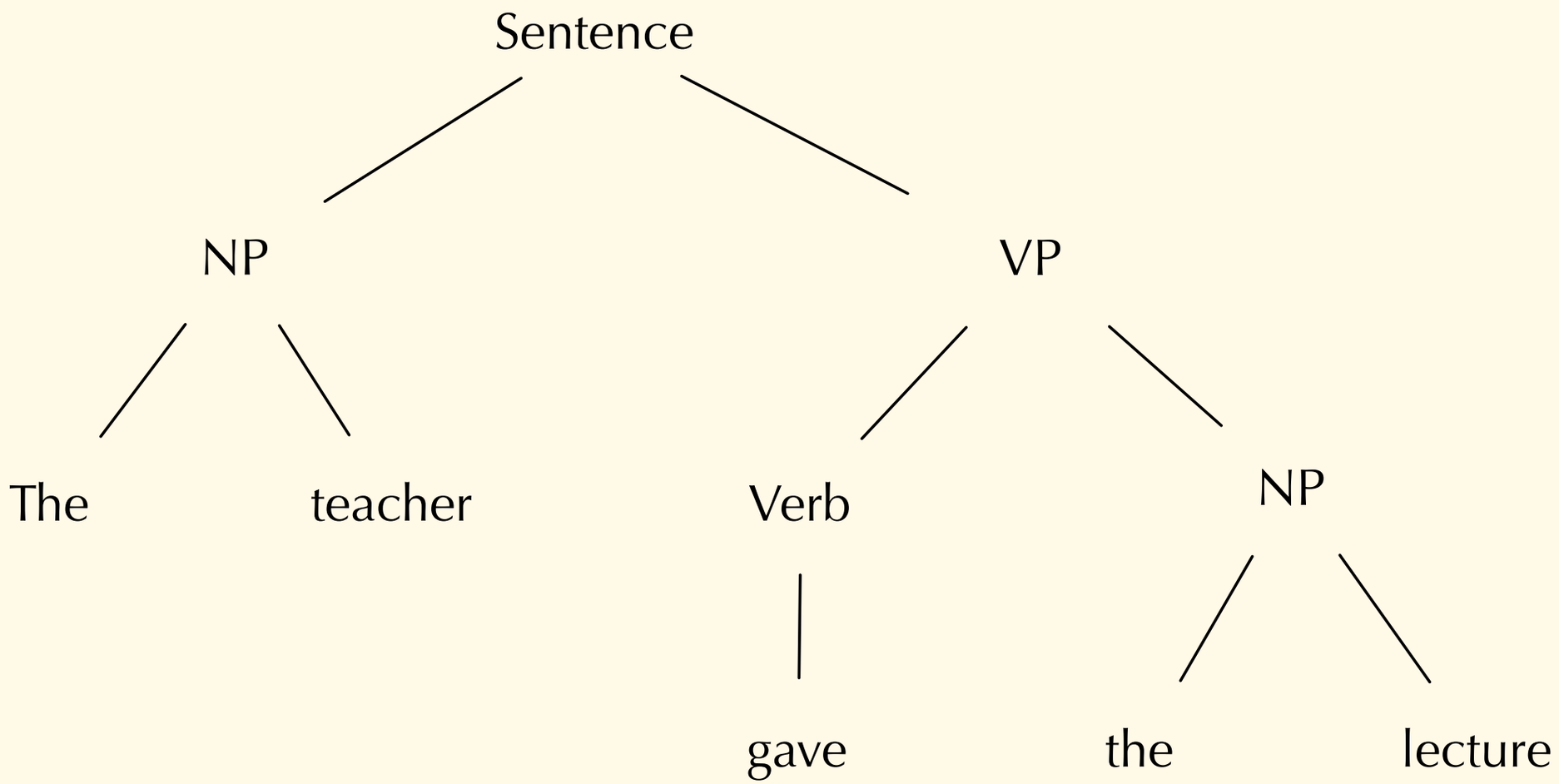
Grammars & Parsing, Part 1:

Rules, representations, and transformations- oh my!



Game plan for today:

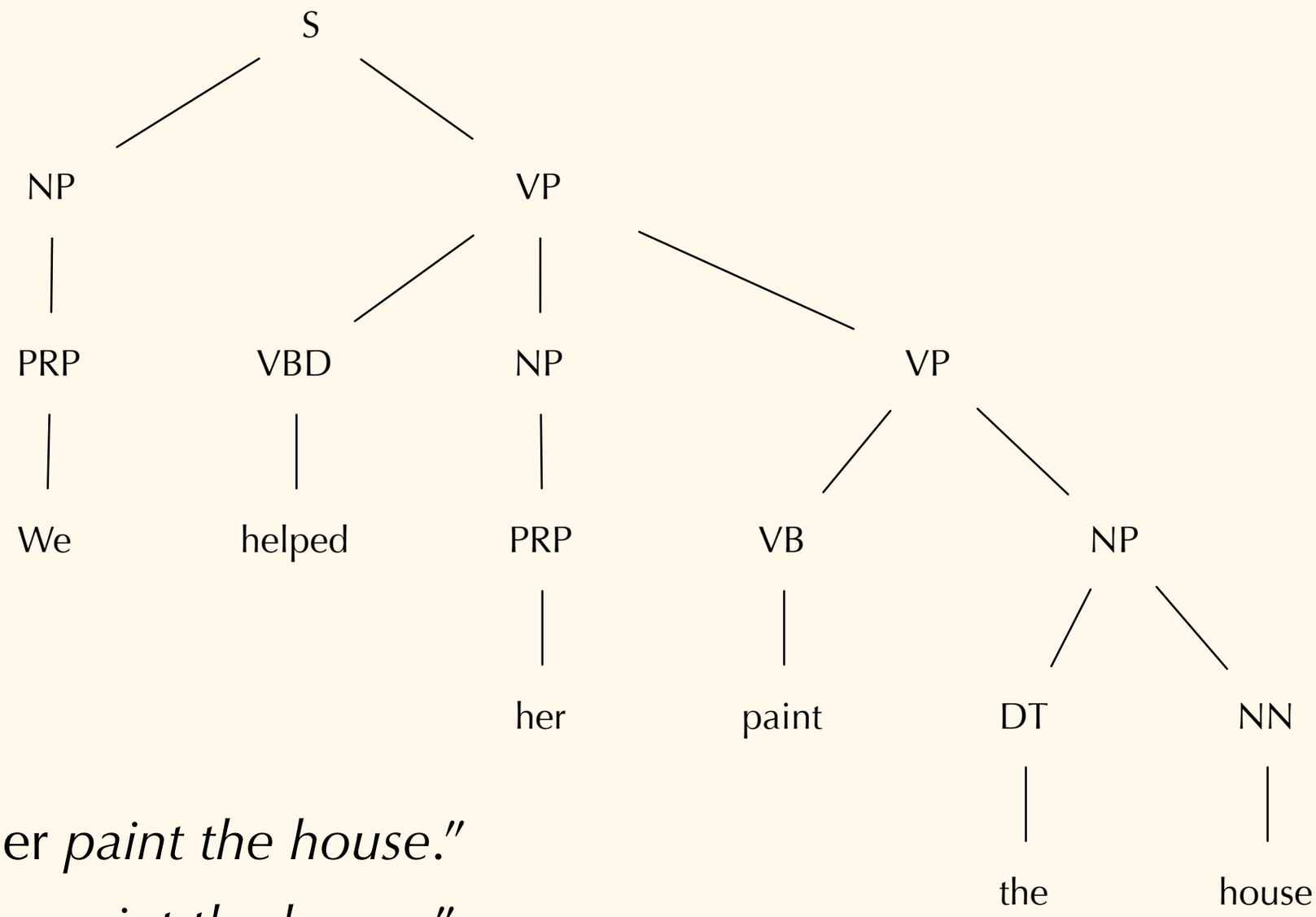
Review of constituents, and why we care

Your friend, the context-free grammar

Introduction to parsing

Tree transformations for fun and profit

Constituents are a sequence of words that behave as a unit.*



"We helped her *paint the house*."

