Context-free Grammar

and

Backtrack Parsing

Martin Kay

Stanford University

and University of the Saarland

Problems with Regular Language

Is English a regular language?

- Bad question! We do not even know what English is!
  - Two eggs and bacon make(s) a big breakfast
  - Can you slide me the salt?
  - He didn't ought to do that
- But—No!
  - I put the wine you brought in the fridge
  - I put the wine you brought for Sandy in the fridge
  - Should we bring the wine you put in the fridge out now?
  - You said you thought nobody had the right to claim that they were above the law

Problems with Regular Language

You said you thought nobody had the right to claim that they were above the law

Problems with Regular Language

[You said you thought [nobody had the right [to claim that [they were above the law]]]]

Problems with Regular Language

Is English mophology a regular language?

Bad question! We do not even know what English morphology is!

- They sell collectables of all sorts
- This concerns unredecontaminatability
- This really is an untiable knot.

But—Probably!

- (Not sure about Swahili, though)

Context-free Grammar

Nonterminal symbols ~ grammatical categories

Terminal Symbols ~ words

Productions ~ (unordered) (rewriting) rules

Distinguished Symbol

Not all that important

- Terminals and nonterminals are disjoint
- Distinguished symbol

Context-free Grammar

Nonterminal symbols ~ grammatical categories

S, NP, VP, Det, N, V

Terminal Symbols ~ words

the, dog, cat, chased

Productions ~ (unordered) (rewriting) rules

S → NP VP	Det → the
NP → Det N	N → dog
VP → V NP	N → cat
	V → chased

Distinguished Symbol

S

Context-free Grammar

Nonterminal symbols ~ grammatical categories

S, NP, VP, Det, N, V

Terminal Symbols ~ words

the, dog, cat, chased

Productions ~ (unordered) (rewriting) rules

S → NP VP	Det → the
NP → Det N	N → dog
VP → V NP	N → cat
	V → chased

Distinguished Symbol

S

Context-free Grammar

Nonterminal symbols ~ grammatical categories

S, NP, VP, Det, N, V

Terminal Symbols ~ words

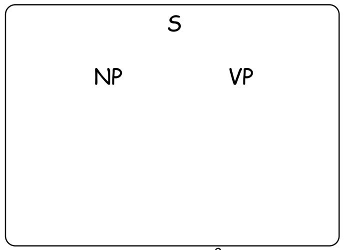
the, dog, cat, chased

Productions ~ (unordered) (rewriting) rules

S → NP VP	Det → the
NP → Det N	N → dog
VP → V NP	N → cat
	V → chased

Distinguished Symbol

S



Context-free Grammar

Nonterminal symbols ~ grammatical categories

S, NP, VP, Det, N, V

Terminal Symbols ~ words

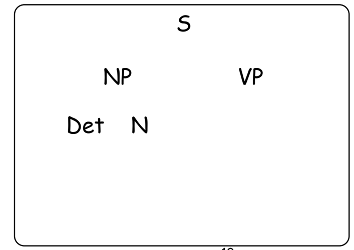
the, dog, cat, chased

Productions ~ (unordered) (rewriting) rules

S → NP VP	Det → the
NP → Det N	N → dog
VP → V NP	N → cat
	V → chased

Distinguished Symbol

S



Context-free Grammar

Nonterminal symbols ~ grammatical categories

S, NP, VP, Det, N, V

Terminal Symbols ~ words

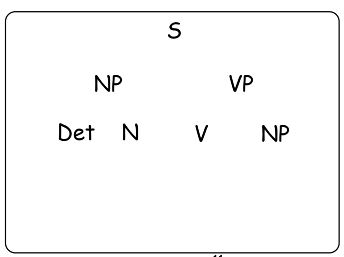
the, dog, cat, chased

Productions ~ (unordered) (rewriting) rules

S → NP VP	Det → the
NP → Det N	N → dog
VP → V NP	N → cat
	V → chased

Distinguished Symbol

S



Context-free Grammar

Nonterminal symbols ~ grammatical categories

S, NP, VP, Det, N, V

Terminal Symbols ~ words

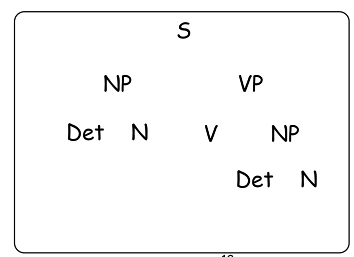
the, dog, cat, chased

Productions ~ (unordered) (rewriting) rules

S → NP VP	Det → the
NP → Det N	N → dog
VP → V NP	N → cat
	V → chased

Distinguished Symbol

S



Context-free Grammar

Nonterminal symbols ~ grammatical categories

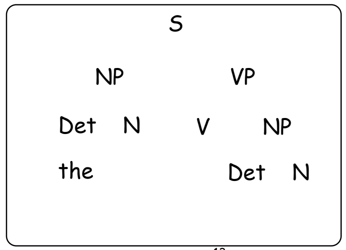
S, NP, VP, Det, N, V

Terminal Symbols ~ words

the, dog, cat, chased

Productions ~ (unordered) (rewriting) rules

S → NP VP	Det → the
NP → Det N	N → dog
VP → V NP	N → cat
	V → chased



Distinguished Symbol

S

Context-free Grammar

Nonterminal symbols ~ grammatical categories

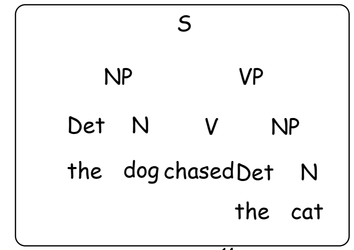
S, NP, VP, Det, N, V

Terminal Symbols ~ words

the, dog, cat, chased

Productions ~ (unordered) (rewriting) rules

S → NP VP	Det → the
NP → Det N	N → dog
VP → V NP	N → cat
	V → chased



Distinguished Symbol

S

Context-free Grammar

Nonterminal symbols ~ grammatical categories

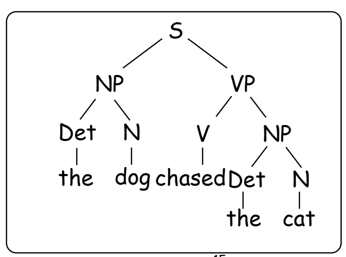
S, NP, VP, Det, N, V

Terminal Symbols ~ words

the, dog, cat, chased

Productions ~ (unordered) (rewriting) rules

S → NP VP	Det → the
NP → Det N	N → dog
VP → V NP	N → cat
	V → chased



Distinguished Symbol

S

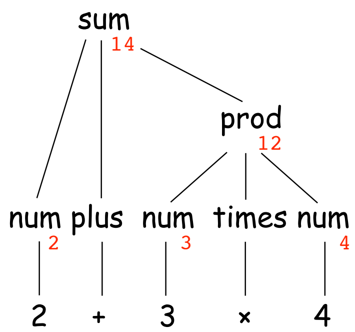
Context-free Grammar

- **Defines a language (set of strings)**

  <u>and</u>

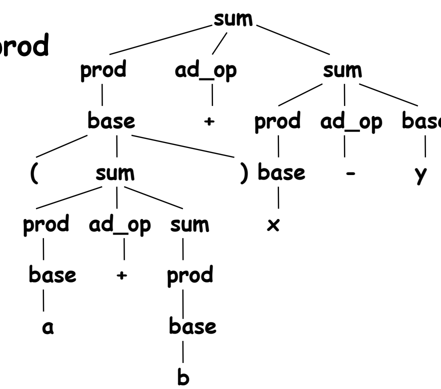
- **Defines a corresponding set of structures**

