

Computational Linguistics:

Defining, calculating, using, and learning linguistic structures

Edward P. Stabler
Lx185/209 lecture notes, 2013-06-14 13:41

0	Introduction	1
I	Simple syntax: Context free phrase structure	9
1	Top-down CF recognition	11
2	Top-down CF parsing	23
3	Beam instead of backtrack, and more alternatives	45
4	Bottom-up CF parsing	53
5	Left-corner CF parsing	67
6	Generalized left-corner CF parsing	73
7	Dynamic methods: CKY, Earley	81
II	Succinct syntax: Beyond context free	91
8	Minimalist grammar	93
9	MGs, MCFGs, CKY and Earley	107
10	Derivations, derived trees, and tree transductions	117
11	MG GLC beam recognition	131
12	MG elaborated	135
13	MG simplified	159
14	MG with copying	161
III	Interfaces, origins, summary	169
15	Below syntax: Morphology, phonology	171
16	Above syntax: Natural logic, discourse dynamics	177
17	Around syntax: Selecting the best parse	193
18	Before syntax: Learning the language	195
19	Summary and key open problems	197

Chapter 0 Introduction

ABSTRACT. These notes will introduce formal grammars and parsing methods for human languages, beginning with syntax, then considering how the parsing models should extend to include, or at least interface with, phonology on one side and reasoning on the other. Finally, time permitting, some recent methods for learning grammars will be explored.

- The notes begin with a survey of basic methods of left-to-right phrase structure (context free, CF) parsing. There is a large space of possibilities, and it is easy to show
 - they vary greatly in time/space requirements,
 - they vary in their suitability for plausible models of incremental interpretation, and
 - they set the stage for parsing languages that are not CF definable.

Here and throughout these notes, the discussions will be mainly informal, aiming for an intuitive understanding of the options. Though the basics in this first section of the notes were discovered mainly in the 1960's and 1970's, they are still not widely known.

- CF parsing methods can be extended to minimalist grammars (MGs), representatives of the class that Joshi'85 calls 'mildly context sensitive' (MCS), with attention to
 - how MGs capture linguistic regularities that CFGs miss, and hence can be much more succinct,
 - how MGs capture a wide range of Chomskian analyses, but are very similar to tree adjoining grammars and certain phrase structure grammars,
 - their plausibility in models for human incremental interpretation, and
 - setting the stage for parsing languages that are not MCS.
- The MG parsers will then be extended to (non-MCS) minimalist grammars with copying (MGCs), with attention to
 - how these grammars capture regularities MGs miss (namely: copying, sharing!),
 - their plausibility in models of incremental interpretation.
- Some preliminary attention will then be given to
 - (under syntax) extending MGCs to (or at least towards) phonology/orthography,
 - (above syntax) using MGC analyses in reasoning and discourse,
 - (around syntax) how phonetics and reasoning about discourse – factors outside of syntax – can incrementally guide the selection among parsing options.
- Finally, we explore how MGCs could be learned from examples and teachers.

So our goals are scientific, very far short of a whole understanding but implementable as far as they go. Science enables the engineers; engineers are the game-changers.

ACKNOWLEDGEMENTS. The material in these notes has been incubating for a very long time. A project like this is a team effort with more participants than most readers would ever imagine. This would not have been possible without help from Ed Keenan, Marcus Kracht, Jens Michaelis, Greg Kobele, Thomas Graf, Hilda Koopman, Dominique Sportiche, Aravind Joshi, Bob Berwick, Jerry Fodor, Mike Harnish, Kristine Yu, the students in my classes, *and so many others* – many of whom do not approve! Certainly, many errors and important omissions remain here, and should be blamed only on me.

0.1 Language, logic, algorithms, computation

- (1) Let's define a logic as something with these three parts:
- i. a language (a set of expressions) that has
 - ii. a “derives” relation \vdash defined for it (a syntactic relation on expressions), and
 - iii. a semantics: expressions of the language have meanings.
 - The meaning of an expression is usually specified with a “model” that determines a semantic valuation function that is often written with double brackets:

$$\llbracket \text{socrates_is_mortal} \rrbracket = \text{True}$$
 - Once the meanings are given, we can usually define an “entails” relation \models , so that for any set of expressions Γ and any expression A , $\Gamma \models A$ means that every model that makes all sentences in Γ true also makes A true.

So a logic has three parts: it's (i) a language, with (ii) a derives relation \vdash , and with (iii) meanings.

We expect the derives relation \vdash should correspond to the \models relation in some way. The logic is **sound** iff from axioms Γ the derivable results are all entailed by Γ . The logic is **complete** iff everything entailed by Γ is derivable.

- (2) We propose, following Montague and many others to study human language from this perspective:
- Each human language is a logic.

This perspective carves up various aspects of language in a useful way for our computational approach.

- (3) We also propose:
- Programming languages are naturally represented as logics.

The programs have a syntax, and a meaning, and each program defines an inference relation, a mapping from input to output (the mapping may be partial, since a program can fail to terminate on some inputs).
 - Systems that can recognize expressions as grammatical or not are also naturally represented as logics.

A grammar is the language of a ‘recognizer’, the grammar semantically denotes a certain language, and the grammar defines an inference relation between recognizer steps. The recognizer is **sound** iff it accepts only strings in the language that the grammar denotes, and it is **complete** iff it accepts every string in the language that the grammar denotes.

A ‘parser’ is a recognizer that returns some kind of structural description of any sequences that are accepted.
- (4) Computer science was born in logic. Computations generally can be regarded as sequences that have certain properties: the steps in the sequence are syntactically defined (\vdash) but they are typically semantically meaningful (\models).
- A physical system realizes/implements a computation iff the causal relations among interpreted states match the defined relations among the interpreted syntactic objects.¹
- (5) An algorithm is a ‘recipe’ that specifies the steps in a computation, steps that stand in a similar ‘derivational’ relation.
- (6) Turing argues that every ‘mechanical calculation’ can be carried out by a very simple kind of device, a Turing machine. Remarkably, it turned out that these calculations also correspond to the evaluations of Church’s ‘recursive’ functions. And many other kinds of devices turned out to be capable of defining exactly the same computations: 2-register machines [9], an infinite abacus with finite recipes [7] ‘general purpose analog machines’ [12], and many other things. Moschovakis has a concise technical presentation the Church-Turing thesis in his lecture notes [11].
- (7) We will use the programming language python (version 2) to implement our recognizers and parsers. Discussions of implementation will be more prominent at the beginning, where we are getting started, than at the end, where we will be expert enough to understand the code without as much explicit discussion.

¹The question of when two computations should be regarded as the same, or one as a compiled version of another, is a delicate one [10, 1]. Optimizing compilers confuse the situation even further.



Figure 1: Gregor Reisch (1508): The new versus the old arithmetic algorithms: Boëthius against Pythagoras, Hindu-Arabic numerals versus the counting board. Theories of sequences are mutually interpretable with theories of number [14, 15, 2], and in computational linguistics too, the choice of representation is key.

0.2 The object of study: human language

Human languages have many surprising properties. Some of these may be due to historical accidents, but the kinds of patterns which seem fairly common and stable across human languages must be due to peculiarities of the way we learn and compute grammatical structure. This is something that (almost) everyone begins to notice when they study linguistics. I like to keep in mind a few of these surprising things – what the linguist Peter Culicover calls ‘syntactic nuts’ [4]:

- In natural tense logics, there are operators that can be prefixed to sentences

Fp	at some time in the future(p)
Pp	at some time in the past(p)

But in human languages like English, the expression of temporal relations is most often marked on a verb or modal or auxiliary, and the way this interacts with agreement, negation, contraction, ellipsis and other aspects of structure is not simple.²

* Is not Abrams hiring Brown?	Isn’t Abrams hiring Brown?
* Abrams has had hired Brown?	* Don’t anyone be trustworthy?
Could Abrams not have hired Brown?	Could Abrams have not hired Brown?
* Abrams does merely not work	
I could not understand that	I couldn’t understand that
But I am not surprised	* But I amn’t surprised
Had he hired me, I would be happy	* Hadn’t he hired me, I would be happy
Abrams need not hire Brown	* Abrams need hire Brown

Our most common verbs and auxiliaries are most irregular, and in many other languages, things are worse.

- It is perhaps not surprising that material that is in some sense predictable can be left out in human languages, but the details are very tricky!

Joe was murdered by someone, but we don’t know who
 Joe was murdered by someone, but we don’t know by who
 Joe was murdered, but we don’t know by who
 * Joe was murdered, but we don’t know who

He proved something, but I don’t know what
 He was evaluating a proof, but I don’t know of what
 * He was evaluating a proof, but I don’t know what

- English has a range of slightly idiomatic copying constructions [8]:

(X-or-no-X)	Linguistics or no linguistics, let’s party	(X-shmX)	Linguistics shminguistics
(X-is-a-X-is-a-X)	A dog is a dog is a dog	(X or X?)	Is she beautiful or is she beautiful?

These sound odd when the two X’s are not exact copies, except (at least for some speakers) for certain expletive insertions [13]:

- Linguistics test or no damn linguistics test, I am going home
- Long linguistics test or no long bloody linguistics test, I am going home

Many other languages have reduplication that is much more central in their grammars. More on this later!

0.3 Introducing python

0.3.1 Printing, arithmetic, strings, variables, and other basics

```
>>> print 'hello world'
hello world
>>> 2*3
6
>>> 2/3
0
```

²Many of these examples from [5].


```
>>> 2./3.
0.6666666666666666
>>> 'x'
'x'
>>> 'x'+ 'y'
'xy'
>>> w+w
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'w' is not defined
```

Since we use = for assignment, we use == for equality.

```
>>> 3 = 2+1
File "<stdin>", line 1
SyntaxError: can't assign to literal
>>> 3 == 2+1
True
>>> 2 >= 1
True
>>> 2 >= 2
True
>>> 2 >= 3
False
```

0.3.2 Importing modules

Suppose we create this file

```
"""
file: test1.py
"""

# I include just this one assignment
ncex = ['colorless', 'green', 'ideas', 'sleep', 'furiously']
```

Then:

```
>>> from test1 import *
>>> ncex
['colorless', 'green', 'ideas', 'sleep', 'furiously']
```

Alternatively:

```
>>> import test1
>>> ncex
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'ncex' is not defined
>>> test1.ncex
['colorless', 'green', 'ideas', 'sleep', 'furiously']
```

0.3.3 Recursion, loops, defining functions

Euclid's algorithm for finding the greatest common divisor of two integers is among the oldest explicitly given. Recalling that $a \bmod b$ is the remainder of a divided by b , it is given this way in a modern book of algorithms [3, §31.2]:

```
EUCLID( $a, b$ )
1  if  $b == 0$ 
2     return  $a$ 
3  else
4     return EUCLID( $b, a \bmod b$ )
```

For example, we can calculate the greatest common divisor of 12 and 18:

<u>step</u>	<u>calculation</u>
0	EUCLID(12,18), that is, a=12 and b=18
1	since $b \neq 0$, we calculate. . .
2	EUCLID(18,12 MOD 18)=EUCLID(18,12), that is, a=18 and b=12
3	since $b \neq 0$, we calculate. . .
4	EUCLID(12,18 MOD 12)=EUCLID(12,6), that is, a=12 and b=6
5	since $b \neq 0$, we calculate. . .
6	EUCLID(6,12 MOD 6)=EUCLID(6,0), that is, a=6 and b=0
7	so we return 6

This formulation of Euclid's algorithm is recursive in the sense that it calls itself, but we can get the same effect with a loop (and loops are generally more efficient in python than recursion is):

```

EUCLID(a, b)
1  while b ≠ 0
2      r = a mod b
3      a = b
4      b = r
5  return a

```

For example, let's see how the previous calculation looks, now done with loops:

<u>step</u>	<u>calculation</u>
0	EUCLID(12,18), that is, a=12 and b=18
1	since $b \neq 0$, we enter the loop and calculate. . .
2	r=12 mod 18=12
3	a=18
4	b=12
5	since $b \neq 0$, we enter the loop and calculate. . .
6	r=18 mod 12=6
7	a=12
8	b=6
9	since $b \neq 0$, we enter the loop and calculate. . .
10	r=12 mod 6=0
11	a=6
12	b=0
13	since $b \neq 0$, we exit the loop and return 6

We can write this algorithm in python quite directly (using % for mod):

```

""" Euclid's algorithm
"""

```

```

def euclid(a, b):
    while b != 0:
        r = a % b
        a = b
        b = r
    return a

```

Now python will do the work for us:

```

>>> from test2 import *
>>> euclid(12,18)
6
>>> euclid(21,9)
3
>>>

```

0.3.4 Lists and more loops

There are many ways to represent sequences of elements, but we will begin with python's lists [1,2,3]. (They are actually what computer scientists usually call 'arrays'.) To apply a function to every element of a list, it is common to use a *for*-loop:

```

""" lists, for-loops
"""
def unordList(l):
    for e in l:
        print e

def enumList(l):
    for (i,e) in enumerate(l):
        print i,e

def spacedList(l):
    for e in l:
        print e,
    print

```

Then:

```

>>> from test3 import *
>>> unordList([2,5,7])
2
5
7
>>> spacedList([2,5,7])
2 5 7

```

We can efficiently get any element out of a python list by its integer position, counting from 0:

```

>>> x=[2,3,5]
>>> x[0]
2
>>> x[1]
3
>>> x[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

```

We can also count from the right, with ‘negative’ positions:

```

>>> x[-1]
5
>>> x[-2]
3

```

And we can take out a subsequence (a ‘slice’):

```

>>> x[0:1]
[2]
>>> x[0:2]
[2, 3]
>>> x[1:]
[3, 5]
>>> x[:2]
[2, 3]

```

The + operator appends two lists:

```

>>> x=[2,3,5]
>>> y=[9,10]
>>> x+y
[2, 3, 5, 9, 10]

```

That calculation does not change either x or y, but we can also extend list x with y:

```

>>> x.extend(y)
>>> x
[2, 3, 5, 9, 10]

```

The empty sequence often written ϵ (or sometimes λ) in math books is of course [] in python.

References

- [1] BLASS, A., DERSHOWITZ, N., AND GUREVICH, Y. When are two algorithms the same? *Bulletin of Symbolic Logic* 15, 2 (2009), 145–168.
- [2] CORCORAN, J., FRANK, W., AND MALONEY, M. String theory. *Journal of Symbolic Logic* 39 (1974), 625–637.
- [3] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms, 2nd Edition*. MIT Press, Cambridge, Massachusetts, 2001.
- [4] CULICOVER, P. W. *Syntactic Nuts: Hard Cases, Syntactic Theory, and Language Acquisition*. Oxford University Press, NY, 1999.
- [5] FLICKINGER, D., NERBONNE, J., SAG, I., AND WASOW, T. Toward evaluation of NLP systems. Tech. rep., Hewlett-Packard Laboratories, 1997.
- [6] JOSHI, A. K. How much context-sensitivity is necessary for characterizing structural descriptions. In *Natural Language Processing: Theoretical, Computational and Psychological Perspectives*, D. Dowty, L. Karttunen, and A. Zwicky, Eds. Cambridge University Press, NY, 1985, pp. 206–250.
- [7] LAMBEK, J. How to program an (infinite) abacus. *Canadian Mathematical Bulletin* 4 (1961), 295–302.
- [8] MANASTER-RAMER, A. Copying in natural languages, context freeness, and queue grammars. In *Proceedings of the 1986 Meeting of the Association for Computational Linguistics* (1986).
- [9] MINSKY, M. L. *Computation: Finite and Infinite Machines*. Prentice-Hall, Englewood Cliffs, New Jersey, 1967.
- [10] MOSCHOVAKIS, Y. N. What is an algorithm? In *Mathematics unlimited – 2001 and beyond*, B. Engquist and W. Schmid, Eds. Springer, NY, 2001, pp. 919–936.
- [11] MOSCHOVAKIS, Y. N. *Recursion and Computation*. University of California, Los Angeles, 2013. <http://www.math.ucla.edu/~ynm/114c.1.13w/nirteng.pdf>.
- [12] POULY, A., BOURNEZ, O., AND GRAÇA, D. S. Analog models of computations. *CoRR abs/1203.4667* (2012).
- [13] PULLUM, G. K., AND RAWLINS, K. Argument or no argument? *Linguistics and Philosophy* 30, 2 (2007), 277–287.
- [14] QUINE, W. Concatenation as a basis for arithmetic. *Journal of Symbolic Logic* 11 (1946), 105–114. Reprinted in Willard V.O. Quine, *Selected Logic Papers*, NY: Random House, 1961.
- [15] RABIN, M. O. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society* 141 (1969), 1–35.

Part I

Simple syntax: Context free phrase structure

Chapter 1 Top-down CF recognition

After introducing various substitution tests as structural probes, Fromkin's [3, p.175] introduction to linguistics builds a grammar like this:¹

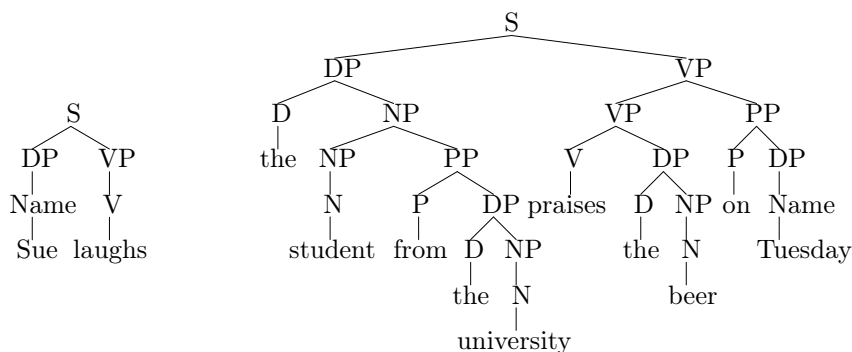
basic rules for selected elements		rules for modifiers
(137)	$S \rightarrow DP VP$	
(124a)	$DP \rightarrow \left\{ \begin{array}{l} (D) NP \\ \text{Name} \\ \text{Pronoun} \end{array} \right\}$	
(123)	$NP \rightarrow N (PP)$	
(130)	$VP \rightarrow V (DP) \left(\left\{ \begin{array}{l} PP \\ CP \\ VP \end{array} \right\} \right)$	
(120)	$PP \rightarrow P (DP)$	
(122)	$AP \rightarrow A (PP)$	
(138)	$CP \rightarrow C S$	
	$AdvP \rightarrow Adv$	
	(73c.iii)	$NP \rightarrow AP NP$
		$NP \rightarrow NP PP$
		$NP \rightarrow NP CP$
		$VP \rightarrow AdvP VP$
		$VP \rightarrow VP PP$
		$AP \rightarrow AdvP AP$
	$\alpha \rightarrow \alpha \text{ Coord } \alpha$ (for $\alpha=D,V,N,A,P,C,Adv,VP,NP,DP,AP,PP,AdvP,S,CP$)	

The numbering in parentheses comes from the Fromkin text. Recall that parentheses in the rules indicate optionality and braces mean 'choose exactly one'. The lexical items are given in a different format, but the same format could be used:

$D \rightarrow \left\{ \begin{array}{l} \text{the} \\ \text{a} \\ \text{some} \\ \text{every} \\ \text{one} \\ \text{two} \end{array} \right\}$		$A \rightarrow \left\{ \begin{array}{l} \text{gentle} \\ \text{clear} \\ \text{honest} \\ \text{compassionate} \\ \text{brave} \\ \text{kind} \end{array} \right\}$	$N \rightarrow \left\{ \begin{array}{l} \text{student} \\ \text{teacher} \\ \text{city} \\ \text{university} \\ \text{beer} \\ \text{wine} \end{array} \right\}$	$V \rightarrow \left\{ \begin{array}{l} \text{laughs} \\ \text{cries} \\ \text{praises} \\ \text{criticizes} \\ \text{says} \\ \text{knows} \end{array} \right\}$	$Adv \rightarrow \left\{ \begin{array}{l} \text{happily} \\ \text{sadly} \\ \text{impartially} \\ \text{generously} \end{array} \right\}$
$Name \rightarrow \left\{ \begin{array}{l} \text{Bill} \\ \text{Sue} \\ \text{José} \\ \text{Maria} \\ \text{Presidents Day} \\ \text{Tuesday} \end{array} \right\}$		$Pronoun \rightarrow \left\{ \begin{array}{l} \text{he} \\ \text{she} \\ \text{it} \\ \text{her} \\ \text{him} \end{array} \right\}$	$P \rightarrow \left\{ \begin{array}{l} \text{in} \\ \text{on} \\ \text{with} \\ \text{by} \\ \text{to} \\ \text{from} \end{array} \right\}$	$C \rightarrow \left\{ \begin{array}{l} \text{that} \\ \epsilon \\ \text{whether} \end{array} \right\}$	$Coord \rightarrow \left\{ \begin{array}{l} \text{and} \\ \text{or} \\ \text{but} \end{array} \right\}$

This grammar defines an infinite set of derivations like these:

¹A grammar like this is a standard first step in linguistics texts. We find similar grammars in [4, pp.33-4] and [6, p.189-192] and [2, p.94] and [7, p.124].



This is a *context free grammar* (CFG) since on the left side of each rule we have just a category, which can ‘expand to’ (or ‘be built up from’) any of the sequences on the right side of the rules in the grammar. This particular grammar has many defects, besides just being radically incomplete:

- There are many regularities in English that this grammar does not enforce, like subject-verb agreement, case requirements on pronouns, etc.
- There are regularities in the grammar which appear non-accidental: e.g. that VPs have Vs in them, PPs have Ps in them. That is, the fundamental properties of phrases often seem to be (largely) determined by one of their elements, the ‘head’.

Nevertheless, this grammar is a good starting point for us. We will get to better grammars later.

If a derivation roughly like the one shown above is computed when you hear that sentence in ordinary fluent speech, how could that happen?

Q. What algorithms can map orthographic or phonetic representations of sentences to their structures?

This is a first, rough indication of the main question for the class. We will see that the question is not stated precisely enough, and will formulate it more carefully on page 39 in §2, below.

1.1 Top-down backtrack CF recognition

The way we recognize a sentence as grammatical, as one that is allowed by the grammar, is to find a derivation of the sentence. This step, *recognition* is often separated from the formulation of a structural representation which is called *parsing*. A parser is usually just a recognizer that keeps a record of the steps used in the successful derivation, but still it is useful to think just about the recognizer first.

So the recognition problem is: given a grammar and a string of words, return True if the string has a derivation and False otherwise.

One way to proceed is to begin with the sentence category S (or whatever it is) and rewrite the leftmost elements until we get to a word, at which point we can check that word against the input, and so on. So, given the grammar and sentence derived above, we begin, step 0, by predicting S, and then expand the S as our first step:

<u>step</u>	<u>predicted</u>	<u>input</u>
0.	S	Sue laughs
1.	DP VP	Sue laughs

At this point we get stuck, though, since there are several ways to expand the DP. A simple strategy for handling this problem – the standard top-down, backtracking strategy – is to take all of the next steps, put them in a list, and then work on one of the possibilities. If that possibility does not work out, then we will try one of the other possibilities. So for the next step we actually take 5 steps (listed here in the order they appear in the grammar above):

<u>step</u>	<u>predicted</u>	<u>input</u>
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
2e.	DP Coord DP VP	Sue laughs

At this point, we choose one of 2a-2e, for example the first one 2a, and expand the leftmost category, and again we have choices:

<u>step</u>	<u>predicted</u>	<u>input</u>
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
2e.	DP Coord DP VP	Sue laughs
3aa.	D Coord D NP VP	Sue laughs
3ab.	the NP VP	Sue laughs
3ac.	a NP VP	Sue laughs
3ad.	some NP VP	Sue laughs
3ae.	every NP VP	Sue laughs
3af.	one NP VP	Sue laughs
3ag.	two NP VP	Sue laughs

Now we could expand 3aa in all possible ways:

<u>step</u>	<u>predicted</u>	<u>input</u>
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
2e.	DP Coord DP VP	Sue laughs
3aa.	D Coord D NP VP	Sue laughs
3ab.	the NP VP	Sue laughs
3ac.	a NP VP	Sue laughs
3ad.	some NP VP	Sue laughs
3ae.	every NP VP	Sue laughs
3af.	one NP VP	Sue laughs
3ag.	two NP VP	Sue laughs
4aaa.	D Coord D Coord D NP VP	Sue laughs
4aab.	the Coord D NP VP	Sue laughs
4aab.	a Coord D NP VP	Sue laughs
4aab.	some Coord D NP VP	Sue laughs
4aab.	every Coord D NP VP	Sue laughs
4aab.	one Coord D NP VP	Sue laughs
4aab.	two Coord D NP VP	Sue laughs

Now it is clear we are in trouble... this procedure will never terminate!

The problem here is a famous one: it comes from ‘left recursion’ through the category D. Let’s say that a **category X is recursive** iff in one or more steps we can derive something else containing X (we use the superscript + to indicate one or more steps \Rightarrow):²

$$\text{(recursion)} \quad X \Rightarrow^+ \dots X \dots$$

We call this kind of recursion **left recursion** iff the category X can contain another category X as its first element (equivalently, on a leftmost branch):

$$\text{(left recursion)} \quad X \Rightarrow^+ X \dots$$

Similarly for **right recursion**:

$$\text{(right recursion)} \quad X \Rightarrow^+ \dots X$$

In the grammar above, we see for example that

²This terminology is standard in this kind of context [1, p.153], but there are other important senses of ‘recursive’. A function definition that calls itself is recursive. And a language (a set of strings) is often said to be recursive iff it is Turing decidable [5, 8].

- Coord is not recursive
- D is both left and right recursive because of coordination
- NP is left recursive in coordination and in PP modification

Left recursion is a tricky problem, so for the moment, let's just remove all left recursion from the grammar, as follows:

	basic rules for selected elements	rules for modifiers
(137)	$S \rightarrow DP VP$	
(124a)	$DP \rightarrow \left\{ \begin{array}{l} (D) NP \\ Name \\ Pronoun \end{array} \right\}$	
(123)	$NP \rightarrow N (PP)$	
(130)	$VP \rightarrow V (DP) \left(\left\{ \begin{array}{l} PP \\ CP \\ VP \end{array} \right\} \right)$	
(120)	$PP \rightarrow P (DP)$	
(122)	$AP \rightarrow A (PP)$	
(138)	$CP \rightarrow C S$	
	$AdvP \rightarrow Adv$	
	(73c.iii)	
		$NP \rightarrow AP NP$
		$NP \rightarrow NP PP$
		$NP \rightarrow NP CP$
		$VP \rightarrow AdvP VP$
		$VP \rightarrow VP PP$
		$AP \rightarrow AdvP AP$
	$\alpha \rightarrow \alpha \text{ Coord } \alpha$ (for $\alpha = D, V, N, A, P, C, Adv, VP, NP, DP, AP, PP, AdvP, S, CP$)	

So let's remove the left recursion from the example we started and continue from step 3 which now looks like this:

step	predicted	input
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
3aa.	the NP VP	Sue laughs
3ab.	a NP VP	Sue laughs
3ac.	some NP VP	Sue laughs
3ad.	every NP VP	Sue laughs
3ae.	one NP VP	Sue laughs
3af.	two NP VP	Sue laughs

At this point the first symbol in step 3aa begins with a word, but it does not match the input, so we throw 4aaa away:

step	predicted	input
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
3aa.	the NP VP	Sue laughs
3ab.	a NP VP	Sue laughs
3ac.	some NP VP	Sue laughs
3ad.	every NP VP	Sue laughs
3ae.	one NP VP	Sue laughs
3af.	two NP VP	Sue laughs

3ab is no good either, nor are any of our attempts at step 3, so we throw them all away and try to proceed from 2b:

<u>step</u>	<u>predicted</u>	<u>input</u>
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
3ba.	N VP	Sue laughs
3bb.	N PP VP	Sue laughs

Clearly neither of these will work either, and so skipping some steps, we get to the point where we consider steps from 2c:

<u>step</u>	<u>predicted</u>	<u>input</u>
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
3ca.	Bill VP	Sue laughs
3cb.	Sue VP	Sue laughs
3cc.	José VP	Sue laughs
3cd.	Maria VP	Sue laughs
3ce.	Presidents Day VP	Sue laughs
3cf.	Tuesday VP	Sue laughs

3ca is thrown out, but 3cc give us a match against our input. When we scan an element from the input (indicated by the prime mark), let's indicate the successful analysis in the input by crossing out the scanned element:

<u>step</u>	<u>predicted</u>	<u>input</u>
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
3cb.	Sue VP	Sue laughs
3cc.	José VP	Sue laughs
3cd.	Maria VP	Sue laughs
3ce.	Presidents Day VP	Sue laughs
3cf.	Tuesday VP	Sue laughs
4cb'	VP	Sue laughs

Continuing:

<u>step</u>	<u>predicted</u>	<u>input</u>
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
3cb.	Sue VP	Sue laughs
3cc.	José VP	Sue laughs
3cd.	Maria VP	Sue laughs
3ce.	Presidents Day VP	Sue laughs
3cf.	Tuesday VP	Sue laughs
4cb'	VP	Sue laughs
4cb'a	V	Sue laughs
4cb'b	V DP	Sue laughs
4cb'c	V PP	Sue laughs
4cb'd	V CP	Sue laughs
4cb'e	V VP	Sue laughs
4cb'f	V DP PP	Sue laughs
4cb'g	V DP CP	Sue laughs
4cb'h	V DP VP	Sue laughs

Taking 4cb'a first, we expand:

<u>step</u>	<u>predicted</u>	<u>input</u>
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
3cb.	Sue VP	Sue laughs
3cc.	José VP	Sue laughs
3ce.	Presidents Day VP	Sue laughs
3ce.	Monday VP	Sue laughs
3cf.	Tuesday VP	Sue laughs
4cb'	VP	Sue laughs
4cb'a	V	Sue laughs
4cb'b	V DP	Sue laughs
4cb'c	V PP	Sue laughs
4cb'd	V CP	Sue laughs
4cb'e	V VP	Sue laughs
4cb'f	V DP PP	Sue laughs
4cb'g	V DP CP	Sue laughs
4cb'h	V DP VP	Sue laughs
4cb'aa	laughs	Sue laughs
4cb'aa	cries	Sue laughs
4cb'aa	praises	Sue laughs
4cb'aa	criticizes	Sue laughs
4cb'aa	says	Sue laughs
4cb'aa	knows	Sue laughs

Now when we check 4cb'aa against the input, we fulfill the last prediction and also consume the last input symbol, which means we have found a derivation:

<u>step</u>	<u>predicted</u>	<u>input</u>
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
3cb.	Sue VP	Sue laughs
3cc.	José VP	Sue laughs
3cd.	Maria VP	Sue laughs
3ce.	Presidents Day VP	Sue laughs
3cf.	Tuesday VP	Sue laughs
4cb'	VP	Sue laughs
4cb'a	V	Sue laughs
4cb'b	V DP	Sue laughs
4cb'c	V PP	Sue laughs
4cb'd	V CP	Sue laughs
4cb'e	V VP	Sue laughs
4cb'f	V DP PP	Sue laughs
4cb'g	V DP CP	Sue laughs
4cb'h	V DP VP	Sue laughs
4cb'aa	laughs	Sue laughs
4cb'aa	cries	Sue laughs
4cb'aa	praises	Sue laughs
4cb'aa	criticizes	Sue laughs
4cb'aa	says	Sue laughs
4cb'aa	knows	Sue laughs
4cb'aa'	ϵ	Sue laughs

Success!

Even with this simple example, the procedure is tedious, so it is impractical to use reasonably sized grammars without a calculator. But first, let's describe more carefully what we did. The usual 'pseudocode' description is this something like this. First each analysis (each line in our calculation above) contains a sequence of predicted categories together with the remaining input – two lists. And the whole sequence of lines in our calculation is then a list of pairs of lists – the so-called 'backtrack stack'. There are just two basic parsing steps. The first one is called **expand** (which as we will say later means 'reduce complete'):

$$(\text{expand}) \frac{\text{input}, X\alpha}{\text{input}, \beta\alpha} \text{ if } X \rightarrow \beta$$

That is, when the predicted sequence begins with a predicted category X and we have a rule that says X rewrites as β , we can predict β followed by whatever else follows the X, leaving the input unchanged. The second rule is **scan** (which as we will say later means 'shift complete'):

$$(\text{scan}) \frac{w \text{ input}, w\alpha}{\text{input}, \alpha}$$

That is, when the predicted sequence begins with a word w and the input also begins with w, we can delete both of them (sometimes indicated informally by crossing them out). When either of these steps applies to α to produce β we write $\alpha \Rightarrow_{td} \beta$.

The following pseudocode uses the variable names chosen to be helpful:

- ds for the (typically incomplete) 'derivations'; ds is the 'backtrack stack'
- cs for the sequence of predicted categories in each derivation; cs is the 'stack'
- i for the remaining words of the input

Given a list of elements $l=[3,5,7]$, the rightmost (or sometimes leftmost) element is called the top element. Then we can pop the top element off the list and use that element like this

$$x = \text{pop}(l)$$

This means that the last element of l is popped off and assigned to x. And the operation

push(x,l)

means that l is extended with the top element x .

Then our first recognition algorithm is this one, for CFGs G in which no category is left-recursive and any (possibly empty) input i :

```

TOP-DOWN BACKTRACK CF RECOGNITION( $G, i$ )
0  ds=[(i,S)] where S is the start category
1  while ds≠ [] and ds[0]≠([],[]):
2      ds=[d| ds[0] ⇒td d] + ds[1:]
3  if ds==[] then False else True

```

The loop entered on line 1 is given simply by line 2: if there is a first derivation $ds[0]$, compute the list of derivations $[d| ds[0] ⇒_{td} d]$, that is, the list of derivations d such that d can be derived with one top-down step from $ds[0]$, and then let ds be these new derivations together with any other derivations in the list $ds[1:]$.