"He helped her *paint the house*."

"They watched her *paint the house* while they drank lemonade."

* This is a somewhat fuzzy definition.

The same constituent often can appear in different contexts:

"*On September seventeenth*, I'd like to fly from Atlanta to Denver."

"I'd like to fly *on September seventeenth* from Atlanta to Denver."

I'd like to fly from Atlanta to Denver *on September seventeenth*."

Why do we care?

Often, the important information in a sentence can only be understood in terms of constituents:

"*On September seventeenth*, I'd like to fly from Atlanta to Denver."

When do they want to fly?

Why do we care?

Often, the important information in a sentence can only be understood in terms of constituents:

"On September seventeenth, I'd like to fly *from Atlanta to Denver*."

Where do they want to go?

Why do we care?

Sometimes, template-filling and regular expressions do the trick...

"*On September seventeenth*, I'd like to fly from Atlanta to Denver."

"I'd like to fly *on September seventeenth* from Atlanta to Denver."

I'd like to fly from Atlanta to Denver *on September seventeenth*."

... often, though, we need a more robust syntactic analysis.

Many NLP tasks make use of syntactic information:

Grammar checking (in e.g., MS Word)

(If a sentence's syntax looks "wrong," it might be ungrammatical)

Information extraction & retrieval

Who/what is the article talking about? When do the events described take place? Where is the user trying to go?

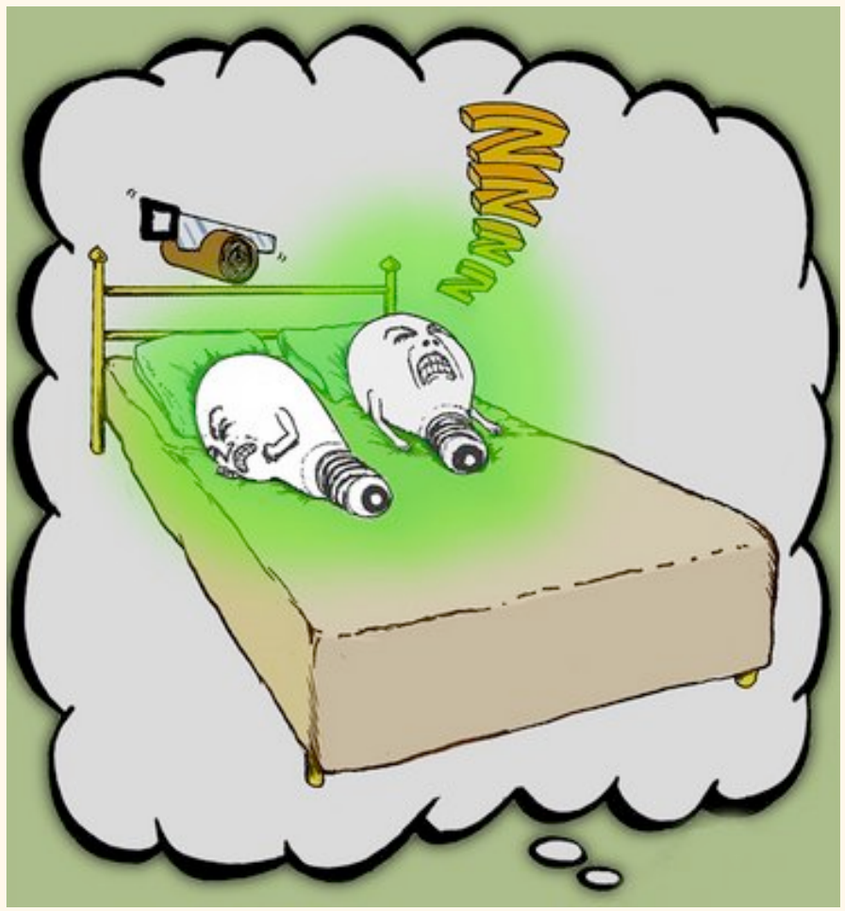
Machine translation

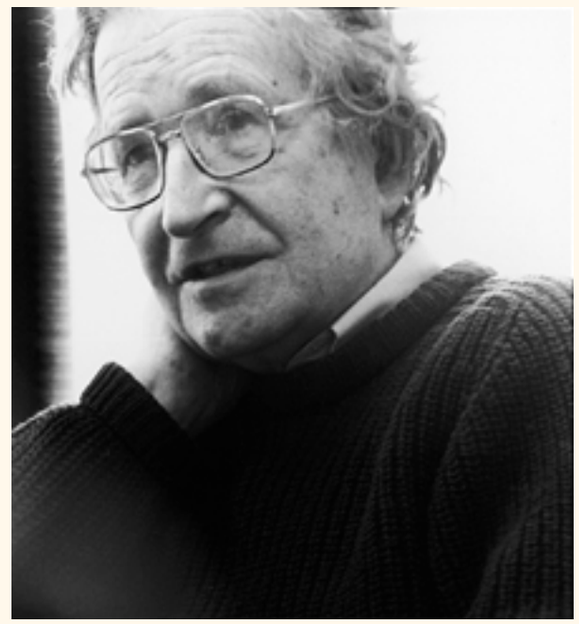
Going from SVO to SOV is easier if you know which words/ constituents are which!

Hwæt! Syntax is very useful...

... but it ain't everything.

"Colorless green ideas sleep furiously."