Structure and Semantics



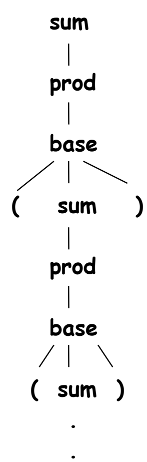
Aithmetic Expressions

sum ⇒ prod add_op sum
sum ⇒ prod
prod ⇒ base mul_op prod
prod ⇒ base
base ⇒ a | b | c ...
base ⇒ (sum)
ad_op ⇒ +
ad_op ⇒ -
mul_op ⇒ *
mul_op ⇒ /



Aithmetic Expressions

sum ⇒ prod add_op sum
sum ⇒ prod
prod ⇒ base mul_op prod
prod ⇒ base
base ⇒ a | b | c ...
base ⇒ (sum)
ad_op ⇒ +
ad_op ⇒ -
mul_op ⇒ *
mul_op ⇒ /

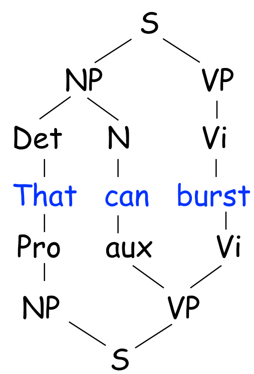


Embedding

Arbitrarily embedded parentheses are not possible in a regular language because the number of open parentheses that still requiring matching close parentheses can be greater than the number of states in the automaton, whatever that number is. Hence, by the pumping lemma, an automaton with only a finite number of states cannot characterize the language.

Ambiguity

More than one structure can correspond to a single string



Lots of Problems!

That it rained slept soundly.

Mary told the boy Mary

John told that it rained me

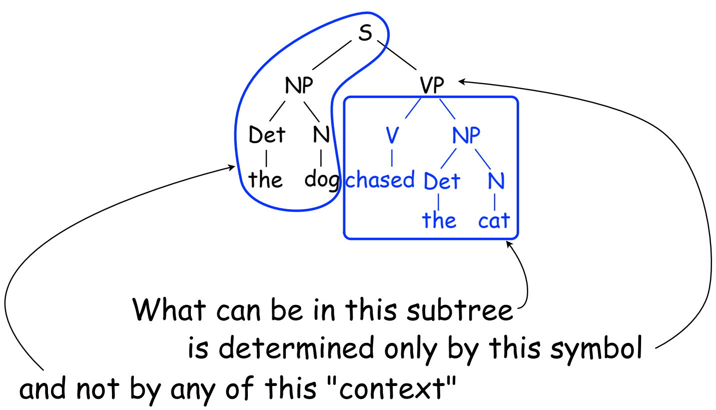
That it rained were certainly true.

His arguments convinces me.

Me told Mary it.

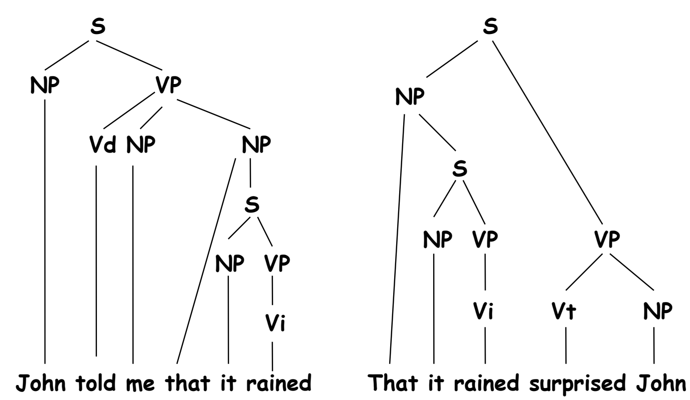
For the moment, just forget the problems

Why "Context-free"?

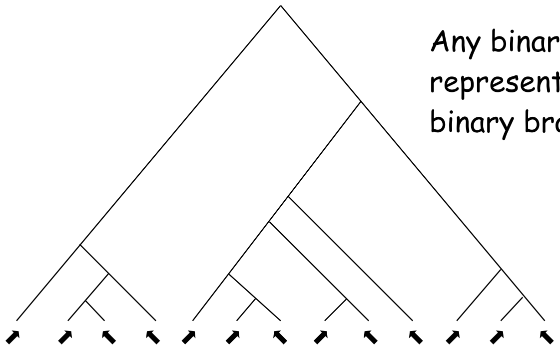


Convenience

You can define something once, and then use it in many places in the grammar



How many (binary) trees with n terminals?



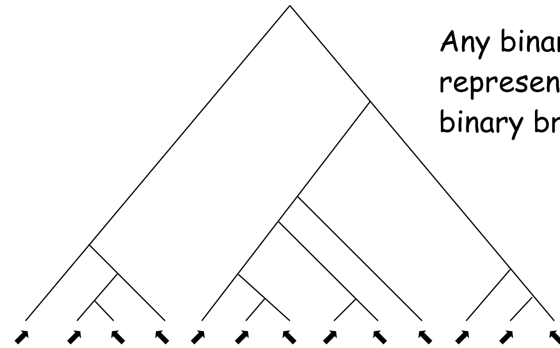
1 1 1 0 0 1 1 0 1 0 0 1 1 0

Any binary number represents a set of binary branching trees

2^{n-2}

Exponential

How many (binary) trees with n terminals?



1 1 1 0 0 1 1 0 1 0 0 1 1 0

Any binary number represents a set of binary branching trees

2^{n-2}

Exponential

Catalan Numbers

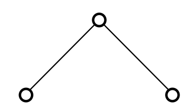
1 terminal

Terminals	Trees
1	1

Catalan Numbers

2 terminals

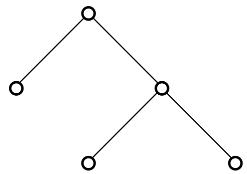
Terminals	Trees
1	1
2	1



Catalan Numbers

3 terminals

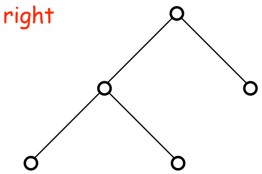
	Terminals	Trees	
left ➡	1	1	
	2	1	⬅ right



Catalan Numbers

3 terminals

	Terminals	Trees	
	1	1	⬅ right
left ➡	2	1	

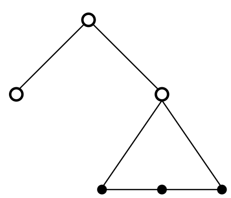


Catalan Numbers

4 terminals

	Terminals	Trees	
left ➡	1	1	
	2	1	
	3	2	⬅ right

$1 \times 2 = 2$



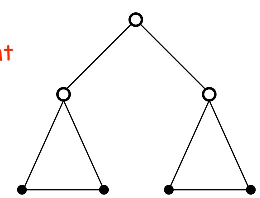
Catalan Numbers

4 terminals

	Terminals	Trees	
	1	1	
left ➡	2	1	⬅ right
	3	2	

$1 \times 2 = 2$

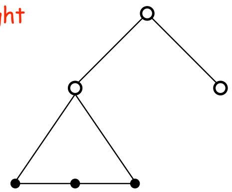
$1 \times 1 = 1$



Catalan Numbers

4 terminals

	Terminals	Trees	
	1	1	⬅ right
	2	1	
left ➡	3	2	



$1 \times 2 = 2$
$1 \times 1 = 1$
$2 \times 1 = 2$

Catalan Numbers

4 terminals

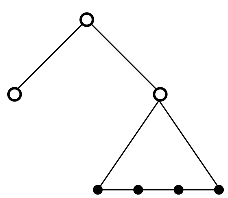
	Terminals	Trees	
	1	1	⬅ right
	2	1	
left ➡	3	2	
	4	5	