1.2 A naive python implementation

First we represent the grammar from page 11, but without any left recursive rules:

```

1  """ file: g1.py
2      a grammar with no left recursion
3  """
4  g1 = [('S', ['DP', 'VP']), # categorial rules
5         ('DP', ['D', 'NP']),
6         ('DP', ['NP']),
7         ('DP', ['Name']),
8         ('DP', ['Pronoun']),
9         ('NP', ['N']),
10        ('NP', ['N', 'PP']),
11        ('VP', ['V']),
12        ('VP', ['V', 'DP']),
13        ('VP', ['V', 'PP']),
14        ('VP', ['V', 'CP']),
15        ('VP', ['V', 'VP']),
16        ('VP', ['V', 'DP', 'PP']),
17        ('VP', ['V', 'DP', 'CP']),
18        ('VP', ['V', 'DP', 'VP']),
19        ('PP', ['P']),
20        ('PP', ['P', 'DP']),
21        ('AP', ['A']),
22        ('AP', ['A', 'PP']),
23        ('CP', ['C', 'S']),
24        ('AdvP', ['Adv']),
25        ('NP', ['AP', 'NP']),
26        ('VP', ['AdvP', 'VP']),
27        ('AP', ['AdvP', 'AP']),
28        ('D', ['the']), # now the lexical rules
29        ('D', ['a']),
30        ('D', ['some']),
31        ('D', ['every']),
32        ('D', ['one']),
33        ('D', ['two']),
34        ('A', ['gentle']),
35        ('A', ['clear']),
36        ('A', ['honest']),
37        ('A', ['compassionate']),
38        ('A', ['brave']),
39        ('A', ['kind']),
40        ('N', ['student']),
41        ('N', ['teacher']),
42        ('N', ['city']),
43        ('N', ['university']),
44        ('N', ['beer']),
45        ('N', ['wine']),
46        ('V', ['laughs']),
47        ('V', ['cries']),
48        ('V', ['praises']),
49        ('V', ['criticizes']),
50        ('V', ['says']),

```

```

51     ('V', ['knows']),
52     ('Adv', ['happily']),
53     ('Adv', ['sadly']),
54     ('Adv', ['impartially']),
55     ('Adv', ['generously']),
56     ('Name', ['Bill']),
57     ('Name', ['Sue']),
58     ('Name', ['Jose']),
59     ('Name', ['Maria']),
60     ('Name', ['Presidents', 'Day']),
61     ('Name', ['Tuesday']),
62     ('Pronoun', ['he']),
63     ('Pronoun', ['she']),
64     ('Pronoun', ['it']),
65     ('Pronoun', ['him']),
66     ('Pronoun', ['her']),
67     ('P', ['in']),
68     ('P', ['on']),
69     ('P', ['with']),
70     ('P', ['by']),
71     ('P', ['to']),
72     ('P', ['from']),
73     ('C', ['that']),
74     ('C', []),
75     ('C', ['whether']),
76     ('Coord', ['and']),
77     ('Coord', ['or']),
78     ('Coord', ['but'])

```

Then we can implement the recognizer like this:

```

1  """ file: td.py
2     a simple top-down backtrack CF recognizer
3  """
4  def showGrammar(g): # pretty print grammar
5      for (lhs,rhs) in g:
6          print lhs, '->',
7          for cat in rhs:
8              print cat,
9          print
10
11 def showDerivations(ds): # pretty print the 'backtrack stack'
12     for (n,(i,cs)) in enumerate(reversed(ds)):
13         print n, '(' ,
14         for w in i: # print each w in input
15             print w,
16         print ', ',
17         for c in cs: # print each predicted c in cs
18             print c,
19         print ') '
20     print '-----'
21
22 def tdstep(g,(i,cs)): # compute all possible next steps from (i,cs)
23     if len(cs)>0:
24         cs1=cs[1:] # copy of predicted categories except cs[0]
25         nextsteps=[]
26         for (lhs,rhs) in g:
27             if lhs == cs[0]:
28                 print 'expand',lhs,'->',rhs # for trace
29                 nextsteps.append((i,rhs+cs1))
30             if len(i)>0 and i[0] == cs[0]:
31                 print 'scan',i[0] # for trace
32                 nextsteps.append((i[1:],cs1))
33         return nextsteps
34     else:
35         return []
36
37 def recognize(g,i):
38     ds = [(i,['S'])]
39     while ds != [] and ds[-1] != ([],[]):
40         showDerivations(ds) # for trace
41         d = ds.pop()
42         ds.extend(tdstep(g,d))
43     if ds == []:

```

```

44     return False
45 else:
46     showDerivations(ds) # for trace
47     return True
48
49 # Examples:
50 # recognize(g1,['Sue','laughs'])
51 # recognize(g1,['Bill','knows','that','Sue','laughs'])
52 # recognize(g1,['Sue','laughed'])
53 # recognize(g1,['the','student','from','the','university','praises','the','beer','on','Tuesday'])
54 # recognize(g1,['the','student','from','the','university','praises','the'])
55 # recognize(g1,['Sue','knows','that','Maria','laughs'])

```

Now in idle, we can load `g1.py` and then `td.py` using F5, or else we can type the load commands like this:

```

>>> from g1 import *
>>> from td import *

```

Now we are ready to try some examples – a few of these are given in comments at the bottom of `td.py`:

```

>>> recognize(g1,['Sue','laughs'])
0 ( Sue laughs , S )
-----
expand S -> ['DP', 'VP']
0 ( Sue laughs , DP VP )
-----
expand DP -> ['D', 'NP']
expand DP -> ['NP']
expand DP -> ['Name']
expand DP -> ['Pronoun']
0 ( Sue laughs , D NP VP )
1 ( Sue laughs , NP VP )
2 ( Sue laughs , Name VP )
3 ( Sue laughs , Pronoun VP )
-----
...
True

```

We omit many lines here, since the program does quite a lot of work on even this simple example.

This program correctly implements standard top-down backtrack parsing, but it has some problems! Listing some of them, from least to most serious:

P1. It is unnecessarily slow! Checking the ‘innermost’ loop:

- a. In line 26 we see that finding the *expand* steps requires looping through the whole list of rules.
- b. In line 27 we see an identity checks on the string names for categories. Since strings are sequences, this check is slower than, for example, checking the identity of two integers.

P2. It is necessarily slow! That is, even if we fix the problems in P1, the number of steps required to process an input of length n can be on the order of k^n for some $k > 1$.³ To fix this problem, we need to somehow reduce or eliminate ambiguity that leads to backtracking. If only we had an oracle who could tell us what to do whenever there is a choice.⁴

P3. With left recursion, it can be non-terminating! This is just problem P2 again in its most extreme form. We need an oracle or something similar to help us avoid making the wrong choices repeatedly.

1.3 Exercises

Get one of our implementations of the top-down recognizer (`td.py`, or one of the others we looked at) and the grammar `g1.py`. Rename the grammar `g<YOUR INITIALS>.py` and then modify your version as follows:

1. Unfortunately, the grammar `g1` accepts ** Sue laughs the student*. In the Fromkin text, this is excluded by the lexical entry for *laughs* which does not allow this verb to occur with a direct object. There, the verbs come with additional information about what they select: intransitives like *laughs* do not take DP complements, but transitives like *praises* do. Verbs like *knows* can select DP or CP complements, but verbs like *laugh* cannot. Fix the grammar in `td0` to get all these things right, for the 6 verbs given.

³For proof see [1, p.299].

⁴We will build oracles later. And notice that with an artificial languages, we can simply modify the language to eliminate ambiguity!

2. If you did the previous step right, it should turn out that your new grammar will never use the rule $VP \rightarrow V VP$. That rule was included in the Fromkin text for auxiliaries, and grammar g1 does not include any auxiliaries in its list of verbs.

Add auxiliaries (using 3rd person present for the finite forms) and bare verbs (*laugh*) and participle forms (*laughed*, *laughing*), so that the recognizer accepts the examples the left, but not the ungrammatical forms on the right:

Sue will laugh	* Sue will laughs
Sue has laughed	* Sue has laughs
Sue is laughing	* Sue is laughed
Sue has been laughing	* Sue will been laughing
Sue will be laughing	* Sue has be laughing

Your changes should be as minimal and as linguistically reasonable as possible. Test your new grammar with the recognizer, and when it is working as required, send it to me as an attachment with “HW1 185a” in the subject line.

References

- [1] AHO, A. V., AND ULLMAN, J. D. *The Theory of Parsing, Translation, and Compiling. Volume 1: Parsing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1972.
- [2] CARNIE, A. *Syntax: A Generative Introduction (Second edition)*. Blackwell, Oxford, 2006.
- [3] FROMKIN, V., Ed. *Linguistics: An Introduction to Linguistic Theory*. Blackwell, Oxford, 2000.
- [4] JOHNSON, K. Introduction to transformational grammar. Tech. rep., University of Massachusetts, Amherst, 2007.
- [5] LEWIS, H. R., AND PAPADIMITRIOU, C. H. *Elements of the Theory of Computation*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.
- [6] O'GRADY, W., DOBROVOLSKY, M., AND KATAMBA, F. *Contemporary Linguistics: An Introduction (Third Edition)*. Pearson Education, Essex, UK, 1996.
- [7] RADFORD, A. *Transformational Grammar: A First Course*. Cambridge University Press, Cambridge, 1988.
- [8] ROGERS, H. *Theory of recursive functions and effective computability*. MIT Press, Cambridge, Massachusetts, 1967.

Chapter 2 Top-down CF parsing

2.1 TD backtrack parsing (complete derivations)

It is a simple matter to modify our recognizer to record a history of the steps taken in the derivation. We modify the pseudocode from page 18 as follows:

```
TOP-DOWN BACKTRACK CF PARSING( $G, i$ )
0  ds=[(i,S,[(i,S)])] where S is the start category
1  while ds≠ [] and ds[0]≠([],[]):
2      ds=[d | ds[0] ⇒tdh d] + ds[1:]
3  if ds==[] then False else True
```

where the steps \Rightarrow_{tdh} are defined as follows:

$$\text{(expand)} \frac{\text{input, } X\alpha, h}{\text{input, } \beta\alpha, h(\text{input, } \beta\alpha)} \quad \text{if } X \rightarrow \beta \quad \text{(scan)} \frac{w \text{ input, } w\alpha, h}{\text{input, } \alpha, h(\text{input, } \alpha)}$$

The implementation is also an easy change from the `td.py`. Now that we are looking at the derivations, it is sometimes of interest to see more than one of them, and so we separate the main `derive` step from the `parse` function which now asks the user whether to look for another parse:

```
1  """ file: tdh.py
2     first parser: collect history
3  """
4  def tdhstep(g,(i,cs,h)): # compute all possible next steps from (i,cs)
5      if len(cs)>0:
6          cs1=cs[1:] # copy of predicted categories except cs[0]
7          nextsteps=[]
8          for (lhs,rhs) in g:
9              if lhs == cs[0]:
10                 print 'expand',lhs,'->',rhs # for trace
11                 h1 = h[:] # copy of history
12                 h1.append((i,rhs+cs1))
13                 nextsteps.append((i,rhs+cs1,h1))
14             if len(i)>0 and i[0] == cs[0]:
15                 print 'scan',i[0] # for trace
16                 i1=i[1:]
17                 h1 = h[:] # copy of history
18                 h1.append((i1,cs1))
19                 nextsteps.append((i1,cs1,h1))
20         return nextsteps
21     else:
22         return []
23
24  def derive(g,ds):
25     while ds != [] and not (ds[-1][0] == [] and ds[-1][1] == []):
26         d = ds.pop()
27         ds.extend(tdhstep(g,d))
28
29  def parse(g,i):
30     ds = [(i,['S'],[(i,['S'])])]
31     while ds != []:
32         derive(g,ds)
33         if ds == []:
34             return 'False'
35     else:
36         d=ds.pop()
```

```

37         for n,step in enumerate(d[2]):
38             print n,step
39             ans = raw_input('more? ')
40             if len(ans)>0 and ans[0]=='n':
41                 return d[2]
42
43 # Examples:
44 # parse(g1,['Sue','laughs'])
45 # parse(g1,['the','student','laughs'])
46 # parse(g1,['the','student','praises','the','beer'])
47 # parse(g1,['Bill','knows','Sue','laughs'])

```

With this code, we get sessions like this one (we have removed many output lines):

```

>>> from g1 import *
>>> from tdh import *
>>> parse(g1,['Bill','knows','Sue','laughs'])
expand S -> ['DP', 'VP']
expand DP -> ['D', 'NP']
...
scan laughs
0 (['Bill', 'knows', 'Sue', 'laughs'], ['S'])
1 (['Bill', 'knows', 'Sue', 'laughs'], ['DP', 'VP'])
2 (['Bill', 'knows', 'Sue', 'laughs'], ['Name', 'VP'])
3 (['Bill', 'knows', 'Sue', 'laughs'], ['Bill', 'VP'])
4 (['knows', 'Sue', 'laughs'], ['VP'])
5 (['knows', 'Sue', 'laughs'], ['V', 'DP', 'VP'])
6 (['knows', 'Sue', 'laughs'], ['knows', 'DP', 'VP'])
7 (['Sue', 'laughs'], ['DP', 'VP'])
8 (['Sue', 'laughs'], ['Name', 'VP'])
9 (['Sue', 'laughs'], ['Sue', 'VP'])
10 (['laughs'], ['VP'])
11 (['laughs'], ['V'])
12 (['laughs'], ['laughs'])
13 ([], [])
more? y
expand NP -> ['N']
...
scan laughs
0 (['Bill', 'knows', 'Sue', 'laughs'], ['S'])
1 (['Bill', 'knows', 'Sue', 'laughs'], ['DP', 'VP'])
2 (['Bill', 'knows', 'Sue', 'laughs'], ['Name', 'VP'])
3 (['Bill', 'knows', 'Sue', 'laughs'], ['Bill', 'VP'])
4 (['knows', 'Sue', 'laughs'], ['VP'])
5 (['knows', 'Sue', 'laughs'], ['V', 'CP'])
6 (['knows', 'Sue', 'laughs'], ['knows', 'CP'])
7 (['Sue', 'laughs'], ['CP'])
8 (['Sue', 'laughs'], ['C', 'S'])
9 (['Sue', 'laughs'], ['S'])
10 (['Sue', 'laughs'], ['DP', 'VP'])
11 (['Sue', 'laughs'], ['Name', 'VP'])
12 (['Sue', 'laughs'], ['Sue', 'VP'])
13 (['laughs'], ['VP'])
14 (['laughs'], ['V'])
15 (['laughs'], ['laughs'])
16 ([], [])
more? y
expand NP -> ['N']
...
expand D -> ['two']
'False'

```

We see there are two derivations of this string! Two points to notice:

- Neither derivation is intended! That is, neither derivation corresponds to an analysis of the string which we expect competent speakers of English to formulate. We got these derivations because the grammar `g1.py` is not enforcing the selection requirements of the verbs.
- If you check, you will see that the number of steps in each derivation is exactly the number of nodes in the corresponding derivation tree! More on this later.

2.2 TD backtrack parsing (rules used)

We could get the whole derivation tree from the history, but it is more convenient to use a smaller representation: the list of rules used in the derivation, in order. A very minor change in the previous program achieves this:

```

TOP-DOWN BACKTRACK CF PARSING( $G, i$ )
0  ds=[(i,S,[])] where S is the start category
1  while ds≠ [] and ds[0]≠([],[]):
2      ds=[d | ds[0] ⇒tdp d] + ds[1:]
3  if ds==[] then False else True

```

where the steps \Rightarrow_{tdp} are defined as follows:

$$(\text{expand}) \frac{\text{input}, X\alpha, h}{\text{input}, \beta\alpha, h(X \rightarrow \beta)} \quad \text{if } X \rightarrow \beta \quad (\text{scan}) \frac{w \text{ input}, w\alpha, h}{\text{input}, \alpha, h}$$

The implementation is easy:

```

1  """ file: tdp.py  stabler@ucla.edu
2     return the rules used in successful derivation
3  """
4  def tdpstep(g,(i,cs,p)): # compute all possible next steps from (i,cs)
5     if len(cs)>0:
6         cs1=cs[1:] # copy of predicted categories except cs[0]
7         p1 = p[1:] # copy of rewrites so far
8         nextsteps=[]
9         for (lhs,rhs) in g:
10            if lhs == cs[0]:
11                #print 'expand',lhs,'->',rhs # for trace
12                nextsteps.append((i,rhs+cs1,p1+[[lhs]+rhs]))
13            if len(i)>0 and i[0] == cs[0]:
14                #print 'scan',i[0] # for trace
15                i1=i[1:]
16                nextsteps.append((i1,cs1,p1))
17        return nextsteps
18    else:
19        return []
20
21  def derive(g,ds):
22    while ds != [] and not (ds[-1][0] == [] and ds[-1][1] == []):
23        d = ds.pop()
24        ds.extend(tdpstep(g,d))
25
26  def parse(g,i):
27    ds = [(i,['S'],[])]
28    while ds != []:
29        derive(g,ds)
30        if ds == []:
31            return 'False'
32        else:
33            d=ds.pop()
34            print 'll=',d[2]
35            ans = raw_input('another? ')
36            if len(ans)>0 and ans[0]=='n':
37                return d[2]
38
39  # Examples:
40  # parse(g1,['Sue','laughs'])
41  # parse(g1,['the','student','laughs'])
42  # parse(g1,['the','student','praises','the','beer'])
43  # parse(g1,['Bill','knows','Sue','laughs'])

```

With this code, we get sessions like this one:

```

>>> from g1 import *
>>> from tdp import *
>>> parse(g1,['Sue','laughs'])
ll= [['S', 'DP', 'VP'], ['DP', 'Name'], ['Name', 'Sue'], ['VP', 'V'], ['V', 'laughs']]
another? n
[['S', 'DP', 'VP'], ['DP', 'Name'], ['Name', 'Sue'], ['VP', 'V'], ['V', 'laughs']]
>>>

```

2.3 Pretty print list format trees

A standard format for trees is to use a list where the first element is the root and the rest of the list is a list of its subtrees. So for example, [S] is the tree containing just one node. [S DP VP] is the tree with root S and subtrees DP VP. When trees get larger, the list notation is less easy to read, so we can “pretty print” these trees in a more readable form:

```

1  """ file: pptree.py  stabler@ucla.edu
2  """
3  """
4  pretty print a tree given in list format
5  """
6  def pptree(n,t): # pretty print t indented n spaces
7      if isinstance(t,list) and len(t)>0:
8          print n*' ', t[0] # print root
9          for subtree in t[1:]: # then subtrees indented by 4
10             pptree(n+4,subtree)
11      else:
12          print '\n'+ ' '*n,t
13      """ example:
14      pptree(0,['TP', ['DP', ['John']], ['VP', ['V','praises']], ['DP', ['Mary']]])
15      pptree(0,[1, 2, [3, 4], [5, 6]])
16      """

```

With this code, we get sessions like this one:

```

>>> from pptree import *
>>> t = ['S', ['DP', ['Name', 'Sue']], ['VP', ['V', 'laughs']]]
>>> pptree(0,t)
S
  DP
    Name
      Sue
  VP
    V
      laughs
>>>

```

2.4 From rules to derivation trees in list format

Our parser returns the list of rules used in a successful derivation, listed in leftmost derivation order, sometimes called ‘preorder’ or LL order. Getting from the rules to derivation trees requires that we be able to recognize terminal productions. Here, we simply assume that the categories are disjoint from the nonterminals, and so if we are building a left branch, the terminal at the leaf will always be distinct from the category to be expanded next. With this convention, we do not need an independent way to tell whether a string is a category or terminal:

```

1  """ ll2dt.py
2  """
3  """
4  convert LL list of rules to list-format tree
5  """
6  def ll2dt(ll):
7      print 'll=',ll
8      if isinstance(ll,list):
9          ll.reverse()
10         return llr2dt(ll)
11     else:
12         return False
13
14 def llr2dt(llr):
15     t=llr.pop()
16     for i in range(1,len(t)):
17         if len(llr)>0 and len(llr[-1])>0 and t[i]==llr[-1][0]:
18             t[i]=llr2dt(llr) # recursive def most natural, and not too deep
19     return t
20
21 # example:
22 # ll=['S', 'DP', 'VP'], ['DP', 'Name'], ['Name', 'Sue'], ['VP', 'V'], ['V', 'laughs']]
23 # ll2dt(ll)
24 # pptree(0,ll2dt(parse(g1noe,['Sue','laughs'])))
25 # pptree(0,ll2dt(parse(g1noe,['the','student','laughs'])))

```

With this code, we get sessions like this one:

```

>>> from g1 import *
>>> from tdp import *
>>> from ll2dt import *
>>> ll = parse(g1,['Sue','laughs'])
ll= [['S', 'DP', 'VP'], ['DP', 'Name'], ['Name', 'Sue'], ['VP', 'V'], ['V', 'laughs']]
another n
>>> ll
[['S', 'DP', 'VP'], ['DP', 'Name'], ['Name', 'Sue'], ['VP', 'V'], ['V', 'laughs']]
>>> ll2dt(ll)
ll= [['S', 'DP', 'VP'], ['DP', 'Name'], ['Name', 'Sue'], ['VP', 'V'], ['V', 'laughs']]
['S', ['DP', ['Name', 'Sue']], ['VP', ['V', 'laughs']]]
>>>

```

2.5 From list format trees to NLTK trees

It is easy to convert our trees to the format used by the python NLTK library, and then we can use their graphical display utilities:

```

1  """ list2nlktree.py
2      convert a list-format tree to an NLTK tree
3  """
4  from nltk.util import in_idle
5  from nltk.tree import Tree
6  from nltk.draw import *
7
8  def list2nlktree(listtree):
9      if isinstance(listtree,list):
10         if listtree==[]:
11             return []
12         else:
13             subtrees=[list2nlktree(e) for e in listtree[1:]]
14             if subtrees == []:
15                 return listtree[0]
16             else:
17                 return Tree(listtree[0],subtrees)
18     else:
19         return listtree
20
21     """ With an NLTK tree, we can use NLTK tree display:
22     t0 = ['S', ['DP', ['Name', 'Sue']], ['VP', ['V', 'laughs']]]
23     list2nlktree(t0).draw()
24     TreeView(t0)
25     TreeView(list2nlktree(ll2dt(parse(g1,['Sue','laughs']))))
26     """

```

With this code, we get sessions like this one:

```

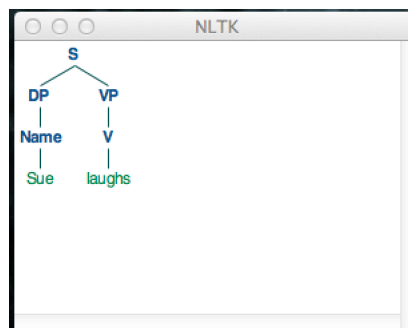
>>> from list2nlktree import *
>>> t = ['S', ['DP', ['Name', 'Sue']], ['VP', ['V', 'laughs']]]
>>> list2nlktree(t)
Tree('S', [Tree('DP', [Tree('Name', ['Sue'])]), Tree('VP', [Tree('V', ['laughs'])])])
>>>

```

The advantage of the nltk format is that we can then use the nltk graphical display tools. If we type

```
list2nlktree(t).draw()
```

We get a tree display like this:



2.6 Standard TD backtrack parsing

If you do not have NLTK:

```

1  """ file: tdp_setup.py
2     load grammar, td parser and tree utilities
3  """
4  from g1 import *
5  from tdp import *
6  from ll2dt import *
7  from pptree import *
8
9  def eg0():
10     t = ll2dt(parse(g1,['Sue','laughs']))
11     pptree(0,t)
12
13  def eg1(): # nb: g1 allows 2 unintended parses for this one!
14     t = ll2dt(parse(g1,['Bill','praises','the','student','on','Tuesday']))
15     pptree(0,t)
16
17  def rpp(): # simple read-parse-print
18     line = raw_input(': ')
19     i = line.split() # built-in python function, splits line at spaces
20     t = ll2dt(parse(g1,i))
21     pptree(0,t)

```

Now we can have sessions like this:

```

>>> rpp()
: the student praises the beer on Tuesday
ll= [['S', 'DP', 'VP'], ['DP', 'D', 'NP'], ['D', 'the'], ['NP', 'N'], ['N', 'student'], ['VP', 'V', 'DP', 'PP'], ['V',
another? n
ll= [['S', 'DP', 'VP'], ['DP', 'D', 'NP'], ['D', 'the'], ['NP', 'N'], ['N', 'student'], ['VP', 'V', 'DP', 'PP'], ['V',
S
  DP
    D
      the
    NP
      N
        student
  VP
    V
      praises
    DP
      D
        the
      NP
        N
          beer
    PP
      P
        on
      DP
        Name
          Tuesday
>>>

```

If you have NLTK, you can do the same thing but with a nicer, graphical display of successful parses:

```

1  """ file: tdp_setup2.py
2     load grammar, td parser and NLTK tree utilities
3  """
4  from g1 import *
5  from tdp import *
6  from ll2dt import *
7  from pptree import *

```



```
8 from nltk.util import in_idle
9 from nltk.tree import Tree
10 from nltk.draw import *
11 from list2nlktree import *
12
13 def eg0():
14     t = ll2dt(parse(g1,['Sue','laughs']))
15     if isinstance(t,list):
16         list2nlktree(t).draw()
17
18 def eg1(): # nb: g1 allows 2 unintended parses for this one!
19     t = ll2dt(parse(g1,['Bill','praises','the','student','on','Tuesday']))
20     list2nlktree(t).draw()
21
22 def rpp(): # simple read-parse-print
23     line = raw_input(': ')
24     i = line.split()
25     t = ll2dt(parse(g1,i))
26     if isinstance(t,list):
27         list2nlktree(t).draw() # NLTK tree drawing function
```

2.7 Assessment: TD time and space requirements

We have seen that getting from a TD recognizer to a parser is fairly easy: we simply record the steps in the derivation, in a way which has no influence on the course of the derivation but only on the output. The parser makes it easier to explore properties of the recognizer, so let's return now to consider more carefully the basic properties of the top-down backtracking recognizer.

2.7.1 Soundness, completeness, and search

It is easy to see (and not difficult to prove conclusively) that the rules \Rightarrow_{td} are sound and complete, for any context free grammar (CFG) in the following senses:

(soundness) For any CFG, if $(i, S) \Rightarrow_{td}^* (\epsilon, \epsilon)$ then $i \in L(G)$.

(completeness) For any CFG, if $i \in L(G)$, then $(i, S) \Rightarrow_{td}^* (\epsilon, \epsilon)$

On the other hand, the backtrack search has these undesirable properties:

(nontermination) For CFGs that have left recursion, the backtrack search can fail to terminate.

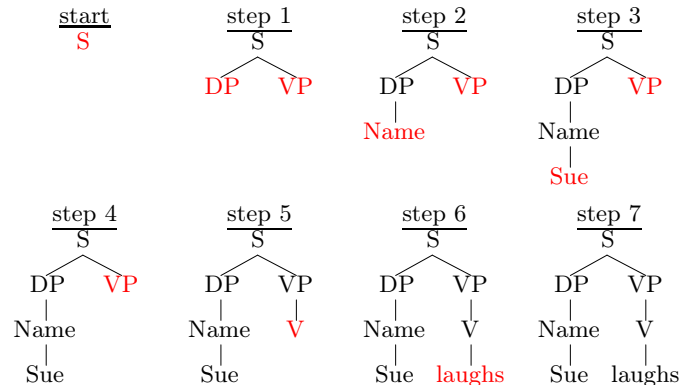
(inefficiency) Even for CFGs lacking left recursion, the number of steps required to recognize a string $i \in L(G)$ is not bounded by any polynomial function of the length of i .

These problems do not arise if the grammar has no left recursion and is deterministic, as discussed further in §2.7.7 below.

2.7.2 Predictiveness and incrementality

Although we have only printed out tree representations at the end of a parse, it is clear that we could have formulated the tree representations built at every step. When we do that, we see that the predicted categories at every step are, at every point, connected to the root. At every point, we have one, connected structure with its predicted categories in the recognizer's memory. This is a possible advantage for making sense of the fact that people normally understand what they hear on a word-by-word incremental basis [4, 35, 37].

Consider, for example, the seven step recognition of *Sue laughs* using our grammar g1. We can depict these steps by showing in red what is in the parser memory at each point:



At each point, the completed parts (shown in black) form connected, incrementally interpretable structures. With TD parsing, the completed structures always have this character, no matter what the grammar is. As we will see later, most other parsers do not have this property. Now that we have parsers, it is easier to see, as scientists or formal-language-theorists, what is going on in top-down recognition.

2.7.3 One parse at a time: garden pathing

It is often proposed that humans have difficulty with certain local ambiguities (or fail completely), resulting in the familiar "garden path" effects:¹ The following sentences exhibit extreme difficulty, but other less extreme variations in difficulty may also evidence the greater or less backtracking involved:

¹There are many studies of garden path effects in human language understanding. Some of the prominent early studies are the following: [3, 12, 14, 11, 7, 30].

- a. The horse raced past the barn fell
- b. Horses raced past barns fall
- c. The man who hunts ducks out on weekends
- d. Fat people eat accumulates
- e. The boat floated down the river sank
- f. The dealer sold the forgery complained
- g. Without her contributions would be impossible

In all of these cases, it seems that we initially are considering an analysis that must later be rejected, and sometimes it is difficult to recover from our mistake. This initially very plausible idea has not been easy to defend. One kind of problem is that some constructions which should involve backtracking are relatively easy: see for example [30, 13].

2.7.4 Right branching derivations, regularity, and space

Among the grammars that the TD recognizer can handle are some that define only regular languages. In particular, we know that if a grammar has only right branching (or only left branching), the language it defines is regular.² A regular language is one that can be recognized with only finite memory, so let's explore how much memory the TD recognizer needs for regular languages.

Our derivations divide the memory requirements into 3 parts:

(input buffer, recognizer memory, output buffer),

where the recognizer memory is what contains the predictions that must be remembered in order to complete the recognition correctly. The syntax acts only on contents of the recognizer memory: expanding an element using a rule of the grammar. Let's set aside the "input buffer" and the "output buffers" for a moment, and consider just the recognizer memory.

Since many grammars are non-deterministic, presenting choices in how to expand categories, we need another kind of memory too, the backtrack stack that remembers choices that have not yet been pursued. Let's set non-determinism aside for a moment too, returning to it in §2.7.7 below, and consider the recognizer memory.

First, consider how much memory is needed to parse grammars that have only simple right branching parses – where the left categories in binary productions are all lexical. That is, every rule has the one of the forms

$$\begin{array}{ll} A \rightarrow BC & \text{where } B \rightarrow w \text{ but not } B \rightarrow CD \\ A \rightarrow w & \text{for } w \text{ a vocabulary element} \\ A \rightarrow [] \end{array}$$

Grammars of this form can be regarded as finite state machines, where there is a transition from state A to C labeled with w iff there is a rule $A \rightarrow BC$ where $B \rightarrow w$; the start category of the grammar is the unique start state of the machine; and a state A is a final state iff there is a rule $A \rightarrow []$.

Consider the following grammar, for example:

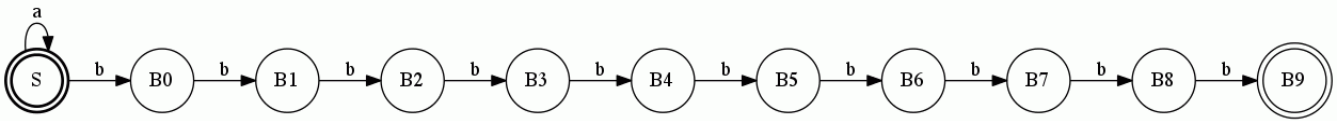
```

1  """ file: g2.py
2  """ a right branching grammar (no left recursion!)
3  """
4  g2 = [( 'S', [ 'A', 'S' ] ),
5        ( 'S', [ ] ),
6        ( 'S', [ 'B', 'B0' ] ),
7        ( 'B0', [ 'B', 'B1' ] ),
8        ( 'B1', [ 'B', 'B2' ] ),
9        ( 'B2', [ 'B', 'B3' ] ),
10       ( 'B3', [ 'B', 'B4' ] ),
11       ( 'B4', [ 'B', 'B5' ] ),
12       ( 'B5', [ 'B', 'B6' ] ),
13       ( 'B6', [ 'B', 'B7' ] ),
14       ( 'B7', [ 'B', 'B8' ] ),
15       ( 'B8', [ 'B', 'B9' ] ),
16       ( 'B9', [ ] ),
17       ( 'A', [ 'a' ] ),
18       ( 'B', [ 'b' ] ),
19  ]

```

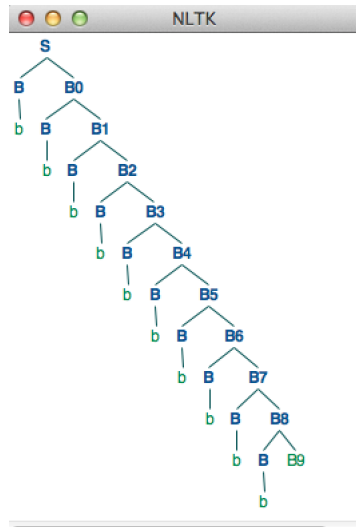
²In fact, a 'regular' or 'finite state' language is often defined to be one that can be generated by a right branching (or left branching) grammar. The equivalence between these grammars and 'finite state machines' is then an easy proof. See for example, [34, 36, 27, 18].

If we draw this grammar as a finite state machine it looks like this:



If we look at the derivation tree for b^{10} , we see that it has 31 nodes:

```
>>> from tdp_setup2 import *
>>> ll=parse(g2,['b','b','b','b','b','b','b','b','b','b','b'])
ll= [['S', 'B', 'B0'], ['B', 'b'], ['B0', 'B', 'B1'], ['B', 'b'], ['B1', 'B', 'B2'], ['B', 'b'], ['B2', 'B', 'B3'], [
another] n
>>> t=ll2dt(ll)
ll= [['S', 'B', 'B0'], ['B', 'b'], ['B0', 'B', 'B1'], ['B', 'b'], ['B1', 'B', 'B2'], ['B', 'b'], ['B2', 'B', 'B3'], [
>>> list2nlktree(t).draw()
```



And so we know that our TD recognition will require 31 steps too:

```
>>> from g2 import *
>>> from tdh import *
>>> parse(g2,['b','b','b','b','b','b','b','b','b','b','b'])
...
0 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['S'])
1 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B', 'B0'])
2 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['b', 'B0'])
3 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B0'])
4 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B', 'B1'])
5 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['b', 'B1'])
6 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B1'])
7 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B', 'B2'])
8 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['b', 'B2'])
9 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B2'])
10 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B', 'B3'])
11 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['b', 'B3'])
12 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B3'])
13 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B', 'B4'])
14 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['b', 'B4'])
15 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B4'])
16 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B', 'B5'])
17 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['b', 'B5'])
18 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B5'])
19 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B', 'B6'])
20 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['b', 'B6'])
21 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B6'])
22 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B', 'B7'])
23 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['b', 'B7'])
24 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B7'])
25 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B', 'B8'])
26 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['b', 'B8'])
```

```

27 ([ 'b' ], [ 'B8' ])
28 ([ 'b' ], [ 'B', 'B9' ])
29 ([ 'b' ], [ 'b', 'B9' ])
30 ([ ], [ 'B9' ])
31 ([ ], [ ])
more? n

```

It is easy to see that with simple right branching derivations like this one, the amount of space required in the stack of predicted categories is bounded by a finite constant. For this grammar, the bound is 2. We never need more space than that to remember what is required in the rest of the string. This makes sense, since the language is finite state.

2.7.5 Digression: Finite state models of human language

Evidence for non-finite-ateness of human languages. Let's write

HG for the grammars of human languages,
 HL for the set of human languages, defined by the grammars HG,
 Reg for the set of finite state (i.e. regular) grammars, and
 L(Reg) for the set of regular languages, defined by the grammars Reg.

These two ideas are widely accepted:

- a. (**HG \notin Reg**) The syntax (i.e. the grammar) used by speakers of human languages is not finite state.
- b. (**HL \notin L(Reg)**) The languages (i.e. the set of sentences generated by the grammars) used by speakers of human languages are not finite state.

It is quite possible to use a non-finite-state CFG to define a finite or finite state language. In fact, CFG definitions of finite and regular sets can be exponentially more succinct than the smallest regular, finite state grammar (Reg) definitions of the same languages [17]. Consequently, accepting (HG \notin Reg) does not entail (HL \notin L(Reg)). In the other direction though, we do have entailment: if (HL \notin L(Reg)), then obviously (HG \notin Reg).

- a. **Evidence for (HL \notin L(Reg)).** The evidence for this claim is usually given by taking some example languages, and arguing that they are not regular in a way that is naturally stated using the following important result:

(**Nerode-Myhill theorem**) Given any language L over alphabet Σ , for any string of words x , define the 'good finals' of x with respect to L:

$$\text{goodFinals}_L(x) = \{y \mid xy \in L\}.$$

And let's say $x \equiv y$ iff $\text{goodFinals}_L(x) = \text{goodFinals}_L(y)$, and for any string x let's write

$$[x]_L = \{y \mid x \equiv y\}.$$

Then L is regular iff the set of equivalence classes is finite:³

$$\{[x]_L \mid x \in \Sigma^*\} \text{ is a finite set.}$$

For English, the argument can be given by presenting an infinite sequence of sequences of words that all have different good finals, like this:

oysters
 oysters oysters
 oysters oysters oysters
 ...

³The Myhill-Nerode theorem is presented in Khoussainov&Nerode'01 [22, Thm 2.4.1]; in Salomaa'69 [33, Thm 5.4]; in Sakarovitch'09 [32, Props 3.11,3.12]; and in Hopcroft&Ullman'79 [19, §3.4] at the end of their second chapter on finite automata. The Myhill-Nerode theorem and Kleene's theorem are perhaps the most important results in formal language theory. Kleene's theorem says that the regular languages are the closure of the set $\{\emptyset\} \cup \{a \mid a \in \Sigma\}$ with respect to concatenation, union, and Kleene star. A nice, modern presentation of Kleene's theorem can be found in [32, Thm 2.1].