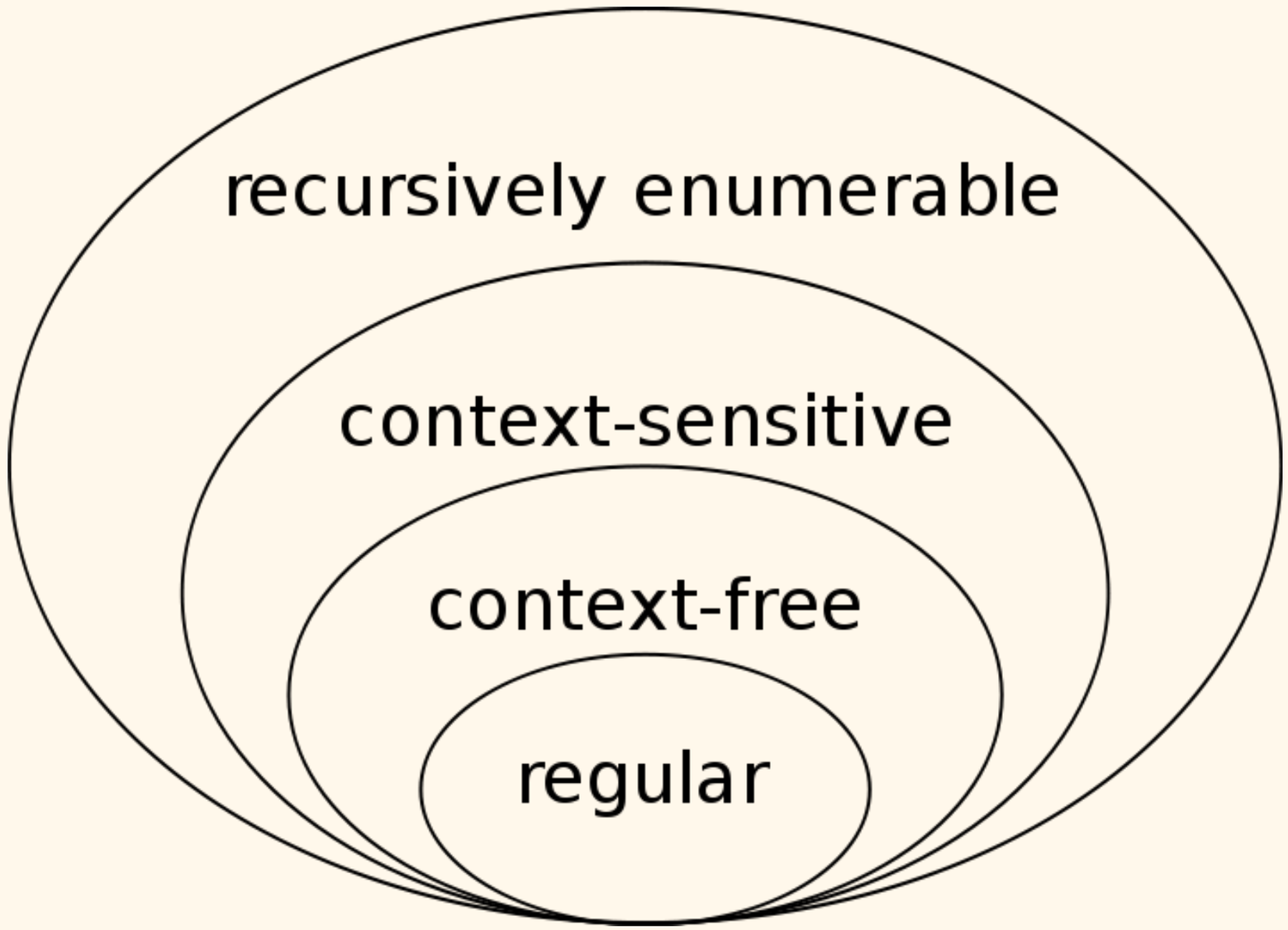




Noam Chomsky
1928 – present

http://wmjasco.blogspot.com/2008/11/colorless-green-ideas-do-not-sleep.html
http://itre.cis.upenn.edu/%7Emyl/languagelog/archives/000025.html

The “Chomsky Hierarchy” describes several classes of formal grammars:



Each superclass can express more complex constructions than its children.

https://en.wikipedia.org/wiki/File:Chomsky-hierarchy.svg

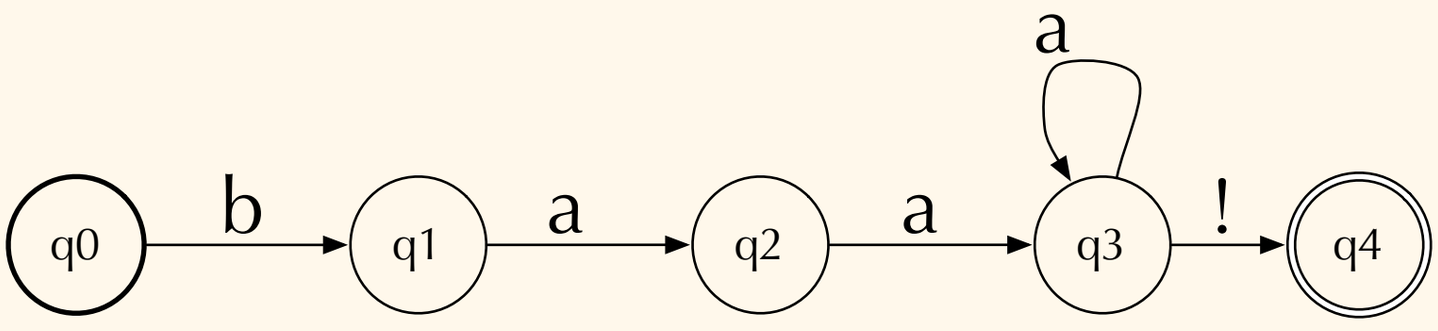
We've already talked about regular grammars:

baaa!

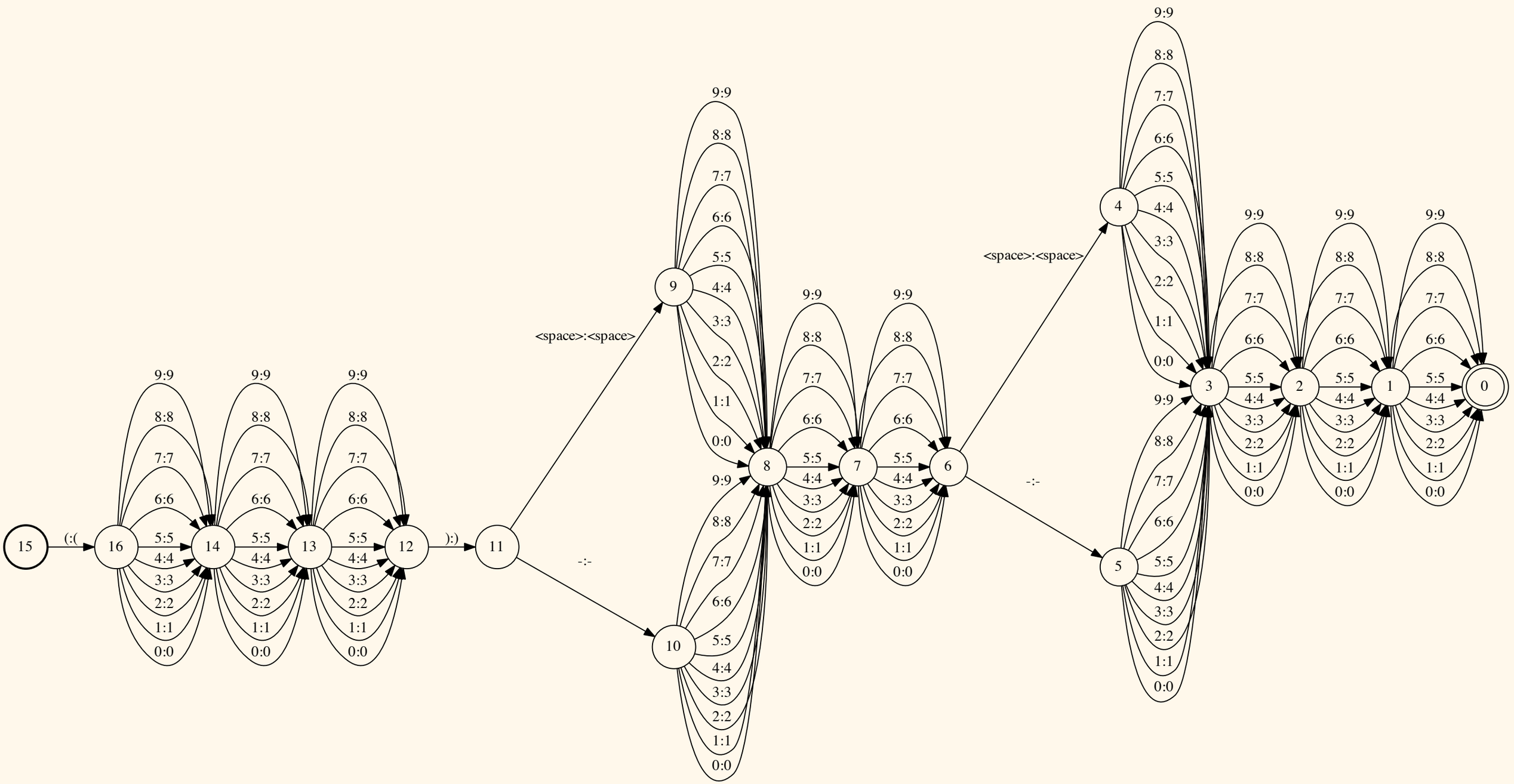
baaaaaaaa!

baa!