$1 \times 2 = 2$
$1 \times 1 = 1$
$\underline{2 \times 1 = 2}$
5

Catalan Numbers

5 terminals

	Terminals	Trees	
left ➡	1	1	
	2	1	
	3	2	
	4	5	⬅ right

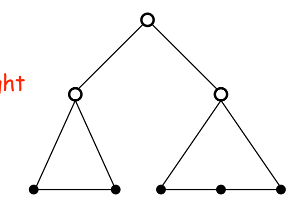


$1 \times 5 = 5$

Catalan Numbers

4 terminals

	Terminals	Trees	
	1	1	
left ➡	2	1	
	3	2	⬅ right
	4	5	

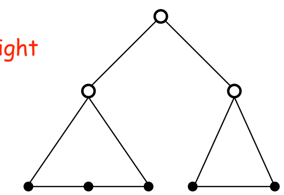


$1 \times 5 = 5$
$1 \times 2 = 2$

Catalan Numbers

4 terminals

	Terminals	Trees	
	1	1	
	2	1	⬅ right
left ➡	3	2	
	4	5	

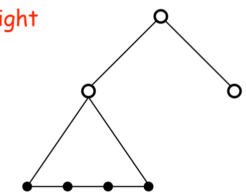


1 × 5 = 5
1 × 2 = 2
2 × 1 = 2

Catalan Numbers

4 terminals

	Terminals	Trees	
	1	1	⬅ right
	2	1	
	3	2	
left ➡	4	5	



1 × 5 = 5
1 × 2 = 2
2 × 1 = 2
5 × 1 = 5
14

Catalan Numbers

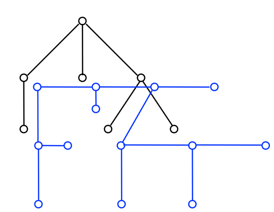
Terminals	Trees
1	1
2	1
3	2 (= 1*1 + 1*1)
4	5 (= 1*2 + 1*1 + 2*1)
5	14 (= 1*5 + 1*2 + 2*1 + 5*1)
6	42 (= 1*14 + 1*5 + 2*2 + 5*1 + 14*1)
...	

$$Cat(n) = \frac{\binom{2n-1}{n}}{2n-1}$$

Catalan Numbers

The number of binary trees with n terminals

The number of arbitrarily branching trees with n nodes.

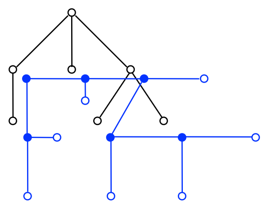


Any tree can be encoded in a binary tree with the same number of terminals as the original tree has nodes.

Catalan Numbers

There are n nonterminals in the binary tree for the n nodes in the original tree, except for the root.

There are n+1 terminals in a binary tree with n nonterminals. □



Let there be:

a arcs,
n nonterminals, and
t terminals

There are two arcs downward from each nonterminal, so

$a = 2n$

There is one arc upward from each terminal and nonterminal except the root, so

$a = n + t - 1$

Ergo

$n = t - 1$

Analyzing Sentences

- Recognition: determining if a given string is a member of the language defined by the grammar
- Parsing: Determining what structure(s) a grammar assigns to a string
- Note: Parsing encompasses recognition.

Top-down

❶ the ❷ dog ❸ chased ❹ the ❺ cat ❻

S → NP VP	Det → the
NP → Det N	N → dog
VP → V NP	N → cat
	V → chased

If there is a sentence between ❶ and ❻, then there must be an NP between ❶ and some ⓧ and a VP between ⓧ and ❻.

If there is an NP between ❶ and ⓧ, then there must be a Det between ❶ and some ⓨ and a N between ⓨ and ⓧ.

More procedurally

❶ the ❷ dog ❸ chased ❹ the ❺ cat ❻

S → NP VP	Det → the
NP → Det N	N → dog
VP → V NP	N → cat
	V → chased

If there is a sentence between ❶ and ❻, look for an NP beginning at ❶ and and let ⓧ be where it ends.

Look for a VP beginning at ⓧ and ending at ❻.

Definite-clause grammar

```
s(A, C) :- np(A, B), vp(B, C).
vp(A, C) :- v(A, B), np(B, C).
np(A, C) :- det(A, B), n(B, C).
det(A, B) :- the(A, B).
n(A, B) :- dog(A, B).
n(A, B) :- cat(A, B).
v(A, B) :- chased(A, B).

the(1, 2).
dog(2, 3).
chased(3, 4).
the(4, 5).
cat(5, 6).
```

```
| ?- s(X, Y).
X = 1,
Y = 6 ?
| ?- np(X, Y).
X = 1,
Y = 3 ? ;
X = 4,
Y = 6 ? ;
no
| ?-
```


Top-down

These do not have to be numbers. They could be ... anything! Just so they are different from one another.

❶ the ❷ dog ❸ chased ❹ the ❺ cat ❻

S → NP VP	Det → the
NP → Det N	N → dog
VP → V NP	N → cat
	V → chased

Definite-clause grammar

```
s(A, C) :- np(A, B), vp(B, C).
vp(A, C) :- v(A, B), np(B, C).
np(A, C) :- det(A, B), n(B, C).
det(A, B) :- the(A, B).
n(A, B) :- dog(A, B).
n(A, B) :- cat(A, B).
v(A, B) :- chased(A, B).

the(start, the1).
dog(the1, dog).
chased(dog, chased)
the(chased, the2).
cat(the2, end).
```

```
| ?- s(X, Y).
X = start,
Y = end ? ;
no
| ?- np(X, Y).
X = start,
Y = dog ? ;
X = chased,
Y = end ? ;
no
```

The sentence has to be part of the program!

Top-down

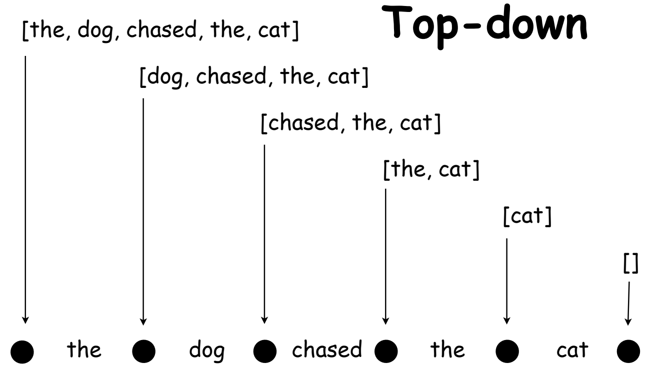
These do not have to be numbers. They could be ... anything! Just so they are different from one another.

❶ the ❷ dog ❸ chased ❹ the ❺ cat ❻

S → NP VP	Det → the
NP → Det N	N → dog
VP → V NP	N → cat
	V → chased

So let's use tails of the string itself!

Top-down



If there is a sentence between [the, dog, chased, the, cat] and [], then there must be an NP between [the, dog, chased, the, cat] and some X and a VP between that X and [].