Now, the empirical claim is that, in the languages defined by the grammars of English speakers, no two elements of this sequence has the same good finals. For example, clearly *oysters eat* is a good sentence. And oysters are cannibalistic, so the sentence *oysters (that) oysters eat eat* is not only grammatical but sensible and true. Now the relevant facts for are claim are these:

eat \in goodFinals(oysters), but eat \notin goodFinals(oysters oysters) or any other seq in the list
eat eat \notin goodFinals(oysters) or any other seq except goodFinals(oysters oysters)

and similarly for every other pair. The expressions *oystersⁿeatⁿ* exhibit a kind of center-embedded dependency that cannot be defined with a regular grammar. This kind of argument is standard in the field (see for example [20]), but I do not regard it as persuasive by itself! It depends on judgements about center embeddings that become unacceptable already around depth 2! This argument is only persuasive when we add the consideration that $HG \notin \text{Reg}$, to which we turn now.

- b. **Evidence for ($HG \notin \text{Reg}$).** In the 1960's various kinds of evidence were offered for this claim. The strongest kind of evidence comes from the idea that finite state grammars cannot define the kinds of constituents which are abundantly evidenced by semantic considerations (defining meaningful units), syntactic arguments (defining grammars in which discontinuous dependencies are recognized appropriately) and by various sorts of psycholinguistic evidence. For example, various studies of recalling substrings of sentences show that constituents are easier to remember than substrings that straddle constituent boundaries.⁴ Miller'67 summarized the upshot of these studies, saying:

... constituent structure languages are more natural, easier to cope with, than regular languages... The hierarchical structure of strings generated by constituent-structure grammars is characteristic of much other behavior that is sequentially organized; it seems plausible that it would be easier for people than would the left-to-right organization characteristic of strings generated by regular grammars. [29]

Non-finite-state constituency is also evidenced by the perceptual effects demonstrated by the 'click' studies [24, 9], and many other phenomena.

Certain aspects of human languages appear to be finite state though:

- *The set of derivations defined by a CFG is a regular tree set (modulo renaming categories).* This was proven by Thatcher'67 [38]. We may return to this idea later, since we also have...
- *The set of derivations defined by a 'minimalist grammar' (MG) is a regular tree set (modulo renaming categories).* This idea follows directly from the results of Michaelis'98 [28]. Recently Kobele'11 and Graf'11 [23, 16] strengthened that insight by showing that the class of MG derivation trees is closed under intersection with regular tree languages.
- *Phonotactics is finite-state.* The SPE phonology of Chomsky&Halle'68 [5] used very powerful rewrite rules, but later analysis by Kaplan&Kay'94 [21] showed that finite state power suffices for almost everything, and this remains true in OT phonology [31, 2, 8] if reduplication is set aside.

2.7.6 Left branching derivations, regularity, and space

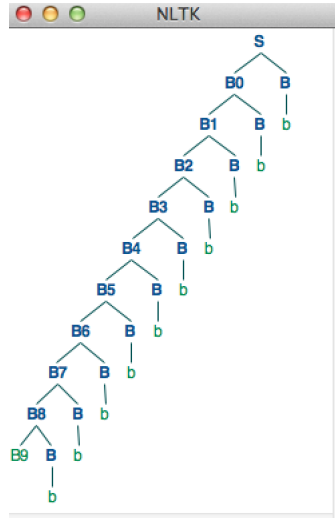
Now let's consider this grammar, in which all branching is to the left:

```

1  """ file: g3.py
2     a left branching grammar (no left recursion!)
3  """
4  g3 = [(('S', ['B0', 'B']),
5         ('B0', ['B1', 'B']),
6         ('B1', ['B2', 'B']),
7         ('B2', ['B3', 'B']),
8         ('B3', ['B4', 'B']),
9         ('B4', ['B5', 'B']),
10        ('B5', ['B6', 'B']),
11        ('B6', ['B7', 'B']),
12        ('B7', ['B8', 'B']),
13        ('B8', ['B9', 'B']),
14        ('B9', []),
15        ('B', ['b']))]
```

⁴See e.g. Fodor, Bever and Garrett'74 [10] for a good review of this early work.

This grammar accepts b^{10} too, with this derivation:



If we look at the parser memory requirements for the 31 step derivation, though, we get a very different picture (running off the right edge of the page – but it should be clear what is happening!)

```
>>> from g3 import *
>>> from tdh import *
>>> parse(g3,['b','b','b','b','b','b','b','b','b','b','b'])
expand S -> ['B0', 'B']
expand B0 -> ['B1', 'B']
...
0 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['S'])
1 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B0', 'B'])
2 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B1', 'B', 'B'])
3 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B2', 'B', 'B', 'B'])
4 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B3', 'B', 'B', 'B', 'B'])
5 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B4', 'B', 'B', 'B', 'B', 'B'])
6 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B5', 'B', 'B', 'B', 'B', 'B', 'B'])
7 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B6', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
8 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B7', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
9 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B8', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
10 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B9', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
11 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
12 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['b', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
13 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
14 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['b', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
15 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
16 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['b', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
17 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
18 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['b', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
19 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
20 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['b', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
21 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
22 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['b', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
23 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
24 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['b', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
25 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
26 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['b', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
27 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
28 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['b', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
29 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
30 (['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'], ['b', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B', 'B'])
31 ([], [])
```

Notice that the number of elements in the parser memory increases as we go down the left branch. The longer the left branch is, using any left branching grammar, the more parser memory required, without bound. So although left branching grammars define only finite state languages, the TD recognizer requires unbounded memory to handle them. Do humans have great difficulty with left-branching? It seems not [15].

2.7.7 Determinism: time, space, nontermination

As we mentioned in the previous chapter, the TD recognizer will fail to terminate if the grammar has left recursion, and even when it doesn't, the number of steps the parser takes can be on the order of k^n for some $k > 1$, as shown in [1, p.299]. This is because of nondeterminism! What can we do?

'Lookahead' can reduce local ambiguity, but English is not $LL(k)$ for any k .

Consider the following simplistic grammar for sentences like *the dog [that I told you about] barks*:

$$\begin{aligned} S &\rightarrow DP VP \\ DP &\rightarrow D NP \\ D &\rightarrow \text{the} \\ NP &\rightarrow NP CP \\ NP &\rightarrow N \\ N &\rightarrow \text{dog} \\ CP &\rightarrow \text{that I told you about} \\ VP &\rightarrow \text{barks} \end{aligned}$$

A TD recognizer processing this example gets to the point:

(dog that I told you about barks, NP VP)

Here, by looking 2 words ahead, we see *that*, and so we might think that this lets us know that we should use the first NP rule:

(dog that I told you about barks, NP CP VP)

But here again, looking 2 words ahead, we still see *that*, and now it would be a mistake to use the first NP rule. Clearly, if there is a finite limit k on how many words ahead we can look, we will not be able to decide which rule to use. A language is said to be $LL(k)$ if we can always decide which rule to use with k symbols of lookahead, so this and many other constructions show that English is not $LL(k)$ for any k .

Quickly surveying some of the other cases of local ambiguity which could present problems:

Subject-verb agreement. In simple English clauses, the subject and verb agree, even though the subject and verb can be arbitrarily far apart:

- a. The deer {are, is} in the field
- b. The deer, the guide claims, {are, is} in the field
- c. The deer who prance about are/*is in the field
- d. The deer who prances about *are/is in the field

Bound pronoun agreement. The number of the embedded subject is unspecified in the following sentence:

- a. I expect the deer to admire the pond.

But in similar sentences it can be specified by binding relations:

- b. I expect the deer to admire {itself,themselves} in the reflections of the pond.

Head movement can move a head away from disambiguating context:

- a.
 - i. Have the students take the exam!
 - ii. Have the students taken the exam?
- b.
 - i. Is the block sitting in the box?
 - ii. Is the block sitting in the box red?

A' movement can create 'doubtful gaps', separated from disambiguating context:

- a. who_i did you expect t_i to make a potholder
- b. who_i did you $_j$ expect PRO_j to make a potholder for t_i

English has intractable global ambiguity, like other human languages.

A context free grammar can have infinitely many derivations for one string. For example, this grammar has one string but infinitely many derivations:

$$S \rightarrow S \quad S \rightarrow a.$$

How much structural ambiguity do sentences of human languages really have? We can get a first impression of how serious the structural ambiguity problem is by looking at simple artificial grammars for these constructions.

1. **PP attachment** [_{VP} **V D N PP1 PP2 ...**]. Consider a grammar with these rules:

$$\begin{aligned} VP &\rightarrow VP PP \\ NP &\rightarrow NP PP \end{aligned}$$

If we calculate how many structures are allowed in strings when there are n PPs, we find:⁵

$$\begin{array}{r} n = \\ \#trees = \end{array} \left| \begin{array}{cccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 2 & 5 & 14 & 132 & 469 & 1430 & 4862 & 16796 & 1053686 & \dots \end{array} \right.$$

2. **N compounds** [_N **N N**]. These can be generated by the rule

$$N \rightarrow N N$$

If we calculate how many structures are allowed in strings when there are n Ns, we find the same series as for PPs,

$$\begin{array}{r} n \\ \#trees \end{array} \left| \begin{array}{cccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 1 & 1 & 2 & 5 & 14 & 42 & 132 & 429 & 1420 & 4862 \end{array} \right.$$

3. **Coordination** [_X **X and X**]. If coordination is always binary, the ambiguities are similar to the previous cases. To make things interesting, suppose we assume that in addition to binary coordination, English allows arbitrary lists of coordinates, with a rule like this:

$$NP \rightarrow NP \text{ (and NP)}^*$$

Note that this is not a standard context-free rule. It is equivalent to a grammar with infinitely many rules:

$$\begin{aligned} NP &\rightarrow NP \text{ and NP} \\ NP &\rightarrow NP \text{ and NP and NP} \\ NP &\rightarrow NP \text{ and NP and NP and NP} \\ &\dots \end{aligned}$$

Then with n NPs, the number of structures allowed grows like this:

$$\begin{array}{r} n \\ \#trees \end{array} \left| \begin{array}{cccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 1 & 1 & 3 & 11 & 45 & 197 & 903 & 4279 & 20793 & 103049 \end{array} \right.$$

Do these kinds of ambiguity-creating elements really crop up in substantial numbers? Consider the 9 PPs and other modifiers in this phrase that is engraved in stone at the UBC Museum of Anthropology:

The government of the province of British Columbia has contributed to the building of this museum to honour the centenary of the province's entry into confederation in the conviction that it will serve and bring pleasure to the people of the nation.

Or consider the following example

Enriched with minerals and vitamins, the purified soybean meal is colored, flavored, pressed, shaped and cut into bits that look and taste like bacon chips or strips, pork sausage, ground beef, sliced ham or chicken and are cheaper and just as nourishing as the real thing. (from the American Publishing House for the Blind (APHB) corpus) [25]

⁵This is the Catalan series $Cat(n) = \binom{2}{n} - \binom{2n}{n-1}$. Its growth is not bounded by any polynomial function of n . Classic discussions appear in [6, 26].

2.7.8 Implementing a timer, counting number of steps

To explore the number of steps a grammar takes, it is easy to add a ‘timer’, a count of the number of steps, to any of our previous programs. One way to do this in python is with a ‘global variable’, that is, a variable that scopes over the entire computation. Here we add a timer and print it out with lines 6,13,17,24,25,31:

```

1  """ file: tdt.py
2      a simple top-down backtrack CF recognizer,
3      with a step-counter ('time', measured in steps)
4  """
5  def tdstep(g,(i,cs)): # compute all possible next steps from (i,cs)
6      global steps
7      if len(cs)>0:
8          cs1=cs[1:] # copy of predicted categories except cs[0]
9          nextsteps=[]
10         for (lhs,rhs) in g:
11             if lhs == cs[0]:
12                 #print 'expand',lhs,'->',rhs # for trace
13                 steps = steps+1
14                 nextsteps.append((i,rhs+cs1))
15         if len(i)>0 and i[0] == cs[0]:
16             #print 'scan',i[0] # for trace
17             steps = steps+1
18             nextsteps.append((i[1:],cs1))
19         return nextsteps
20     else:
21         return []
22
23 def recognize(g,i):
24     global steps
25     steps = 0
26     ds = [(i,['S'])]
27     while ds != [] and ds[-1] != ([],[]):
28         #showDerivations(ds) # for trace
29         d = ds.pop()
30         ds.extend(tdstep(g,d))
31     print 'steps=',steps
32     if ds == []:
33         return False
34     else:
35         #showDerivations(ds) # for trace
36         return True
37
38 # Examples:
39 # recognize(g1,['Sue','laughs'])
40 # recognize(g1,['Bill','knows','that','Sue','laughs'])
41 # recognize(g1,['Sue','laughed'])
42 # recognize(g1,['the','student','from','the','university','praises','the','beer','on','Tuesday'])
43 # recognize(g1,['the','student','from','the','university','praises','the'])
44 # recognize(g1,['Sue','knows','that','Maria','laughs'])

```

Adding a timer to our basic recognizer makes it clear that backtracking to find alternative parsers also takes many steps! The changes to `tdp.py` are just the lines 6,14,18,31,32,36:

```

1  """ file: tdpt.py stabler@ucla.edu
2      return the rules used in succesful derivation
3      with a step-counter ('time', measured in steps)
4  """
5  def tdpstep(g,(i,cs,p)): # compute all possible next steps from (i,cs)
6      global steps
7      if len(cs)>0:
8          cs1=cs[1:] # copy of predicted categories except cs[0]
9          p1 = p[:] # copy of rewrites so far
10         nextsteps=[]
11         for (lhs,rhs) in g:
12             if lhs == cs[0]:
13                 #print 'expand',lhs,'->',rhs # for trace
14                 steps = steps+1
15                 nextsteps.append((i,rhs+cs1,p1+[[lhs]+rhs]))
16         if len(i)>0 and i[0] == cs[0]:
17             #print 'scan',i[0] # for trace
18             steps = steps+1
19             i1=i[1:]
20             nextsteps.append((i1,cs1,p1))

```

```

21     return nextsteps
22 else:
23     return []
24
25 def derive(g,ds):
26     while ds != [] and not (ds[-1][0] == [] and ds[-1][1] == []):
27         d = ds.pop()
28         ds.extend(tdpstep(g,d))
29
30 def parse(g,i):
31     global steps
32     steps = 0
33     ds = [(i,['S'],[])]
34     while ds != []:
35         derive(g,ds)
36         print 'steps=',steps
37         if ds == []:
38             return 'False'
39         else:
40             d=ds.pop()
41             print 'll=',d[2]
42             print 'parse length=',len(d[2])
43             ans = raw_input('another? ')
44             if len(ans)>0 and ans[0]=='n':
45                 return d[2]
46
47 # Examples:
48 # parse(g1,['Sue','laughs'])
49 # parse(g1,['the','student','laughs'])
50 # parse(g1,['the','student','praises','the','beer'])
51 # parse(g1,['Bill','knows','Sue','laughs'])

```

2.8 Conclusions: We have to revise our goal!

The previous section established the following things:

- Computing a list of all parses is not possible for CF grammars in general, since a single string can have infinitely many parses!
- Even when every string has finitely many parses, computing a list of all parses is not tractable for CF grammars in general, since there can be exponentially many of them! In fact, this happens in human languages...
- In human languages, given standard assumptions about constituency, no polynomial function of n bounds the number of parses of strings of length n .

These facts require that we change our idea about our main question, from page 12. It is not reasonable to ask how people compute (all) grammatical structures from orthographic or phonetic representations, since it is not plausible that they do! There are a number of possible responses to this worry. Making our assumptions explicit, consider these alternatives:

- Q1a. Since humans cannot map orthographic or phonetic input to complete, explicit analyses, they must use some more compact representation of (all) those structures. What algorithms can compute those?⁶
- Q1b. It is unreasonable to assume that we compute all the structures of the sentences we hear, and it is well known that people systematically fail to notice many ambiguities in sentences they hear. SO language users must somehow rank candidate analyses, implicitly (and perhaps probabilistically), and then restrict their attention to the most probable one(s), in context. What algorithms can do that?
- Q1c. Obviously, each language user has limited memory. There must be some particular bound k on the number of elements that can be remembered at any time in analyzing a sentence – a bound on the parser memory. When

⁶When computer scientists say that context free grammars can be efficiently parsed, they usually have this kind of idea in mind. They do not mean that all the parse structures can be listed efficiently. Instead, their parsers output something other than explicit trees, or explicit lists of rules used in each parse. We will consider such alternatives below, when we study chart parsers and consider their plausibility as models of human sentence processing.

the parser reaches that bound, the derivation must just crash, ceasing to be available for consideration.⁷ So we might ask: what algorithms do this in a human-like way?

There are many other responses to the basic facts about human sentence recognition, each of which poses slightly different questions. We will not try to decide among them here. Certain core assumptions are shared by all of them:

- (i) parsing involves finding derivations from the grammar (i.e. from some finite memory of linguistic structure),
- (ii) to a first approximation, derivations are built up incrementally from left to right
- (iii) to a first approximation, meaning is calculated incrementally from left to right

We can focus on achieving these in various ways, postponing decisions about which of Q1a-Q1c is the right perspective.

⁷In fact, there are reasons to think that the number of elements that can be remembered depends in part on what those elements are. Remembering 3 clearly distinct things may be easier than remembering 3 very similar things. This may be one reason that sentences like these are so difficult:

- Buffalo buffalo buffalo Buffalo buffalo
with the syntax of: [California girls] like [Pacific waves]
- Dogs dogs dog dog dogs
with the syntax of: [Mice (that) cats chase eat cheese]

There is a big literature exploring memory and interference effects of various kinds in parsing [].

Exercises: Memory requirements for our TD recognizer

Get `tdh.py` and copy it to `tdh<your-initials>.py`. Then modify the program as follows (small changes!)...

1. Instead of just the steps in the parse, modify the program to also print out the number of states in parser memory, like this:

```
0 1 (['Sue', 'laughs'], ['S'])
1 2 (['Sue', 'laughs'], ['DP', 'VP'])
2 2 (['Sue', 'laughs'], ['Name', 'VP'])
3 2 (['Sue', 'laughs'], ['Sue', 'VP'])
4 1 (['laughs'], ['VP'])
5 1 (['laughs'], ['V'])
6 1 (['laughs'], ['laughs'])
7 0 ([], [])
```

Here each line has the number of the step and then the number of elements in parser memory at that step.

2. Now make a second change. Instead of printing out just the steps in the proof, print out the number of states in parser memory and the number of derivations in the backtrack stack just before the current step was taken, like this:

```
0 1 1 (['Sue', 'laughs'], ['S'])
1 2 1 (['Sue', 'laughs'], ['DP', 'VP'])
2 2 1 (['Sue', 'laughs'], ['Name', 'VP'])
3 2 3 (['Sue', 'laughs'], ['Sue', 'VP'])
4 1 4 (['laughs'], ['VP'])
5 1 4 (['laughs'], ['V'])
6 1 4 (['laughs'], ['laughs'])
7 0 4 ([], [])
```

Here each line has the number of the step, the number of elements in parser memory at that step (as before), and then the number of elements in the backtrack stack at each point.

We can check this with `td.py`, since that program lists all the derivations at every point. For example, at the last step in the parse of `Sue laughs`, we can see that there are, in fact, 4 derivations in the backtrack stack:

```
...
-----
scan laughs
0 ( , )
1 ( Sue laughs , Bill VP )
2 ( Sue laughs , NP VP )
3 ( Sue laughs , D NP VP )
-----
True
```

3. Using `g1.py` and your answer to the previous question:
 - a. What sentence of 10 words or less requires the most parser memory? That is, find a sentence that, at some point in calculating one of its parses, has n elements in parser memory for n as high as you can get it. Tell me what the sentence is and what n is in a comment at the bottom of your file `tdh<your-initials>.py`. (I will check your answer!)
 - b. In a sentence or two, explain why your answer to the previous question maximizes parser memory use – again put this in a comment at the bottom of your file `tdh<your-initials>.py`.
 - c. What sentence of 10 words or less requires the most backtrack memory? That is, find a sentence that, at some point in calculating one of its parses, has n derivations elements in its backtrack stack for n as high as you can get it. Tell me what the sentence is, and what n is in a comment at the bottom of your file `tdh<your-initials>.py`. (I will check your answer!)
 - d. In a sentence or two, explain why your answer to the previous question maximizes backtrack stack use – again put this in a comment at the bottom of your file `tdh<your-initials>.py`.

So you will have a modified parser, with 4 short answers to problem 3 at the bottom of the file. Email me your work by midnight next Wednesday. Monday is Martin Luther King, Jr. day. (If you have trouble, first do the easy steps 1, 3a, 3b. Then go back to 2, 3c, 3d. I will give some hints in class too.)

More exercises: Time requirements for our TD recognizer

*** Not assigned this year! but you can do these for fun if you want ***

4. Modify `tdp.py` to produce the program `tdpt.py` that is listed in §2.7.8 above, name it `tdpt<YOURINITIALS>.py`, and put your name at the top of the file too.
5. At the top of this file, add a grammar with the name `g4` with the following rules:

$$\begin{aligned} S &\rightarrow aSS \\ S &\rightarrow \epsilon \end{aligned}$$

6. Use your `tdpt.py` to see how many steps it takes to find all parses of each of these sentences:

a, aa, aaa, aaaa, aaaaa, aaaaaa, aaaaaaa

Cut and paste the timer counts for each of these sentences into a comment at the bottom of your file.

7. Also, in the same comment at the bottom of your file, answer this question: For which values of n does the number of steps required to find all parses exceed 2^n ? Briefly defend your answer as persuasively as you can.
8. Optionally: sketch a proof that establishes the correctness of your answer to 7.

References

- [1] AHO, A. V., AND ULLMAN, J. D. *The Theory of Parsing, Translation, and Compiling. Volume 1: Parsing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1972.
- [2] ALBRO, D. M. *Studies in Computational Optimality Theory, with Special Reference to the Phonological System of Malagasy*. PhD thesis, UCLA, 2005.
- [3] BEVER, T. G. The cognitive basis for linguistic structures. In *Cognition and the Development of Language*, J. Hayes, Ed. Wiley, NY, 1970.
- [4] CHAMBERS, C. G., TANENHAUS, M. K., EBERHARD, K. M., FILIP, H., AND CARLSON, G. N. Actions and affordances in syntactic ambiguity resolution. *Journal of Experimental Psychology: Learning, Memory and Cognition* 30, 3 (2004), 687–696.
- [5] CHOMSKY, N., AND HALLE, M. *The Sound Pattern of English*. MIT Press, Cambridge, Massachusetts, 1968.
- [6] CHURCH, K., AND PATIL, R. How to put the block in the box on the table. *Computational Linguistics* 8 (1982), 139–149.
- [7] CRAIN, S., AND STEEDMAN, M. On not being led up the garden path. In *Natural Language Parsing*, D. Dowty, L. Karttunen, and A. Zwicky, Eds. Cambridge University Press, NY, 1985.
- [8] EISNER, J. Doing OT in a straightjacket. Presented at UCLA, 1999.
- [9] FODOR, J. A., AND BEVER, T. G. The psychological reality of linguistic segments. *Journal of Verbal Learning and Verbal Behavior* 4 (1965), 414–420.
- [10] FODOR, J. A., BEVER, T. G., AND GARRETT, M. F. *The Psychology of Language: An Introduction to Psycholinguistics and Generative Grammar*. McGraw-Hill, NY, 1974.
- [11] FORD, M., BRESNAN, J., AND KAPLAN, R. M. A competence-based theory of syntactic closure. In *The Mental Representation of Grammatical Relations*, J. Bresnan, Ed. MIT Press, Cambridge, Massachusetts, 1982.
- [12] FRAZIER, L. *On Comprehending Sentences: Syntactic Parsing Strategies*. PhD thesis, University of Massachusetts, Amherst, 1978.
- [13] FRAZIER, L., AND CLIFTON, C. *Construal*. MIT Press, Cambridge, Massachusetts, 1996.
- [14] FRAZIER, L., AND RAYNER, K. Making and correcting errors during sentence comprehension. *Cognitive Psychology* 14 (1982), 178–210.
- [15] FRAZIER, L., AND RAYNER, K. Parameterizing the language system: Left- vs. right-branching within and across languages. In *Explaining Linguistic Universals*, J. A. Hawkins, Ed. Blackwell, NY, 1988, pp. 247–279.
- [16] GRAF, T. Closure properties of minimalist derivation tree languages. In *Logical Aspects of Computational Linguistics, LACL'11* (2011).
- [17] HARTMANIS, J. On the succinctness of different representations of languages. *SIAM Journal on Computing* 9 (1980), 114–120.

- [18] HOPCROFT, J. E., MOTWANI, R., AND ULLMAN, J. D. *Introduction to Automata Theory, Languages and Computation (2nd Edition)*. Addison-Wesley, Reading, Massachusetts, 2000.
- [19] HOPCROFT, J. E., AND ULLMAN, J. D. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.
- [20] JÄGER, G., AND ROGERS, J. Formal language theory: Refining the Chomsky hierarchy. *Philosophical Transactions of the Royal Society B* 367 (2012), 1956–1970.
- [21] KAPLAN, R., AND KAY, M. Regular models of phonological rule systems. *Computational Linguistics* 20 (1994), 331–378.
- [22] KHOUSSAINOV, B., AND NERODE, A. *Automata Theory and Its Applications*. Birkhäuser, Boston, 2001.
- [23] KOBELE, G. M. Minimalist tree languages are closed under intersection with recognizable tree languages. In *Logical Aspects of Computational Linguistics, LACL'11* (2011), S. Pogodalla and J.-P. Prost, Eds., pp. 129–144.
- [24] LADEFOGED, P., AND BROADBENT, D. Perception of sequence in auditory events. *Quarterly Journal of Experimental Psychology* 13 (1960), 162–170.
- [25] LANGENDOEN, D. T. Limitations on embedding in coordinate structures. *Journal of Psycholinguistic Research* 27 (1998), 235–259.
- [26] LANGENDOEN, D. T., MCDANIEL, D., AND LANGSAM, Y. Preposition-phrase attachment in noun phrases. *Journal of Psycholinguistic Research* 18 (1989), 533–548.
- [27] LEWIS, H. R., AND PAPADIMITRIOU, C. H. *Elements of the Theory of Computation*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.
- [28] MICHAELIS, J. Derivational minimalism is mildly context-sensitive. In *Proceedings, Logical Aspects of Computational Linguistics, LACL'98* (NY, 1998), Springer, pp. 179–198.
- [29] MILLER, G. A. Project grammarama. In *Psychology of Communication*. Basic Books, NY, 1967.
- [30] PRITCHETT, B. L. *Grammatical Competence and Parsing Performance*. University of Chicago Press, Chicago, 1992.
- [31] RIGGLE, J. *Generation, Recognition, and Learning in Finite State Optimality Theory*. PhD thesis, University of California, Los Angeles, 2004.
- [32] SAKAROVITCH, J. *Elements of Automata Theory*. Cambridge University Press, NY, 2009.
- [33] SALOMAA, A. *The Theory of Automata*. Pergamon, NY, 1969.
- [34] SALOMAA, A. *Formal Languages*. Academic, NY, 1973.
- [35] SHIEBER, S., AND JOHNSON, M. Variations on incremental interpretation. *Journal of Psycholinguistic Research* 22 (1994), 287–318.
- [36] SIPSER, M. *Introduction to the Theory of Computation*. PWS Publishing, Boston, 1997.
- [37] STABLER, E. P. The finite connectivity of linguistic structure. In *Perspectives on Sentence Processing*, C. Clifton, L. Frazier, and K. Rayner, Eds. Lawrence Erlbaum, Hillsdale, New Jersey, 1994, pp. 245–266.
- [38] THATCHER, J. W. Characterizing derivation trees of context-free grammars through a generalization of finite automata theory. *Journal of Computer and System Sciences* 1, 4 (1967), 317 – 322.

Chapter 3 Beam instead of backtrack, and more alternatives

A beam parser is one where the search is restricted to a finite collection, a finite ‘beam’ of options at each point [11, 10, 7, 6, 13, 12, 8]. One variety of these strategies is called ‘ k -best’: the search is limited to the k most probable parses. A simple beam parser is introduced here. It provides a kind of top-down analysis that can ‘handle’ left recursion, in a sense, though it does so in a manner that we will see is less than satisfactory. The basic idea is simply to always evaluate the simplest alternative parse (or the parse that is ‘most probable’ in some other sense), within a bound set by a parameter k . Any parses with probability less than k are discarded.

3.1 The TD beam recognizer

The data structure we need for the beam can be thought of as a sorted list, sorted by probability, from which we always pop one of the elements with highest probability. Since this list must be sorted, it is not a stack, but is sometimes called a *priority queue* or *heap*.¹ The other ingredient we need is some way of determining the probability of each parse. We will return to this question later, but for now let’s just say that each option is equally likely. That is, whenever a parse with probability p can expand in n ways, we assign each of the n possible parses probability p/n . And in order to make sure that the beam stays a reasonable size, we will set a finite bound k on the minimum probability parse that we want to consider.

```
TOP-DOWN BEAM CF RECOGNITION( $G, i, k$ )
0  beam=[(1,i,S)], a priority queue, where S is the start category
1  while beam≠ [] and max(bean)≠(p,[],[]) (any p):
2      (p0,i0,cs0)=pop(bean), the maximum element
3      nextsteps=[d | (i0,cs0) ⇒td d]
4      p1=p0* $\frac{1}{\text{len}(\text{nextsteps})}$ 
5      if p >  $k$ :
6          for each (i1,cs1) in nextsteps, push (p1,i1,cs1) onto beam
7  if beam==[] then False else True
```

where the steps \Rightarrow_{td} are unchanged:

$$\text{(expand)} \quad \frac{\text{input}, X\alpha}{\text{input}, \beta\alpha} \quad \text{if } X \rightarrow \beta \qquad \text{(scan)} \quad \frac{w \text{ input}, w\alpha}{\text{input}, \alpha}$$

If k is negative, then all derivations will be kept in the beam, as was done in top-down recognition. But the beam recognizer always expands one of the maximum probability parses, so unlike the top-down recognizer, the beam recognizer will always terminate on a grammatical input. Like the top-down parser, though, if k is negative, the recognizer can fail to terminate on ungrammatical inputs if the grammar has left recursion. (And even with positive k , with some unusual left recursive grammars, the recognizer fail to terminate – In what cases can this happen?)

3.2 Implementing the TD beam recognizer

Python provides a priority queue, a heap. To use this data structure, we load the basic capabilities with the command:

```
>>> import heapq
```

That command adds the `heapq` library, which allows us to create heaps. Any list can be converted into a heap:

¹The implementation of these structures is interesting, a standard topic in classes on data structures. See, for example, [1, §6].

```
>>> l1=[4,1,3,2]
>>> heapq.heapify(l1)
```

Now we can pop the minimum element, and push new elements:

```
>>> x = heapq.heappop(l1)
>>> x
1
>>> l1
[2, 4, 3]
>>> y = heapq.heappop(l1)
>>> y
2
>>> x = heapq.heappop(l1)
>>> x
3
>>> l1
[4]
heapq.heappush(l1,-3)
>>> l1
[-3, 4]
heapq.heappush(l1,5)
>>> l1
[-3, 4, 5]
```

We can use this for our probabilities, but since we want to pop the maximum probability elements, we keep the probability negated, and we let represent our derivations with triples

(-probability, remaining input, predicted categories).

The *expand* and *scan* steps can remain unchanged, but whenever there are n possible next steps from a parse with probability p , we assign each possibility p/n .