/baa+!/



\(\d{3}\)[-]\d{3}[-]\d{4}


15
(:(
16
14
13
12
):)
11
-:-
10
9
8
7
6
<space>:<space>
4
5
3
2
1
0
9:9
8:8
7:7
6:6
5:5
4:4
3:3
2:2
1:1
0:0

Regular languages can be very powerful...

... but have their limitations.

For example:

Write a regular expression to tell if a string's nested parentheses match up.

((2 + 3) * 4) Yes!

((2 + 3) * 4 No!

Python obviously manages to do it, somehow...

```
% cat python_syntax_example.py
print ( ( 2 + 3 ) * 4 )
print ( ( 2 + 3 ) * 4

% python python_syntax_example.py
  File "python_syntax_example.py", line 3

                                ^
SyntaxError: invalid syntax
```

But it can't do it using a regular grammar.

Another example:

Try and use a regular grammar to match the family of strings $a^n b^n$.

E.g., match “aaabbb”, “aaaabbbb”, etc...

... but not “aaabb”, “aabbb”, etc.

A useful way to think about it: can you make an FSA to do this?

Both cases are examples of languages that can be described using *context-free grammars* but not with *regular grammars*.

A context-free grammar (CFG) is a 4-tuple consisting of:

N	A set of non-terminal symbols
Σ	A set of terminal symbols
R	Set of rules of the form $A \rightarrow \alpha^*$ where α is a string of symbols from $(\Sigma \cup N)$
S	A designated start symbol

Any string from a context-free language can be produced by recursively applying the rewrite rules in its grammar...

... and any string that *cannot* be so produced is not part of that language!

A (very) simple example: basic arithmetic.

Let's write a grammar that can tell us whether an arithmetic expression (e.g. "2 + (3 - 4)") is well-formed.

The simplest expression is just a number:

Exp → number

Valid unary operators are "+" and "-" (e.g., "-4"), and their result is also an expression:

UnOp → "+" | "-"
Exp → UnOp Exp

Binary operators work similarly:

BinOp → "+" | "-" | "*" | "/"
Exp → Exp BinOp Exp

A (very) simple example: basic arithmetic.

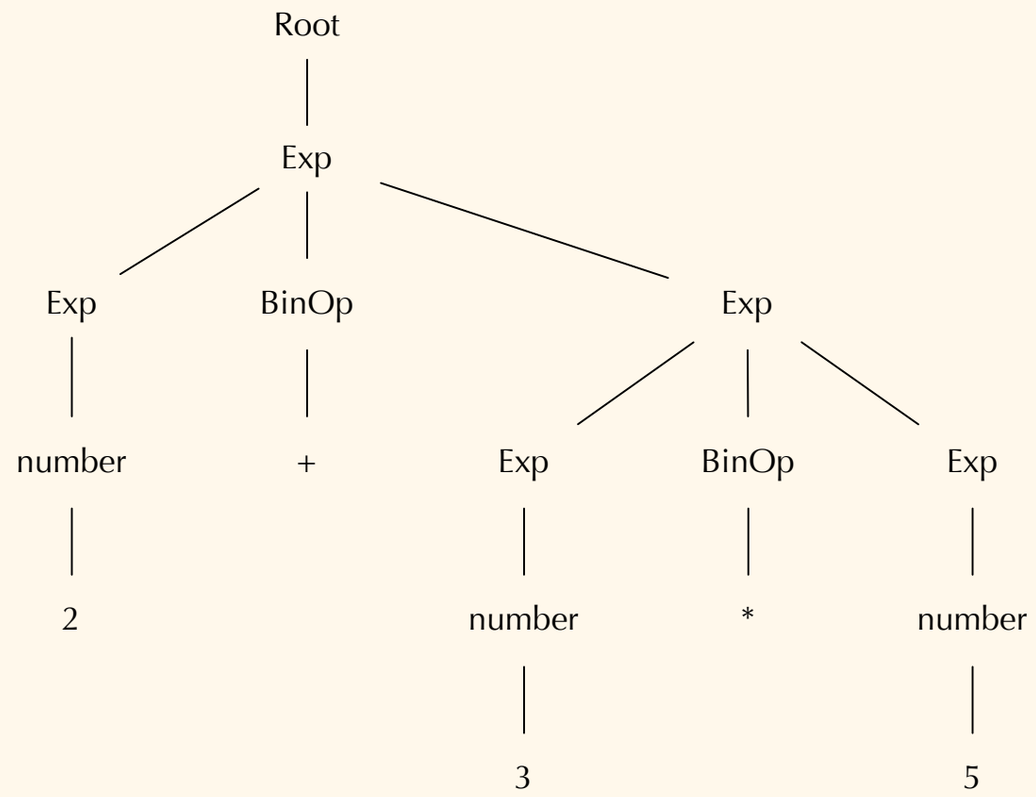
Finally, expressions can be wrapped in matched parentheses: Exp → “(“ Exp “)”

Root	→	Exp
number	→	“1” | “2” | “3” ... “0”
BinOp	→	“+” | “-” | “*” | “/”
UnOp	→	“+” | “-”
Exp	→	number
Exp	→	UnOp Exp
Exp	→	Exp BinOp Exp
Exp	→	“(“ Exp “)”

Terminal

Non-terminal

2 + 3 * 5



Can you spot the problem?

Useful aside:

As *finite-state* automata (FSA) are to regular grammars...

Push-down automata (PDA) are to context-free grammars.

All CFGs have an equivalent PDA.

PDAs are very similar to FSAs, but with one major difference: they have “memory” in the form of a stack.

Transition rules can specify stack actions and stack criteria as well as input symbols.