Renamed String Positions

```
s(A, C) :- np(A, B), vp(B, C).
vp(A, C) :- v(A, B), np(B, C).
np(A, C) :- det(A, B), n(B, C).
det([the | X], X).
n([dog | X], X).
n([cat | X], X).
v([chased | X], X).
```

There is a noun between positions A and B if A is a string beginning with the word "dog" and B is the remainder of that string.

Renamed String Positions

```
s(A, C) :- np(A, B), vp(B, C).
vp(A, C) :- v(A, B), np(B, C).
np(A, C) :- det(A, B), n(B, C).
det([the | X], X).
n([dog | X], X).
n([cat | X], X).
v([chased | X], X).
```

```
| ?- s([the,dog,chased,the,cat], []).
yes
```


But its only a recognizer!

```
s(A, C, s(NP, VP)) :- np(A, B, NP), vp(B, C, VP).
vp(A, C, vp(V, NP)) :- v(A, B, V), np(B, C, NP).
np(A, C, np(Det, N)) :- det(A, B, Det), n(B, C, N).
det([the | X], X, det(the)).
n([dog | X], X, n(dog)).
n([cat | X], X, n(cat)).
v([chased | X], X, v(chased)).
```

```
| ?- s([the,dog,chased,the,cat], [], S).
S = s(np(det(the),n(dog)),vp(v(chased),np(det(the),n
(cat)))) ?
```

Caveat Parsor

```
s(A, C, s(NP, VP)) :- np(A, B, NP), vp(B, C, VP).
vp(A, C, vp(V, NP)) :- v(A, B, V), np(B, C, NP).
np(A, C, np(Det, N)) :- det(A, B, Det), n(B, C, N).
det([the | X], X, det(the)).
pp(A, C, pp(P, NP)) :- p(A, B, P), np(B, C, NP).
np(A, C, np(NP, PP)) :- np(A, B, NP), pp(B, C, PP).
vp(A, C, vp(VP, PP)) :- vp(A, B, VP), pp(B, C, PP).
n([dog | X], X, n(dog)).
n([cat | X], X, n(cat)).
n
v
p([round | X], X, p(round)).
```

```
| ?- s([the,dog,chased,the,cat], [], S).
S = s(np(det(the),n(dog)),vp(v(chased),np(det(the),n(cat)))) ? ;
! Resource error: insufficient memory
```


Left Recursion

NP --> NP PP

To find an NP, first find an NP !!!

NP --> Det N

Det --> NP apostrophe-s

Left Recursion

A grammar t contains instances of left recursion c ced by one with no left recursion whic

- describes the same
- assigns the same structu he sentences.

COMING SOON!

For the moment, we will just look for a different parsing algorithm that does not have the problem.

Also, for a while, we will concentrate on recognizers because we know how to turn them into parsers.

Stack-String Models

Stack	String
[s/[np, vp]]	[the, dog, chased, ...]

Access only from the middle

Stack-String Models

Stack	String
[s/[np, vp]]	[the, dog, chased, ...]
[s/[np,vp],np/[det,n]]	[the, dog, chased, ...]
[s/[np,vp],np/[det,n]],det/[the]]	[the, dog, chased, ...]

Pattern 1: Push

Stack-String Models

Stack	String
[s/[np,vp],np/[det,n]],det/[the]	[the, dog, chased, ...]

Pattern 4: Complete rule

Stack	String
[s/[np,vp],np/[det,n]]	[det, dog, chased, ...]

Pattern 3: Pop

Stack	String
[s/[np,vp],np/[n]]	[dog, chased, the, cat]

0.	[s/[np,vp]]	[the,dog,chased,the,cat]	0
1.	[s/[np,vp],np/[det,n]]	[the,dog,chased,the,cat]	1
2.	[s/[np,vp],np/[det,n],det/[the]]	[the,dog,chased,the,cat]	1
3.	[s/[np,vp],np/[det,n]]	[det,dog,chased,the,cat]	4
4.	[s/[np,vp],np/[n]]	[dog,chased,the,cat]	3
5.	[s/[np,vp],np/[n],n/[dog]]	[dog,chased,the,cat]	1
6.	[s/[np,vp],np/[n]]	[n,chased,the,cat]	4
7.	[s/[np,vp]]	[np,chased,the,cat]	4
8.	[s/[vp]]	[chased,the,cat]	3
9.	[s/[vp],vp/[v,np]]	[chased,the,cat]	1
10.	[s/[vp],vp/[v,np],v/[chased]]	[chased,the,cat]	1
11.	[s/[vp],vp/[v,np]]	[v,the,cat]	4
12.	[s/[vp],vp/[np]]	[the,cat]	3
13.	[s/[vp],vp/[np],np/[det,n]]	[the,cat]	1
14.	[s/[vp],vp/[np],np/[det,n],det/[the]]	[the,cat]	1
15.	[s/[vp],vp/[np],np/[det,n]]	[det,cat]	4
16.	[s/[vp],vp/[np],np/[n]]	[cat]	3
17.	[s/[vp],vp/[np],np/[n],n/[cat]]	[cat]	
18.	[s/[vp],vp/[np],np/[n]]	[n]	
19.	[s/[vp],vp/[np]]	[np]	
20.	[s/[vp]]	[vp]	
21.	[]	[s]	4

Top down recognizer

1: Push

For convenience, write
A → B C ...

as
A/[B, C ...]

Looking for Q

Put it on the stack

A rule that could find Q

```
move([P/[Q | R | Stack], String,                ❶
     [P/[Q | R, Q/S] | Stack], String)
  :- rule(Q/S).
move([P/[Q | R | Stack], [S | String],          ❷
     [P/[Q | R | Stack], [Q | String])
  :- rule(Q/[S]).
move([P/[Q , R | S | Stack], [Q | String],      ❸
     [P/[R | S | Stack], String).
move([P/[Q] | Stack, [Q | String],              ❹
     Stack, [P | String]).
```


2: Reduce with rule

Looking for Q

This would satisfy the rule

So replace it

A rule that could find Q

```
move([P/[Q | R | Stack], String,                ❶
     [Q/S, P/[Q | R | Stack], String)
  :- rule(Q/S).
move([P/[Q | R | Stack], [S | String],          ❷
     [P/[Q | R | Stack], [Q | String])
  :- rule(Q/[S]).
move([P/[Q , R | S | Stack], [Q | String],      ❸
     [P/[R | S | Stack], String).
move([P/[Q] | Stack], [Q | String],             ❹
     Stack, [P | String]).
```


3: Pop

Looking for Q

```
move([P/[Q | R] | Stack], String,               ❶
     [Q/S, P/[Q | R] | Stack], String)
  :- rule(Q/S).
move([P/[Q | R] | Stack], [S | String],         ❷
     [P/[Q | R] | Stack], [Q | String])
  :- rule(Q/[S]).
move([P/[Q , R | S] | Stack], [Q | String],     ❸
     [P/[R | S] | Stack], String).
move([P/[Q] | Stack], [Q | String],             ❹
     Stack, [P | String]).
```

Here it is

So delete both of them

4: Complete Rule

Looking for Q

```
move([P/[Q | R] | Stack], String,               ❶
     [Q/S, P/[Q | R] | Stack], String)
  :- rule(Q/S).
move([P/[Q | R] | Stack], [S | String],         ❷
     [P/[Q | R] | Stack], [Q | String])
  :- rule(Q/[S]).
move([P/[Q , R | S] | Stack], [Q | String],     ❸
     [P/[R | S] | Stack], String).
move([P/[Q] | Stack], [Q | String],             ❹
     Stack, [P | String]).
```

This will satisfy the whole rule

So delete both of them

The Interpreter

```
parse(Stack0, String0) :-
      move(Stack0, String0, Stack, String),
      parse(Stack, String).
parse([], [_]).
```


Three Main Paradigms

- Top-down
- Bottom-up
  - Shift-reduce
  - Left-corner

We can manage with less than four cases

Top-down (again)

```
parse([], String, String).
parse([Goal | Goals], [Goal | String], Rest) :-
  parse(Goals, String, Rest).
parse([Goal | Goals], String0, Rest) :-
  rule(Goal, Rhs),
  parse(Rhs, String0, String1),
  parse(Goals, String1, Rest).
```

The empty set of goals is satisfied by the empty string, leaving everything else as the remainder

Top-down

```
parse([], String, String).
parse([Goal | Goals], [Goal | String], Rest) :-
  parse(Goals, String, Rest).
parse([Goal | Goals], String0, Rest) :-
  rule(Goal, Rhs),
  parse(RHS, String0, String1),
  parse(Goals, String1, Rest).
```

If the Goal is found at the beginning of the string, try to show that the remainder of the string satisfies the remainder of the goals.