In the following implementation, the function `tdstep` is unchanged from the top-down parser `td.py`. The only changes are in `recognize`, especially in the lines 25, 27 and 31-36:

```
1  """ file: tdb.py
2     A simple top-down beam CF recognizer.
3     The minor changes from td.py are labeled "for beam"
4  """
5  import heapq
6
7  def tdstep(g,(i,cs)): # compute all possible next steps from (i,cs)
8      if len(cs)>0:
9          cs1=cs[1:] # copy of predicted categories except cs[0]
10         nextsteps=[]
11         for (lhs,rhs) in g:
12             if lhs == cs[0]:
13                 print 'expand',lhs,'->',rhs # for trace
14                 nextsteps.append((i,rhs+cs1))
15         if len(i)>0 and i[0] == cs[0]:
16             print 'scan',i[0] # for trace
17             nextsteps.append((i[1:],cs1))
18         return nextsteps
19     else:
20         return []
21
22 def recognize(g,i,k):
23     beam = [(-1.,i,['S'])] # probability 1., negated
24     heapq.heapify(beam) # make list into a "min-heap"
25     while beam != [] and not(min(beam)[1]==[] and min(beam)[2]==[]):
26         (prob0,i0,cs0) = heapq.heappop(beam)
27         print 'popped',(prob0,i0,cs0) # for trace
28         nextsteps = tdstep(g,(i0,cs0))
29         print 'next steps=',nextsteps
30         if len(nextsteps) > 0:
31             prob1 = prob0/float(len(nextsteps))
32             if -(prob1) > k:
33                 for (i1,cs1) in nextsteps:
34                     heapq.heappush(beam,(prob1,i1,cs1))
35                     print 'pushed',(prob1,i1,cs1) # for trace
36         print '|beam|=',len(beam) # for trace
37     if beam == []:
38         return False
```

```

39     else:
40         return True
41
42 # Examples:
43 # recognize(g1,['Sue','laughs'],0.05)
44 # recognize(g1,['Sue','laughs'],0.5)
45 # recognize(g1,['Bill','knows','that','Sue','laughs'],0.10)
46 # recognize(g1,['Bill','knows','that','Sue','laughs'],0.01)

```

3.3 Implementing the TD beam parser

It is easy to make the same changes to the top-down parser `tdp.py` to get this top-down beam parser:

```

1  """ file: tdbp.py  stabler@ucla.edu
2     beam parser
3  """
4  import heapq
5
6  def tdpstep(g,(ws,cs,p)): # compute all possible next steps from (ws,cs)
7      if len(cs)>0:
8          cs1=cs[1:] # copy of predicted categories except cs[0]
9          p1 = p[:] # copy of rules used so far
10         nextsteps=[]
11         for (lhs,rhs) in g:
12             if lhs == cs[0]:
13                 #print 'expand',lhs,'->',rhs # for trace
14                 nextsteps.append((ws,rhs+cs1,p1+[[lhs]+rhs]))
15         if len(ws)>0 and ws[0] == cs[0]:
16             #print 'scan',ws[0] # for trace
17             ws1=ws[1:]
18             nextsteps.append((ws1,cs1,p1))
19         return nextsteps
20     else:
21         return []
22
23  def derive(g,beam,k):
24     while beam != [] and not (min(beam)[1] == [] and min(beam)[2] == []):
25         (prob0,ws0,cs0,p0) = heapq.heappop(beam)
26         nextsteps = tdpstep(g,(ws0,cs0,p0))
27         #print 'nextsteps=',nextsteps
28         if len(nextsteps) > 0:
29             prob1 = prob0/float(len(nextsteps))
30             if -(prob1) > k:
31                 for (ws1,cs1,p1) in nextsteps:
32                     heapq.heappush(beam,(prob1,ws1,cs1,p1))
33                     #print 'pushed',(prob1,ws1,cs1) # for trace
34             #print '|beam|=',len(beam) # for trace
35
36  def parse(g,ws,k):
37     beam = [(-1.,ws,['S'],[])]
38     heapq.heapify(beam) # make list of derivations into a "min-heap"
39     while beam != []:
40         derive(g,beam,k)
41         if beam == []:
42             return 'False'
43         else:
44             d=heapq.heappop(beam)
45             print 'll=',d[3]
46             ans = raw_input('another? ')
47             if len(ans)>0 and ans[0]=='n':
48                 return d[3]
49
50 # Examples:
51 # parse(g1,['Sue','laughs'],-1.)
52 # parse(g1,['Bill','knows','that','Sue','laughs'],-1.)
53 # parse(g0,['Sue','laughs'],0.01)
54 # parse(g0,['Sue','laughs'],0.0001)
55 # parse(g0,['the','student','laughs'],0.0001)
56 # parse(g0,['the','student','laughs'],0.000001)
57 # parse(g0min,['the','kind','student','laughs'],0.0000001)

```

3.4 Implementing the original grammar, with left recursion

Since the top-down beam parser can now handle left recursion, we can go back to the original grammar from Fromkin, which we listed on page 11:

```

1  """ file: g0.py
2     our first grammar. It has left recursion and empty productions.
3  """
4  g0 = [('S', ['DP', 'VP']), # categorial rules
5        ('DP', ['D', 'NP']),
6        ('DP', ['NP']),
7        ('DP', ['Name']),
8        ('DP', ['Pronoun']),
9        ('NP', ['N']),
10       ('NP', ['N', 'PP']),
11       ('VP', ['V']),
12       ('VP', ['V', 'DP']),
13       ('VP', ['V', 'PP']),
14       ('VP', ['V', 'CP']),
15       ('VP', ['V', 'VP']),
16       ('VP', ['V', 'DP', 'PP']),
17       ('VP', ['V', 'DP', 'CP']),
18       ('VP', ['V', 'DP', 'VP']),
19       ('PP', ['P']),
20       ('PP', ['P', 'DP']),
21       ('AP', ['A']),
22       ('AP', ['A', 'PP']),
23       ('CP', ['C', 'S']),
24       ('AdvP', ['Adv']),
25       ('NP', ['AP', 'NP']),
26       ('NP', ['NP', 'PP']), # left rec
27       ('NP', ['NP', 'CP']), # left rec
28       ('VP', ['AdvP', 'VP']),
29       ('VP', ['VP', 'PP']), # left rec
30       ('AP', ['AdvP', 'AP']),
31       ('D', ['D', 'Coord', 'D']), # left rec
32       ('V', ['V', 'Coord', 'V']), # left rec
33       ('N', ['N', 'Coord', 'N']), # left rec
34       ('A', ['A', 'Coord', 'A']), # left rec
35       ('P', ['P', 'Coord', 'P']), # left rec
36       ('C', ['C', 'Coord', 'C']), # left rec
37       ('Adv', ['Adv', 'Coord', 'Adv']), # left rec
38       ('VP', ['VP', 'Coord', 'VP']), # left rec
39       ('NP', ['NP', 'Coord', 'NP']), # left rec
40       ('DP', ['DP', 'Coord', 'DP']), # left rec
41       ('AP', ['AP', 'Coord', 'AP']), # left rec
42       ('PP', ['PP', 'Coord', 'PP']), # left rec
43       ('AdvP', ['AdvP', 'Coord', 'AdvP']), # left rec
44       ('S', ['S', 'Coord', 'S']), # left rec
45       ('CP', ['CP', 'Coord', 'CP']), # left rec
46       ('D', ['the']), # now the lexical rules
47       ('D', ['a']),
48       ('D', ['some']),
49       ('D', ['every']),
50       ('D', ['one']),
51       ('D', ['two']),
52       ('A', ['gentle']),
53       ('A', ['clear']),
54       ('A', ['honest']),
55       ('A', ['compassionate']),
56       ('A', ['brave']),
57       ('A', ['kind']),
58       ('N', ['student']),
59       ('N', ['teacher']),
60       ('N', ['city']),
61       ('N', ['university']),
62       ('N', ['beer']),
63       ('N', ['wine']),
64       ('V', ['laughs']),
65       ('V', ['cries']),
66       ('V', ['praises']),
67       ('V', ['criticizes']),
68       ('V', ['says']),
69       ('V', ['knows']),
70       ('Adv', ['happily']),

```

```

71 ('Adv', ['sadly']),
72 ('Adv', ['impartially']),
73 ('Adv', ['generously']),
74 ('Name', ['Bill']),
75 ('Name', ['Sue']),
76 ('Name', ['Jose']),
77 ('Name', ['Maria']),
78 ('Name', ['Presidents', 'Day']),
79 ('Name', ['Tuesday']),
80 ('Pronoun', ['he']),
81 ('Pronoun', ['she']),
82 ('Pronoun', ['it']),
83 ('Pronoun', ['him']),
84 ('Pronoun', ['her']),
85 ('P', ['in']),
86 ('P', ['on']),
87 ('P', ['with']),
88 ('P', ['by']),
89 ('P', ['to']),
90 ('P', ['from']),
91 ('C', ['that']),
92 ('C', []), # empty production!
93 ('C', ['whether']),
94 ('Coord', ['and']),
95 ('Coord', ['or']),
96 ('Coord', ['but'])

```

With this grammar, we get sessions like this:

```

>>> from tdbp import *
>>> from g0 import *
>>> parse(g0, ['Sue', 'laughs'], 0.0001)
11= [['S', 'DP', 'VP'], ['DP', 'Name'], ['Name', 'Sue'], ['VP', 'V'], ['V', 'laughs']]
another? y
'False'
>>> parse(g0, ['the', 'student', 'laughs'], 0.0001)
'False'
>>> parse(g0, ['the', 'student', 'laughs'], 0.00001)
'False'
>>> parse(g0, ['the', 'student', 'laughs'], 0.000001)
11= [['S', 'DP', 'VP'], ['DP', 'D', 'NP'], ['D', 'the'], ['NP', 'N'], ['N', 'student'], ['VP', 'V'], ['V', 'laughs']]
another? n

```

3.5 Assessment: Time, space, and nondeterminism

The situation is only slightly changed from the TD parser, so we list the basic properties quickly here.

1. The rules \Rightarrow_{td} are sound and complete, as before – see §2.7.1.
2. The structures parsed at each step are connected, allowing for incremental interpretation, as before – see §2.7.2.
3. This method develops one parse at a time – the maximum probability parse in the beam, at each step – and so is similar to the top-down recognizer in suggesting a treatment of ‘garden path’ sentences – see 2.7.3.
4. Right branching derivations only require a finite amount of parser memory, no matter how long the input is – see §2.7.6
5. Left branching derivations can require unbounded amounts of parser memory, increasing with the length of the input – see §2.7.4
6. Lookahead can reduce indeterminacy, but not effectively for languages like English. English is not $LL(k)$ for any k – see §2.7.7.
7. And of course, English has intractable ambiguity, no matter what recognizer you use – see §2.7.7.

The new thing is this:

8. Given a fixed, positive $0 < k \leq 1$, most left recursive grammars will not cause non-termination. (But see exercise 6 on page 50 below.

3.6 More alternatives, and conclusions

TD parsing has lots of nice properties, and the beam parser shares many of them. And certain cases of non-termination and intractability could be avoided with this approach.

But the beam parser is still exploring many parses which a ‘smarter’ parser might reject, e.g. parses that could have been avoided by looking one word ahead, or parses that could have been avoided by better judging plausibility in context. To take just one example: Do we really predict all the nouns (within suitably generous probability bounds) and then check our predictions? Not likely! And do we really want a parser to have to predict how many modifiers a phrase has, before seeing any of the phrase? That seems unlikely too. There are many other search methods: simple breadth-first, A* [9], Hale’s adaptive methods [5], and more. And any of these algorithms could be implemented in ‘hardware’ that provides more or less parallelism, complicating the relation between the number of steps required (‘time’ in the computer scientists’ sense) and the real time required by the hardware (and in particular, by any neurophysiological implementation). We should keep the basic questions about search, the questions about how to handle indeterminacy, in mind as we consider other parsing strategies.

Exercises: Time requirements for our TDB recognizer

Due by the end of the day on Monday (i.e. by midnight) January 28.

1. Modify `tdbp.py` to produce the program `tdbpt.py` in the same way that we modified the simple top-down parser in §2.7.8 (in the updated version of the week 2 notes). Name your file `tdbpt(YOURINITIALS).py`, and put your name at the top of the file too.
2. At the top of this file, add a grammar with the name `g4` with the following rules:

$$\begin{aligned} S &\rightarrow aSS \\ S &\rightarrow \epsilon \end{aligned}$$

3. Use your `tdbpt.py` to see how many steps it takes to find all parses of each of these sentences, using `g4`:

a, aa, aaa, aaaa, aaaaa, aaaaaa, aaaaaaa

Cut and paste the step counts for each of these sentences into a comment at the bottom of your file.

4. In the same comment at the bottom of your file, answer this question: For which values of n does the number of steps required to find all parses of the sentence a^n exceed 2^n ?
5. Optionally: Sketch a proof that establishes the correctness of your answer to 4.
6. Optionally: Can you design a grammar which can cause this parser to be nonterminating when $k < 0$? Present the grammar and explain the problem.
7. A prominent tradition in psycholinguistics holds that typical sentence processing involves development of a single analysis, where that parse is subjected to ‘reanalysis’ when required. For example, Frazier and Clifton say:

Serial theories of sentence processing specify that the processor pursues just a single analysis of a sentence until that analysis becomes implausible or untenable, at which point revision of the first analysis occurs. . . Here it is argued that revision cost cannot be calculated in purely structural terms. . . It is also argued that a theory of revisions must include a Minimal Revisions principle. . . [3, p.193]

In beam search, the most probable parse is developed at each So then, is a beam processor a serial model with reanalysis? Compare this idea to Frazier and Clifton’s or other psycholinguistic proposals about reanalysis [2, 4, etc]. Staub’s [14] proposal is especially interesting: he suggests that we see the influence of discarded parses at later stages of processing. Is there an alternative interpretation of his results?

References

- [1] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms, 2nd Edition*. MIT Press, Cambridge, Massachusetts, 2001.
- [2] FODOR, J. D., AND INOUE, A. Garden path reanalysis: Attach (anyway) and revision as last resort. In *Cross-linguistic Perspectives on Language Processing*, M. De Vincenzi and V. Lombardo, Eds. Kluwer, Boston, 2000, pp. 21–61.

- [3] FRAZIER, L., AND CLIFTON, C. Sentence reanalysis and visibility. In *Reanalysis in Sentence Processing*, J. D. Fodor and F. Ferreira, Eds. Kluwer, Boston, 1998, pp. 143–176.
- [4] GRODNER, D., GIBSON, E., ARGAMAN, V., AND BABYONSHEV, M. Against repair-based reanalysis in sentence comprehension. *Journal of Psycholinguistic Research* 32, 2 (2003), 141–166.
- [5] HALE, J. T. What a rational parser would do. *Cognitive Science* 35, 3 (2011), 399–443.
- [6] HUANG, L., AND CHIANG, D. Better k -best parsing. In *Proceedings of the International Workshop on Parsing Technologies (IWPT)* (2005), pp. 53–64.
- [7] JIMÉNEZ, V. M., AND MARZAL, A. Computation of the n best parse trees for weighted and stochastic context-free grammars. In *Proceedings of the Joint IAPR International Workshops on Advances in Pattern Recognition* (London, UK, 2000), Springer-Verlag, pp. 183–192.
- [8] JURAFSKY, D. A probabilistic model of lexical and syntactic access and disambiguation. *Cognitive Science* 20 (1996), 137–194.
- [9] KLEIN, D., AND MANNING, C. D. A* parsing: Fast exact Viterbi parse selection. In *Human Language Technology Conference - North American chapter of the Association for Computational Linguistics annual meeting (HLT-NAACL)* (2003).
- [10] PAULS, A., AND KLEIN, D. K-best A* parsing. In *Proceedings of the Association for Computational Linguistics* (2009).
- [11] PAULS, A., KLEIN, D., AND QUIRK, C. Top-down k-best A* parsing. In *Proceedings of the Association for Computational Linguistics* (2010), pp. 275–298.
- [12] ROARK, B. Probabilistic top-down parsing and language modeling. *Computational Linguistics* 27, 2 (2001), 249–276.
- [13] ROARK, B. Robust garden path parsing. *Natural Language Engineering* 10, 1 (2004), 1–24.
- [14] STAUB, A. The return of the repressed: Abandoned parses facilitate syntactic reanalysis. *Journal of Memory and Language* 57, 2 (2007), 299 – 323.

Chapter 4 Bottom-up CF parsing

Bottom-up parsing is sometimes called shift-reduce parsing, since its basic operations are shift and reduce. It is also sometimes called LR, because it proceeds left-to-right, building a rightmost parse in reverse [15], but LR parsing often refers to BU parsing done with a “chart”, as we will discuss in §7.3.

4.1 BU backtrack recognition

The algorithm for BU recognition deals with non-determinacies in the same way that the TD recognizer does (i.e. by keeping a list of all choices), but the parsing steps are different. In TD recognition, everything is predicted and then scanned, but in bottom-up parsing everything except the prediction of a complete sentence is heard first, shifted, and then the heard things are reduced to categories by using the rewrite rules backwards. There is one special case of the reduce rule, called reduce-complete, which applies only at the last step of the recognition to fulfill the prediction of the S. To indicate which categories (and words) in the parser memory have been found, keeping them distinct from the categories (and words) that have been predicted, let’s put the found categories into tuples. That is, when we have heard the word **Sue**, we put (Sue) into the stack, and when we have heard the category DP we put (DP) into the stack.¹

```
BOTTOM-UP BACKTRACK CF RECOGNITION( $G, ws$ )
0  ds=[(ws,S)] where S is the start category
1  while ds≠ [] and ds[0]≠([],[]):
2      ds=[d| ds[0] ⇒bu d] + ds[1:]
3  if ds==[] then False else True
```

where the steps \Rightarrow_{bu} are defined as follows:

$$\begin{aligned} \text{(reduce)} \quad & \frac{\text{input}, (X_n) \dots (X_1)\alpha}{\text{input}, (X)\alpha} \quad \text{if } X \rightarrow X_1 \dots X_n \\ \text{(reduce-complete)} \quad & \frac{\text{input}, (X_n) \dots (X_1) X\alpha}{\text{input}, \alpha} \quad \text{if } X \rightarrow X_1 \dots X_n \\ \text{(shift)} \quad & \frac{w \text{ input}, \alpha}{\text{input}, (w) \alpha} \end{aligned}$$

With these rules, we get derivations like the following. (This derivation is calculated by `buh.py` from §4.2 below, and so python puts an extra comma into the completed categories.)

```
0  (['Sue', 'laughs'], ['S'])
1  (['laughs'], [( 'Sue', ), 'S'])
2  (['laughs'], [( 'Name', ), 'S'])
3  (['laughs'], [( 'DP', ), 'S'])
4  ([], [( 'laughs', ), ( 'DP', ), 'S'])
5  ([], [( 'V', ), ( 'DP', ), 'S'])
6  ([], [( 'VP', ), ( 'DP', ), 'S'])
7  ([], [])
```

¹This very slightly non-standard presentation of the bottom-up recognizer, with a reduce-complete rule, will set the stage for the important generalization of the method later.

The implementation is an easy change from `td.py`, simply replacing the top-down rules by the bottom-up ones:

```

1  """ file: bu.py
2     a simple bottom-up backtrack CF recognizer
3  """
4  def showDerivations(ds): # pretty print the 'backtrack stack'
5      for (n,(i,cs)) in enumerate(reversed(ds)):
6          print n,'( ',
7              for w in i:
8                  print w,
9                  print ', ',
10             for c in cs:
11                 print c,
12             print ') '
13     print '-----'
14
15  def revTupled(rhs):
16     tmp = rhs[:]
17     tmp.reverse()
18     return [(x,) for x in tmp]
19
20  def bustep(g,(i,cs)): # compute all possible next steps from (i,cs)
21     nextsteps=[]
22     for (lhs,rhs) in g:
23         if len(rhs) < len(cs) and cs[:len(rhs)+1]==revTupled(rhs)+[lhs]:
24             print 'reduce-complete',lhs,'->',rhs # for trace
25             nextsteps.append((i,cs[len(rhs)+1:]))
26         if len(rhs) <= len(cs) and cs[:len(rhs)]==revTupled(rhs):
27             print 'reduce',lhs,'->',rhs # for trace
28             nextsteps.append((i,[(lhs,)]+cs[len(rhs):]))
29     if len(i)>0:
30         print 'shift',i[0] # for trace
31         i1=i[1:]
32         nextsteps.append((i1,[(i[0],)]+cs))
33     return nextsteps
34
35  def recognize(g,i):
36     ds = [(i,['S'])]
37     while ds != [] and ds[-1] != ([],[]):
38         showDerivations(ds) # for trace
39         d = ds.pop()
40         ds.extend(bustep(g,d))
41     if ds == []:
42         return False
43     else:
44         showDerivations(ds) # for trace
45         return True
46
47  # Examples:
48  # recognize(g0noe,['Sue','laughs'])
49  # recognize(g0noe,['Sue','laughs','on','Tuesday'])
50  # recognize(g0noe,['the','student','from','the','university','praises','the','beer','on','Presidents','Day'])
51  # recognize(g0noe,['the','student','from','the','university','praises','the'])
52  # recognize(g0noe,['Sue','knows','that','Maria','or','Bill','laughs'])

```

Notice that we cannot allow an empty category with this standard backtracking BU recognizer, because it can cause non-termination. For example, with the rule $C \rightarrow []$, we can reduce to find any number of C 's at any point in any parse. For now, we simply eliminate that rule. Left recursion, on the other hand, no longer causes non-termination. So we add the left recursive rules from our original grammar but leave out the empty production (and a few other rules, for simplicity):

```

1  """ file: g0noe.py
2     a grammar with left recursion, but no empty productions
3  """
4  g0noe = [(('S', ['DP', 'VP']), # categorial rules
5           ('DP', ['D', 'NP']),
6           ('DP', ['NP']),
7           ('DP', ['Name']),
8           ('DP', ['Pronoun']),
9           ('NP', ['N']),
10          ('VP', ['V']),
11          ('VP', ['V', 'DP']),

```

```

12 ('VP', ['V', 'CP']),
13 ('VP', ['V', 'VP']),
14 ('VP', ['V', 'DP', 'PP']),
15 ('VP', ['V', 'DP', 'VP']),
16 ('PP', ['P']),
17 ('PP', ['P', 'DP']),
18 ('AP', ['A']),
19 ('AP', ['A', 'PP']),
20 ('CP', ['C', 'S']),
21 ('AdvP', ['Adv']),
22 ('NP', ['AP', 'NP']),
23 ('NP', ['NP', 'PP']), # left rec
24 ('NP', ['NP', 'CP']), # left rec
25 ('VP', ['AdvP', 'VP']),
26 ('VP', ['VP', 'PP']), # left rec
27 ('AP', ['AdvP', 'AP']),
28 ('D', ['D', 'Coord', 'D']), # left rec
29 ('V', ['V', 'Coord', 'V']), # left rec
30 ('N', ['N', 'Coord', 'N']), # left rec
31 ('A', ['A', 'Coord', 'A']), # left rec
32 ('P', ['P', 'Coord', 'P']), # left rec
33 ('C', ['C', 'Coord', 'C']), # left rec
34 ('Adv', ['Adv', 'Coord', 'Adv']), # left rec
35 ('VP', ['VP', 'Coord', 'VP']), # left rec
36 ('NP', ['NP', 'Coord', 'NP']), # left rec
37 ('DP', ['DP', 'Coord', 'DP']), # left rec
38 ('AP', ['AP', 'Coord', 'AP']), # left rec
39 ('PP', ['PP', 'Coord', 'PP']), # left rec
40 ('AdvP', ['AdvP', 'Coord', 'AdvP']), # left rec
41 ('S', ['S', 'Coord', 'S']), # left rec
42 ('CP', ['CP', 'Coord', 'CP']), # left rec
43 ('D', ['the']), # now the lexical rules
44 ('D', ['a']),
45 ('D', ['some']),
46 ('D', ['every']),
47 ('D', ['one']),
48 ('D', ['two']),
49 ('A', ['gentle']),
50 ('A', ['clear']),
51 ('A', ['honest']),
52 ('A', ['compassionate']),
53 ('A', ['brave']),
54 ('A', ['kind']),
55 ('N', ['student']),
56 ('N', ['teacher']),
57 ('N', ['city']),
58 ('N', ['university']),
59 ('N', ['beer']),
60 ('N', ['wine']),
61 ('V', ['laughs']),
62 ('V', ['cries']),
63 ('V', ['praises']),
64 ('V', ['criticizes']),
65 ('V', ['says']),
66 ('V', ['knows']),
67 ('Adv', ['happily']),
68 ('Adv', ['sadly']),
69 ('Adv', ['impartially']),
70 ('Adv', ['generously']),
71 ('Name', ['Bill']),
72 ('Name', ['Sue']),
73 ('Name', ['Jose']),
74 ('Name', ['Maria']),
75 ('Name', ['Presidents', 'Day']),
76 ('Name', ['Tuesday']),
77 ('Pronoun', ['he']),
78 ('Pronoun', ['she']),
79 ('Pronoun', ['it']),
80 ('Pronoun', ['him']),
81 ('Pronoun', ['her']),
82 ('P', ['in']),
83 ('P', ['on']),
84 ('P', ['with']),
85 ('P', ['by']),
86 ('P', ['to']),

```

```

87     ('P', ['from']),
88     ('C', ['that']),
89     # ('C', []), # no empty productions allowed in BU parsing
90     ('C', ['whether']),
91     ('Coord', ['and']),
92     ('Coord', ['or']),
93     ('Coord', ['but'])

```

Now we can get sessions like this:

```

>>> from g0noe import *
>>> from bu import *
>>> recognize(g0noe, ['Sue', 'laughs'])
0 ( Sue laughs , S )
-----
shift Sue
0 ( laughs , ('Sue',) S )
-----
reduce Name -> ['Sue']
shift laughs
0 ( , ('laughs',) ('Sue',) S )
1 ( laughs , ('Name',) S )
-----
reduce V -> ['laughs']
0 ( , ('V',) ('Sue',) S )
1 ( laughs , ('Name',) S )
-----
reduce VP -> ['V']
0 ( , ('VP',) ('Sue',) S )
1 ( laughs , ('Name',) S )
-----
0 ( laughs , ('Name',) S )
-----
reduce DP -> ['Name']
shift laughs
0 ( , ('laughs',) ('Name',) S )
1 ( laughs , ('DP',) S )
-----
reduce V -> ['laughs']
0 ( , ('V',) ('Name',) S )
1 ( laughs , ('DP',) S )
-----
reduce VP -> ['V']
0 ( , ('VP',) ('Name',) S )
1 ( laughs , ('DP',) S )
-----
0 ( laughs , ('DP',) S )
-----
shift laughs
0 ( , ('laughs',) ('DP',) S )
-----
reduce V -> ['laughs']
0 ( , ('V',) ('DP',) S )
-----
reduce VP -> ['V']
0 ( , ('VP',) ('DP',) S )
-----
reduce-complete S -> ['DP', 'VP']
reduce S -> ['DP', 'VP']
0 ( , ('S',) S )
1 ( , )
-----
0 ( , )
-----
True

```

4.2 BU backtrack parsing (complete derivations)

```

BOTTOM-UP BACKTRACK CF PARSING( $G, ws$ )
0  ds=[(ws,S,(ws,S))] where S is the start category
1  while ds≠ [] and ds[0]≠([],[]):
2      ds=[d| ds[0] ⇒buh d] + ds[1:]
3  if ds==[] then False else True

```

where the steps \Rightarrow_{buh} are defined as follows:

$$\begin{aligned}
 (\text{reduce}) \quad & \frac{\text{input}, (X_n) \dots (X_1)\alpha, h}{\text{input}, (X)\alpha, h(\text{input}, (X)\alpha)} \quad \text{if } X \rightarrow X_1 \dots X_n \\
 (\text{reduce-complete}) \quad & \frac{\text{input}, (X_n) \dots (X_1) X\alpha, h}{\text{input}, \alpha, h(\text{input}, \alpha)} \quad \text{if } X \rightarrow X_1 \dots X_n \\
 (\text{shift}) \quad & \frac{\text{w input}, \alpha, h}{\text{input}, (w)\alpha, h(\text{input}, w\alpha)}
 \end{aligned}$$

The implementation is also an easy change from the `bu.py`. With `buh.py` code, we get sessions like this one (we have removed many output lines):

```

>>> from g0noe import *
>>> from buh import *
>>> parse(g0noe,['Bill','knows','Sue','laughs'])
shift Bill
reduce Name -> ['Bill']
...
reduce VP -> ['V', 'DP', 'VP']
reduce-complete S -> ['DP', 'VP']
reduce S -> ['DP', 'VP']
0 (['Bill', 'knows', 'Sue', 'laughs'], ['S'])
1 (['knows', 'Sue', 'laughs'], [( 'Bill', ), 'S'])
2 (['knows', 'Sue', 'laughs'], [( 'Name', ), 'S'])
3 (['knows', 'Sue', 'laughs'], [( 'DP', ), 'S'])
4 (['Sue', 'laughs'], [( 'knows', ), ( 'DP', ), 'S'])
5 (['Sue', 'laughs'], [( 'V', ), ( 'DP', ), 'S'])
6 (['laughs'], [( 'Sue', ), ( 'V', ), ( 'DP', ), 'S'])
7 (['laughs'], [( 'Name', ), ( 'V', ), ( 'DP', ), 'S'])
8 (['laughs'], [( 'DP', ), ( 'V', ), ( 'DP', ), 'S'])
9 ([], [( 'laughs', ), ( 'DP', ), ( 'V', ), ( 'DP', ), 'S'])
10 ([], [( 'V', ), ( 'DP', ), ( 'V', ), ( 'DP', ), 'S'])
11 ([], [( 'VP', ), ( 'DP', ), ( 'V', ), ( 'DP', ), 'S'])
12 ([], [( 'VP', ), ( 'DP', ), 'S'])
13 ([], [])
more? y
reduce-complete S -> ['DP', 'VP']
reduce S -> ['DP', 'VP']
shift laughs
...
reduce VP -> ['V']
reduce S -> ['DP', 'VP']
False

```

Notice that the string `['Bill','knows','Sue','laughs']` now has only one derivation, while on page 24 we noticed that with the grammar `g1.py` it had two derivations. It has one now, because we removed the empty production for the complementizer `C`. But the sentence `['Bill','praises','the','student','on','Tuesday']` now has 4 derivations! The new derivations come from the left recursive rules that allow the PP to modify either the NP or the VP, and we still get the previous two derivations with the PP as complement of N or of V. And again, if you check, you will see that the number of steps in each derivation is exactly the number of nodes in the corresponding derivation tree.

4.3 BU backtrack parsing (collecting rules)

We could get the whole derivation tree from the history, but it is more convenient to use a smaller representation: the list of rules used in the derivation. A very minor change in the previous program achieves this:

```

BOTTOM-UP BACKTRACK CF PARSING( $G, ws$ )
0  ds=[(ws,S,[])] where S is the start category
1  while ds≠ [] and ds[0]≠([],[]):
2      ds=[d| ds[0] ⇒bup d] + ds[1:]
3  if ds=[] then False else True

```

where the steps \Rightarrow_{bup} are defined as follows:

$$\begin{aligned}
 (\text{reduce}) \quad & \frac{\text{input}, (X_n) \dots (X_1)\alpha, h}{\text{input}, (X)\alpha, h(X \rightarrow X_1 \dots X_n)} \quad \text{if } X \rightarrow X_1 \dots X_n \\
 (\text{reduce-complete}) \quad & \frac{\text{input}, (X_n) \dots (X_1)X\alpha, h}{\text{input}, \alpha, h(X \rightarrow X_1 \dots X_n)} \quad \text{if } X \rightarrow X_1 \dots X_n \\
 (\text{shift}) \quad & \frac{w \text{ input}, \alpha, h}{\text{input}, (w)\alpha, h}
 \end{aligned}$$

The implementation is easy, and with `bup.py` we get sessions like this one:

```

>>> from g0noe import *
>>> from bup import *
>>> parse(g0noe, ['Sue', 'laughs'])
lr= [['Name', 'Sue'], ['DP', 'Name'], ['V', 'laughs'], ['VP', 'V'], ['S', 'DP', 'VP']]
another? n
[['Name', 'Sue'], ['DP', 'Name'], ['V', 'laughs'], ['VP', 'V'], ['S', 'DP', 'VP']]
>>>

```

4.4 From LR rules to derivation trees in list format

The conversion from rules used in LR order to derivation trees is similar to the conversion from LL rules described in §2.4. The code is not too hard, but trickier than things we have done so far so I will not try to explain it now. It is enough to be able to use it, and that's easy:

```

1  """ lr2dt.py
2      convert LR list of rules to list-format tree
3
4  We will assume that x is a nonterminal iff there is no rule rewriting it.
5  We assume that lexical rules have only lexical items on rhs.
6  Non-lexical rules have no lexical items on rhs.
7  Empty rhs is treated as the empty sequence of lexical items, as usual.
8  """
9  from g0noe import *
10
11 def nonterminal(g,x):
12     for (lhs,rhs) in g:
13         if lhs==x:
14             return False # breaks from loop
15     return True
16
17 def lexical_rule(g,r):
18     if len(r[1:])==0:
19         return True
20     else:
21         return nonterminal(g,r[1])
22
23 def lr2dt(g,lr): # for top call, tmp should be []
24     if isinstance(lr,list) and len(lr)>0:
25         t=lr.pop()
26         if lexical_rule(g,t):
27             return t
28         else: # if it is nonlexical
29             for i,x in enumerate(reversed(t[1:])):
30                 t[len(t)-i-1]=lr2dt(g,lr) # recursive def most natural, and not too deep
31             return t
32     else:
33         return lr
34
35 """example:
36 lr1 = [['Name', 'Sue'], ['DP', 'Name'], ['V', 'laughs'], ['VP', 'V'], ['S', 'DP', 'VP']]

```

```

37 lr2dt(g0noe,lr1[:])
38 from pptree import *
39 from bup import *
40 pptree(0,lr2dt(g0noe,parse(g0noe,['Sue','laughs'])))
41 pptree(0,lr2dt(g0noe,parse(g0noe,['Sue','laughs','on','Presidents','Day'])))
42 from list2nlktree import *
43 list2nlktree(lr2dt(g0noe,parse(g0noe,['Sue','laughs','on','Tuesday']))).draw()
44 list2nlktree(lr2dt(g0noe,parse(g0noe,['Sue','laughs','and','cries']))).draw()
45 list2nlktree(lr2dt(g0noe,parse(g0noe,['the','student','with','Sue','knows','that','Bill','laughs','on','Presidents','P
46 list2nlktree(lr2dt(g0noe,parse(g0noe,['the','student','with','Sue','knows','or','says','that','Bill','laughs','on','P
47 """

```

With this code, we get sessions like this one:

```

>>> from g0noe import *
>>> from bup import *
>>> from lr2dt import *
>>> from pptree import *
>>> lr2dt(g0noe,parse(g0noe,['Sue','or','Bill','laughs']))
lr= [['Name', 'Sue'], ['DP', 'Name'], ['Coord', 'or'], ['Name', 'Bill'],
      ['DP', 'Name'], ['DP', 'DP', 'Coord', 'DP'], ['V', 'laughs'], ['VP', 'V'],
      ['S', 'DP', 'VP']]
another? n
['S', ['DP', ['DP', ['Name', 'Sue']], ['Coord', 'or'], ['DP', ['Name', 'Bill']]],
      ['VP', ['V', 'laughs']]]
>>> pptree(0,lr2dt(g0noe,parse(g0noe,['Sue','laughs','on','Tuesday'])))
lr= [['Name', 'Sue'], ['DP', 'Name'], ['V', 'laughs'], ['VP', 'V'], ['P', 'on'],
      ['Name', 'Tuesday'], ['DP', 'Name'], ['PP', 'P', 'DP'], ['VP', 'VP', 'PP'],
      ['S', 'DP', 'VP']]
another? n
S
  DP
    Name
      Sue
  VP
    VP
      V
        laughs
    PP
      P
        on
      DP
        Name
          Tuesday

```

4.5 BU beam parsing

We could easily implement BU beam parsing too, but let's stop here with the BU ideas for now.

4.6 Assessment: time, space, nondeterminism

We have seen that getting from a BU recognizer to a parser is fairly easy: we simply record the steps in the derivation, in a way which has no influence on the course of the derivation but only on the output. And now that we have parsers, it is easier to see, as scientists or formal-language-theorists, what is going on in bottom-up recognition.

4.6.1 Predictiveness and incrementality

Although we have only printed out tree representations at the end of a parse, it is clear that we could have formulated the tree representations built at every step. When we do that, we see that, at many points, the stack has parts of the derivation which are not connected to the root or to each other by rules that have already been used.

⇒ picture coming ⇐

4.6.2 Right branching derivations, regularity, and space

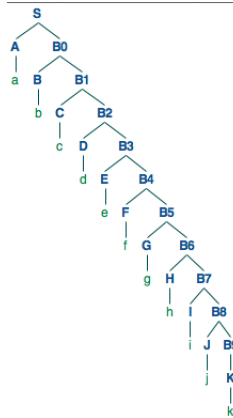
Among the grammars that the BU recognizer can handle are some that define only regular languages. In particular, we know that if a grammar has only right branching (or only left branching), the language it defines is regular. A regular language is one that can be recognized with only finite memory, so let's explore how much memory the BU recognizer needs for regular languages, as we did for the top-down recognizer.

Consider again a simple grammar that generates only 1 string, with a simple right branching derivation:

```

1  """ file: g2noe.py
2      this grammar has no empty productions
3      parse(g2noe, ['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'])
4  """
5  g2noe = [(('S', ['B', 'B0']),
6            ('B0', ['B', 'B1']),
7            ('B1', ['B', 'B2']),
8            ('B2', ['B', 'B3']),
9            ('B3', ['B', 'B4']),
10           ('B4', ['B', 'B5']),
11           ('B5', ['B', 'B6']),
12           ('B6', ['B', 'B7']),
13           ('B7', ['B', 'B8']),
14           ('B8', ['B', 'B9']),
15           ('B9', ['B']),
16           ('B', ['b'])])]
```

If we look at the derivation tree for the only sentence in this language, we see that it has 33 nodes:



This sentence is accepted on a long right branch. Watch the parser memory use in the following derivation:

```

>>> from buht import *
>>> from g2noe import *
>>> parse(g2noe, ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'])
steps= 6142
0 (['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], ['S'])
1 (['b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('a',), 'S')])
2 (['b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('A',), 'S')])
3 (['c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('b',), ('A',), 'S')])
4 (['c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('B',), ('A',), 'S')])
5 (['d', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('c',), ('B',), ('A',), 'S')])
6 (['d', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('C',), ('B',), ('A',), 'S')])
7 (['e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('d',), ('C',), ('B',), ('A',), 'S')])
8 (['e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('D',), ('C',), ('B',), ('A',), 'S')])
9 (['f', 'g', 'h', 'i', 'j', 'k'], [(('e',), ('D',), ('C',), ('B',), ('A',), 'S')])
10 (['f', 'g', 'h', 'i', 'j', 'k'], [(('E',), ('D',), ('C',), ('B',), ('A',), 'S')])
11 (['g', 'h', 'i', 'j', 'k'], [(('f',), ('E',), ('D',), ('C',), ('B',), ('A',), 'S')])
12 (['g', 'h', 'i', 'j', 'k'], [(('F',), ('E',), ('D',), ('C',), ('B',), ('A',), 'S')])
13 (['h', 'i', 'j', 'k'], [(('g',), ('F',), ('E',), ('D',), ('C',), ('B',), ('A',), 'S')])
14 (['h', 'i', 'j', 'k'], [(('G',), ('F',), ('E',), ('D',), ('C',), ('B',), ('A',), 'S')])
15 (['i', 'j', 'k'], [(('h',), ('G',), ('F',), ('E',), ('D',), ('C',), ('B',), ('A',), 'S')])
16 (['i', 'j', 'k'], [(('H',), ('G',), ('F',), ('E',), ('D',), ('C',), ('B',), ('A',), 'S')])
17 (['j', 'k'], [(('i',), ('H',), ('G',), ('F',), ('E',), ('D',), ('C',), ('B',), ('A',), 'S')])
18 (['j', 'k'], [(('I',), ('H',), ('G',), ('F',), ('E',), ('D',), ('C',), ('B',), ('A',), 'S')])
19 (['k'], [(('j',), ('I',), ('H',), ('G',), ('F',), ('E',), ('D',), ('C',), ('B',), ('A',), 'S')])
20 (['k'], [(('J',), ('I',), ('H',), ('G',), ('F',), ('E',), ('D',), ('C',), ('B',), ('A',), 'S')])
21 ([], [(('k',), ('J',), ('I',), ('H',), ('G',), ('F',), ('E',), ('D',), ('C',), ('B',), ('A',), 'S')])
```



```

22 ([], [( 'K' ), ( 'J' ), ( 'I' ), ( 'H' ), ( 'G' ), ( 'F' ), ( 'E' ), ( 'D' ), ( 'C' ), ( 'B' ), ( 'A' ), 'S'])
23 ([], [( 'B9' ), ( 'J' ), ( 'I' ), ( 'H' ), ( 'G' ), ( 'F' ), ( 'E' ), ( 'D' ), ( 'C' ), ( 'B' ), ( 'A' ), 'S'])
24 ([], [( 'B8' ), ( 'I' ), ( 'H' ), ( 'G' ), ( 'F' ), ( 'E' ), ( 'D' ), ( 'C' ), ( 'B' ), ( 'A' ), 'S'])
25 ([], [( 'B7' ), ( 'H' ), ( 'G' ), ( 'F' ), ( 'E' ), ( 'D' ), ( 'C' ), ( 'B' ), ( 'A' ), 'S'])
26 ([], [( 'B6' ), ( 'G' ), ( 'F' ), ( 'E' ), ( 'D' ), ( 'C' ), ( 'B' ), ( 'A' ), 'S'])
27 ([], [( 'B5' ), ( 'F' ), ( 'E' ), ( 'D' ), ( 'C' ), ( 'B' ), ( 'A' ), 'S'])
28 ([], [( 'B4' ), ( 'E' ), ( 'D' ), ( 'C' ), ( 'B' ), ( 'A' ), 'S'])
29 ([], [( 'B3' ), ( 'D' ), ( 'C' ), ( 'B' ), ( 'A' ), 'S'])
30 ([], [( 'B2' ), ( 'C' ), ( 'B' ), ( 'A' ), 'S'])
31 ([], [( 'B1' ), ( 'B' ), ( 'A' ), 'S'])
32 ([], [( 'B0' ), ( 'A' ), 'S'])
33 ([], [])
more? n

```

It is easy to see that with simple right branching derivations like this one, the amount of space required in the stack of predicted categories is not bounded by a finite constant – the longer the string, the more parser memory we will need.