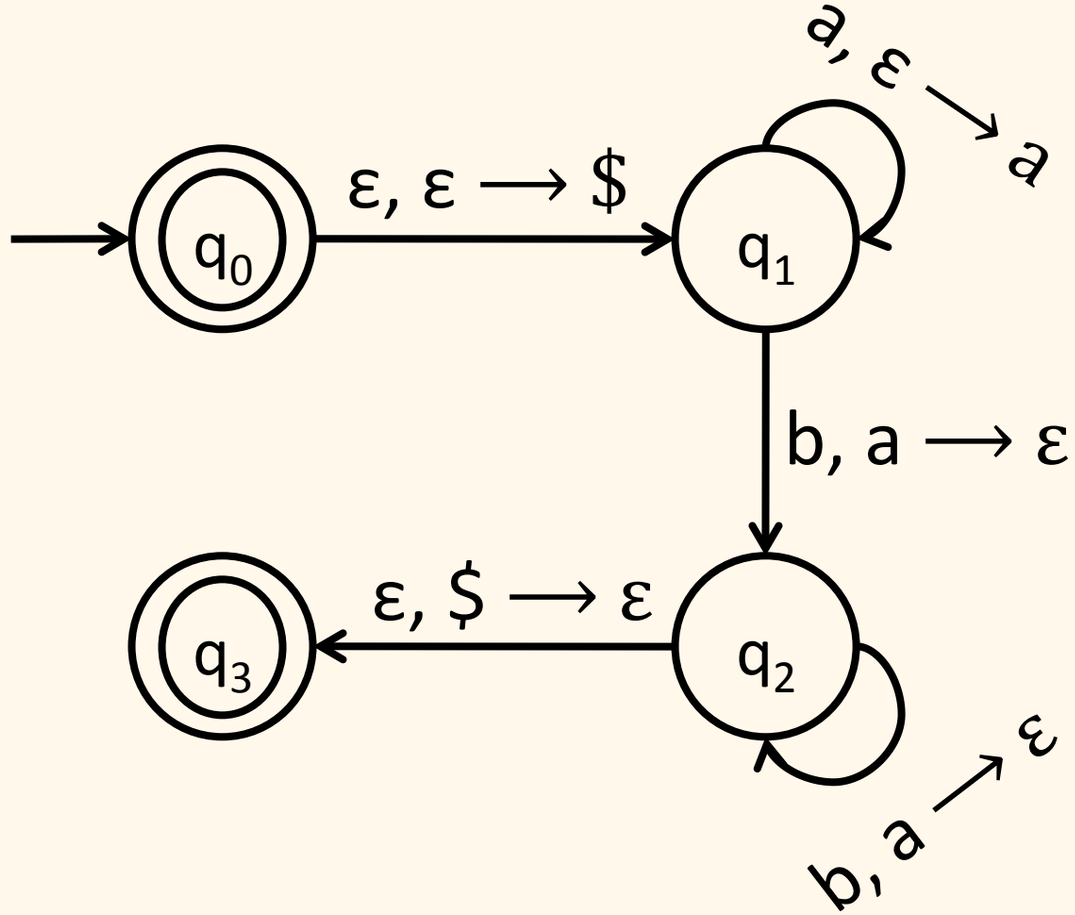
An example PDA for $a^n b^n$ for $n \geq 0$:

$a, \varepsilon \rightarrow a$

"next symbol must be 'a', and push 'a' on stack after transition."

$b, a \rightarrow \varepsilon$

"next symbol must be 'a', top of stack must be 'b', and pop top element off of stack after transition."



"read a's, push each on the stack; when the b's start, read each one and pop an 'a' off the stack each time; keep reading until we run out of b's or the stack is empty. If either one happens by itself, fail."

http://www-cs.ccny.cuny.edu/~vmitsou/304spring10/

Back to CFGs...

This is one way to represent them, and is what the book uses.

Root	→	Exp
number	→	“1” | “2” | “3” ... “0”
BinOp	→	“+” | “-” | “*” | “/”
UnOp	→	“+” | “-”
Exp	→	number
Exp	→	UnOp Exp
Exp	→	Exp BinOp Exp
Exp	→	“(“ Exp “)”

Another way uses a standardized notation, Backus-Naur Form:

```
<lhs> ::= <rhs> “terminal”
```

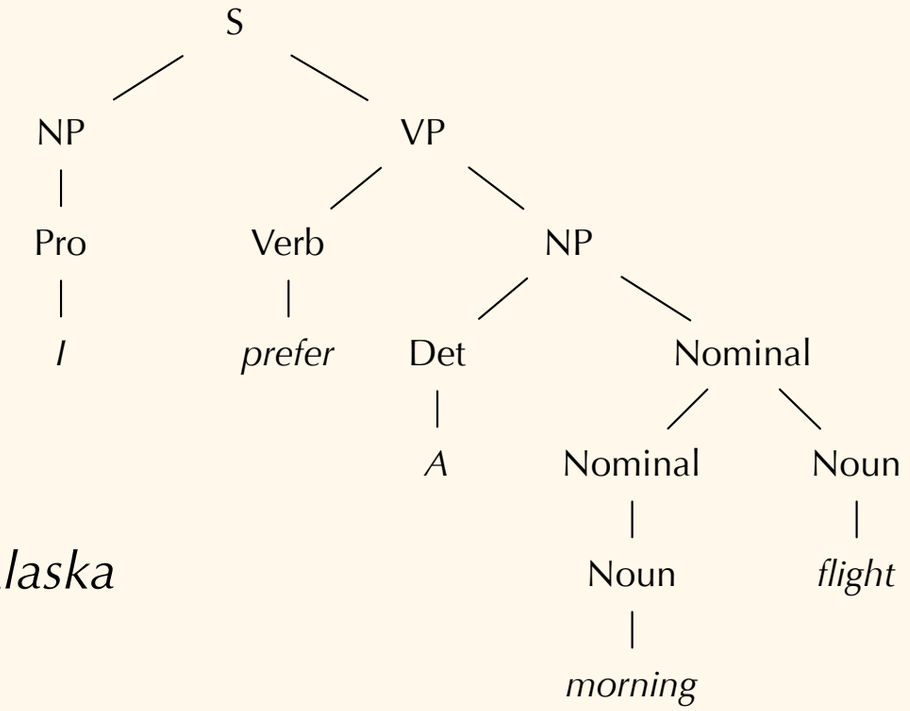
```
<Root> ::= <Exp>
<number> ::= “1” | “2” | ... “0”
...
<Exp> ::= <UnOp> <Exp>
...
```

Our arithmetic example is not very language-y...

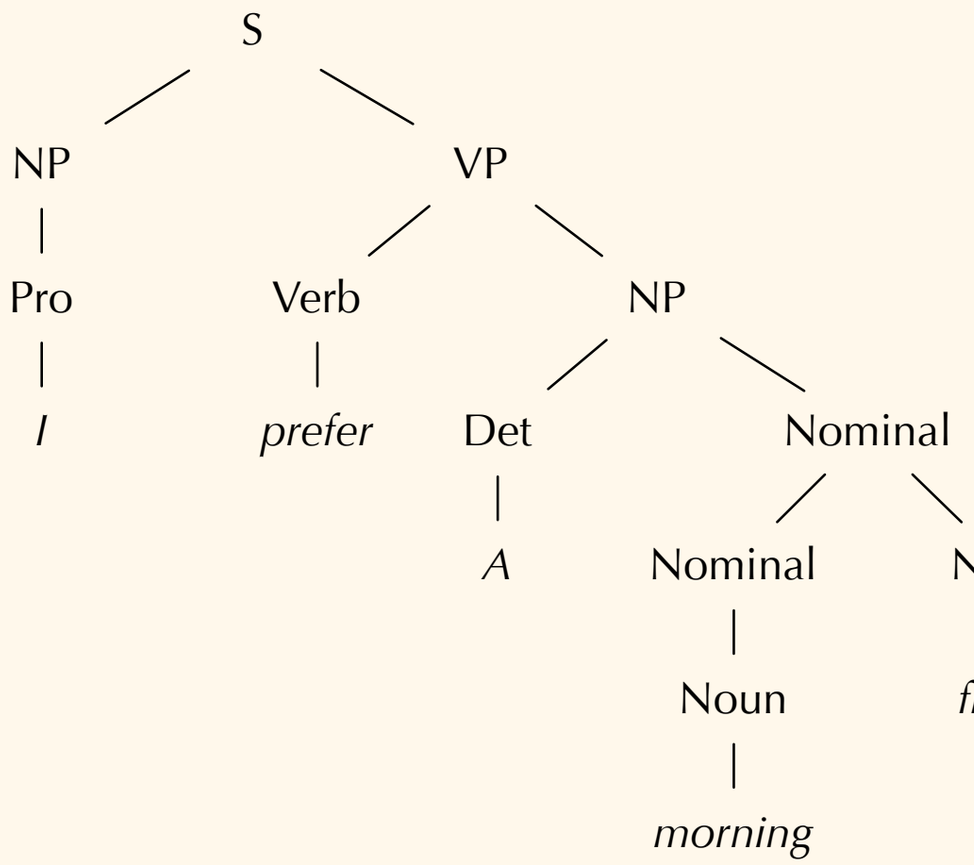
Let's try a more interesting example.