Top-down

```
parse([], String, String).
parse([Goal | Goals], [Goal | String], Rest) :-
  parse(Goals, String, Rest).
parse([Goal | Goals], String0, Rest) :-
  rule(Goal, Rhs),
  parse(RHS, String0, String1),
  parse(Goals, String1, Rest).
```

If there is a rule that would replace the next goal by some more specific goals (RHS), then

Top-down

```
parse([], String, String).
parse([Goal | Goals], [Goal | String], Rest) :-
  parse(Goals, String, Rest).
parse([Goal | Goals], String0, Rest) :-
  rule(Goal, Rhs),
  parse(RHS, String0, String1),
  parse(Goals, String1, Rest).
```

If there is a rule that would replace the next goal by some more specific goals (RHS), then
try to satisfy these more specific goals, and then

Top-down

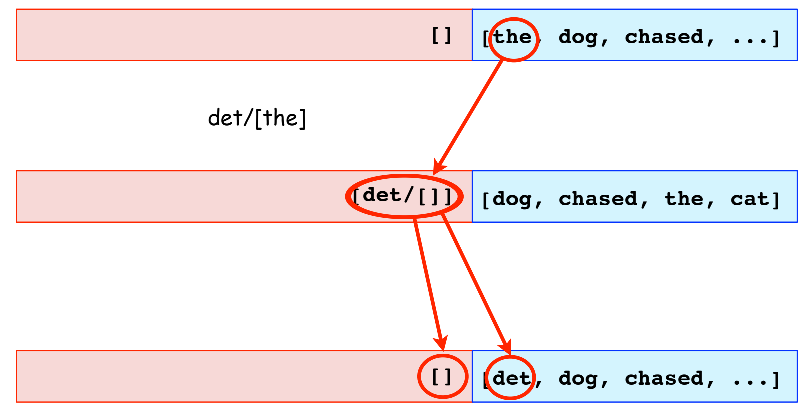
```
parse([], String, String).
parse([Goal | Goals], [Goal | String], Rest) :-
  parse(Goals, String, Rest).
parse([Goal | Goals], String0, Rest) :-
  rule(Goal, Rhs),
  parse(RHS, String0, String1),
  parse(Goals, String1, Rest).
```

If there is a rule that would replace the next goal by some more specific goals (RHS), then try to satisfy these more specific goals, and then
try to satisfy the rest of the original set of goals

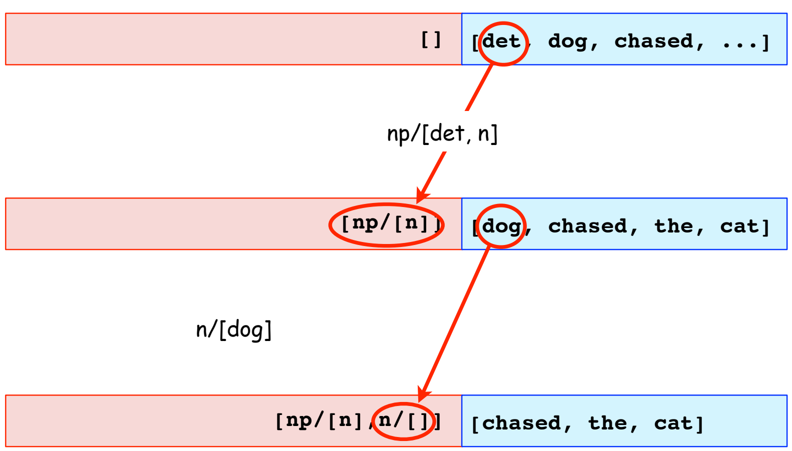
Backtracking

0.	[s/[np,vp]]	[the,dog,chased,the,cat]	0
1.	[np/[det,n],s/[np,vp]]	[the,dog,chased,the,cat]	1
2.	[det/[the],np/[det,n],s/[np,vp]]	[the,dog,chased,the,cat]	1
3.	[np/[det,n],s/[np,vp]]	[det,dog,chased,the,cat]	4
4.	[det/[the],np/[det,n],s/[np,vp]]	[det,dog,chased,the,cat]	1
4.	[np/[n],s/[np,vp]]	[dog,chased,the,cat]	3
5.	[n/[cat],np/[n],s/[np,vp]]	[dog,chased,the,cat]	1
5.	[n/[dog],np/[n],s/[np,vp]]	[dog,chased,the,cat]	1
6.	[np/[n],s/[np,vp]]	[n,chased,the,cat]	4
7.	[n/[cat],np/[n],s/[np,vp]]	[n,chased,the,cat]	1
7.	[n/[dog],np/[n],s/[np,vp]]	[n,chased,the,cat]	1
7.	[n/[house],np/[n],s/[np,vp]]	[n,chased,the,cat]	1
7.	[s/[np,vp]]	[np,chased,the,cat]	4
8.	[np/[det,n],s/[np,vp]]	[np,chased,the,cat]	1
...			

Bottom-up



Bottom-up



0.	[]	[the,dog,chased,the,cat]	0
1.	[det/[]]	[dog,chased,the,cat]	2
2.	[]	[det,dog,chased,the,cat]	1
3.	[np/[n]]	[dog,chased,the,cat]	2
4.	[np/[n],n/[]]	[chased,the,cat]	2
5.	[np/[n]]	[n,chased,the,cat]	1
6.	[np/[]]	[chased,the,cat]	3
7.	[]	[np,chased,the,cat]	1
8.	[s/[vp]]	[chased,the,cat]	2
9.	[s/[vp],v/[]]	[the,cat]	2
10.	[s/[vp]]	[v,the,cat]	1
11.	[s/[vp],vp/[np]]	[the,cat]	2
12.	[s/[vp],vp/[np],det/[]]	[cat]	2
13.	[s/[vp],vp/[np]]	[det,cat]	1
14.	[s/[vp],vp/[np],np/[n]]	[cat]	2
15.	[s/[vp],vp/[np],np/[n],n/[]]	[]	2
16.	[s/[vp],vp/[np],np/[n]]	[n]	1
17.	[s/[vp],vp/[np],np/[]]	[]	3
18.	[s/[vp],vp/[np]]	[np]	1
19.	[s/[vp],vp/[np],s/[vp]]	[]	2
19.	[s/[vp],vp/[]]	[]	3
20.	[s/[vp]]	[vp]	1
21.	[s/[]]	[]	3
22.	[]	[s]	1