4.6.3 Left branching derivations, regularity, and space

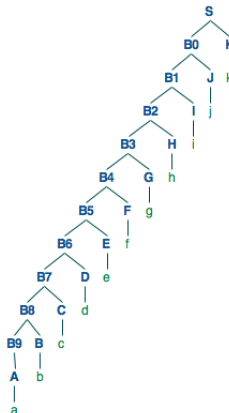
Now let's look at left branching:

```

1  """ file: g3noe.py
2      a left branching grammar (no left recursion!)
3      parse(g3noe, ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'])
4  """
5  g3noe = [( 'S', [ 'B0', 'K' ] ),
6           ( 'B0', [ 'B1', 'J' ] ),
7           ( 'B1', [ 'B2', 'I' ] ),
8           ( 'B2', [ 'B3', 'H' ] ),
9           ( 'B3', [ 'B4', 'G' ] ),
10          ( 'B4', [ 'B5', 'F' ] ),
11          ( 'B5', [ 'B6', 'E' ] ),
12          ( 'B6', [ 'B7', 'D' ] ),
13          ( 'B7', [ 'B8', 'C' ] ),
14          ( 'B8', [ 'B9', 'B' ] ),
15          ( 'B9', [ 'A' ] ),
16          ( 'A', [ 'a' ] ),
17          ( 'B', [ 'b' ] ),
18          ( 'C', [ 'c' ] ),
19          ( 'D', [ 'd' ] ),
20          ( 'E', [ 'e' ] ),
21          ( 'F', [ 'f' ] ),
22          ( 'G', [ 'g' ] ),
23          ( 'H', [ 'h' ] ),
24          ( 'I', [ 'i' ] ),
25          ( 'J', [ 'j' ] ),
26          ( 'K', [ 'k' ] )

```

If we look at the derivation tree for the only sentence in this language, we see that it has 33 nodes:



The left branching in this grammar requires unbounded memory top-down, but bottom-up we find:

```

>>> from buht import *
>>> from g3noe import *
steps= 8178
0 (['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], ['S'])
1 (['b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('a',), 'S')])
2 (['b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('A',), 'S')])
3 (['b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('B9',), 'S')])
4 (['c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('b',), ('B9',), 'S')])
5 (['c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('B',), ('B9',), 'S')])
6 (['c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('B8',), 'S')])
7 (['d', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('c',), ('B8',), 'S')])
8 (['d', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('C',), ('B8',), 'S')])
9 (['d', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('B7',), 'S')])
10 (['e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('d',), ('B7',), 'S')])
11 (['e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('D',), ('B7',), 'S')])
12 (['e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('B6',), 'S')])
13 (['f', 'g', 'h', 'i', 'j', 'k'], [(('e',), ('B6',), 'S')])
14 (['f', 'g', 'h', 'i', 'j', 'k'], [(('E',), ('B6',), 'S')])
15 (['f', 'g', 'h', 'i', 'j', 'k'], [(('B5',), 'S')])
16 (['g', 'h', 'i', 'j', 'k'], [(('f',), ('B5',), 'S')])
17 (['g', 'h', 'i', 'j', 'k'], [(('F',), ('B5',), 'S')])
18 (['g', 'h', 'i', 'j', 'k'], [(('B4',), 'S')])
19 (['h', 'i', 'j', 'k'], [(('g',), ('B4',), 'S')])
20 (['h', 'i', 'j', 'k'], [(('G',), ('B4',), 'S')])
21 (['h', 'i', 'j', 'k'], [(('B3',), 'S')])
22 (['i', 'j', 'k'], [(('h',), ('B3',), 'S')])
23 (['i', 'j', 'k'], [(('H',), ('B3',), 'S')])
24 (['i', 'j', 'k'], [(('B2',), 'S')])
25 (['j', 'k'], [(('i',), ('B2',), 'S')])
26 (['j', 'k'], [(('I',), ('B2',), 'S')])
27 (['j', 'k'], [(('B1',), 'S')])
28 (['k'], [(('j',), ('B1',), 'S')])
29 (['k'], [(('J',), ('B1',), 'S')])
30 (['k'], [(('B0',), 'S')])
31 ([], [(('k',), ('B0',), 'S')])
32 ([], [(('K',), ('B0',), 'S')])
33 ([], [])
more n

```

There are never more than 3 elements in the parser memory, and clearly this would hold, no matter how long the left branch was.

4.6.4 Determinism: time, space, nontermination

As we mentioned in the previous chapter, the BU recognizer will fail to terminate if the grammar has left recursion, and even when it doesn't, the number of steps the parser takes can be on the order of k^n for some $k > 1$. This is because of nondeterminism!

'Lookahead' can reduce local ambiguity, but English is not LR(k) for any k .

Consider the simplistic grammar from §2.7.7 for sentences like *the dog [that I told you about] barks*:

$$\begin{aligned}
 S &\rightarrow \text{DP VP} \\
 \text{DP} &\rightarrow \text{D NP} \\
 \text{D} &\rightarrow \text{the} \\
 \text{NP} &\rightarrow \text{NP CP} \\
 \text{NP} &\rightarrow \text{N} \\
 \text{N} &\rightarrow \text{dog} \\
 \text{CP} &\rightarrow \text{that I told you about} \\
 \text{VP} &\rightarrow \text{barks}
 \end{aligned}$$

Parsing TD, the first choice arises when the parse needs to choose which rule to use to expand the NP. A BU parser, though, has choices before that. For example, after shifting the first element

(dog that I told you about barks, (the) S)

There is a choice about whether to shift again or to reduce. Looking at the grammar (and not the input), an oracle could tell us that we should reduce (because no rule has **the** and something else on the right hand side). – We will

talk about how to compute that oracle later. But we face a more serious challenge in the BU parse when we get to this point:

(that I told you about barks, (NP) (D) S)

Now, in order to tell whether we should shift or reduce, we need to look at the next word. If that word is **barks**, we should reduce. If the next word is **that**, we should shift. In fact, looking 1 word ahead always suffices to tell us what we should do next, and so the grammar is LR(1).

Is English LR(1)? No. When we elaborate the grammar further, it is not hard to see that English is not LR(k) for any k . To show this, we just need to find cases where a decision about what to do with a constituent will depend on something that can appear arbitrarily later in the string. Marcus's famous argument [17] used the sentences:²

(you) have the students take the exam!
 have the students taken the exam?

In the former sentence, but not the latter, the parser should first find the empty DP subject of the imperative. But the parser cannot tell what to do without seeing past the subject to the verb, *take* vs. *taken*, and there is no bound on the length of subjects in English, so no fixed finite lookahead will suffice.

4.7 Conclusions

We are struggling with a few problems with our parsers:

- Given a particular grammar G , let's say that the recognizer (or parser) has a problem with **consistency** if it is pursuing derivations which could not possibly succeed, no matter what the input was. For example, given the grammar at the beginning of §4.6.4 on page 62, we noticed that the recognizer could start with the steps:

0 (the dog that I told you about barks, S)
 1 (dog that I told you about barks, (the) S) shift
 2 (that I told you about barks, (dog) (the) S) shift

Looking at the grammar, we can see that *no matter what the input was*, no successful recognition can come from step 2! The recognizer is wasting a lot of time with options that could not possibly succeed, no matter what the input was.

- Given a particular grammar G and a particular input i , let's say that the recognizer has a problem with **local ambiguity** if it is pursuing dead ends that could not work given input i . Using the example from §4.6.4 again, consider the steps:

0 (the dog that I told you about barks, S)
 1 (dog that I told you about barks, (the) S) shift
 2 (dog that I told you about barks, (D) S) shift
 3 (that I told you about barks, (dog) (D) S) shift
 4 (that I told you about barks, (NP) (D) S) shift
 5 (that I told you about barks, (DP) S) shift

This parse cannot succeed with the input shown here, but if the sentence had been *the dog barks*, then step 5 would have been the right one. In this case, one symbol of lookahead – to see the word *that* – would be enough to let us avoid this error, but Marcus shows various constructions where no finite amount of lookahead will be enough. The recognizer is wasting a lot of time with options that could not possibly succeed given the particular input we are analyzing. But the BU parser does much better here than the TD one does, because – intuitively – it has seen more of the input at the point when it has to make choices among alternative derivations.

²Marcus [17] proposes that when confronted with such situations, the human parser delays the decision about what to do about the possible implicit subject (you) until after the next phrase is constructed. In effect, this allows the parser to look some finite number of constituents ahead, instead of just a finite number of words ahead. This idea is further developed by Berwick and Weinberg'97. Parsing strategies of this kind are sometimes called "non-canonical." They were noticed by Knuth [15], and studied further by Szymanski and Williams [23]. They are briefly discussed in the classic reference by Aho and Ullman [1, §6.2]. A formal study of Marcus's linguistic proposals is carefully done by Nozohoor-Farshi [19]. This is an appealing idea which may deserve further consideration in the context of more recent proposals about human languages. We return to related perspectives in §7

- Given a particular grammar G and a particular input i , let's say that the recognizer has a problem with **global ambiguity** if there are multiple good parses. This problem become intractable when the number of parses grows exponentially with the length of the input. This problem comes from the grammar, so no parser can do anything about it! But it raises a question: *how does the human parser avoid getting hopelessly bogged down in irrelevant ambiguities?* Obviously, we often notice ambiguities, and it could be that sometimes they could make comprehension of grammatical sentences difficult, but usually we do not notice, or at least they do not make comprehension difficult. Some ideas that could be part of a solution are considered in the exercises below.

Exercises: BU models and human-like parsing

1. Since all parsing models face problems with indeterminacies, there has been great interest in how humans tend to resolve local and global ambiguities, and how they recover when they have resolved them incorrectly. One idea is that people prefer certain analyses over others, only considering alternatives when the preferred analysis does not work. For PPs and other modifying phrases, there seems to be a preference to attach PPs in the deepest possible positions, at least in English. For example, there is evidence that people initially misanalyze *in the library* in sentences like³

Jill put the book that her son had been reading in the library,

and the phrase *the sock* in

While Mary was mending the sock fell off her lap.

Frazier calls this a “late closure” preference: listeners prefer to “close” each phrase as late as possible, so if they can attach a constituent low, they will.

How could you adapt the bottom-up parser above to exhibit this preference? Pereira offers an answer here: [20]). prefer shift over reduce, whenever both options are consistent. **Implement a version of Pereira’s idea by modifying `bup.py`. Explain why it would be impossible to modify the top-down parser to exhibit the same preference.**

2. The “late closure” idea mentioned in the previous exercise is a preference that depends on structure, but there is some evidence that the availability of analyses can depend on cooccurrence frequencies, meanings, and even details about what is going on in the discourse. Altman and Steedman [3] propose that people will, even on the first pass through a phrase, favor analyses that are “referentially supported.” For example, if I say first

A psychologist was counseling two women. He was worried about one of them but not about the other.

then you are more likely to get the correct analysis of

The psychologist told the woman that he was having trouble with to visit him again.

Describe informally (but as precisely and briefly as possible) how this idea could be implemented. Then see if you can implement a simple version of the idea.

Altmann and others [2, 4, 14] later showed certain limitations in the contexts in which an effect of referential support can be seen. Briefly discuss whether these proposals could also make sense in a BU computational model of parsing.

3. Cuetos&Mitchell’88 present arguments that in some other languages, late closure effects do not hold [6]. Various theories about this have been proposed: Mitchell&Cuetos’91 propose that speakers are just sensitive to the most frequent attachments in their languages [18]; Gibson&al’96 propose that late closure interacts with another preference they call ‘predicate proximity’ [12]; Konieczny&al’97 integrates 3 separate preferences in a ‘parameterized head attachment’ theory, later extended by Hemforth&al’98 [16, 13]; another kind of preference comes from what Janet Fodor calls ‘implicit prosody’ [9, 10]; and additional factors seem to be implicated in a recent study of Bulgarian [22]. Review one or more of these studies and consider whether the proposals are compatible with BU or any of our other recognition methods.

³These examples of “garden paths” are from Frazier’s classic survey article [11]. Cf. also the recent Pickering & van Gompel survey [21].

References

- [1] AHO, A. V., AND ULLMAN, J. D. *The Theory of Parsing, Translation, and Compiling. Volume 1: Parsing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1972.
- [2] ALTMANN, G. T. M., GARNHAM, A., AND DENNIS, Y. Avoiding the garden path: Eye movements in context. *Journal of Memory and Language* 31 (1992), 685–712.
- [3] ALTMANN, G. T. M., AND STEEDMAN, M. Interaction with context during human sentence processing. *Cognition* 30 (1988), 191–238.
- [4] ALTMANN, G. T. M., VAN NICE, K. Y., GARNHAM, A., AND HENSTRA, J.-A. Late closure in context. *Journal of Memory and Language* 38, 4 (1998), 459–484.
- [5] BERWICK, R. C., AND WEINBERG, A. S. A computational model of minimalist syntactic theory. In *Proceedings of the 1997 Conference on Computational Psycholinguistics* (Berkeley, California, 1997).
- [6] CUETOS, F., AND MITCHELL, D. C. Crosslinguistic differences in parsing: Restrictions on the use of the late closure strategy in Spanish. *Cognition* 30 (1988), 73–105.
- [7] DEREMER, F. L. *Practical Translators for LR(k) Parsers*. PhD thesis, Massachusetts Institute of Technology, 1969.
- [8] DEREMER, F. L. Simple LR(k) grammars. *Communications of the ACM* 14, 7 (1971), 453–460.
- [9] FODOR, J. D. Learning to parse? *Journal of Psycholinguistic Research* 27 (1998), 285–319.
- [10] FODOR, J. D. Psycholinguistics cannot escape prosody. In *Speech Prosody* (2002).
- [11] FRAZIER, L. Syntactic complexity. In *Natural Language Parsing*, D. Dowty, L. Karttunen, and A. Zwicky, Eds. Cambridge University Press, NY, 1985.
- [12] GIBSON, E., PEARLMUTTER, N., CANSECO-GONZÁLEZ, E., AND HICKOK, G. Recency preference in the human sentence processing mechanism. *Cognition* 59 (1996), 23–59.
- [13] HEMFORTH, B., KONIECZNY, L., SCHEEPERS, C., AND STRUBE, G. Syntactic ambiguity resolution in German. In *Sentence Processing: A Crosslinguistic Perspective*, D. Hillert, Ed., Syntax and Semantics, Vol. 31. Academic Press, San Diego, 1998, pp. 293–312.
- [14] KAMIDE, Y., ALTMANN, G. T. M., AND HAYWOOD, S. L. The time-course of prediction in incremental sentence processing: Evidence from anticipatory eye movements. *Journal of Memory and Language* 49 (2003), 133–156.
- [15] KNUTH, D. E. On the translation of languages from left to right. *Information and Control* 8 (1965), 607–639.
- [16] KONIECZNY, L., HEMFORTH, B., SCHEEPERS, C., AND STRUBE, G. The role of lexical heads in parsing: evidence from German. *Language and Cognitive Processes* 12 (1997), 307–348.
- [17] MARCUS, M. *A Theory of Syntactic Recognition for Natural Language*. MIT Press, Cambridge, Massachusetts, 1980.
- [18] MITCHELL, D. C., AND CUETOS, F. The origins of parsing strategies. In *Current Issues in Natural Language Processing*, C. Smith, Ed. 1991, pp. 1–12.
- [19] NOZOOHOR-FARSHI, R. *LRRL(k) grammars: a left to right parsing technique with reduced lookaheads*. PhD thesis, University of Alberta, 1986.
- [20] PEREIRA, F. C. N. A new characterization of attachment preferences. In *Natural Language Parsing*, D. Dowty, L. Karttunen, and A. Zwicky, Eds. Cambridge University Press, NY, 1985.
- [21] PICKERING, M. J., AND VAN GOMPEL, R. P. G. Syntactic parsing. In *Handbook of Psycholinguistics, Second Edition*, M. J. Traxler and M. A. Gernsbacher, Eds. Academic Press, NY, 2006, pp. 539–580.
- [22] SEKERINA, I. A., FERNÁNDEZ, E. M., AND PETROVA, K. A. Relative clause attachment in Bulgarian. In *Proceedings of the 12th Annual Workshop on Formal Approaches to Slavic Linguistics* (2003), O. Arnaudova, W. Browne, M. L. Rivero, and Stojanović, Eds., Michigan Slavic Publications, pp. 375–394.
- [23] SZYMANSKI, T. G., AND WILLIAMS, J. H. Noncanonical extensions of bottom-up parsing techniques. *SIAM Journal on Computing* 5 (1976), 231–250.
- [24] TOMITA, M. *Efficient parsing for natural language: a fast algorithm for practical systems*. Kluwer, Boston, 1985.

Chapter 5 Left-corner CF parsing

5.1 LC backtrack recognition

The algorithm for LC recognition deals with non-determinacies in the same way that the TD,BU recognizers do (i.e. by keeping a list of all choices), but the parsing steps are different. In TD recognition, everything is predicted and then scanned. bottom-up parsing everything except the prediction of a complete sentence is heard first, shifted, and then the heard things are reduced to categories by using the rewrite rules backwards. In LC parsing, we use a mix of these same ideas.

```

LEFT-CORNER BACKTRACK CF RECOGNITION( $G, i$ )
0  ds=[(i,S)] where S is the start category
1  while ds $\neq$  [] and ds[0] $\neq$ ([],[]):
2      ds=[d| ds[0]  $\Rightarrow_{lc}$  d] + ds[1:]
3  if ds==[] then False else True
    
```

where the steps \Rightarrow_{lc} are defined as follows:

$$\begin{array}{l}
 \text{(lc-reduce)} \quad \frac{\text{input}, (X_1)\alpha}{\text{input}, X_2 \dots X_n(X)\alpha} \quad \text{if } X \rightarrow X_1 \dots X_n \\
 \text{(lc-reduce-complete)} \quad \frac{\text{input}, (X_1)X\alpha}{\text{input}, X_2 \dots X_n\alpha} \quad \text{if } X \rightarrow X_1 \dots X_n \\
 \text{(shift)} \quad \frac{\text{w input}, \alpha}{\text{input}, (\text{w}) \alpha} \\
 \text{(shift-complete)} \quad \frac{\text{w input}, \text{w}\alpha}{\text{input}, \alpha}
 \end{array}$$

The reduce rule says that if you have a completed element (X_1) on top of the stack, the “left corner” of the rule $X \rightarrow X_1 \dots X_n$, then you can predict $X_2 \dots X_n$ in order to have a completed (X) . The reduce-complete rule is similar, except that the X that is completed is already there as a predicted element, and so they cancel each other out.¹ When $n = 0$, that is, when the right side of a rewrite rule is empty, then the left corner is empty and there is nothing to predict, and so the LC parser and the BU parse behave the same in these cases.

With these rules, we get derivations like the following.

¹This LC parsing strategy, in which the decision about whether to complete is made at the reduce step, is sometimes called ‘arc-eager’ [1].

0	(Sue laughs on Tuesday, S)		
1	(laughs on Tuesday, (Sue) S)	shift	
2	(laughs on Tuesday, (Name) S)	lc-reduce:	Name→Sue
3	(laughs on Tuesday, (DP) S)	lc-reduce:	DP→Name
4	(laughs on Tuesday, VP)	lc-reduce-complete:	S→DP VP
5	(on Tuesday, (laughs) VP)	shift	
6	(on Tuesday, (V) VP)	lc-reduce:	V→laughs
7	(on Tuesday, (VP) VP)	lc-reduce	VP→V
8	(on Tuesday, PP)	lc-reduce-complete:	VP→VP PP
9	(Tuesday, (on) PP)	shift	
10	(Tuesday, (P) PP)	lc-reduce:	P→on
11	(Tuesday, DP)	lc-reduce-complete:	PP→P DP
12	(ϵ , (Tuesday) DP)	shift	
13	(ϵ , (Name) DP)	lc-reduce:	Name→Tuesday
14	(ϵ , ϵ)	lc-reduce-complete:	DP→Name

The implementation is an easy change from `td.py` and `bu.py`.

Notice that we cannot allow an empty category with this standard backtracking LC recognizer, because it can cause non-termination. For example, with the rule $C \rightarrow \epsilon$ that we have in the Fromkin grammar on page 11, we can reduce to find any number of C 's at any point in any parse, as with BU recognition. For now, we simply eliminate that rule. Left recursion, on the other hand, though problematic for TD recognition, is not a problem for BU or LC. So for practice we can use the grammar `g0noe` that was presented on page 54. We have sessions like this, using a parser `lch.py` that returns the complete history of the parse:

```
>>> from lch import *
>>> from g0noe import *
>>> parse(g0noe, ['Sue', 'laughs'])
0 (['Sue', 'laughs'], ['S'])
1 (['laughs'], [( 'Sue', ), 'S'])
2 (['laughs'], [( 'Name', ), 'S'])
3 (['laughs'], [( 'DP', ), 'S'])
4 (['laughs'], ['VP'])
5 ([], [( 'laughs', ), 'VP'])
6 ([], [( 'V', ), 'VP'])
7 ([], [])
more? y
>>>
```

5.2 Assessment: time, space, and nondeterminism

5.2.1 Predictiveness and incrementality

Although we have only printed out tree representations at the end of a parse, it is clear that we could have formulated the tree representations built at every step. When we do that, we see that, at various points, the stack has parts of the derivation which are not connected to the root or to each other by rules that have already been used.

⇒ picture coming ⇐

5.2.2 Right branching derivations, regularity, and space

Among the grammars that the LC recognizer can handle are some that define only regular languages. In particular, we know that if a grammar has only right branching (or only left branching), the language it defines is regular.

Consider again the simple grammar from page 60, repeated here, that generates only 1 string, with a simple right branching derivation:

```
1 """ file: g2noe.py
2     this grammar has no empty productions
3     parse(g2noe, ['b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b', 'b'])
4     """
5 g2noe = [( 'S', ['B', 'B0']),
6          ( 'B0', ['B', 'B1']),
7          ( 'B1', ['B', 'B2']),
8          ( 'B2', ['B', 'B3']),
9          ( 'B3', ['B', 'B4']),
10         ( 'B4', ['B', 'B5']),
```



```

11         ('B5', ['B', 'B6']),
12         ('B6', ['B', 'B7']),
13         ('B7', ['B', 'B8']),
14         ('B8', ['B', 'B9']),
15         ('B9', ['B']),
16         ('B', ['b'])

```

This grammar accepts its single string on a long right branch:

```

>>> from g2noe import *
>>> from licht import *
>>> parse(g2noe, ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'])
steps= 1254462
0 (['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], ['S'])
1 (['b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('a',), 'S')])
2 (['b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('A',), 'S')])
3 (['b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], ['B0'])
4 (['c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('b',), 'B0')])
5 (['c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('B',), 'B0')])
6 (['c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], ['B1'])
7 (['d', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('c',), 'B1')])
8 (['d', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('C',), 'B1')])
9 (['d', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], ['B2'])
10 (['e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('d',), 'B2')])
11 (['e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('D',), 'B2')])
12 (['e', 'f', 'g', 'h', 'i', 'j', 'k'], ['B3'])
13 (['f', 'g', 'h', 'i', 'j', 'k'], [(('e',), 'B3')])
14 (['f', 'g', 'h', 'i', 'j', 'k'], [(('E',), 'B3')])
15 (['f', 'g', 'h', 'i', 'j', 'k'], ['B4'])
16 (['g', 'h', 'i', 'j', 'k'], [(('f',), 'B4')])
17 (['g', 'h', 'i', 'j', 'k'], [(('F',), 'B4')])
18 (['g', 'h', 'i', 'j', 'k'], ['B5'])
19 (['h', 'i', 'j', 'k'], [(('g',), 'B5')])
20 (['h', 'i', 'j', 'k'], [(('G',), 'B5')])
21 (['h', 'i', 'j', 'k'], ['B6'])
22 (['i', 'j', 'k'], [(('h',), 'B6')])
23 (['i', 'j', 'k'], [(('H',), 'B6')])
24 (['i', 'j', 'k'], ['B7'])
25 (['j', 'k'], [(('i',), 'B7')])
26 (['j', 'k'], [(('I',), 'B7')])
27 (['j', 'k'], ['B8'])
28 (['k'], [(('j',), 'B8')])
29 (['k'], [(('J',), 'B8')])
30 (['k'], ['B9'])
31 ([], [(('k',), 'B9')])
32 ([], [(('K',), 'B9')])
33 ([], [])
more? n

```

It is easy to see that with simple right branching derivations like this one, the amount of space required in the stack of predicted categories is bounded by a finite constant – no matter how long the string, the parser never needs more memory. However, calculating this parse takes longer than it should. . . even though each left corner is unambiguous! . . . efficiency is still a serious problem.

5.2.3 Left branching derivations, regularity, and space

Now let's look at left branching, using the grammar from page 61, repeated here:

```

1 """ file: g3noe.py
2     a left branching grammar (no left recursion!)
3     parse(g3noe, ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'])
4 """
5 g3noe = [(('S', ['B0', 'K']),
6         ('B0', ['B1', 'J']),
7         ('B1', ['B2', 'I']),
8         ('B2', ['B3', 'H']),
9         ('B3', ['B4', 'G']),
10        ('B4', ['B5', 'F']),
11        ('B5', ['B6', 'E']),
12        ('B6', ['B7', 'D']),
13        ('B7', ['B8', 'C']),
14        ('B8', ['B9', 'B']),

```

```

15      ('B9', ['A']),
16      ('A', ['a']),
17      ('B', ['b']),
18      ('C', ['c']),
19      ('D', ['d']),
20      ('E', ['e']),
21      ('F', ['f']),
22      ('G', ['g']),
23      ('H', ['h']),
24      ('I', ['i']),
25      ('J', ['j']),
26      ('K', ['k'])

```

The left branching in this grammar requires unbounded memory top-down, but bottom-up we find:

```

>>> from lcht import *
>>> from g3noe import *
>>> parse(g3noe, ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'])
steps= 12263
0 (['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], ['S'])
1 (['b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('a',), 'S')])
2 (['b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('A',), 'S')])
3 (['b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('B9',), 'S')])
4 (['b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('B', ('B8',), 'S'))])
5 (['c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('b',), 'B', ('B8',), 'S')])
6 (['c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('B8',), 'S')])
7 (['c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('C', ('B7',), 'S'))])
8 (['d', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('c',), 'C', ('B7',), 'S')])
9 (['d', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('B7',), 'S')])
10 (['d', 'e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('D', ('B6',), 'S'))])
11 (['e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('d',), 'D', ('B6',), 'S')])
12 (['e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('B6',), 'S')])
13 (['e', 'f', 'g', 'h', 'i', 'j', 'k'], [(('E', ('B5',), 'S'))])
14 (['f', 'g', 'h', 'i', 'j', 'k'], [(('e',), 'E', ('B5',), 'S')])
15 (['f', 'g', 'h', 'i', 'j', 'k'], [(('B5',), 'S')])
16 (['f', 'g', 'h', 'i', 'j', 'k'], [(('F', ('B4',), 'S'))])
17 (['g', 'h', 'i', 'j', 'k'], [(('f',), 'F', ('B4',), 'S')])
18 (['g', 'h', 'i', 'j', 'k'], [(('B4',), 'S')])
19 (['g', 'h', 'i', 'j', 'k'], [(('G', ('B3',), 'S'))])
20 (['h', 'i', 'j', 'k'], [(('g',), 'G', ('B3',), 'S')])
21 (['h', 'i', 'j', 'k'], [(('B3',), 'S')])
22 (['h', 'i', 'j', 'k'], [(('H', ('B2',), 'S'))])
23 (['i', 'j', 'k'], [(('h',), 'H', ('B2',), 'S')])
24 (['i', 'j', 'k'], [(('B2',), 'S')])
25 (['i', 'j', 'k'], [(('I', ('B1',), 'S'))])
26 (['j', 'k'], [(('i',), 'I', ('B1',), 'S')])
27 (['j', 'k'], [(('B1',), 'S')])
28 (['j', 'k'], [(('J', ('B0',), 'S'))])
29 (['k'], [(('j',), 'J', ('B0',), 'S')])
30 (['k'], [(('B0',), 'S')])
31 (['k'], [(('K')])
32 ([], [(('k',), 'K')])
33 ([], [])
more? n

```

There are never more than 3 elements in the parser memory, and clearly this would hold, no matter how long the left branch was.

(Optional) Exercise: Notice the step count shown here, and compare it to the step count shown for the right-branching example in the previous section. Why is calculating this left-branching parse so much faster? Why didn't we find a similar difference between right- and left-branching with the bottom-up parser (check the steps that we reported for these same examples for the BU parser).

5.2.4 Determinism: time, space, nontermination

As we mentioned in the previous chapter, the LC recognizer will fail to terminate if the grammar has empty categories, and even when it doesn't, the number of steps the parser takes can be on the order of k^n for some $k > 1$. This is because of nondeterminism!

'Lookahead' can reduce local ambiguity, but English is not LC(k) for any k .

⇒ more coming ⇐

Exercises: LC parsing

1. Implement left corner recognition in a file called `lc<YOURINITIALS>.py`, and put your name at the top of the file too.
2. Test your recognizer with some of the examples that are mentioned above (and any others you like!)
3. Email me the recognizer when it is working perfectly.

Optional exercises or squib topics for later:

4. After implementing the recognizer `lc.py`, implement as many of these as you can:
 - a recognizer with a step counter (a ‘timer’) `lct.py`,
 - a parser that collects complete histories `lch.py`,
 - a parser that collects complete histories, with a step counter `lcht.py`,
 - a parser that collects rules used `lcp.py`,
 - a parser with a step counter `lcpt.py`,
 - a function that converts the list of rules in LC order to a parse tree `lc2dt.py`.
5. Hemforth&al’98 mentions LC parsing as one of 3 ideas that are “still central in psycholinguistic research” [3, p299]. Are the parsing preferences suggested there really compatible with the LC model? (See me if you have trouble finding this paper.)
6. Manning&Schutze’99 say in their text that “left-corner parsing is a particularly interesting case: left-corner parsers work incrementally from left-to-right, combine top-down and bottom-up prediction, and hold pride of place in the family of Generalized Left Corner Parsing models” [5, p427]. What do they mean by “incrementally”? As discussed in class, it is not true that the completed parts of a parse are always connected. That is, there are grammars with sentences s such that the LC recognizer, in the correct parse, finds an unconnected part and does not connect it until k steps later, for $k > 0$. In this case let’s say that, given G , sentence s has disconnection time k . Define a grammar that accepts an infinite series of sentences s_0, s_1, \dots such that disconnection times k for these sentences increases without bound. (Choose a grammar that is linguistically natural, and sentences that are easy to understand.)

References

- [1] ABNEY, S. P., AND JOHNSON, M. Memory requirements and local ambiguities of parsing strategies. *Journal of Psycholinguistic Research* 20 (1991), 233–249.
- [2] AHO, A. V., AND ULLMAN, J. D. *The Theory of Parsing, Translation, and Compiling. Volume 1: Parsing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1972.
- [3] HEMFORTH, B., KONIECZNY, L., SCHEEPERS, C., AND STRUBE, G. Syntactic ambiguity resolution in German. In *Sentence Processing: A Crosslinguistic Perspective*, D. Hillert, Ed., Syntax and Semantics, Vol. 31. Academic Press, San Diego, 1998, pp. 293–312.
- [4] MANNING, C. D., AND CARPENTER, B. Probabilistic parsing using left corner language models. In *Proceedings of the 1997 International Workshop on Parsing Technologies* (1997). Reprinted in H. Bunt and A. Nijholt (eds.) *Advances in Probabilistic and Other Parsing Technologies*, Boston, Kluwer: pp. 105-124.
- [5] MANNING, C. D., AND SCHÜTZE, H. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, Massachusetts, 1999.
- [6] MOORE, R. C. Improved left-corner chart parsing for large context-free grammars. In *New Developments in Parsing Technology*, H. Bunt, J. Carroll, and G. Satta, Eds. Boston, Kluwer, 2004.
- [7] RESNIK, P. Left-corner parsing and psychological plausibility. In *Proceedings of the 14th International Conference on Computational Linguistics, COLING 92* (1992), pp. 191–197.
- [8] ROARK, B., AND JOHNSON, M. Efficient probabilistic top-down and left-corner parsing. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics* (1999), pp. 421–428.