S	→	NP VP
NP	→	Pronoun | ProperNoun | Det Nominal
Nominal	→	Nominal Noun | Noun
VP	→	Verb | Verb NP | Verb NP PP | Verb PP
PP	→	Preposition NP
Noun	→	*flight* | *breeze* | *morning* | *trip* | ...
Verb	→	*is* | *prefer* | *like* | *need* | *want* | ...
Pronoun	→	*me* | *I* | *you* | *it*
ProperNoun	→	*Baltimore* | *Los Angeles* | *Chicago* | *United* | *Alaska*
Det	→	*the* | *a* | *an* | *this* | *these* | *that*
Preposition	→	*from* | *to* | *on* | *hear*

"I prefer a morning flight."



Producing a grammar from a tree is called "induction"...



S	→	NP VP
NP	→	Pro | Det Nominal
Nominal	→	Nominal Noun | Noun
VP	→	Verb NP
Noun	→	*flight* | *morning*
Verb	→	*prefer*
Pronoun	→	*I*
Det	→	*a*

If only we had some sort of data-*bank* of *trees* from which to induce grammars...

The Penn WSJ Treebank provides a standard set of non-terminals to use (this table only shows the major ones):

ADJP	Adjective Phrase	ADVP	Adverbial Phrase	CONJP	Conjunction Phrase
FRAG	Fragment	INTJ	Interjection	LST	List marker
NAC	Not a Constituent	NP	Noun Phrase	NX	Complex NP
PP	Prepositional Phrase	PRN	Parenthetical	PRT	Particle
QP	Quantifier Phrase	RRC	Reduced Relative Clause	S	Simple Clause
SBAR	Subordinate Clause	SBARQ	Subordinate Question Clause	SINV	Inverted Clause
SQ	Inverted Question	UCP	Unlike Coordinated Phrase	VP	Verb Phrase
WHADJP	Wh-adjective Phrase	WHAVP	Wh-adverb Phrase	WHNP	Wh-noun Phrase
WHPP	Wh-prepositional Phrase	X	Unknown		

This is in addition to the standard "pre-terminal" tags (PoS tags: NN, JJ, etc.).

One common criticism of PTB's tag set is that it is too "flat," and makes it hard to encode certain things.

One important extension to CFGs is the addition of probability: how "likely" is a certain production?

If we have a rule, e.g. S → NP VP, a PCFG would also tell us P(S → NP VP).

$$\begin{aligned} \mathrm{P}(\mathrm{S} \rightarrow \mathrm{NP\ VP}) &= \mathrm{P}(\mathit{rhs} = (\mathrm{NP\ VP}) \mid \mathit{lhs} = \mathrm{S}) \\ &= \mathrm{P}(\mathit{NP\ VP} \mid \mathit{S}) \end{aligned}$$

When inducing such a grammar, we keep track of how many times each LHS & RHS appear, and use these counts to compute probabilities.

Grammars can be "equivalent" in several different ways.

Two CFGs G and G' are *strongly equivalent* if they describe the same language, and they produce identical trees for strings (modulo some details about labels).

Two CFGs G and G' are *weakly equivalent* if they describe the same language.

Sometimes, we want to convert G into a weakly equivalent G' that might have useful properties.

One common transformation is into *Chomsky Normal Form* (CNF):

A grammar $G=(N, \Sigma, R, S)$ is in CNF if all productions in R are in one of two forms:

A → B C s.t. A, B, and C ∈ N (all are non-terminals)

A → *a* s.t. A ∈ N and *a* ∈ Σ (unary nonterm-term production)

Another is Griebach Normal Form (GNF):

A grammar $G=(N, \Sigma, R, S)$ is in GNF if all productions in R are in one of two forms:

A → *a X* s.t. A ∈ N, *a* ∈ Σ, and X ∈ N* No left-branching allowed!

CNF is named for Noam Chomsky... about whom we've heard a lot already...

GNF is named for Sheila Greibach, a noted pioneer in the field of automata theory, and discoverer of Greibach's Theorem.

Sheila Greibach
1939 – present

All CFGs have weakly equivalent CNF and GNF forms.

Another family of transformations: factorization.

When we factorize a rule, we are taking a single rule and factorizing it into multiple rules.

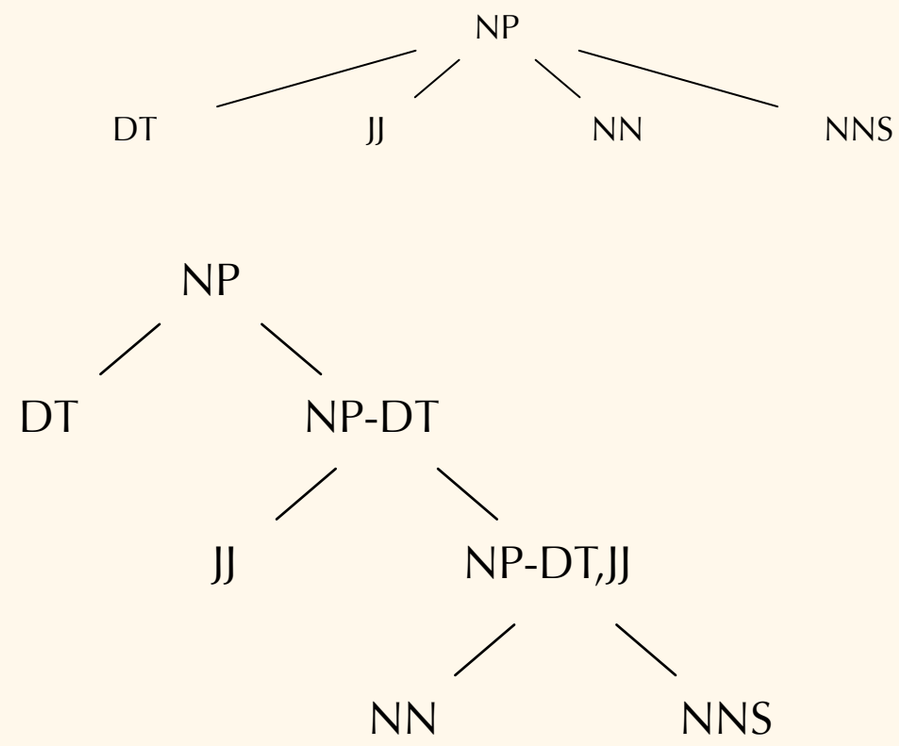
There are two main ways of doing this: from the left, or from the right.

NP → DT JJ NN NNS

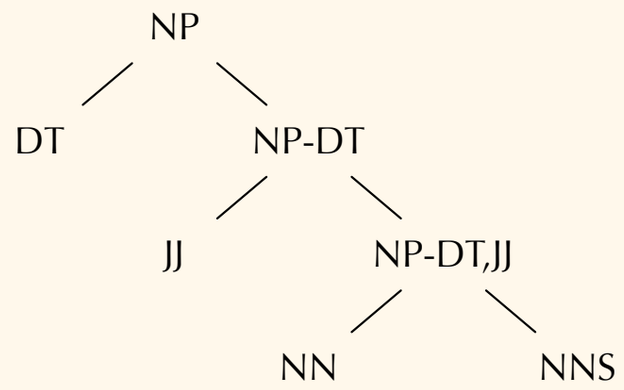
NP → DT NP-DT

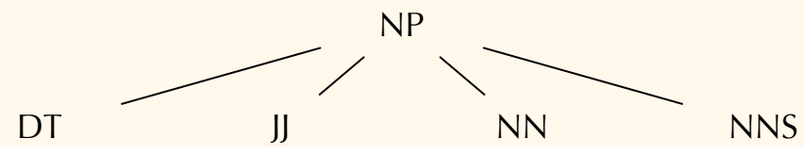
NP-DT → JJ NP-DT,JJ

NP-DT,JJ → NN NNS



There are two main ways of doing this: from the left, or from the right.