Left Corner

This rule is satisfied

So put the result in the string

```
move([P/[] | Stack], String,
     Stack, [P |String]) :- !.
move(Stack, [Q | String],
     [P/R | Stack], String) :- rule(P/[Q | R]).
move([P/[Q | R] | Stack], [Q | String],
     [P/R | Stack], String).
```


Left Corner

Need to consume a Q

Here is a rule that could do it

```
move([P/[] | Stack], String,
     Stack, [P |String]) :- !.
move(Stack, [Q | String],
     [P/R | Stack], String) :- rule(P/[Q | R]).
move([P/[Q | R] | Stack], [Q | String],
     [P/R | Stack], String).
```

No more Q

Try to satisfy rest of the rule

Left Corner

Need to consume a Q

Here is one

```
move([P/[] | Stack], String,
     Stack, [P |String]) :- !.
move(Stack, [Q | String],
     [P/R | Stack], String) :- rule(P/[Q | R]).
move([P/[Q | R] | Stack], [Q | String],
     [P/R | Stack], String).
```

No more Q

Left-corner

Same as for top-down.

```
parse([], String, String).
parse([Goal | Goals], [Goal | String], Rest) :-
  parse(Goals, String, Rest).
parse(Goals, [Item | String0], Rest) :-
  rule(P, [Item | Rhs]),
  parse(RHS, String0, String1),
  parse(Goals, [P | String1], Rest).
```

Left-corner

Find a rule that could form a phrase beginning with the leftmost item in the string (the left corner).

```
parse([], String, String).
parse([Goal | Goals], [Goal | String], Rest) :-
  parse(Goals, String, Rest).
parse(Goals, [Item | String0], Rest) :-
  rule(P, [Item | Rhs]),
  parse(RHS, String0, String1),
  parse(Goals, [P | String1], Rest).
```


Left-corner

Find a rule that could form a phrase beginning with the leftmost item in the string (the left corner).
With the right-hand side of the rule as a sequence of goals, try to find such a phrase.

```
parse([], String, String).
parse([Goal | Goals], [Goal | String], Rest) :-
  parse(Goals, String, Rest).
parse(Goals, [Item | String0], Rest) :-
  rule(P, [Item | Rhs]),
  parse(RHS, String0, String1),
  parse(Goals, [P | String1], Rest).
```


Left-corner

Find a rule that could form a phrase beginning with the leftmost item in the string (the left corner).
With the right-hand side of the rule as a sequence of goals, try to find such a phrase.
Replace the phrase with its category, and continue trying to satisfy the original goals.

```
parse([], String, String).
parse([Goal | Goals], [Goal | String], Rest) :-
  parse(Goals, String, Rest).
parse(Goals, [Item | String0], Rest) :-
  rule(P, [Item | RHS]),
  parse(RHS, String0, String1),
  parse(Goals, [P | String1], Rest).
```


Top-down vs. Left-corner

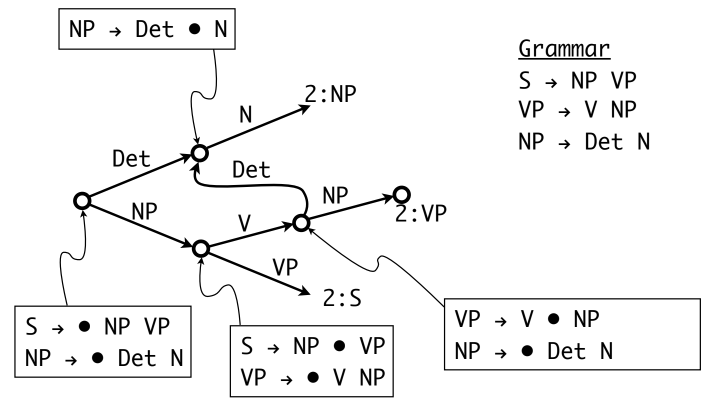
```
parse([], String, String).
parse([Goal | Goals], [Goal | String], Rest) :-
  parse(Goals, String, Rest).
parse([Goal | Goals], String0, Rest) :-
  rule(Goal, RHS),
  parse(RHS, String0, String1),
  parse(Goals, String1, Rest).
```

```
parse([], String, String).
parse([Goal | Goals], [Goal | String], Rest) :-
  parse(Goals, String, Rest).
parse(Goals, [Item | String0], Rest) :-
  rule(P, [Item | RHS]),
  parse(RHS, String0, String1),
  parse(Goals, [P | String1], Rest).
```

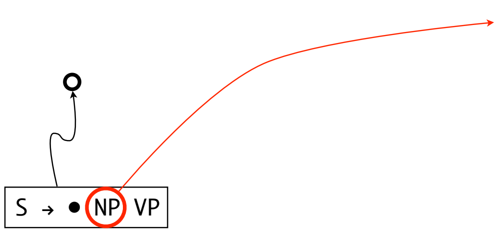
Shift reduce

	the dog chased the cat
det	dog chased the cat
det n	chased the cat
np	chased the cat
np v	the cat
np v det	cat
np v det n	
np v np	
np vp	
s	