Chapter 6 Generalized left-corner CF parsing

TD, BU, and LC differ in the point at which they trigger the use of a rule. This observation is made by Brosgol'73 [4], and by Demers'77 [7]. This observation sets the stage for a generalization. We can specify, for each rule, how much of the right side of the rule needs to be completed before we predict the rest to complete the left side. Let's call this 'generalized left-corner' recognition (GLC), even though various other generalizations of LC parsing are in the literature.¹

6.1 GLC backtrack recognition

Let's assume that each rule has its right hand side broken into two parts like this:

$S \rightarrow ,DP VP$	top-down
$S \rightarrow DP, VP$	left-corner
$S \rightarrow DP VP,$	bottom-up

In general for a rule with a right hand side of length n , there are $n + 1$ possible triggers. The triggers can differ for each rule. For example, let's call this grammar g121111221111, where each number corresponds to the length of the trigger of the rule, with the rules are given row-by-row as follows:

$S \rightarrow DP , VP$	$DP \rightarrow D NP ,$	$NP \rightarrow NP , PP$	$NP \rightarrow N ,$
$VP \rightarrow V , DP$	$DP \rightarrow V ,$	$CP \rightarrow C S ,$	$PP \rightarrow P DP ,$
$P \rightarrow of ,$	$D \rightarrow the ,$	$N \rightarrow student ,$	$V \rightarrow knows ,$

Using backtrack search, the algorithm is this:

```

GLC BACKTRACK CF RECOGNITION( $G, i$ )
0  ds=[(i,S)] where S is the start category
1  while ds $\neq$  [] and ds[0] $\neq$ ([],[]):
2      ds=[d| ds[0]  $\Rightarrow_{glc}$  d] + ds[1:]
3  if ds==[] then False else True
    
```

where the steps \Rightarrow_{glc} are defined as follows:

$$\begin{aligned}
 & \text{(glc-reduce)} \frac{\text{input}, (X_i) \dots (X_1) \alpha}{\text{input}, X_{i+1} \dots X_n (X) \alpha} \text{ if } X \rightarrow X_1 \dots X_i, X_{i+1} \dots X_n \\
 & \text{(glc-reduce-complete)} \frac{\text{input}, (X_i) \dots (X_1) X \alpha}{\text{input}, X_{i+1} \dots X_n \alpha} \text{ if } X \rightarrow X_1 \dots X_i, X_{i+1} \dots X_n \\
 & \text{(shift)} \frac{w \text{ input}, \alpha}{\text{input}, (w) \alpha} \\
 & \text{(shift-complete)} \frac{w \text{ input}, w \alpha}{\text{input}, \alpha}
 \end{aligned}$$

The reduce rule says that if you have a completed trigger $(X_i) \dots (X_1)$ on top of the stack, the "left corner" of the rule $X \rightarrow X_1 \dots X_i X_{i+1} \dots X_n$, then you can predict $X_{i+1} \dots X_n$ in order to have a completed (X) . The reduce-complete rule is similar, except that the X that is completed is already there as a predicted element, and so they cancel each other out, as in LC parsing. Also notice that the trigger, the rest of the right hand side, or both can have length 0.

¹Ritchie calls our parsing strategies 'extended generalized left corner' [12, p.482]. Note that these parsing methods are 'arc-eager' in the sense of Abney and Johnson'91.

We have sessions like this, using a parser `glch.py` that returns the complete history of the parse:

```
>>> from glch import *
>>> from g121111221111 import *
>>> parse(g10,['the','student','knows','the','student'])
0 (['the', 'student', 'knows', 'the', 'student'], ['S'])
1 (['student', 'knows', 'the', 'student'], [(('the',), 'S')])
2 (['student', 'knows', 'the', 'student'], [(('D',), 'S')])
3 (['knows', 'the', 'student'], [(('student',), ('D',), 'S')])
4 (['knows', 'the', 'student'], [(('N',), ('D',), 'S')])
5 (['knows', 'the', 'student'], [(('NP',), ('D',), 'S')])
6 (['knows', 'the', 'student'], [(('DP',), 'S')])
7 (['knows', 'the', 'student'], ['VP'])
8 (['the', 'student'], [(('knows',), 'VP')])
9 (['the', 'student'], [(('V',), 'VP')])
10 (['the', 'student'], ['DP'])
11 (['student'], [(('the',), 'DP')])
12 (['student'], [(('D',), 'DP')])
13 ([], [(('student',), ('D',), 'DP')])
14 ([], [(('N',), ('D',), 'DP')])
15 ([], [(('NP',), ('D',), 'DP')])
16 ([], [])
more?
```

If we use a grammar with empty triggers, the GLC backtracking parser can fail to terminate.

A simple beam parser can be constructed by adapting the TD beam parser §3 so that it uses GLC rules instead of TD rules. Then the beam parser can avoid nontermination if every parse that can be extended at all can be extended in more than one way. But it can still be very inefficient. For example, consider this tiny grammar:

```
1 """ file: g2111.py
2     for the glc recognizer
3 """
4 g2111 = [(('DP', ['D', 'NP'], []),
5         ('NP', ['N'], []),
6         ('D', ['the'], []),
7         ('N', ['student'], []))]
```

With our GLC beam parser we find:

```
>>> from glcvt import *
>>> from g2111 import *
>>> recognize(g2111,['the','student'],-0.1)
steps= 10
probability= 0.25
True
```

If we change the grammar so that just one rule has an empty left corner, nontermination can be a problem, so let's modify the triggers so that two rules have empty left corners:

```

1  """ file: g0101.py
2      for the glc recognizer
3  """
4  g0101 = [('DP', [], ['D', 'NP']),
5           ('NP', ['N'], []),
6           ('D', [], ['the']),
7           ('N', ['student'], [])]

```

Now, with our GLC beam parser we find:

```

>>> from glc import *
>>> from g0101 import *
>>> recognize(g0101, ['the', 'student'], -0.1)
steps= 5366
probability= 0.000434027777778
True

```

Things are very much worse with a larger grammar like `g0` or `g1` – GLC parsing is typically infeasible if there are any empty left corners.

6.2 Assessment

6.2.1 The space of GLC recognizers

As Demers'77 [7] points out, for any grammar, the collection of trigger functions F_r for each rule r can be naturally partially ordered by top-downness:

$$F_1 \leq F_2 \text{ if and only if for every production } p, \text{ the trigger } F_1(p) \text{ is at least as long as } F_2(p).$$

In other words, a setting of triggers F_1 is as bottom-up as F_2 if and only if for every production p , the triggering point defined by F_1 is at least as far to the right as the triggering point defined by F_2 . It is easy to see that $\langle (F_r), \leq \rangle$ is a lattice, as Demers claims, since for any collection (F_r) of trigger functions for any grammar, the least upper bound of (F_r) is just the function which maps each rule to the trigger which is the shortest of the triggers assigned by any function in (F_r) , and the greatest lower bound of (F_r) is the function which maps each rule to the trigger which is the longest assigned by any function in (F_r) . Furthermore, the lattice is finite.² The simple lattice structure for a 3 rule grammar can be depicted like this:

²It is easy to see that for any single production p with right side of length $|p|$, there is a “chain lattice” of $|p| + 1$ recognition strategies for that rule. The GLC lattice for a grammar is the lattice product of these chains. For a discussion of lattice products, see for example [6, §1.26].

6.2.2 Oracles

We saw in the previous chapter that the TD, BU, and LC recognizers will fail to terminate in certain cases, and even when they do terminate, the number of steps taken on input of length n can be on the order of k^n for some $k > 1$. This is because of nondeterminism! In §4.7 on page 63 we noticed that the nondeterminism can be partitioned into 3 kinds:

- Given a particular grammar G , the recognizer has a problem with **consistency** if it is pursuing derivations which could not possibly succeed, no matter what the input was.
- Given a particular grammar G and a particular input i , let's say that the recognizer has a problem with **local ambiguity** if it is pursuing dead ends that could not work given input i .
- Given a particular grammar G and a particular input i , let's say that the recognizer has a problem with **global ambiguity** if there are multiple grammatical parses.

To get a parser which is as fast and reliable as the human parser, we want to reduce all of these sorts of indeterminacy to manageable levels. Let's consider each of these in turn.

Consistency oracle

Some stacks cannot possibly be reduced to empty, no matter what input string is provided. In particular:

- There is no point in shifting a word if it cannot be part of the trigger of the most recently predicted category.
- There is no point in building a constituent (i.e. using a rule) if the parent category cannot be part of the trigger of the most recently predicted category.

These conditions can be enforced by calculating, for each category C that could possibly be predicted, all of the stack sequences which could possibly be part of a trigger for C .

- For TD, the triggers are always empty.
- For LC, the trigger sequences are one symbol long (unless empty productions are allowed, which must have empty triggers).
- For BU, trigger sequences are the lengths of the rhs of the rules.

Given a context free grammar $G = \langle \Sigma, N, \rightarrow, S \rangle$, we can generate instances of the **is a beginning of** relation with the following logic.

$$\begin{aligned} \text{(trigger)} & \frac{}{(x_1 \dots x_i, C)} C \rightarrow x_1 \dots x_i, x_{i+1} \dots x_n \\ \text{(unshift)} & \frac{(x_1 \dots x_i, C)}{(x_1 \dots x_{i-1}, C)} \\ \text{(unreduce)} & \frac{(x_1 \dots x_i, C)}{(x_1 \dots x_{i-1} y_1 \dots y_j, C)} x_i \rightarrow y_1 \dots y_j, y_{j+1} \dots y_n \end{aligned}$$

Clearly this last rule can recursively define an infinite set of beginnings.

Example: Consider the following grammar g5mix with the triggers indicated:

$$\begin{array}{lll} \text{IP} \rightarrow \text{DP I1}, & \text{I1} \rightarrow \text{,I0 VP} & \text{I0} \rightarrow \text{,will.} \\ \text{DP} \rightarrow \text{D1}, & \text{D1} \rightarrow \text{D0, NP} & \text{D0} \rightarrow \text{the,} \\ \text{NP} \rightarrow \text{,N1} & \text{N1} \rightarrow \text{N0,} & \text{N0} \rightarrow \text{idea,} \\ & \text{N1} \rightarrow \text{,N0 CP} & \\ \text{VP} \rightarrow \text{,V1} & \text{V1} \rightarrow \text{V0,} & \text{V0} \rightarrow \text{,suffice} \\ \text{CP} \rightarrow \text{C1}, & \text{C1} \rightarrow \text{C0, IP} & \text{C0} \rightarrow \text{that,} \end{array}$$

It is not hard to see that in this case, the beginnings of each category are finite. For example, the following proof shows that the beginnings of IP include DP I1, DP, D1, D0, **the**, and ϵ :

$$\frac{(\text{DP I1,IP})}{(\text{DP,IP})} (\text{trigger})$$

$$\frac{(\text{DP,IP})}{(\text{D1,IP})} (\text{unreduce})$$

$$\frac{(\text{D1,IP})}{(\text{D0,IP})} (\text{unreduce})$$

$$\frac{(\text{D0,IP})}{(\text{the,IP})} (\text{unreduce})$$

$$\frac{(\text{the,IP})}{(\epsilon,IP)} (\text{unshift})$$

Notice that the beginnings of IP do not include the idea, the I1, D0 I0, I0, or I1.

GLC recognition with an oracle is defined so that whenever a completed category is placed on the stack, the resulting sequence of completed categories on the stack must be a beginning of the most recently predicted category. So let's say that a sequence C is **reducible** iff the sequence C is the concatenation of two sequences $C = C_1C_2$ where

1. C_1 is a sequence $(A_i), \dots, (A_1)$ of $0 \leq i$ completed elements
2. C_2 begins with a predicted element C , and
3. A_1, \dots, A_i is a beginning of C

When the beginnings of a category can be infinite, though, we cannot precompute an explicit list of them. In that case, as we see in standard approaches to LR parsing, it is common to use an automaton to define them, but now there are other perspectives which we will mention in §7.3 below.

Using this definition of reducibility (which can sometimes but not always be explicitly calculated), we can add a first oracle to GLC parsing by defining the steps \Rightarrow_{glco} as follows:

$$\text{(glc-reduce)} \frac{\text{input}, (X_i) \dots (X_1) \alpha}{\text{input}, X_{i+1} \dots X_n (X) \alpha} \text{ if } X \rightarrow X_1 \dots X_i, X_{i+1} \dots X_n \text{ and } (X)\alpha \text{ is reducible}$$

$$\text{(glc-reduce-complete)} \frac{\text{input}, (X_i) \dots (X_1) X \alpha}{\text{input}, X_{i+1} \dots X_n \alpha} \text{ if } X \rightarrow X_1 \dots X_i, X_{i+1} \dots X_n$$

$$\text{(shift)} \frac{w \text{ input}, \alpha}{\text{input}, (w)\alpha} \text{ if } (w)\alpha \text{ is reducible}$$

$$\text{(shift-complete)} \frac{w \text{ input}, w \alpha}{\text{input}, \alpha}$$

Now let's consider what we can do with a lookahead oracle.

Lookahead oracle

For the top-down steps, when we place predictions onto the stack, there is no point in using an expansion $A \rightarrow \alpha, \beta$, predicting β , if the next symbol to be parsed could not possibly be the first symbol of a β . So by looking ahead we can avoid some dead ends. This “bottom-up” information can be provided with a “lookahead oracle.” Obviously, the lookahead oracle does not look into the future to hear what has not been spoken yet. Rather, the parser lags k words behind the buffering of the input.

In ordinary, clear conversation, humans do not do this! As mentioned before, evidence suggests that, at least typically, we analyze what we hear word-by-word [3, 5, 13]. But let's see how lookahead could be implemented, in case we wanted to use it in unusual situations (or in engineering tasks where modeling human abilities is less important than processing large amounts of data quickly).

We can precompute, for each category p , what the first k symbols of the string could be when we are recognizing that category in a successful derivation of any sentence. Obviously, this will always be finite and can be kept in a table. And obviously, in calculating lookahead, we ignore the triggers, since the triggers are completed elements on the stack. We want lookahead only to keep predictions in check.

One kind of situation that we must allow for is this. If we are predicting A_1, \dots, A_n and $A_1 \dots A_i \Rightarrow^* \epsilon$, then first symbol of A_{i+1} (if there is one) is the first symbol for A_1, \dots, A_n .

Since we allow empty productions, we first calculate which categories of our grammar can be empty using the recursive definition:

$$\text{empty}(A) = \text{True iff } A \rightarrow \epsilon \vee (A \rightarrow A_1 \dots A_n \wedge \text{empty}(A_i) \text{ for all } 1 \leq i \leq n).$$

With that practice, we can recursively define $\text{first}(A)$ for each category A , as follows:

$$w \in \text{first}(A) \text{ iff } \begin{cases} A \rightarrow w\alpha, \text{ or} \\ A \rightarrow B\alpha \wedge w \in \text{first}(B), \text{ or} \\ A \rightarrow B_1 \dots B_{i-1} B_i \alpha \wedge \text{empty}(B_1) \wedge \dots \wedge \text{empty}(B_{i-1}) \wedge w \in \text{first}(B_i). \end{cases}$$

For each nonterminal w , of course, $\text{first}(w) = \{w\}$. So then finally, consider any grammar G with trigger-marked rules

$$A \rightarrow \alpha, \beta.$$

For every suffix $\beta = x_1 \dots x_n$ in the grammar, we define³

$$\text{first}(x_1 \dots x_n) = \{w \mid w\alpha \in \text{first}(x_1) \times \dots \times \text{first}(x_n)\}.$$

That is, $\text{first}(x_1 \dots x_n)$ is the set of first words w of elements of the Cartesian product $\text{first}(x_1) \times \dots \times \text{first}(x_n)$. This strategy can be generalized to compute the first k elements of any sequence of categories, as described in [2, pp357-8] or [8, pp239f,254ff].

Using this definition of first , we can add a second oracle to GLC parsing by defining the steps \Rightarrow_{glco} as follows:

$$\begin{aligned} (\text{glc-reduce}) \quad & \frac{\text{input}, (X_i) \dots (X_1) \alpha}{\text{input}, X_{i+1} \dots X_n (X) \alpha} \text{ if } X \rightarrow X_1 \dots X_i, X_{i+1} \dots X_n, \quad \text{and } (X)\alpha \text{ is reducible,} \\ & \quad \text{and } \text{first}(\text{input}) \subseteq \text{first}(X_{i+1} \dots X_n) \\ (\text{glc-reduce-complete}) \quad & \frac{\text{input}, (X_i) \dots (X_1) X \alpha}{\text{input}, X_{i+1} \dots X_n \alpha} \text{ if } X \rightarrow X_1 \dots X_i, X_{i+1} \dots X_n, \text{ and } \text{first}(\text{input}) \subseteq \text{first}(X_{i+1} \dots X_n) \\ (\text{shift}) \quad & \frac{w \text{ input}, \alpha}{\text{input}, (w)\alpha} \text{ if } (w)\alpha \text{ is reducible} \\ (\text{shift-complete}) \quad & \frac{w \text{ input}, w \alpha}{\text{input}, \alpha} \end{aligned}$$

Global ambiguity oracle

So far, we are using the beam to throw out parses that require too much guessing, but we can do much better! We will return to this later.

³Here, for sets s_1, s_2 , I am writing $s_1 \times s_2$ for the set of strings concatenated from s_1 and s_2 . That is,

$$s_1 \times s_2 = \{x_1 x_2 \mid x_1 \in s_1 \wedge x_2 \in s_2\}.$$

Exercises: GLC parsing and calculating the oracles

1. Copy my `empties.py` to a new file called `first<YOUR-INITIALS>.py`.
2. **The parses we want to get.** In a comment, write out the 16 step GLC derivation of the sentence

the idea will suffice

from `g5mix`, using the triggers indicated in that grammar. This grammar is shown on 77 above and is in the file `g5mix.py`. Write the derivation out with lines numbered from 0 to 16, exactly as done in class. (I am thinking you will type this derivation in by hand, but you can calculate it instead if you see a way to do that.)

3. **The look-ahead oracle that will help us get those parses.** Let's calculate `firstsOfCats(g)` which takes a grammar `g` with rules (lhs, α, β) and returns a list of pairs $(A, firsts)$ where $firsts$ is the list of words that can begin A , `first(A)`. The calculation is done as follows:

- Initialize the first sets to empty lists
- Then process the grammar rules (lhs, rhs) as follows, until nothing is added:
 - if the rhs starts with a terminal w , add w to `first(lhs)`
 - if the rhs starts with nonterminal A , add `first(A)` to `first(lhs)`
 - if the rhs starts with nonterminals $A_1 \dots A_i$ where $empty(A_1) \wedge \dots \wedge empty(A_{i-1})$, add `first(A_i)` to `first(lhs)`

(I will check your code by applying it to the grammar `g5mix.py`. You should check it that way too!)

References

- [1] ABNEY, S. P., AND JOHNSON, M. Memory requirements and local ambiguities of parsing strategies. *Journal of Psycholinguistic Research* 20 (1991), 233–249.
- [2] AHO, A. V., AND ULLMAN, J. D. *The Theory of Parsing, Translation, and Compiling. Volume 1: Parsing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1972.
- [3] AOSHIMA, S., YOSHIDA, M., AND PHILLIPS, C. Incremental processing of coreference and binding in Japanese. *Syntax* 12, 2 (2009), 93–134.
- [4] BROSGOL, B. M. *Deterministic Translation Grammars*. PhD thesis, Harvard University, 1973.
- [5] CHAMBERS, C. G., TANENHAUS, M. K., EBERHARD, K. M., FILIP, H., AND CARLSON, G. N. Actions and affordances in syntactic ambiguity resolution. *Journal of Experimental Psychology: Learning, Memory and Cognition* 30, 3 (2004), 687–696.
- [6] DAVEY, B., AND PRIESTLEY, H. *Introduction to Lattices and Order*. Cambridge University Press, NY, 1990.
- [7] DEMERS, A. J. Generalized left corner parsing. In *Conference Report of the 4th Annual Association for Computing Machinery Symposium on Principles of Programming Languages* (1977), pp. 170–181.
- [8] GRUNE, D., AND JACOBS, C. J. H. *Parsing Techniques: A Practical Guide*. Springer, NY, 2008.
- [9] NEDERHOF, M.-J. Generalized left-corner parsing. In *Proceedings of the 6th Conference of the European Association for Computational Linguistics* (1993), pp. 305–314.
- [10] NIJHOLT, A. *Context Free Grammars: Covers, Normal Forms, and Parsing*. Springer-Verlag, NY, 1980.
- [11] NOZOOHOR-FARSHI, R. GLR parsing for epsilon-grammars. In *Generalized LR Parsing*, M. Tomita, Ed. Kluwer, Boston, 1991, pp. 61–76.
- [12] RITCHIE, G. Completeness conditions for mixed strategy context free parsing. *Computational Linguistics* 25, 4 (1999), 457–486.
- [13] STURT, P. The psycholinguistics of structural composition. In *LINCOM Studies in Theoretical linguistics* (2011), R. K. Mishra and N. Srinivasan, Eds., vol. 5, pp. 14–32.

Chapter 7 Dynamic methods: CKY, Earley

If I had to choose between that and the Matrix, I'd choose the Matrix.
– Cypher, in *The Matrix*

The stack-based GLC approaches to parsing calculate one derivation at a time. When we do not know which path to take, we can choose one and keep a list (a ‘backtrack stack’ or a ‘beam’) of the choices not taken, in case our first choices do not work out. That kind of approach seems to fit with natural hypotheses about incremental interpretation and garden path effects in human language use. And that kind of approach fits with the fact that some (but not all!) very long sentences are fairly easy to understand, even though our grammars are large.¹ However, the stack-based methods face difficult problems which we have not handled adequately yet. Putting it bluntly, the methods we have looked at *do not work*. They are nonterminating with some of the rules we want to use, and even when we eliminate those rules, they are intractable. We have not found a way to use them yet that could be a reasonable starting point for models of human sentence recognition or production.

Rather than developing one possible derivation at a time as GLC methods do, maybe it would be better to work on whole sets of solutions at once. This kind of step is familiar in mathematics, when we graduate from simple arithmetic to algebra. Instead of computing 1 product, we can compute many at once, and reason about whole sets of solutions. A linear equation with 3 variables defines an infinite plane of solutions. With two of these equations, the intersection of two different planes is a continuous line of solutions, and a third equation can restrict the solutions to a point. This kind of calculation is usually done with matrices of real numbers, representing the points in Euclidean space \mathbb{R}^n . But linguistics is (mainly) discrete, not continuous. That is, we humans categorize the continuous flux of our experience and reason categorically, discretely. We are digital, not analog computers. The phrases “the stuffy nose” and “the stuff he knows” are usually acoustically similar, but not syntactically or semantically or pragmatically similar. 350 years after Euclid, Diophantus began the move from reasoning about continuous Euclidean spaces with real numbers to reasoning on the grid, looking for solutions in tuples of integers \mathbb{Z}^n . The ideas in this chapter will be more like that. Numbering our categories with natural numbers $\mathbb{N} = \{0, 1, \dots\}$, we look for solutions in \mathbb{N}^n . The recognition methods in this chapter use matrix-based ‘dynamic programming’ methods to collect all parts of all solutions, in a matrix. The discussion of dynamic programming in a good text on algorithms like [7, §15]² is strongly recommended. We will use some fundamental but surprising strategies to compute ‘all solutions’, that is, the transitive closure of a set with respect to a function or a set of functions).

¹Charniak & al’98 observe that “...for large grammars (such as the PCFG induced from the Penn II [*Wall Street Journal*] corpus, which contains around $1.6 \cdot 10^4$ rules) and longish sentences (say, 40 words and punctuation), even $\mathcal{O}(n^3)$ looks pretty bleak. . . In our work, we have found that exhaustively parsing maximum-40-word sentences from the Penn II treebank requires an average of about 1.2 million edges per sentence.” There is also popular interest in very very long sentences, especially intelligible ones. A column in *The Guardian* (Friday Nov 30, 2007) says: “For many years the longest sentence in English literature belonged to James Joyce, with a 4,391-word section of Molly Bloom’s *Ulysses* soliloquy. Then, in 2001, came Benjamin Trotter’s 13,955-word effort in Jonathan Coe’s *The Rotters’ Club*. Now we have Nigel Tomm’s one-sentence, 469,375-word book, *The Blah Story, Volume 4*.” In the *New York Times* (on page BR27 of the Sunday Book Review, Dec 26, 2010) we read “...Czech novelist Bohumil Hrabal’s *Dancing Lessons for the Advanced in Age* (1964)... unfurls as a single, sometimes maddening sentence that ends after 117 pages without a period, giving the impression that the opinionated, randy old cobbler will go on jawing ad infinitum. But the gambit works. His exuberant ramblings gain a propulsion that would be lost if the comma splices were curbed, the phrases divided into sentences. . . The Polish novelist Jerzy Andrzejewski went even longer in *The Gates of Paradise* (1960), weaving several voices into a lurid and majestic 158-page run-on. (The novel actually consists of two sentences, the final one a mere five words long.) . . . An 800-plus worder in Victor Hugo’s *Les Misérables* (1862) has sometimes (erroneously) been cited as the longest in French literature; its winding description of Louis Philippe as a ruler who hews to the middle of the road in every aspect (‘well read and caring but little for literature,’ ‘incapable of rancor and of gratitude’), damns the king’s modesty with the grandness of its design. Is it coincidence that, as Roger Shattuck points out, the longest sentence in Proust – 944 words – dissects the plight of the homosexual in society? And what about the last of the six immense paragraphs that constitute Gabriel García Márquez’s *Autumn of the Patriarch* (1975), one mammoth sentence concluding with ‘the good news that the uncountable time of eternity had come to an end’?” Checking these examples, most of them have, in effect, coordinated sentences that are neither separated by periods nor connected by *and*, but just listed with commas. Many such sentences are perfectly intelligible, and so any adequate model of human parsing should handle them. See the Márquez example given below on page 88.

²Cf. [28, §8], [25, §37], [1, §10.2].

DIOPHANTI
ALEXANDRINI
ARITHMETICORVM
LIBRI SEX.

ET DE NVMERIS MVLTANGVLIS
LIBER VNVS.

*Nunc primum Græcè & Latinè editi, atque absolutissimis
Commentariis illustrati.*

AVCTORE CLAVDIO GASPARE BACHETO
MEZIRIACO SEBVSIANO, V. C.



LVTETIAE PARISIORVM,
Sumptibus SEBASTIANI CRAMOISY, via
Iacobæ, sub Ciconiis.

M. DC. XXI.

CVM PRIVILEGIO REGIS.

7.1 CKY recognition

For simplicity, we first consider context free grammars in Chomsky normal form. Chomsky normal form grammars have rules of only the following forms, for some nonterminals $A, B, C \in N$, and pronounced elements $w \in \Sigma$,

$$A \rightarrow BC \quad A \rightarrow w$$

If ϵ is in the language, then the following form is also allowed, subject to the requirement that S does not occur on the right side of any rule:

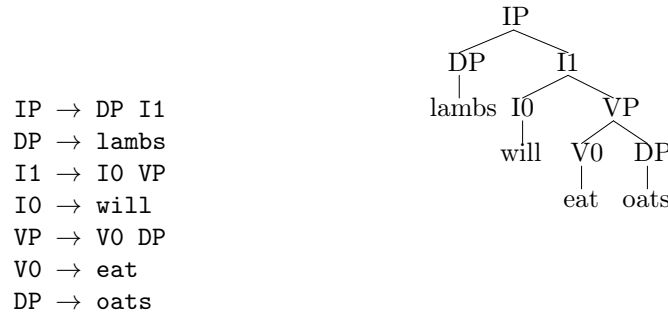
$$S \rightarrow \epsilon$$

Here, we will leave out the case of languages that contain ϵ . (It is of technical interest only, and could easily be covered if needed.) A Chomsky normal form grammar has no unit or empty productions (except possibly $S \rightarrow \epsilon$), there are no “cycles” $A \Rightarrow^+ A$, and no infinitely ambiguous strings. And these grammars allow an especially simple matrix-based recognition method. To parse a string w_1, \dots, w_n , for $n > 0$ we use the following logic:

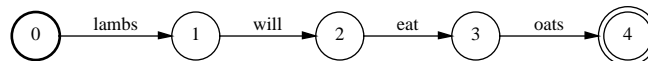
$$\begin{array}{l} \overline{(i-1, i) : A} \quad \text{[reduce1]} \quad \text{if } A \rightarrow w_i \\ \frac{(i, j) : B \quad (j, k) : C}{(i, k) : A} \quad \text{[reduce2]} \quad \text{if } A \rightarrow BC \end{array}$$

Notice how similar these rules are to the BU parser rules: the first is like a shift and reduce, and the second is like a single reduce step, *except that the rules are not being kept in a stack*. Instead, these results are kept in a matrix indexed by positions $0 \dots n$, and individual parses are not kept distinct. Recognition is successful iff $(0, n) : S$ is in the closure of the axioms under these inference rules. In that case, we have found an S between positions 0 and n .

Consider this grammar for example



The axioms can be regarded as specifying a finite state machine representation of the input:



Given an n state finite state machine representation of the input, computing the CKY closure can be regarded as filling in the “upper triangle” of an $n \times n$ matrix, from the (empty) diagonal up:

	0	1	2	3	4
0		DP			IP
1			IO		I1
2				V0	VP
3					DP
4					

(It is easy to generalize the CKY method to accept not strings but arbitrary finite state machines.)

The CKY algorithm provides an efficient way to calculate the complete matrix ‘in one sweep’. The idea is to start by filling the lexical categories, just above the diagonal – these are constituents of length 1. Then we consider constituents of length 2,3,...n. This is a variant of the Floyd-Warshall algorithm for computing transitive closures, emerging from the work of Cocke, Kasami and Younger [14, 34, 6, 2, 26, 27], rather like a single matrix multiplication step, filling the upper triangle of the matrix:

```

CKY CF RECOGNITION( $G, w_0 \dots w_n$ )
0   m = a square matrix of size  $(n+1) \times (n+1)$ 
1   for  $A \rightarrow w_i$ :
2     m[i-1][i].append(A)
3   for width  $\in \{2, \dots, n\}$ :
4     for start  $\in \{0, \dots, n-1\}$ :
5       end = start + width
6       for mid  $\in \{start+1, \dots, mid-1\}$ :
7         for  $A \rightarrow BC$ :
8           if  $B \in m[start][mid]$  and  $C \in m[mid][end]$ :
9             m[start][end].append(lhs)
10  if  $S \in m[0][end]$  then True else False

```

We can picture its operation in the loop that begins in line 4: we go from widths 2 to n-1, building constituents from each start position, and with each possible midpoint between the start and start+width. It is easy to see that this algorithm completes within time proportional to n^3 , since we have the three loops in lines 3,4,6 bounded by n, with a fixed amount of computation in each.

We can implement this algorithm quite directly in python:

```

1  """ cky.py    E. Stabler, 2013-02-15
2      Recognizer for CFG in CNF form
3  """
4  # from g0 import *
5
6  def initializeMatrix(w,PL,matrix): # insert lexical items w=w1 w2 ... wn
7      for (i,wi) in enumerate(w):      # enumerate numbers the elements of w from 0
8          for (lhs,rhs) in PL:
9              if rhs == wi:
10                 matrix[i][i+1].append(lhs) # so if lhs -> w1, lhs in matrix[0][1]
11
12 def closeMatrix(P,matrix,length):
13     positions = length+1 # intuitively, we add position 0
14     for width in range(2,positions):
15         for start in range(positions-width):
16             end = start + width
17             for mid in range(start+1,end): # so then range stops with end-1
18                 for (lhs,y,z) in P:
19                     if y in matrix[start][mid] and z in matrix[mid][end]:
20                         matrix[start][end].append(lhs)
21
22 def accepts((PL,P),w):
23     length = len(w)
24     positions = length + 1 # intuitively, we add position 0
25     matrix = [ [ [] for i in range(positions) ] for j in range(positions) ]
26     initializeMatrix(w,PL,matrix)
27     closeMatrix(P,matrix,length)
28     showMatrix(matrix)
29     return 'S' in matrix[0][length]
30
31 def showMatrix(m):
32     for row in m:
33         print row
34
35 #from g0cnf import *
36 #print accepts((l0cnf,p0cnf),['Sue','laughs'])
37 #print accepts((l0cnf,p0cnf),['Sue','praises','Maria','and','the','student'])
38 #print accepts((l0cnf,p0cnf),['praises','Maria','and','the','student'])
39 #print accepts((l0cnf,p0cnf),['Sue','praises','Maria','and','the','student','knows','it'])

```

Let’s adapt our first grammar so that the rules are all in Chomsky normal form (with no empty productions), and let’s divide the lexical, terminal rules from the binary nonterminal rules:

```

1  """ file: g0cnf.py
2  Here we adapt parts of our first grammar g0.py to CNF, and
3  we separate binary nonlexical from unary lexical productions.
4  This grammar has left (and right) recursion but no empty productions.
5  """
6  p0cnf = [('S', 'DP', 'VP'),
7          ('S', 'DP', 'V'), # for intransitive
8          ('DP', 'D', 'N'),
9          ('DP', 'D', 'NP'),
10         ('NP', 'N', 'PP'),
11         ('NP', 'NP', 'PP'), # left rec
12         ('NP', 'A', 'NP'),
13         ('VP', 'V', 'DP'),
14         ('VP', 'V', 'PP'),
15         ('VP', 'Adv', 'VP'),
16         ('VP', 'VP', 'Adv'), # left rec
17         ('VP', 'V', 'CP'),
18         ('PP', 'P', 'DP'),
19         ('CP', 'C', 'S'),
20         ('A', 'Adv', 'A'),
21         ('S', 'S', 'CoordS'), # left rec
22         ('CoordS', 'Coord', 'S'),
23         ('D', 'D', 'CoordD'), # left rec
24         ('CoordD', 'Coord', 'D'),
25         ('V', 'V', 'CoordV'), # left rec
26         ('CoordV', 'Coord', 'V'),
27         ('N', 'N', 'CoordN'), # left rec
28         ('CoordN', 'Coord', 'N'),
29         ('A', 'A', 'CoordA'), # left rec
30         ('CoordA', 'Coord', 'A'),
31         ('Adv', 'Adv', 'CoordAdv'), # left rec
32         ('CoordAdv', 'Coord', 'Adv'),
33         ('P', 'P', 'CoordP'), # left rec
34         ('CoordP', 'Coord', 'P'),
35         ('Adv', 'Adv', 'CoordAdv'),
36         ('CoordAdv', 'Coord', 'Adv'),
37         ('VP', 'VP', 'CoordVP'), # left rec
38         ('CoordVP', 'Coord', 'VP'),
39         ('NP', 'NP', 'CoordNP'), # left rec
40         ('CoordNP', 'Coord', 'NP'),
41         ('DP', 'DP', 'CoordDP'), # left rec
42         ('CoordDP', 'Coord', 'DP'),
43         ('PP', 'PP', 'CoordPP'), # left rec
44         ('CoordPP', 'Coord', 'PP'),
45         ('CP', 'CP', 'CoordCP'), # left rec
46         ('CP', 'Coord', 'CP')]
47
48  10cnf = [('D', 'the'), # now the lexical rules
49          ('D', 'a'),
50          ('D', 'some'),
51          ('D', 'every'),
52          ('D', 'one'),
53          ('D', 'two'),
54          ('A', 'gentle'),
55          ('A', 'clear'),
56          ('A', 'honest'),
57          ('A', 'compassionate'),
58          ('A', 'brave'),
59          ('A', 'kind'),
60          ('N', 'student'),
61          ('N', 'teacher'),
62          ('N', 'city'),
63          ('N', 'university'),
64          ('N', 'beer'),
65          ('N', 'wine'),
66          ('V', 'laughs'),
67          ('V', 'cries'),
68          ('V', 'praises'),
69          ('V', 'criticizes'),
70          ('V', 'says'),
71          ('V', 'knows'),
72          ('Adv', 'happily'),
73          ('Adv', 'sadly'),
74          ('Adv', 'impartially'),

```

```

75     ('Adv', 'generously'),
76     ('DP', 'Bill'),
77     ('DP', 'Sue'),
78     ('DP', 'Jose'),
79     ('DP', 'Maria'),
80     ('DP', 'Monday'),
81     ('DP', 'Tuesday'),
82     ('DP', 'he'),
83     ('DP', 'she'),
84     ('DP', 'it'),
85     ('DP', 'him'),
86     ('DP', 'her'),
87     ('P', 'in'),
88     ('P', 'on'),
89     ('P', 'with'),
90     ('P', 'by'),
91     ('P', 'to'),
92     ('P', 'from'),
93     ('C', 'that'),
94     ('C', 'whether'),
95     ('Coord', 'and'),
96     ('Coord', 'or'),
97     ('Coord', 'but')]

```

With this grammar, we have sessions like this:

```

>>> from g0cnf import *
>>> from ckyt import *
>>> print accepts((l0cnf,p0cnf),['Sue','laughs'])
chart= [[[], ['DP'], ['S']], [[], [], ['V']], [[], [], []]]
steps= 3
True
>>> print accepts((l0cnf,p0cnf),['laughs','Sue'])
chart= [[[], ['V'], ['VP']], [[], [], ['DP']], [[], [], []]]
steps= 3
False
>>> print accepts((l0cnf,p0cnf),['Sue','praises','Maria','and','the','student'])
chart= [[[], ['DP'], ['S'], ['S'], [], [], ['S']], [[], [], ['V'], ['VP'], [], [], ['VP']], [[], [], [], ['DP'], [], []]]
steps= 15
True
>>> print accepts((l0cnf,p0cnf),['Sue','praises','Maria','and','the','student','knows','it'])
chart= [[[], ['DP'], ['S'], ['S'], [], [], ['S'], ['S'], ['S']], [[], [], ['V'], ['VP'], [], [], ['VP'], [], [], [], []]]
steps= 26
True
>>> print accepts((l0cnf,p0cnf),['Bill','knows','that','Sue','praises','Maria','and',
    'the','student','knows','it'])
chart= [[[], ['DP'], ['S'], [], [], ['S'], ['S'], [], [], ['S'], ['S'], 'S'], ['S', 'S']], [[], [], ['V'], [], [], ['VP'], [], [], []]]
steps= 47
True

```

This is the first efficient recognizer we have used in this class. It still wastes time on useless steps, but not to the extent we saw in the other methods. It always terminates; it can handle left and right recursion; it has no problem with exponential or infinite ambiguity. It does not allow unary and empty productions, but those can be added without making the parser significantly less efficient [15].