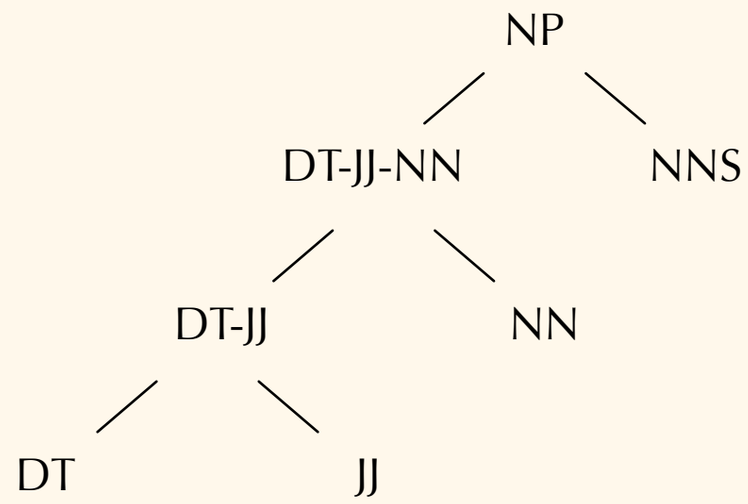


NP → DT JJ NN NNS

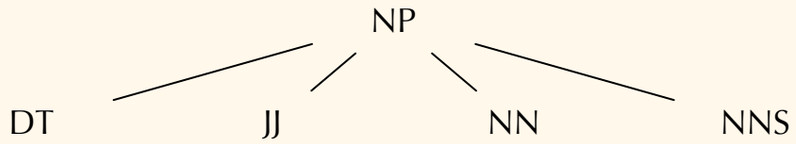
NP → DT-JJ-NN NNS

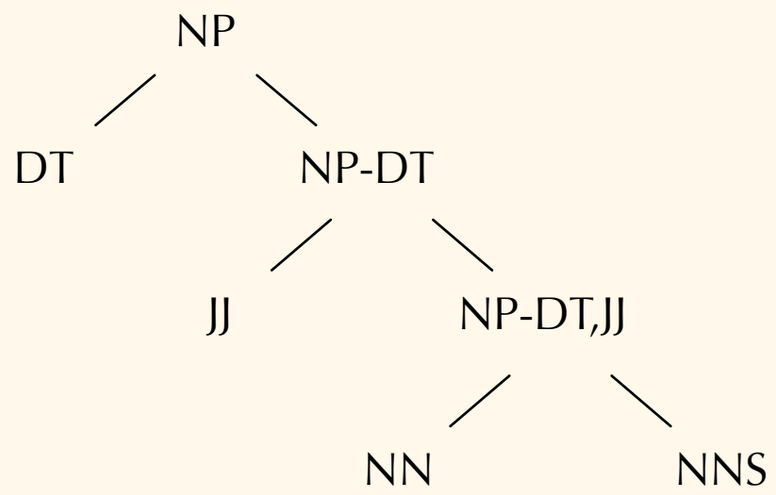
DP-JJ-NN → DT-JJ NN

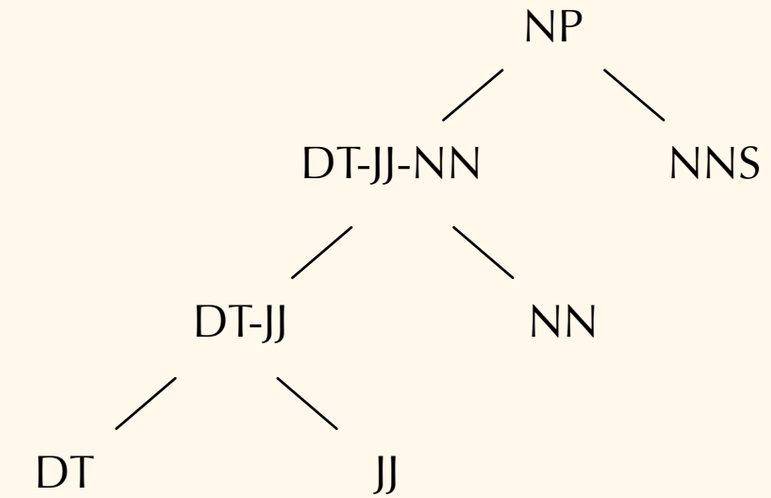
DT-JJ → DT JJ



These are two different ways of *binarizing* a grammar: all productions now have a maximum of two children.







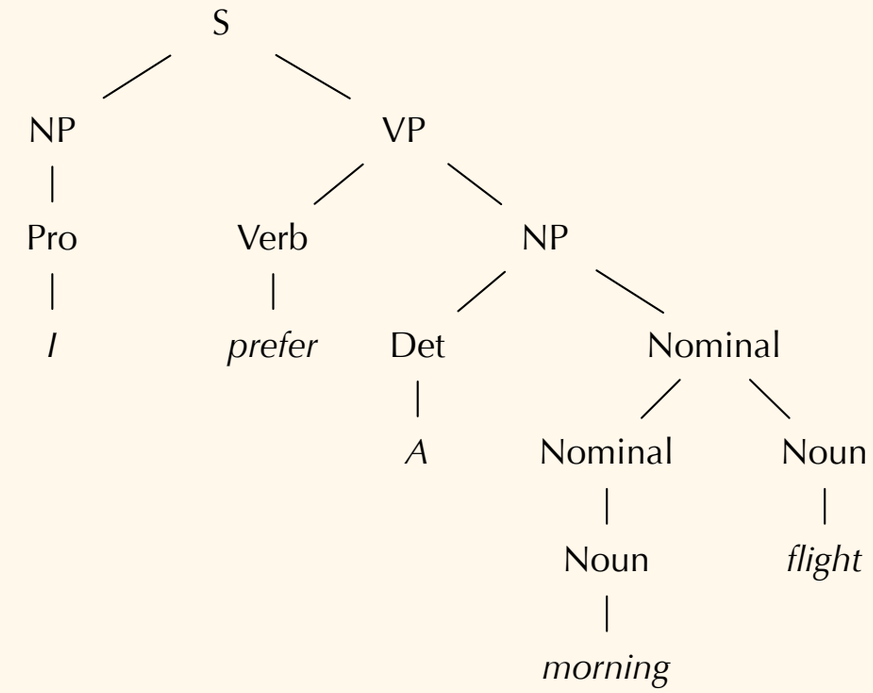
Besides being computationally useful, depending on how you label your new nodes, it may help with rule sparsity!

Going from a tree to a grammar is induction...

... going the other way (from a string to a tree, using a grammar) is parsing.

"I prefer a morning flight."