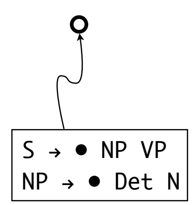
Finite-state guide



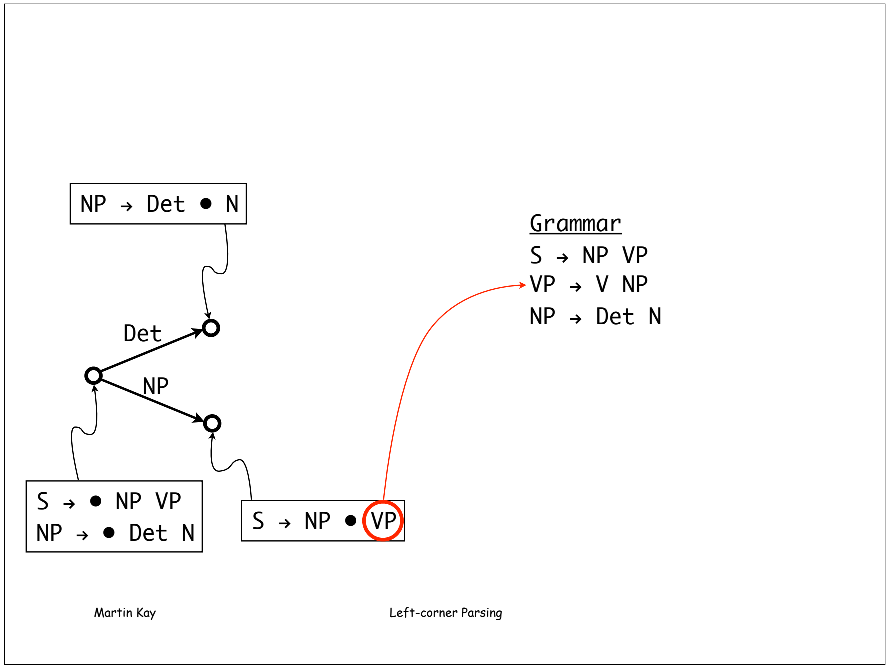
Grammar
S → NP VP
VP → V NP
NP → Det N

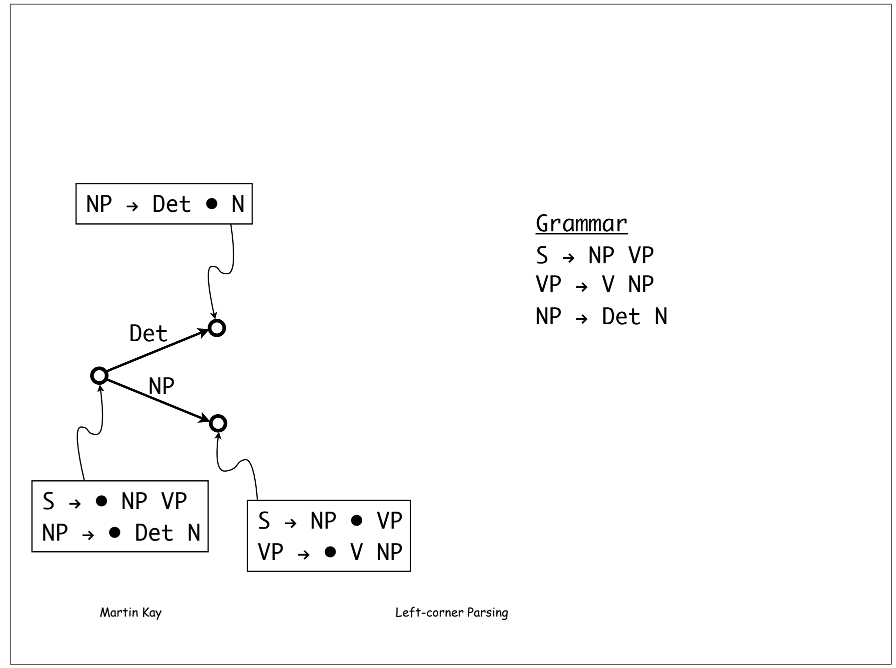


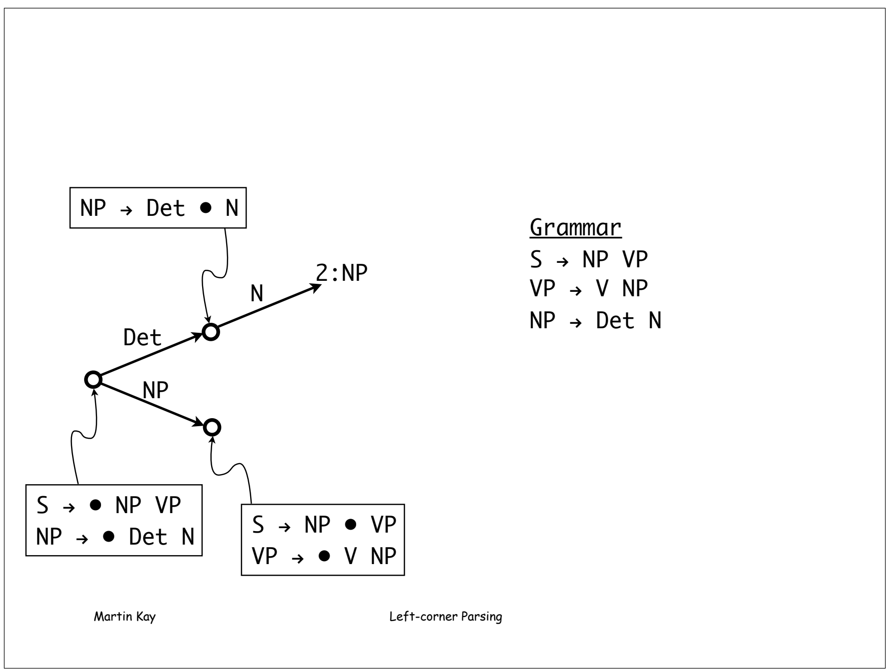
Grammar
S → NP VP
VP → V NP
NP → Det N

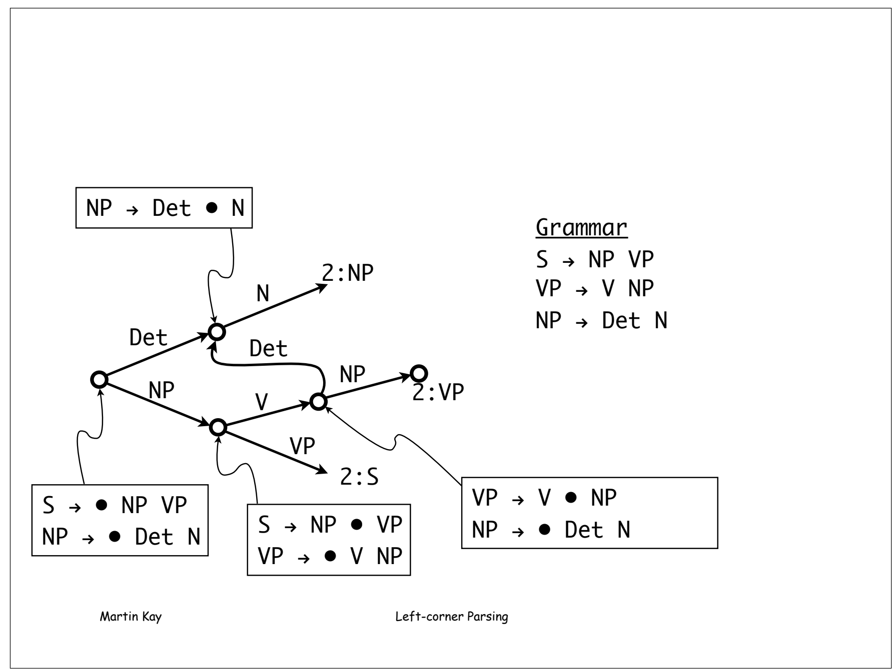


Grammar
S → NP VP
VP → V NP
NP → Det N


NP → Det • N
Det
NP
S → • NP VP
NP → • Det N
S → NP • VP
Grammar
S → NP VP
VP → V NP
NP → Det N
Martin Kay
Left-corner Parsing


NP → Det • N
Det
NP
S → • NP VP
NP → • Det N
S → NP • VP
VP → • V NP
Grammar
S → NP VP
VP → V NP
NP → Det N
Martin Kay
Left-corner Parsing


NP → Det • N
N
2:NP
Det
NP
S → • NP VP
NP → • Det N
S → NP • VP
VP → • V NP
Grammar
S → NP VP
VP → V NP
NP → Det N
Martin Kay
Left-corner Parsing


NP → Det • N
N
2:NP
Det
Det
NP
2:VP
NP
V
VP
2:S
S → • NP VP
NP → • Det N
S → NP • VP
VP → • V NP
VP → V • NP
NP → • Det N
Grammar
S → NP VP
VP → V NP
NP → Det N
Martin Kay
Left-corner Parsing

Reachability—Top down

```
parse([], String, String).
parse([Goal | Goals], [Goal | String], Rest) :-
  parse(Goals, String, Rest).
parse([Goal | Goals], String0, Rest) :-
  rule(Goal, Rhs),
  parse(RHS, String0, String1),
  parse(Goals, String1, Rest).
```


Reachability—Top down

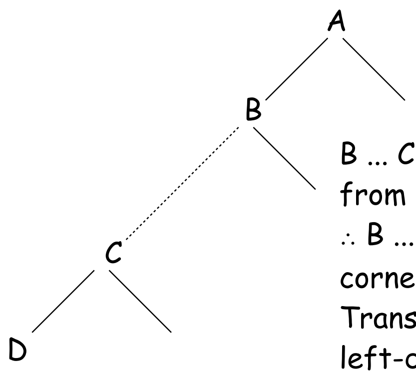
```
parse([], String, String).
parse([Goal | Goals], [Goal | String], Rest) :-
  parse(Goals, String, Rest).
parse([Goal | Goals], [Item1 | String0], Rest) :-
  rule(Goal, [Item2 | Rhs]),
  reachable(Item2, Item1),
  parse(RHS, [Item1 | String0], String1),
  parse(Goals, String1, Rest).
```


Reachability—Bottom-up

```
parse([], String, String).
parse([Goal | Goals], [Goal | String], Rest) :-
  parse(Goals, String, Rest).
parse(Goals, [Item | String0], Rest) :-
  rule(P, [Item | Rhs]),
  parse(RHS, String0, String1),
  parse(Goals, [P | String1], Rest).
```


Reachability-Bottom-up

```
parse([], String, String).
parse([Goal | Goals], [Goal | String], Rest) :-
  parse(Goals, String, Rest).
parse([Goal | Goals], [Item | String0], Rest) :-
  rule(P, [Item | Rhs]),
  reachable(Goal, P),
  parse(RHS, String0, String1),
  parse([Goal | Goals], [P | String1], Rest).
```



B ... C, D are reachable from A.
∴ B ... C, D are (proper) left corners of A.
Transitive closure of the left-daughter relation.