On the other hand, since this recognition method pools all results from all derivations in one table, extracting a description of the derivation now takes more work than before, and getting the trees can be nonterminating in the case where there are infinitely many trees to extract. It is straightforward to adapt the TD parser to do this, for example (this is implemented in `ckyp.py`). So in a sense, it could seem that we have sacrificed the problem we wanted to solve in order to achieve efficiency, but that is not quite fair. When the number of trees is reasonable, we can get them from the chart in a reasonable amount of time.

Setting aside the trees, the recognizer itself is ‘efficient’ in the computer scientists’ sense (i.e. it terminates within a number of steps bounded by a polynomial function of the length n of the input – in fact the number of steps is proportional to n^3 in the worst case. Is that good enough? Aho&Ullman [2] say:

It is essentially a “dynamic programming” method and it is included here because of its simplicity. It is doubtful, however, that it will find practical use, for three reasons:

1. n^3 time is too much to allow for parsing.
2. The method uses an amount of space proportional to the square of the input length.

3. The method of the next section (Earley’s algorithm) does at least as well in all respects as this one, and for many grammars does better.

Valiant’75 shows that the runtime complexity can be reduced below n^3 using fast matrix multiplication [33], but the method is not feasible. Lee’02 shows that CF recognition cannot be faster than matrix multiplication [16]. Satta’94 finds analogous polynomial complexity results for the more expressive formalism of TAGs [24].

7.2 Earley recognition

With the grammar `g0cnf` in the last section, no sentence begins with the verb `laughs`, but still if you give CKY a string like `knows that Sue praises Maria and the student knows it`, it will do a lot of work before realizing that this input cannot be recognized. The steps of the CKY recognizer are bottom-up, applied without any top-down check on whether the constituents being built could possibly appear in those positions. Early’68 shows how a top-down oracle can be built in. This algorithm has the “prefix property,” which means that, processing a string from left to right, an ungrammatical prefix (i.e. a sequence of words that is not a prefix of any grammatical string) will be recognized at the the earliest possible point.

For $A, B, C \in N$ and some designated $S' \notin N$, for $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$, and for input $w_1 \dots w_n \in \Sigma^n$,

$$\begin{array}{ll}
 (0, 0) : S' \rightarrow \square \bullet S & \text{[axiom]} \\
 \\
 \frac{(i, j) : A \rightarrow \alpha \bullet w_{j+1} \beta}{(i, j+1) : A \rightarrow \alpha w_{j+1} \bullet \beta} & \text{[scan]} \\
 \\
 \frac{(i, j) : A \rightarrow \alpha \bullet B \beta}{(j, j) : B \rightarrow \bullet \gamma} & \text{[predict] \quad if } B \rightarrow \gamma \\
 \\
 \frac{(i, k) : A \rightarrow \alpha \bullet B \beta \quad (k, j) : B \rightarrow \gamma \bullet}{(i, j) : A \rightarrow \alpha B \bullet \gamma} & \text{[complete]}
 \end{array}$$

The input is recognized iff $(0, n) : S' \rightarrow S \bullet$ is in the closure of the axioms (in this case, the set of axioms has just one element) under these inference rules. A straightforward implementation is provided in `earley.py`.

7.3 Memoization

The matrix-based parsers we have reviewed here is that they cannot provide a reasonable model of how humans actually parse ordinary, clear, conversation. Even if they could be made to fail in garden paths (e.g. maybe the tree collection would fail in those cases?), still it is clear that these methods need more space for longer sentences, without bound. CKY needs an n^2 matrix for a sentence of length n , which does not seem reasonable.

But let’s reflect on why these parsers can be efficient when the GLC parsers were not: it is because the GLC parsers must guess, and when a guess is wrong, they often have to recompute constituents that had already been computed. So any of the GLC parsers, from top-down (LL) to bottom-up (LR) could be supplemented with a memory mechanism which could store, ‘memoize’, how particular constituents were parsed. Then, when an alternative parse is explored, previously computed subproblems need not be solved again. An appropriate memoization strategy can turn a top-down parser into something like an Earley parser [12, 20, 4, 13], and there is some evidence that a process like this could account for ‘reanalysis’ and other effects in human parsing [30, 11]. If memory for past computations is kept forever, then obviously memory requirements grow without bound, but this need not be assumed. Research on these topics is ongoing. (See footnote 3 on page 115 on achieving Earley-like parsing of MCFGs with memoization in `datalog`.)

⇒ more coming ⇐

There it was, indeed, captive in seven lacquered bibles, so unavoidable and brutal that only a man immune to the spell of glory and alien to the interests of his power dared expose it in living flesh before the impassive old man who listened to him without blinking fanning himself in the wicker rocking chair, who only sighed after each mortal revelation, who only said aha, repeated it, using his hat to shoo away the April flies aroused by the luncheon leftovers, swallowing whole truths, bitter truths, truths which were like live coals that kept on burning in the shadows of his heart, because everything had been a farce, your excellency, a carnival apparatus that he himself had put together without really thinking about it when he decided that the corpse of his mother should be displayed for public veneration on a catafalque of ice long before anyone thought about the merits of her sainthood and only to contradict the evil tongues that said you were rotting away before you died, a circus trick which he had fallen into himself without knowing it ever since they came to him with the news general sir that his mother Bendición Alvarado was performing miracles and he had ordered her body carried in a magnificent procession into the most unknown corners of his vast statueless country so that no one should be left who did not know the worth of your virtues after so many years of sterile mortification, after so many painted birds without benefit, mother, after so much love without thanks, although it never would have occurred to me that the order was to be changed into the jape of the false dropsy victims who were paid to get rid of their water in public, they had paid two hundred pesos to a false dead man who arose from his grave and appeared walking on his knees through the crowd frightened by his ragged shroud and his mouth full of earth, they had paid eighty pesos to a gypsy woman who pretended to give birth in the middle of the street to a two-headed monster as punishment for having said that the miracles had been set up by the government, and that they had been, there wasn't a single witness who hadn't been paid money, an ignominious conspiracy that none the less had not been put together by his adulators with the innocent idea of pleasing him as Monsignor Demetrius Aldous had imagined during his first scrutinies, no, your excellency, it was a duty piece of business on the part of your proselytes, the most scandalous and sacrilegious of all the things they had made proliferate in the shadow of his power, because the ones who had invented the miracles and backed up the testimonies of lies were the same followers of his regime who had manufactured and sold the relics of the dead bride's gown worn by his mother Bendición Alvarado, aha, the same ones who had printed the little cards and coined the medals with her portrait as a queen, aha, the ones who enriched themselves with curls from her head, aha, with the flasks of water drawn from her side, aha, with the shroud of diagonal cloth where they used door paint to sketch the tender body of a virgin sleeping in profile with her hand on her heart and which was sold by the yard in the back rooms of Hindu bazaars, a monstrous lie sustained by the supposition that the corpse remained uncorrupted before the avid eyes of the endless throng that filed through the main nave of the cathedral, when the truth was quite something else, your excellency, it was that the body of his mother was not preserved because of her virtues or through the repair work done with paraffin and the cosmetic tricks that he had decided upon out of pure filial pride but that she had been stuffed according to the worst skills of taxidermy just like the posthumous animals in science museums as he found out with my own hands, mother, I opened the glass casket as the funereal emblems fell apart with the air, I took the crown of orange blossoms from your moldy brow where the stiff filly-mane hairs had been pulled out by the roots strand by strand to be sold as relics, I pulled you out from under the damp gauze of your bridal veil and the dry residue and the difficult saltpeter sunsets of death and you weighed the same as a sun-dried gourd and you had an old trunk-bottom smell and I could sense inside of you a feverish restlessness that was like the sound of your soul and it was the scissor-slicing of the moth larvae who were chewing you up inside, your limbs fell off by themselves when I tried to hold you in my arms because they had removed the innards of everything that held together your live body of a sleeping happy mother with her hand on her heart and they had stuffed you up again with rags so that all that was left of what had been you was only a shell with dusty stuffing that crumbled just by being lifted in the phosphorescent air of your firefly bones and all that could be heard were the flea leaps of the glass eyes on the pavement of the dusk-lighted church, turned to nothing, it was a trickle of the remains of a demolished mother which the bailiffs scooped up from the floor with a shovel to throw it back any way they could into the box under the gaze of monolithic sternness from the indecipherable satrap whose iguana eyes refused to let the slightest emotion show through even when he was all alone in the unmarked berlin with the only man in this world who had dared place him in front of the mirror of truth, both looked out through the haze of the window curtains at the hordes of needy who were finding relief from the heat-ridden afternoon in the dew-cool doorways where previously they had sold pamphlets describing atrocious crimes and luckless loves and carnivorous flowers and inconceivable fruits that compromised the will and where now one only heard the deafening racket of the stalls selling false relics of the clothes and the body of his mother Bendición Alvarado, while he underwent the clear impression that Monsignor Demetrius Aldous had read his thoughts when he turned his sight away from the mobs of invalids and murmured that when all's said and done something good had come out of the rigor of his scrutiny and it was the certainty that these poor people love your excellency as they love their own lives, because Monsignor Demetrius Aldous had caught sight of the perfidy within the presidential palace itself, had seen the greed within the adulation and the wily servility among those who flourished under the umbrella of power, and he had come to know on the other hand a new form of love among the droves of needy who expected nothing from him because they expected nothing from anyone and they professed for him an earthly devotion that could be held in one's hands and a loyalty without illusions that we should only want for God, your excellency, but he did not even blink when faced with that startling revelation which in other times would have made his insides twist, nor did he sigh but meditated to himself with a hidden restlessness that this was all we needed, father, all we need is for nobody to love me now that you're going off to take advantage of the glory of my misfortune under the golden cupolas of your fallacious world while he was left with the undeserved burden of truth without a loving mother who could help him through it, more lonely than a left hand in this nation which I didn't choose willingly but which was given me as an established fact in the way you have seen it which is as it has always been since time immemorial with this feeling of unreality, with this smell of shit, with this un-historied people who don't believe in anything except life, this is the nation they forced on me without even asking me, father, with one-hundred-degree heat and ninety-eight-percent humidity in the upholstered shadows of the presidential berlin, breathing dust, tormented by the perfidy of the rupture that whistled like a teakettle during audiences, no one to lose a game of dominoes to, and no one to believe his truth, father, put yourself in my skin, but he didn't say it, he just sighed, he just blinked for an instant and asked Monsignor Demetrius Aldous that the brutal conversation of that afternoon remain between ourselves, you haven't told me anything, father, I don't know the truth, promise me that, and Monsignor Demetrius Aldous promised him that of course your excellency doesn't know the truth, my word as a man.

Figure 7.1: 1500 words, from Gabriel García Márquez, *Autumn of the Patriarch*. See footnote 1 on p.81.

Exercises

1. CKY and global ambiguity.

- We used the example grammar `g0cnf.py` in Chomsky Normal Form in our experiments with CKY parsing. Copy that grammar to `gSScnf<YOURINITIALS>.py` and then change the grammar so that it simply parses the language $a^+ = \{a, aa, aaa, aaaa, \dots\}$, assigning to each string all binary trees. That is, for the string aa there should be just one parse, but for aaa there should be two parses $\overline{a[aa]}$ and $[[aa]a]$, and so on. Use the parser `ckyp` to check up to $a^7 = aaaaaaa$ to make sure that all parses are found. In a comment at the bottom of your file `gss<YOURINITIALS>.py`, indicate how many parses you find for $aa, aaa, aaaa, \dots, a^7$.
- Turning to `g0cnf.py` again, using `ckyp.py`, try to find the sentence of 10 words or less with the maximum number of ambiguities. Tell me what it is, and how many structures it has, in a comment at the bottom of `gss<YOURINITIALS>.py`.

- Probabilistic CKY.** For any grammar, assume each rule in $X \rightarrow \dots$ has probability $\frac{1}{n}$ where n is the number of rules with X on the left hand side. Modify `cky.py` to `pcky<YOURINITIALS>.py` so that instead of simply adding the lhs A to the matrix in step 2, it adds the category with its probability (A, pa) where pa is the probability of $A \rightarrow w_i$. And then in step 8, if $(B, prob)$ is already in $m[start][end]$, modify that entry if necessary so that B is paired with the maximum of $prob$ and p^*pb^*pc where p is the probability of $A \rightarrow BC$, $(B, pb) \in m[start][mid]$, and $(C, pc) \in m[mid][end]$. For any sentence that has a derivation, then, this parser will compute (S, ps) where ps is the probability of the most probable derivation.

- TD tree collection for the Earley recognizer.** The parser `ckyp.py` extends the recognizer `cky.py` by adding a top-down tree collector, which in effect parses the structures already in the chart with a top-down backtracking parser. Create a file `earleyp<YOURINITIALS>.py` that adds top-down tree collection for the Earley recognizer.

References

- AHO, A., HOPCROFT, J., AND ULLMAN, J. *Data Structure and Algorithms*. Addison-Wesley, Reading, Massachusetts, 1983.
- AHO, A. V., AND ULLMAN, J. D. *The Theory of Parsing, Translation, and Compiling. Volume 1: Parsing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1972.
- BERTSCH, E., AND NEDERHOF, M.-J. Some observations on LR-like parsing with delayed reduction. *Information Processing Letters* 104, 6 (2007), 195–199.
- BERTSCH, E., NEDERHOF, M.-J., AND SCHMITZ, S. On LR parsing with selective delays. In *Compiler Construction, CC'13*, K. D. Bosschere and R. Jhala, Eds., Lecture Notes in Computer Science. Springer, NY, 2013.
- CHARNIAK, E., GOLDWATER, S., AND JOHNSON, M. Edge-based best-first chart parsing. In *Proceedings of the Workshop on Very Large Corpora* (1998).
- COCKE, J., AND SCHWARTZ, J. T. Programming languages and their compilers: Preliminary notes. Tech. rep., Courant Institute of Mathematical Sciences, New York University, 1970.
- CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms, 3rd Edition*. MIT Press, Cambridge, Massachusetts, 2009.
- DUBEY, A., KELLER, F., AND STURT, P. A model of discourse predictions in human sentence processing. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing, EMNLP 2011* (2011), pp. 304–312.
- EARLEY, J. *An Efficient Context-Free Parsing Algorithm*. PhD thesis, Carnegie-Mellon University, 1968.
- EARLEY, J. An efficient context-free parsing algorithm. *Communications of the Association for Computing Machinery* 13 (1970), 94–102.
- FODOR, J. D., AND FERREIRA, F., Eds. *Reanalysis in Sentence Processing*. Kluwer, Boston, 1998.
- FORD, B. Packrat parsing: a practical linear-time algorithm with backtracking. MIT Master's thesis, 2002.
- HALE, J. T. What a rational parser would do. *Cognitive Science* 35, 3 (2011), 399–443.
- KASAMI, T. An efficient recognition and syntax algorithm for context free languages. Tech. Rep. AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA, 1965.
- LANGE, M., AND LEISS, H. To CNF or not to CNF? an efficient yet presentable version of the CYK algorithm. *Informatica Didactica* 8 (2009).
- LEE, L. Fast context-free parsing requires fast Boolean matrix multiplication. *Journal of the ACM* 49, 1 (2002), 1–15.

- [17] LEERMAKERS, R. Non-deterministic recursive ascent parsing. In *Proceedings of the 5th International Conference of the European Association for Computational Linguistics* (1991).
- [18] LEERMAKERS, R. A recursive ascent Earley parser. *Information Processing Letters* 41 (1992), 87–91.
- [19] LEERMAKERS, R. Recursive ascent parsing: From Earley to Marcus. *Theoretical Computer Science* 104 (1992), 299–312.
- [20] LEERMAKERS, R. *The Functional Treatment of Parsing*. Kluwer, Boston, 1993.
- [21] MARCUS, M. *A Theory of Syntactic Recognition for Natural Language*. MIT Press, Cambridge, Massachusetts, 1980.
- [22] MAZZEI, A., LOMBARDO, V., AND STURT, P. Constraining the form of supertags with the strong connectivity hypothesis. In *Supertagging*, S. Bangalore and A. K. Joshi, Eds. MIT Press, Cambridge, Massachusetts, 2010, pp. 407–427.
- [23] PRYOR, J. What’s so bad about living in the matrix? In *Philosophers Explore the Matrix*, C. Grau, Ed. Oxford University Press, NY, 2005, pp. 49–61.
- [24] SATTA, G. Tree adjoining grammar parsing and boolean matrix multiplication. *Computational Linguistics* 20 (1994), 173–232.
- [25] SEDGEWICK, R. *Algorithms*. Addison-Wesley, Menlo Park, California, 1983.
- [26] SHIEBER, S. M., SCHABES, Y., AND PEREIRA, F. C. N. Principles and implementation of deductive parsing. Tech. Rep. CRCT TR-11-94, Computer Science Department, Harvard University, Cambridge, Massachusetts, 1993.
- [27] SIKKEL, K., AND NIJHOLT, A. Parsing of context free languages. In *Handbook of Formal Languages, Volume 2: Linear Modeling*, G. Rozenberg and A. Salomaa, Eds. Springer, NY, 1997, pp. 61–100.
- [28] SKIENA, S. V. *The Algorithm Design Manual*. Springer, NY, 2008.
- [29] SPERBER, M., AND THIEMANN, P. Generation of LR parsers by partial evaluation. *ACM Transactions on Programming Languages and Systems* 22, 2 (2000), 224–264.
- [30] STAUB, A. The return of the repressed: Abandoned parses facilitate syntactic reanalysis. *Journal of Memory and Language* 57, 2 (2007), 299 – 323.
- [31] STURT, P. Incrementality in syntactic processing: Computational models and experimental evidence. In *Proceedings of the ACL Workshop on Incremental Parsing* (2004), F. Keller, S. Clark, M. Crocker, and M. Steedman, Eds., Association for Computational Linguistics, p. 66.
- [32] STURT, P. The psycholinguistics of structural composition. In *LINCOM Studies in Theoretical linguistics* (2011), R. K. Mishra and N. Srinivasan, Eds., vol. 5, pp. 14–32.
- [33] VALIANT, L. G. General context free recognition in less than cubic time. *Journal of Computer and System Sciences* 10 (1975), 308–315.
- [34] YOUNGER, D. Recognition and parsing of context free languages in time $O(n^3)$. *Information and Control* 10 (1967), 189–208.

Part II

Succinct syntax: Beyond context free

Chapter 8 Minimalist grammar

8.1 Mildly context sensitive languages

There are various arguments for assuming that at least some human languages – that is, some sets of strings defined by grammars plausible as models of human language recognition – are not context free:

- Shieber'85 shows that a variety of Swiss-German has case-checking dependencies between verbs and their objects of the form

S O₁ O₂ . . . V₁ V₂ . . .

Previous crossing dependencies had been noted: verbal clusters in Dutch [10, 4], *respectively* in English [2, 17]) but in those cases it is not clear that the dependencies are syntactic.

- Culy'85 shows that Bambara has a morphological process of whole word reduplication. Morphophonological reduplication is very common, but Culy argues that in this instance the reduplication is productive and unbounded.
- Rambow'94 argues that scrambling in some dialects of German is not only not strongly CF definable but also not strongly MG definable [19, 3]
- There seems to be some evidence of full phrase reduplication in some languages too: English X-or-no-X [14], and Chinese A-not-A questions [18, 9, 24].

Some of the relevant examples are displayed in Figure 8.1 on page 105.

But there are many reasons for assuming that the *grammars* of human languages are not CF. When we think about what sorts of rules plausibly define human languages, what sorts of generalizations seem to be respected in human languages, context free grammars do not provide reasonable ways to state them. Chomsky'56 argues not that a reasonable grammar of English will not be CF, and every mainstream linguistic tradition has adopted more expressive notations. More powerful formalisms tend to allow much more compact descriptions [8, 25, 20], that is, they allow us to capture generalizations. Presumably, the existence of simple, insightful descriptions is (not coincidental but) to be explained by something about how the language learner represents the language. So until we get more direct evidence about the mental representation of grammar, we would like a formalism that allows fairly direct representation of the linguists' most insightful descriptions of the patterns found in language. Insightfulness is subjective; if you are worried about that, think: succinctness.¹

Aravind Joshi'85 defines the class of mildly context sensitive (MCS) languages with the following properties:

- They properly include the context free languages
- They can be parsed in polynomial time
- They can define some (but not all) crossing dependencies
- They have the constant growth property: for each MCS language, there is a finite constant k such that if a string s has length i and some string s' has length greater than $i + k$, then there is at least one string of intermediate length between them.

The third condition of this definition is not precise, but the class of languages definable by tree adjoining grammars (TAGs) is generally regarded as MCS, as is the larger class of languages definable by minimalist grammars (MGs). Joshi'85 proposes:

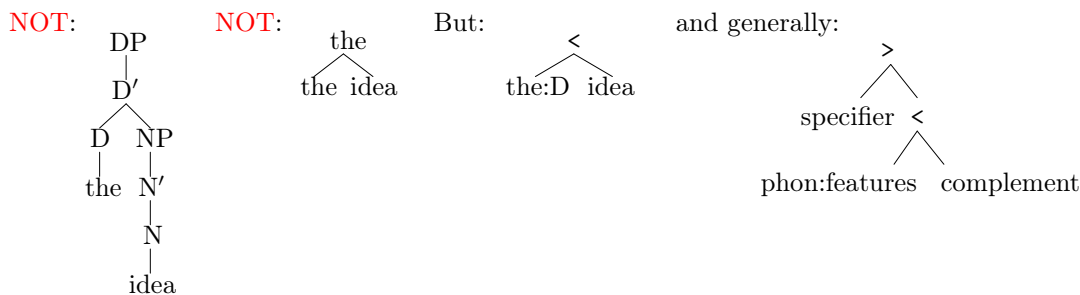
¹Succinctness may not be subjective, but the options for obtaining succinct representations are limited by the mechanisms of the device that interprets/compiles the grammar specification. So, putting the point more carefully, I should have said something like: "if you are worried about insightfulness, think: succinctness relative to plausible assumptions about cognitive architecture." But now we have "plausible assumptions" in there. So what I should really have said is: "it's a mistake to think there's a way to firmer ground."

(MCS hypothesis) Human languages are not only ‘weakly’ MCS in the sense of being in this class, but also that they are ‘strongly’ MCS, in the sense that a formalism powerful enough to define exactly the MCS languages will also suffice to define the correct structures for human languages.

The MCS hypothesis seems quite widely accepted. Even if it is not quite right, it seems very close to what we need. Jäger and Rogers’12 say: “Most experts... assume at this time that all natural languages are MG languages” – See their figure and this quote in Figure 8.2 on page 106. We will present MG languages now, language defined by minimalist grammars.

8.2 Minimalist grammar on bare phrase structure

First of all, we will get rid of some redundancy in conventional tree representations, using not trees on the left where each category symbol appears 3 times, or the second tree where the lexical item appears twice, but trees where the internal nodes have an “arrow” pointing to the head:



The arrows (‘order’ symbols) <, > point towards the **head** of the phrase. The largest subtree with a given head is a **maximal** projection, or **phrase XP**. And we will assume that each phrase can have at most one complement, but any number of specifiers.

minimalist grammar = (Lexicon, {merge, move}) (first pass)

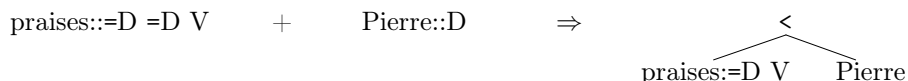
Lexicon: associates vocabulary with feature sequences:

vocabulary (phon+sem)	Marie, Pierre, who, praises, ...
category	N, V, A, P, C, D, I, ...
selector	=N, =V, =A, =P, =C, =D, =I, ...
licensor	+wh, +case, ...
licensee	-wh, -case, ...

in the order **word::features***

Examples:
 Marie::D
 who::D -wh
 praises::=D =D V
 ε::=I +wh C

Merge triggered by =X, attaches X on right if simple, left otherwise



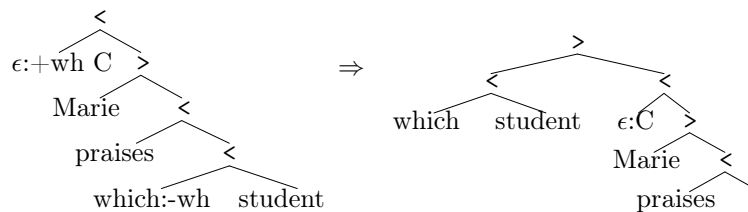


Each structure building operation applies to expressions, deleting a pair of features (shown in red), and building headed binary tree structures like those shown here, with the order symbols “pointing” towards the head.

The features are deleted in order, and the only affected features are those on the head of the two arguments.

Here and throughout, when writing an item of the form *string:features*, when *features* is empty, I often write just *string*.

Move triggered by +f, moving maximal -f subconstituent to specifier:



(SMC) Move cannot apply unless there is exactly 1 head with -f feature first

When the head of an expression begins with +f, and there is exactly one other node N in the tree with -f as its first feature, we take the phrase that has N as its head and move it up to the left.

This is a unary, simplification step, but like merge, it deletes a pair of features and adds an “arrow”.

Notice that we use :: in lexical items and : on the leaves of larger trees – this distinction is necessary now, but will be useful later.

minimalist grammar = (Lexicon, {merge, move}) (second pass, slightly more precise)

- **vocabulary** $\Sigma = \{\text{every, some, student, } \dots\}$ (phon, sem features)
- **types** $T = \{::, :\}$ (“lexical” and “derived,” respectively)
- **syntactic features** F :
 - $\text{C, T, D, N, V, P, } \dots$ (selected categories)
 - $=\text{C, =T, =D, =N, =V, =P, } \dots$ (selector features)
 - $+\text{wh, +case, +focus, } \dots$ (licensors)
 - $-\text{wh, -case, -focus, } \dots$ (licensees)
- **expressions** E : trees with non-root nodes $\{<, >\}$, leaves $\Sigma^* \times T \times F^*$
- **lexicon** $Lex \subset \Sigma^* \times \{::\} \times F^*$, a finite set of 1-node trees
- **Tree Notation**: sometimes we write *word* : ϵ simply as *word*,
and we often leave nodes with $\epsilon : \epsilon$ unlabeled altogether

Notation:

$t[\mathbf{f}]$ is a tree with 1st feature \mathbf{f} at its head,
and the tree t is the result of removing \mathbf{f} and changing $::$ to $:$

$t\{t_1/t_2\}$ is the result of replacing t_1 by t_2 in t

$t_1^>$ is the maximal projection of the head of t_1

ϵ is the 1 node tree labeled with no syntactic or phonetic features

- $Lex \subset \Sigma^* \times \{::\} \times F^*$, a finite set of 1-node trees

-

$$\mathbf{merge}(t_1[=\mathbf{f}], t_2[\mathbf{f}]) = \begin{cases} \begin{array}{c} < \\ t_1 \quad t_2 \end{array} & \text{if } t_1 \text{ has exactly 1 node} \\ \begin{array}{c} > \\ t_2 \quad t_1 \end{array} & \text{otherwise} \end{cases}$$

$$\mathbf{move}(t_1[+\mathbf{f}]) = t_2^> \begin{array}{c} > \\ t_1 \quad t_2[-\mathbf{f}]^>/\epsilon \end{array} \quad \text{if (SMC) only one head has } -\mathbf{f} \\ \text{as its first feature}$$

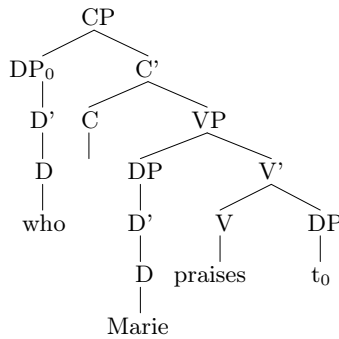
The two generating functions are partial functions from expressions to expressions. Merge is a binary function, and move is a unary function. (If you want, you could take the union of these two function to get one big ‘merge’ function, defined by 3 cases, and let the first two be called ‘external merge’ – defined on pairs of expressions, and the third case – defined on single expressions, be called ‘internal merge’.)

These specifications take some ‘decoding’! Especially in the case of **move**. What it says is this: if you have a tree $t[+f]$ with a head whose first syntactic feature is $+f$, and if that tree has a subtree $t_2[-f]$ whose head has first feature $-f$ (and by the SMC only one head has $-f$ as its first feature), then the result of applying **move** is the tree that has the maximal projection of t_2 as its specifier, with a sister tree that is the result of replacing the maximal projection of t_2 by the empty node. **In sum:**

- There are finitely many lexical items, and each one is a pairing of some pronounceable vocabulary (possibly empty) with a finite sequence of syntactic features.
- Each structure building operation “checks” and cancels a pair of features.
- Features in a sequence are canceled from left to right.
- Merge applies to a simple head and the first constituent it selects by attaching the selected constituent on the right, in **complement** position. If a head selects any other constituents, these are attached to the left in **specifier** positions.
- All movement is overt, phrasal, leftward. A maximal subtree moves to attach on the left as a specifier of the licensing phrase.
- Our restriction (SMC) prevents movement when two outstanding $-f$ requirements would compete for the same position. This is a strong version of the “shortest move” condition discussed in [6].

NB: SMC looks like a stipulation here, but we will reveal it in a different light soon. . .

example MG1

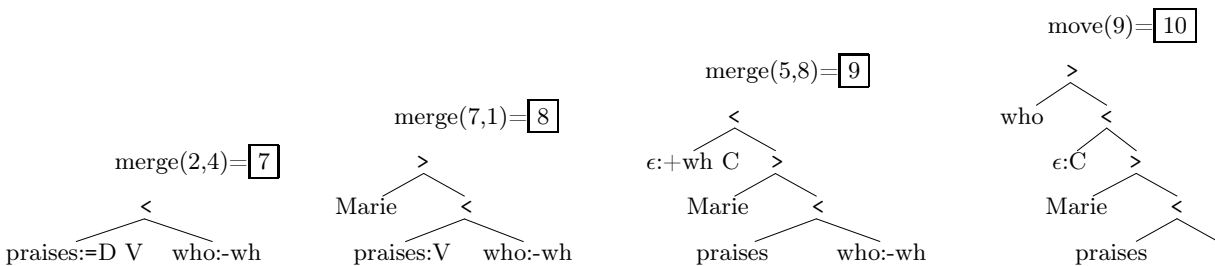


0	Pierre::D	who::D -wh	4
1	Marie::D	ε::=V +wh C	5
2	praises::=D =D V	and::=C =C C	6
3	ε::=V C		

This tiny grammar generates an infinite language, because of the coordinator *and*

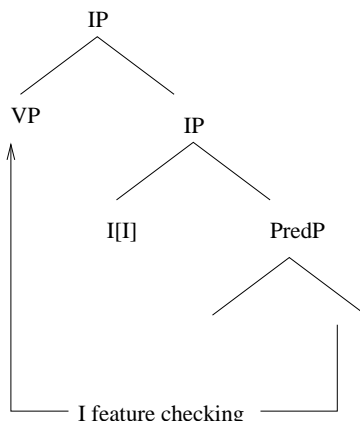
Here we show a “bare tree” of the sort just introduced, but it is easy to compute instead a representation more similar to the ones common in the linguistic literature, as shown on the right.

These structures represent the results of a derivation, but it is also easy to keep a complete record of the whole derivation, and to do this, we do not need the whole tree structures with all those empty nodes. It suffices to have categorized tuples of strings – as below



A tree is **completed** iff it has just 1 category symbol left, the “sentence” or “start” category

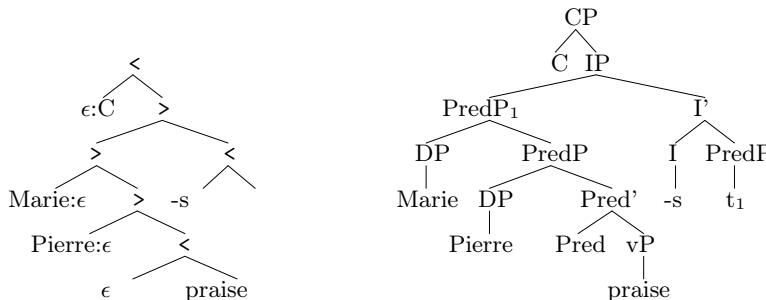
example: SOVI “Naive Tamil” inspired by Mahajan



We have only phrasal movement, not the head movement that many linguists use to position inflection. But some linguists have considered the possibility that inflection is positioned by phrasal movement.

For example, Mahajan [13] considers the possibility that the verb gets placed next to inflection by phrasal movement. Assuming that the Subject and Object are already in the VP (a “VP-shell”) before this movement happens, we can derive a very simple SOVI order with the lexical items here.

Notice that the *-s* in the string component of an expression signals that this is an affix, while the *-v* in the feature sequence of an expression signals that this item must move to a *+v* licensing position.



Pierre::D
 praise::v
 ε::=I C
 ε::=v =D =D Pred -v

Marie::D
 criticize::v
 -s::=Pred +v I

This simple proposal just intended to show one way that the most basic idea of Mahajan’s proposal could be encoded

Example 2 from Baker

Almost any category can combine with a complement...

- a eat [some spinach] (verb)
- b pieces [of cake] (noun)
- c fond [of swimming] (verb)
- d under [the table] (preposition)
- e will/to [eat some spinach] (tense)
- f the [piece of cake] (determiner)
- g too [fond of swimming] (degree)
- h that [Kate ate spinach] (complementizer)

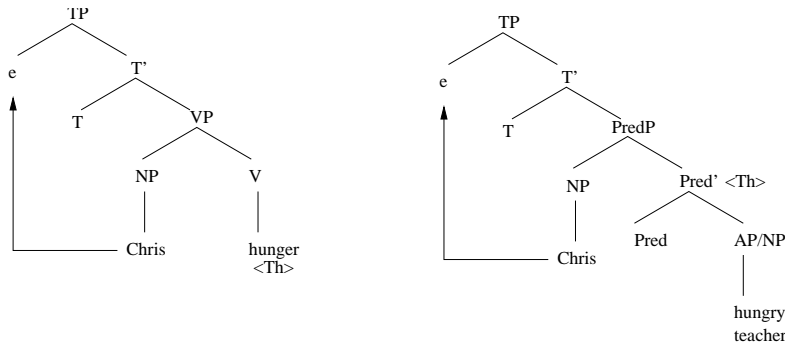
... only verbs are true predicates, with the power to license a specifier, which they typically theta-mark. In contrast, nouns and adjectives need help from a functional category Pred in order to do this.