S	→	NP VP
NP	→	Pronoun | ProperNoun | Det Nominal
Nominal	→	Nominal Noun | Noun
VP	→	Verb | Verb NP | Verb NP PP | Verb PP
PP	→	Preposition NP
Noun	→	*flight* | *breeze* | *morning* | *trip* | ...
Verb	→	*is* | *prefer* | *like* | *need* | *want* | ...
Pronoun	→	*me* | *I* | *you* | *it*
ProperNoun	→	*Baltimore* | *Los Angeles* | *Chicago* | *United* | *Alaska*
Det	→	*the* | *a* | *an* | *this* | *these* | *that*
Preposition	→	*from* | *to* | *on* | *hear*

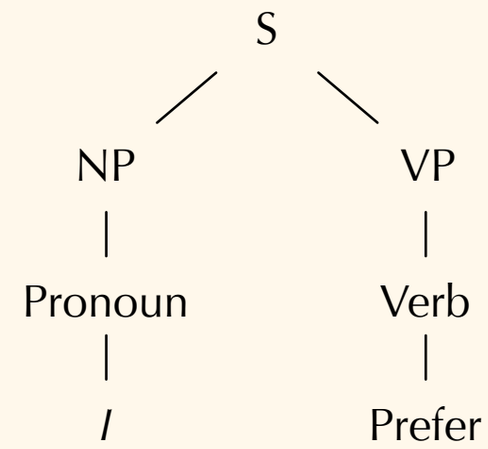


There are two general approaches to parsing: “top-down”, and “bottom-up.”

“Top-down” parsing starts at the top of the tree, and tries combinations of productions until it gets to the end.

“I prefer a morning flight.”

S	→	NP VP
NP	→	Pronoun | ProperNoun | Det Nominal
Nominal	→	Nominal Noun | Noun
VP	→	Verb | Verb NP | Verb NP PP | Verb PP
PP	→	Preposition NP
Noun	→	*flight* | *breeze* | *morning* | *trip* | ...
Verb	→	*is* | *prefer* | *like* | *need* | *want* | ...
Pronoun	→	*me* | *I* | *you* | *it*
ProperNoun	→	*Baltimore* | *Los Angeles* | *Chicago* | *United* | *Alaska*
Det	→	*the* | *a* | *an* | *this* | *these* | *that*
Preposition	→	*from* | *to* | *on* | *hear*

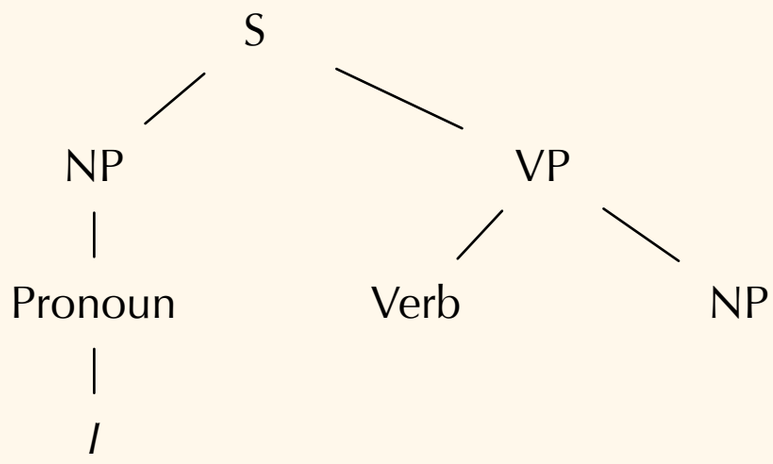


There are two general approaches to parsing: “top-down”, and “bottom-up.”

“Top-down” parsing starts at the top of the tree, and tries combinations of productions until it gets to the end.

“I prefer a morning flight.”

S	→	NP VP
NP	→	Pronoun | ProperNoun | Det Nominal
Nominal	→	Nominal Noun | Noun
VP	→	Verb | Verb NP | Verb NP PP | Verb PP
PP	→	Preposition NP
Noun	→	*flight* | *breeze* | *morning* | *trip* | ...
Verb	→	*is* | *prefer* | *like* | *need* | *want* | ...
Pronoun	→	*me* | *I* | *you* | *it*
ProperNoun	→	*Baltimore* | *Los Angeles* | *Chicago* | *United* | *Alaska*
Det	→	*the* | *a* | *an* | *this* | *these* | *that*
Preposition	→	*from* | *to* | *on* | *hear*



There are two general approaches to parsing: “top-down”, and “bottom-up.”

“Bottom-up” parsing does the opposite, and starts with the words themselves and works upwards:

“I prefer a morning flight.”

S	→	NP VP
NP	→	Pronoun | ProperNoun | Det Nominal
Nominal	→	Nominal Noun | Noun
VP	→	Verb | Verb NP | Verb NP PP | Verb PP
PP	→	Preposition NP
Noun	→	*flight* | *breeze* | *morning* | *trip* | ...
Verb	→	*is* | *prefer* | *like* | *need* | *want* | ...
Pronoun	→	*me* | *I* | *you* | *it*
ProperNoun	→	*Baltimore* | *Los Angeles* | *Chicago* | *United* | *Alaska*
Det	→	*the* | *a* | *an* | *this* | *these* | *that*
Preposition	→	*from* | *to* | *on* | *hear*

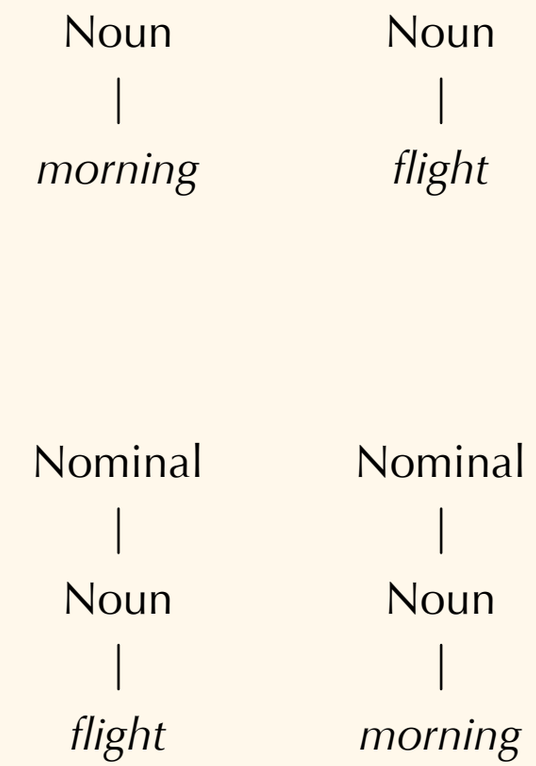
Noun	Noun
|	|
morning	*flight*

There are two general approaches to parsing: “top-down”, and “bottom-up.”

“Bottom-up” parsing does the opposite, and starts with the words themselves and works upwards:

“I prefer a morning flight.”

S	→	NP VP
NP	→	Pronoun | ProperNoun | Det Nominal
Nominal	→	Nominal Noun | Noun
VP	→	Verb | Verb NP | Verb NP PP | Verb PP
PP	→	Preposition NP
Noun	→	*flight* | *breeze* | *morning* | *trip* | ...
Verb	→	*is* | *prefer* | *like* | *need* | *want* | ...
Pronoun	→	*me* | *I* | *you* | *it*
ProperNoun	→	*Baltimore* | *Los Angeles* | *Chicago* | *United* | *Alaska*
Det	→	*the* | *a* | *an* | *this* | *these* | *that*
Preposition	→	*from* | *to* | *on* | *hear*



Top-down parsing:

Disadvantage: potential for lots of backtracking.

Advantage: doesn’t waste time on trees that won’t root.

Bottom-up parsing:

Disadvantage: many possible trees will have to be abandoned, because they won’t root.

Advantage: simpler, less egregious backtracking.

We will discuss specific parsing algorithms in detail next time...