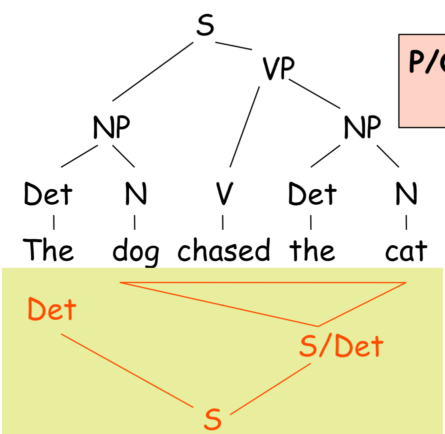
Remember this?

A grammar t… contains instances of left recursion c…ced by one with no left recursion which…

- describes the same…
- assigns the same structu… …he sentences.

COMING SOON!

The Left-corner Transformation



P/Q: The part of a P that follows a Q

Eliminating Left Recursion (step 1)

Rules

```
s --> np, vp.
np --> det, n.
n --> adj, n.
np --> np, pp.
det --> np, ap_s.
pp --> prep, np.
vp --> iv.
vp --> vt, np.
vp --> vp, pp.
```

```
det --> [the].
n --> [dog].
n --> [cat]
n --> [bone].
n --> [field].
adj --> [big].
prep --> [with].
prep --> [in].
vi --> [slept].
vt --> [chased].
```

Reachability

s:	np
	det
np:	det
det:	det
n:	n
vp:	iv
	tv
pp:	prep

```
s --> np, s/np.
s --> det, s/det.
np --> det, np/det.
np --> np, np/np.
det --> np, det/np.
det --> det, det/det.
pp --> prep, pp/prep.
vp --> iv, vp/iv.
vp --> vt, vp/vt.
vp --> vp, vp/vp
```

Step 2

Rules

```
s --> np, vp.
np --> det, n.
n --> adj, n.
np --> np, pp.
det --> np, ap_s.
pp --> prep, np.
vp --> iv.
vp --> vt, np.
vp --> vp, pp.

det --> [the].
n --> [dog].
n --> [cat]
n --> [bone].
n --> [field].
adj --> [big].
prep --> [with].
prep --> [in].
vi --> [slept].
vt --> [chased].
```

```
s --> np, s/np.
s --> det, s/det.
np --> det, np/det.
np --> np, np/np.
det --> np, det/np.
det --> det, det/det.
pp --> prep, pp/prep.
vp --> iv, vp/iv.
vp --> vt, vp/vt.
vp --> vp, vp/vp
```

```
s/np --> vp, s/s.
s/det --> n, s/np.
np/det --> n, np/np.
np/np --> [].
np/np --> ap/s, np/det.
det/np --> ap_s, det/det.
pp/prep --> np, pp/pp
...
```


Step 2

Rules

```
s --> np, vp.
np --> det, n.
n --> adj, n.
np --> np, pp.
det --> np, ap_s.
pp --> prep, np.
vp --> iv.
vp --> vt, np.
vp --> vp, pp.

det --> [the].
n --> [dog].
n --> [cat]
n --> [bone].
n --> [field].
adj --> [big].
prep --> [with].
prep --> [in].
vi --> [slept].
vt --> [chased].
```

```
s --> np, s/np.
s --> det, s/det.
np --> det, np/det.
np --> np, np/np.
det --> np, det/np.
det --> det, det/det.
pp --> prep, pp/prep.
vp --> iv, vp/iv.
vp --> vt, vp/vt.
vp --> vp, vp/vp
```

```
s/np --> vp, s/s.
s/det --> n, s/np.
np/det --> n, np/np.
np/np --> [].
np/np --> ap_s, np/det.
det/np --> ap_s, det/det.
pp/prep --> np, pp/pp
...
```


Caching

```
foo(Input1, Input2, Output1, Output2)
```

Inputs Outputs

```
try(foo(A, B, C, F)) :-
  \+ done(foo(A, B)),
  assert(done(foo(A, B))),
  foo(A, B, C, D),
  assert(result(foo(A, B, C, D))),
  fail.
try(foo(A, B, C, D)) :-
  result(foo(A, B, C, D)).
```

Find all solutions on the first call!

Caching top-down

```
parse([], String, String).
parse([Goal | Goals], [Goal | String], Rest) :-
  try(parse(Goals, String, Rest)).
parse([Goal | Goals], String0, Rest) :-
  rule(Goal, RHS),
  try(parse(RHS, String0, String1)),
  try(parse(Goals, String1, Rest)).
```

```
parser(Goa
  retracta
  retracta
  try(pars
try(parse(Goals, String0, String)) :-
  \+ done(parse(Goals, String0)),
  assert(done(parse(Goals, String0))),
  parse(Goals, String0, String),
  assert(result(parse(Goals, String0, String))),
  fail.
try(parse(Goals, String0, String)) :-
  result(parse(Goals, String0, String)).
```

A repackaged left-corner parser

```
parse([], String, String).
parse([Goal | Goals], [Goal | String0], String) :-
  parse(Goals, String0, String).
parse(Goals, String0, String) :-
  apply_rule(String0, String1),
  parse(Goals, String1, String).

apply_rule([Item | String0], [Phrase | String]) :-
  rule(Phrase, [Item | Rhs]),
  parse(Rhs, String0, String),
```


apply_rule as a memo-function

```
apply_rule(String0, String) :-
  table(String0, Strings),
  !,
  member(String, Strings).
apply_rule(String0, String) :-
  setof(S, apply_rule0(String0, S), Strings),
  assert(table(String0, Strings)),
  member(String, Strings).

apply_rule0([Item | String0], [Phrase |
  String]) :-
  rule(Phrase, [Item | Rhs]),
  parse(Rhs, String0, String).
```


The Cache

```
table([the,dog,chased,the,cat],
  [det,dog,chased,the,cat]).
table([dog,chased,the,cat], [n,chased,the,cat]).
table([det,dog,chased,the,cat],
  [np,chased,the,cat]).
table([chased,the,cat], [v,the,cat]).
table([the,cat], [det,cat]).
table([cat], [n]).
table([det,cat], [np]).
table([v,the,cat], [vp]).
table([np,chased,the,cat], [s]).
```


- **Bottom-up parsers also use a stack, but in this case, the stack represents a summary of the input already seen, rather than a prediction about input yet to be seen.**

Backtracking Complexity

- **Exponential—why?**