- 42 a. Èmèrí *(ye) mòsé
 Mary PRED beautiful_A
 'Mary is beautiful'

Here I paste together clips from [1, pp.24,20,40] in an attempt to very briefly(!) summarize Baker's view.

Example 42 is from the Nigerian language Edo [1, p.40], where, Baker suggests, matters are slightly simpler than in the similar English contrast

- 1. Mary *(is) beautiful

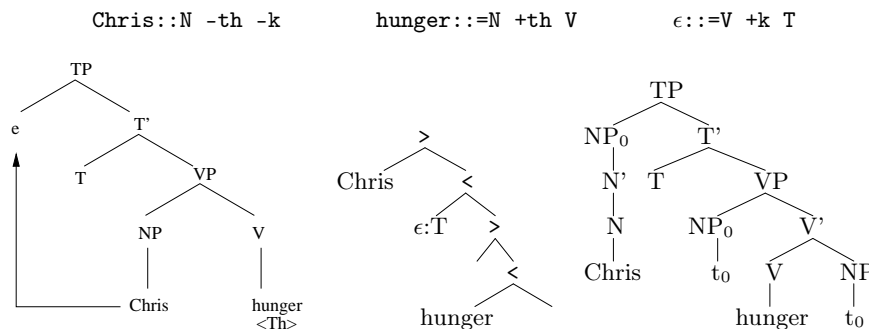


X is a verb iff X is a lexical category and X has a specifier.
 Agent and theme roles can only be assigned to specifier positions.

These diagrams from [1, p35], the claims are from p23 and p26 respectively.

In transcribing the tree, I corrected what I take to be an error in the published version: I put a Pred' in the tree where the published version has a PredP'. (With a machine deriving the trees, such errors would be less common!)

- MGs have **merge**, not [**merge as spec**] and [**merge as comp**]
- we could add [**merge as spec**], but this loses linear asymmetries
- but V can **merge** an N and then check a theta feature in spec

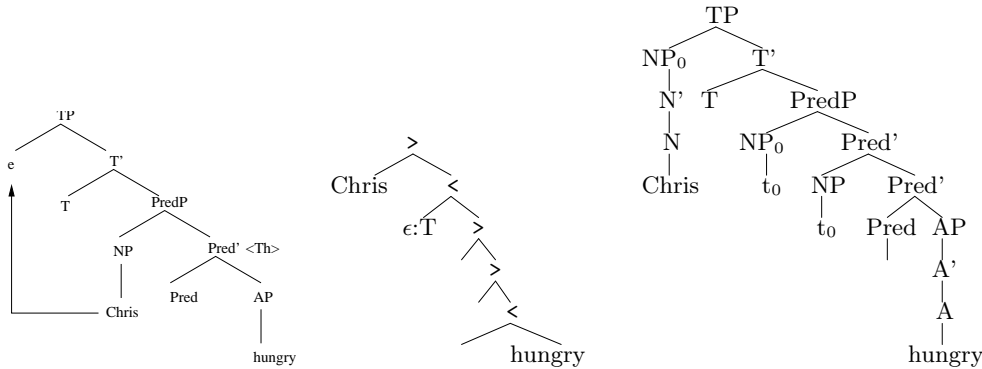


If we added a rule of **merge spec**, we could try to keep an account of the linear asymmetries by stipulating that it is dispreferred, but that is not how the science should work! If this is dispreferred, we would like it to be dispreferred for a reason.

- MGs cannot **merge** an spec and simultaneously **license** it
- but Pred can **merge** an A and then check a theta feature in another spec

Chris::N -th -k hungry::A ε::=A =N +th Pred ε::=Pred +k T

Baker’s proposal (below left), gets a precise implementation with this lexicon, with these structures (below middle and right):



Our slender and elegant bare trees might make you think that the traditional depictions are making things look much more complicated than they are... You’re right! We will see this next time.

An open question: (possible squib topic)

- Suppose there are two types of categories: lexical and functional
- Suppose only 1 lexical category, V, can select a specifier (just one) or check its own complement
- Suppose only 1 lexical category, N, has referential index, so can fill +th arg positions
- Suppose only 1 lexical category, A, is -N -V: no specifier, and cannot fill +th arg positions
- Suppose the functional categories vary across langs. Each can have a complement, and can license (but never select) a specifier (and only at most one specifier).

Q: can grammars of this form generate all the MG languages?

simpler Q: is every MG language generable by a grammar with at most 1 category that can select a specifier?

These ideas inspired by Baker need fleshing out before they will have syntactic impact. For example, is the category with specs also the only one that can check +th features? One strategy for problems like these: Take an arbitrary MG, and see if you can specify a recipe for converting it into another one that generates the same language but satisfies these conditions.

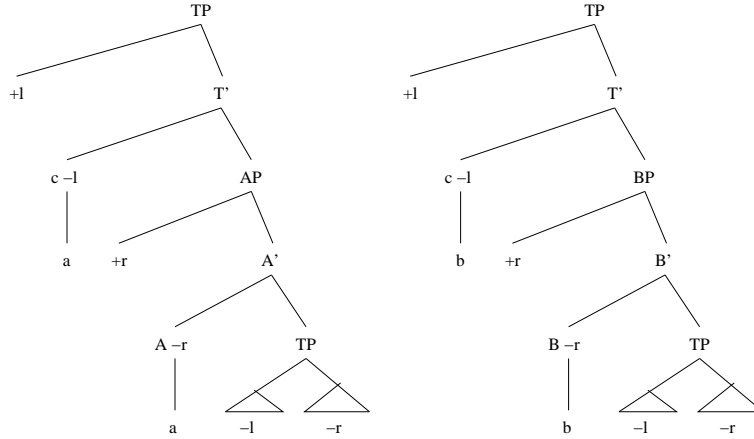
Example: the copy language

To construct an MG for $L_{xx} = \{xx \mid x \in \{a, b\}^+\}$ is slightly complicated. We use movement to keep the two halves of each sentence ‘synchronized’ during the derivation. So we will let each structure have a substructure with two pieces that can move independently.

Call the features triggering the movement

-l(eft) and -r(ight).

and then the recursive step can be pictured as having two cases, one for AP’s with terminal a and one for BP’s with terminal b, like this:



Notice that in these pictures, the start category T has a +l and a -l, while A and B each have a +r and -r. That makes the situation nicely symmetric. We can read the lexical items off of these trees:

$$\begin{aligned} a::=A \ +l \ T \ -l & & b::=B \ +l \ T \ -l \\ a::=T \ +r \ A \ -r & & b::=T \ +r \ B \ -r \end{aligned}$$

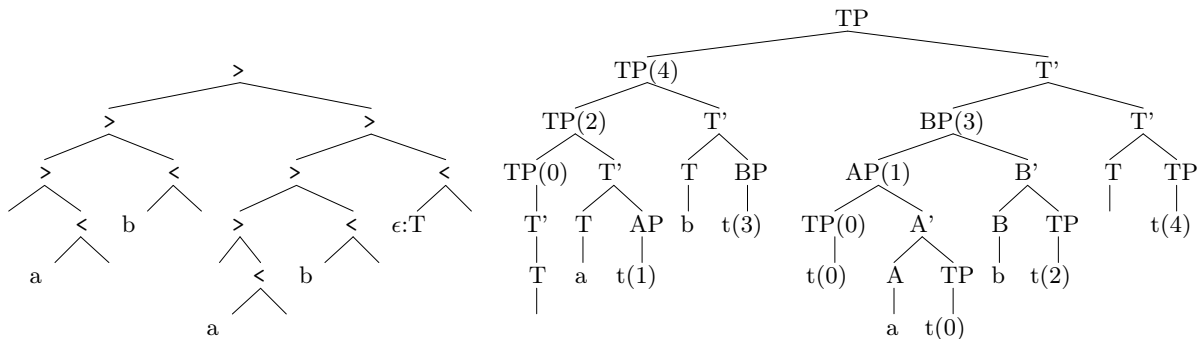
With this recursion, we only need to add the base case. Since we already have recursive structures that expect CPs to have a +r and -r, so we just need

$$\epsilon::=T \ +r \ +l \ T$$

to finish the derivation (at the ‘top’ of the tree), and to begin a derivation, we use this:

$$\epsilon::T \ -r \ -l$$

This grammar has just 6 lexical items. (See if you can find a simpler formulation!) It allows us to do derivations like this, showing the derived structure on the left, and the corresponding conventional tree on the right:



This works as we hoped!²

The derivations with this grammar are rather tedious, and interesting because they have lots of movements! (Is it reasonable to suppose that the kinds of structures shown in Figure 8.1 on page 105 are derived in this way?) We can get the computer to check our calculations. The recognizer `mgcky` will be presented in the next section takes the `xx` grammar in this form:

²Mattescu and Salomaa [15] present a context sensitive grammar for the language $\{xx \mid x \in \{a, b\}^+\}$. If you compare that one to ours, you will see that ours is much easier to understand.

```

1  """      file: mgxx.ml
2           created: 2013-02-19 10:52:17 PDT   E Stabler stabler@ucla.edu
3
4  Grammar for the copy language {XX/ X\in{a,b}*}
5  This grammar has lots of local ambiguity, and does lots of movement
6  (more than in any human language, I think)
7  so it gives the parser a good workout.
8
9  I use (1,f) for select f, (-1,f) for category f
10         (2,f) for license f, (-2,f) for licensee f
11 """
12 mgxx = [([], [(-1, 'T'), (-2, 'r'), (-2, 'l')]),
13         ([], [(1, 'T'), (2, 'r'), (2, 'l'), (-1, 'T')]),
14         (['a'], [(1, 'T'), (2, 'r'), (-1, 'A'), (-2, 'r')]),
15         (['b'], [(1, 'T'), (2, 'r'), (-1, 'B'), (-2, 'r')]),
16         (['a'], [(1, 'A'), (2, 'l'), (-1, 'T'), (-2, 'l')]),
17         (['b'], [(1, 'B'), (2, 'l'), (-1, 'T'), (-2, 'l')])
18 ]

```

And we can use the grammar like this:

```

>>> from mgxx import *
>>> from mgcky import *
>>> prettyMG(mgxx)
::T -r -l
::=T +r +l T
a::=T +r A -r
b::=T +r B -r
a::=A +l T -l
b::=B +l T -l
>>> recognize(mgxx, ['a', 'b', 'a', 'b'], 'T')
--- scanning (0, 0) and then (0, 1) a
( 0 , 0 ) :: T -r -l
( 0 , 0 ) :: =T +r +l T
( 0 , 1 ) :: =T +r A -r
( 0 , 1 ) :: =A +l T -l
( 0 , 1 ) : +r A -r (0,0):-r -l
( 0 , 1 ) : A -r (0,0):-l
( 0 , 0 ) : +r +l T (0,0):-r -l
...
( 4 , 4 ) : +l T (1,1):-l
( 4 , 4 ) : +r +l T (2,4):-r (0,2):-l
( 2 , 4 ) : +l T (0,2):-l
( 0 , 4 ) : T
( 4 , 4 ) : +r +l T (3,4):-r (1,2):-l
...
True
>>> recognize(mgxx, ['a', 'b'], 'T')
--- scanning (0, 0) and then (0, 1) a
( 0 , 0 ) :: T -r -l
( 0 , 0 ) :: =T +r +l T
( 0 , 1 ) :: =T +r A -r
...
False
>>>

```

We will explain how this calculation is done in the next section.

Exercises: MG on bare phrase structure

Do at least 2 of the following. (...or more for extra credit)

1. Provide an MG for VISO, as similar to our SOVI on page 98 as possible, and show a sample derivation in complete detail (as we did at the bottom of page 97 and in class)
2. Provide an MG for SVIO, as similar to our SOVI on page 98 as possible, and show a sample derivation in complete detail (as we did at the bottom of page 97 and in class)

- *Hint for the previous 2 problems:* Notice that, in the SOVI example on page 98, the underlying order, before movements, is ISO(Pred)V. And in that example, [SOV] is a Pred phrase, S is a DP, O is a DP, and V is a VP. Since these are all phrases, any of them can move. To get SOVI, we moved the PredP [SOV] to the specifier of I. You can do the problems 1 and 2 without changing the underlying ISOV order, just by changing what moves where.
3. Provide an MG for Σ^* where $\Sigma = \{a, b\}$, the set of all sentences consisting of a's and b's (including ϵ) and show a sample derivation for a string of length 3 in complete detail (as we did at the bottom of page 97 and in class)

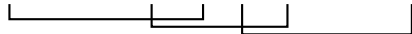
References

- [1] BAKER, M. C. *Lexical Categories: Verbs, Nouns, and Adjectives*. Cambridge University Press, NY, 2003.
- [2] BAR-HILLEL, Y., AND SHAMIR, E. Finite state languages: Formal representations and adequacy problems. *Bulletin of the Research Council of Israel 8F* (1960), 155–166. Reprinted in Y. Bar-Hillel, *Language and Information: Selected Essays on their Theory and Application*. NY: Addison-Wesley, 1964.
- [3] BECKER, T., RAMBOW, O., AND NIV, M. The derivational generative power of formal systems, or, scrambling is beyond LCFRS. IRCS technical report 92-38, University of Pennsylvania, 1992.
- [4] BRESNAN, J., KAPLAN, R. M., PETERS, S., AND ZAENEN, A. Cross-serial dependencies in Dutch. *Linguistic Inquiry 13*, 4 (1982), 613–635.
- [5] CHOMSKY, N. Three models for the description of language. *IRE Transactions on Information Theory IT-2* (1956), 113–124.
- [6] CHOMSKY, N. *The Minimalist Program*. MIT Press, Cambridge, Massachusetts, 1995.
- [7] CULY, C. The complexity of the vocabulary of Bambara. *Linguistics and Philosophy 8*, 3 (1985), 345–352.
- [8] HARTMANIS, J. On the succinctness of different representations of languages. *SIAM Journal on Computing 9* (1980), 114–120.
- [9] HUANG, C.-T. J. Modularity and Chinese A-not-A questions. In *Interdisciplinary Approaches to Language: Essays in Honor of S.-Y. Kuroda*, C. Georgopoulos and R. Ishihara, Eds. Kluwer, Dordrecht, 1991.
- [10] HUYBREGTS, M. Overlapping dependencies in Dutch. Tech. rep., University of Utrecht, 1976. Utrecht Working Papers in Linguistics.
- [11] JÄGER, G., AND ROGERS, J. Formal language theory: Refining the Chomsky hierarchy. *Philosophical Transactions of the Royal Society B 367* (2012), 1956–1970.
- [12] JOSHI, A. K. How much context-sensitivity is necessary for characterizing structural descriptions. In *Natural Language Processing: Theoretical, Computational and Psychological Perspectives*, D. Dowty, L. Karttunen, and A. Zwicky, Eds. Cambridge University Press, NY, 1985, pp. 206–250.
- [13] MAHAJAN, A. Eliminating head movement. In *The 23rd Generative Linguistics in the Old World Colloquium, GLOW '2000, Newsletter #44* (2000), pp. 44–45.
- [14] MANASTER-RAMER, A. Copying in natural languages, context freeness, and queue grammars. In *Proceedings of the 1986 Meeting of the Association for Computational Linguistics* (1986).
- [15] MATEESCU, A., AND SALOMAA, A. Aspects of classical language theory. In *Handbook of Formal Languages, Volume 1: Word, Language, Grammar*, G. Rozenberg and A. Salomaa, Eds. Springer, NY, 1997, pp. 175–251.
- [16] MICHAELIS, J., AND KRACHT, M. Semilinearity as a syntactic invariant. In *Logical Aspects of Computational Linguistics* (NY, 1997), C. Retoré, Ed., Springer-Verlag (Lecture Notes in Computer Science 1328), pp. 37–40.
- [17] PULLUM, G. K., AND GAZDAR, G. Natural languages and context free languages. *Linguistics and Philosophy 4* (1982), 471–504.
- [18] RADZINSKI, D. Unbounded syntactic copying in Mandarin Chinese. *Linguistics and Philosophy 13* (1990), 113–127.
- [19] RAMBOW, O. *Formal and Computational Aspects of Natural Language Syntax*. PhD thesis, University of Pennsylvania, 1994. Computer and Information Science Technical report MS-CIS-94-52 (LINC LAB 278).
- [20] RAVIKUMAR, B., AND IBARRA, O. H. Relating the type of ambiguity of finite automata to the succinctness of their representation. *SIAM Journal on Computing 18*, 6 (1989), 1263–1282.
- [21] SEKI, H., MATSUMURA, T., FUJII, M., AND KASAMI, T. On multiple context-free grammars. *Theoretical Computer Science 88* (1991), 191–229.
- [22] SHIEBER, S. M. Evidence against the context-freeness of natural language. *Linguistics and Philosophy 8*, 3 (1985), 333–344.
- [23] STABLER, E. P. Derivational minimalism. In *Logical Aspects of Computational Linguistics*, C. Retoré, Ed. Springer-Verlag (Lecture Notes in Computer Science 1328), NY, 1997, pp. 68–95.

- [24] STABLER, E. P. Varieties of crossing dependencies: Structure dependence and mild context sensitivity. *Cognitive Science* 93, 5 (2004), 699–720.
- [25] UKKONEN, E. On size bounds for deterministic parsers. In *ICALP 81, Lecture Notes in Computer Science 115* (NY, 1981), Springer, pp. 218–228.

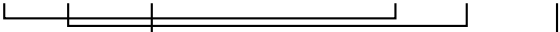
English (Chomsky'56)

will Marie have \emptyset be -en prais -ing Pierre



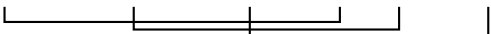
Dutch (Huybregts'76)

... because I Cecilia Henk the hippo saw help feed
 ... omdat ik Cecilia Henk de nijlpaarden zag helpen voeren



Swiss-German (Sheiber'85)

... that we the children Hans the house let help paint
 ... das mer d'chind em Hans es huus lönd hälfe aastriche



Simpler but similar: the "xx" language

a b b a b b

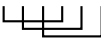


Figure 8.1: Some crossing dependencies

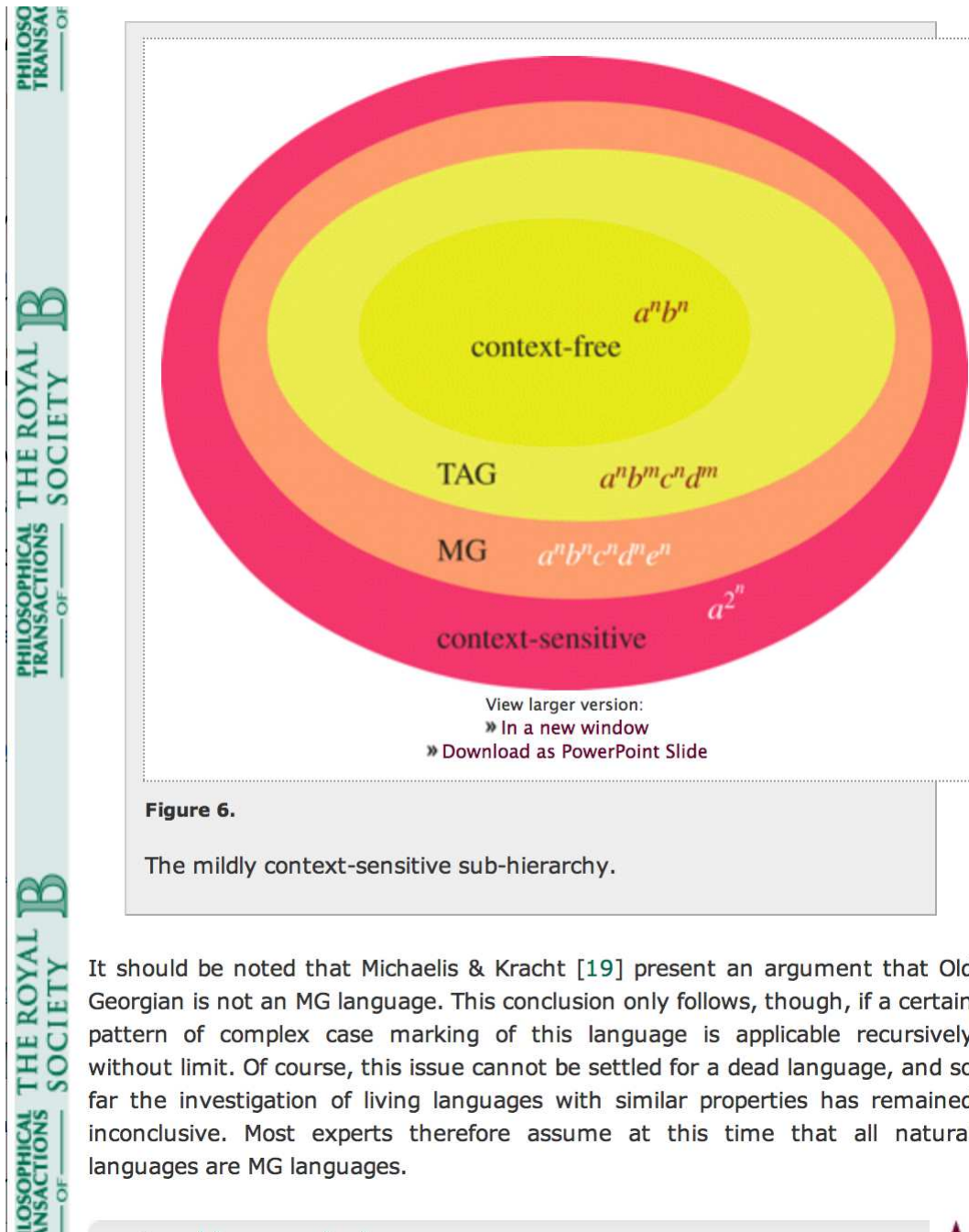


Figure 6.

The mildly context-sensitive sub-hierarchy.

It should be noted that Michaelis & Kracht [19] present an argument that Old Georgian is not an MG language. This conclusion only follows, though, if a certain pattern of complex case marking of this language is applicable recursively without limit. Of course, this issue cannot be settled for a dead language, and so far the investigation of living languages with similar properties has remained inconclusive. Most experts therefore assume at this time that all natural languages are MG languages.

Figure 8.2: from Jäger&Rogers'12

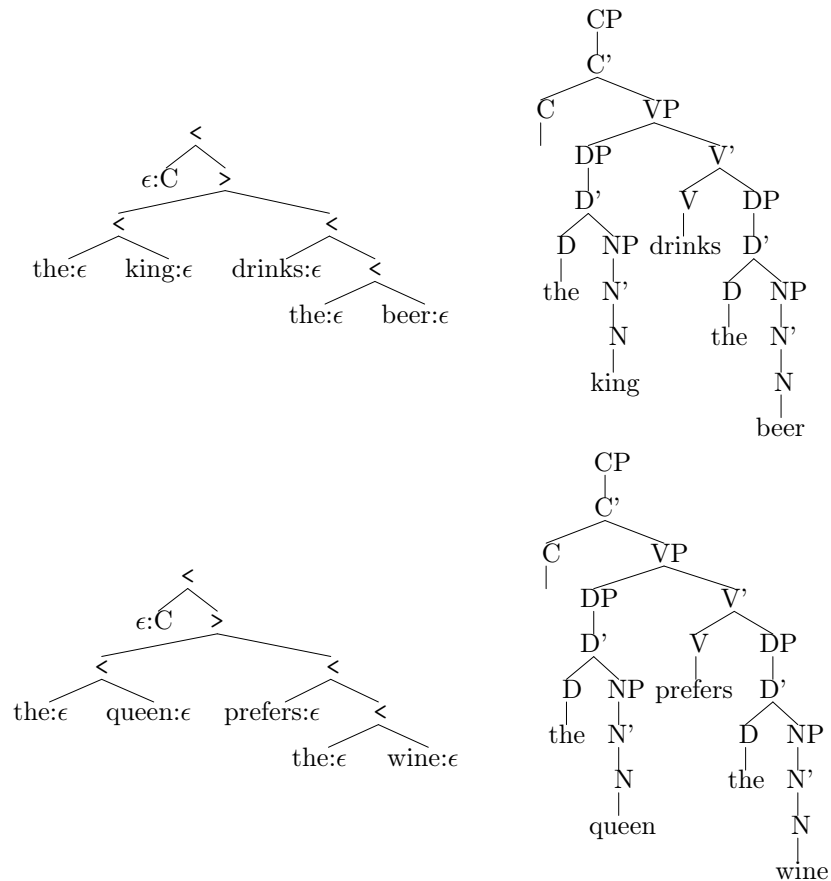
Chapter 9 MGs, MCFGs, CKY and Earley

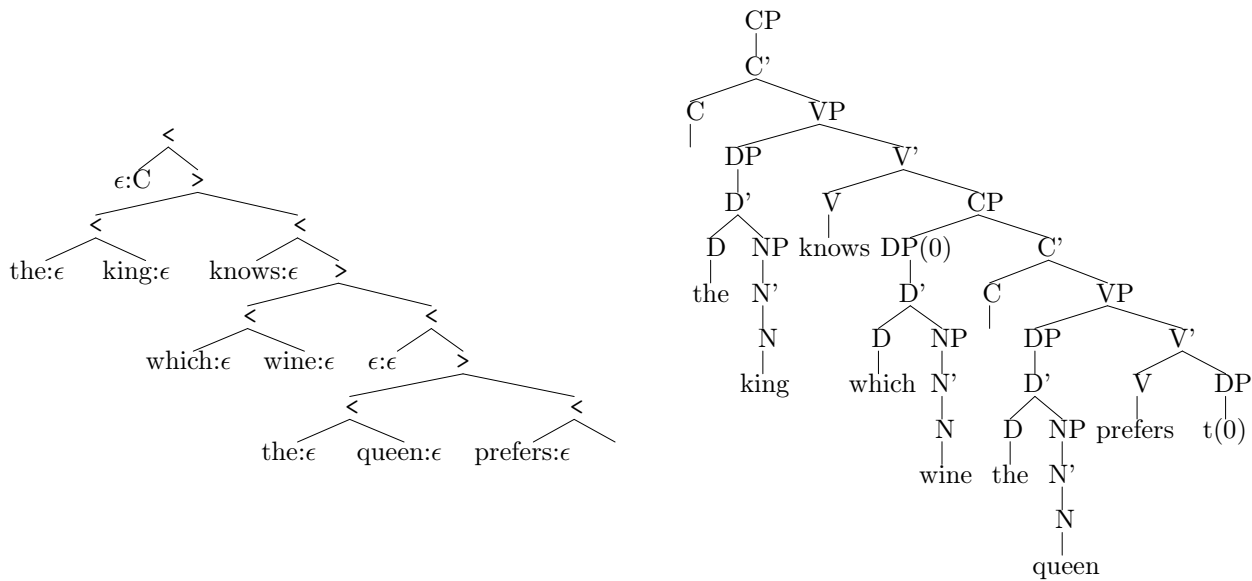
9.1 MG derivations, simplified

Consider this simple grammar, mg_0 :

$\epsilon ::= V C$	$\epsilon ::= V +wh C$
the $::= N D$	which $::= N D -wh$
king $::= N$	queen $::= N$
wine $::= N$	beer $::= N$
drinks $::= D =D V$	prefers $::= D =D V$
knows $::= C =D V$	says $::= C =D V$

From this MG, we can derive structures like the ones on the left, which correspond to the more conventionally represented trees on the right:



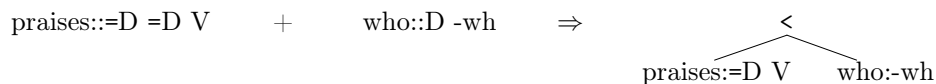
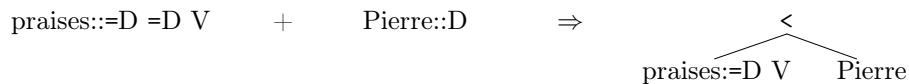


The structures on the left are simpler than the ones on the right – fewer nodes, and less redundancy in marking the relevant categories. But still there is lots of branching in these structures. We want to ask: what does the syntax need to be able to see in these structures?

Looking at the merge and move rules, we can see that the syntax needs know (i) the features of the head, and (ii) it needs to be able to find the subconstituent that is going to move, and (iii) it needs the strings in order to define the linear order properly. But once a phrase has no more syntactic features, the grammar never needs to see its structure any more.

So, roughly: the syntax needs to be able to see (i) the phrases with non-empty syntactic feature sequences at their heads, and (ii) the strings. This allows us to eliminate the trees completely, as long as we keep the phrases with features – in particular, that means phrases that are going to move. To achieve this, we must distinguish another special case of *merge*, namely, merging with a phrase that is going to move. Let's explain this. . .

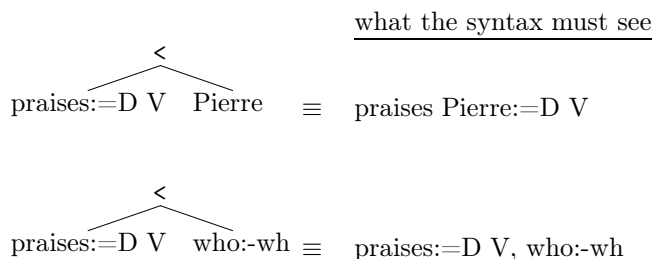
Merge: two different cases



Unlike Pierre, the DP who is a **mover**!

In both cases, merge applies to give us simple 3 node derived trees, but these two results are importantly different. In the first case, nothing in the grammar can ever separate *praises* from *Pierre*. But in the second case, *who* can be separated. This phrase, with its pronounced elements and its features, needs to be visible to the grammar.

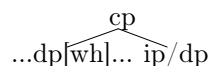
the essential parts of the trees



Each **tree** can be replaced by a **tuple of categorized strings**, as long as we keep moving phrases separate. (To reduce notational clutter, we do not bracket the tuples, but the commas are important!)

Each categorized string in any expression generated by the grammar is sometimes called a “chain.” It represents a constituent that may be related to other positions by movement. So each tree is replaced by a tuple of chains.

The “traditional” approach to parsing movements involves passing dependencies (sometimes called “slash dependencies” because of the familiar slash notation for them) down to c-commanded positions, in configurations roughly like this:



Glancing at the trees in the previous sections, we see that this method cannot work: there is no bound on the number of movements through any given part of a path through the tree, landing sites do not c-command their origins, etc. This intuitive difference also corresponds to an expressive power difference, as pointed out just above: minimalist grammars can define languages like $a^n b^n c^n d^n e^n$ which are beyond the expressive power of TAGs, CCGs (as formalized in Vijay-Shanker and Weir 1994), and standard trace-passing regimes.

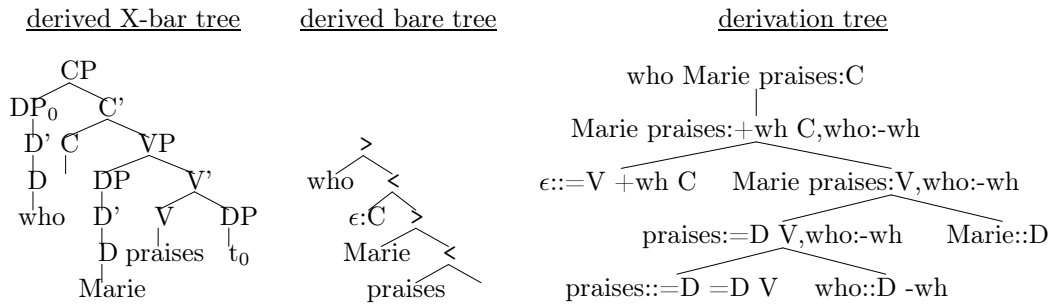
the first example derivation, reformulated

Lexicon:

1	Marie::D	who::D -wh	4
2	praises::=D =D V	ε::=V +wh C	5

Derivation, 4 steps:

merge(2 , 4)	=	praises:=D V, who:-wh	A
merge(A , 1)	=	Marie praises:V, who:-wh	B
merge(5 , B)	=	Marie praises:+wh C, who:-wh	C
move(C)	=	who Marie praises:C	D



In syntax101 we use informal grammars with X-bar trees something like we have on the left.

The formal MGs can, in 4 steps, derive “bare trees” like the one in the middle, defining the same X-bar trees.

Now we see that the bare trees can be replaced by tuples of categorized strings. With this representation, the two trees on the left are represented by the root node of the derivation tree on the right. The first two trees are derived trees, but tree on the right is a derivation tree showing all 4 steps of the derivation. The branching steps are merges, the non-branching step is the move, the leaves are lexical items.

Thm: The lexicons of bare tree MGs and tuple MGs are identical, and the derivations correspond in this way. There is a function f from bare trees to tuples such that

1. f is the identity function on lexical items,
2. every tuple MG derivation tree is just the result of relabeling a bare tree MG derivation tree using f , and
3. f maps any completed tree with yield s to $s : C$, where C is the start category, and
4. for any particular lexicon G , every completed tuple MG derivation tree is the value of f applied to a completed bare tree MG derivation tree.

So the tuple formulation of MGs defines exactly the same sentences in exactly the same way (with a derivation tree of exactly the same shape) as the bare tree formulation of MGs.

9.2 CKY-like recognition

In the previous section, we saw how tree-based MGs can be replaced by tuple-based MGs, defined as follows [28], without changing the language or the shape of the derivations of each string in the language. The tuple-based rules can be formulated like this:

minimalist grammar $G = \langle Lex, \{\text{merge}, \text{move}\} \rangle$

- **vocabulary** $\Sigma = \{\text{every}, \text{some}, \text{student}, \dots\}$
- **types** $T = \{::, : \}$ (‘lexical’ and ‘derived’, respectively)
- **syntactic features** F of four kinds:

C, T, D, N, V, P, \dots	(selected categories)
$=C, =T, =D, =N, =V, =P, \dots$	(selector features)
$+wh, +case, +focus, \dots$	(licensors)
$-wh, -case, -focus, \dots$	(licensees)
- **chains** $C = \Sigma^* \times T \times F^*$
- **expressions** $E = C^*$
- **lexicon** $Lex \subset \Sigma^* \times \{::\} \times F^*$, a finite set

merge: $(E \times E) \rightarrow E$ is the union of the following 3 functions,
for $\cdot \in \{:, ::\}$, $\gamma \in F^*$, $\delta \in F^+$

$$\frac{s :: = f\gamma \quad t \cdot f, \alpha_1, \dots, \alpha_k}{st : \gamma, \alpha_1, \dots, \alpha_k} \text{merge1: lexical item selects a non-mover as complement}$$

$$\frac{s : = f\gamma, \alpha_1, \dots, \alpha_k \quad t \cdot f, \iota_1, \dots, \iota_l}{ts : \gamma, \alpha_1, \dots, \alpha_k, \iota_1, \dots, \iota_l} \text{merge2: derived item selects a non-mover as specifier}$$

$$\frac{s \cdot = f\gamma, \alpha_1, \dots, \alpha_k \quad t \cdot f\delta, \iota_1, \dots, \iota_l}{s : \gamma, \alpha_1, \dots, \alpha_k, t : \delta, \iota_1, \dots, \iota_l} \text{merge3: any item selects a mover}$$

Here, $\alpha_1, \dots, \alpha_k, \iota_1, \dots, \iota_l$ ($0 \leq k, l$) are any chains.

Notice that the domains of merge1, merge2, and merge3 are disjoint, so their union is a function.

move: $E \rightarrow E$ is the union of the following 2 functions,
for $\gamma \in F^*$, $\delta \in F^+$, satisfying the following condition,

(SMC) none of the chains $\alpha_1, \dots, \alpha_{i-1}, \alpha_{i+1}, \dots, \alpha_k$ has $-f$ as its first feature,

$$\frac{s : +f\gamma, \alpha_1, \dots, \alpha_{i-1}, t : -f, \alpha_{i+1}, \dots, \alpha_k}{ts : \gamma, \alpha_1, \dots, \alpha_{i-1}, \alpha_{i+1}, \dots, \alpha_k} \text{move1: final move of licensee}$$

$$\frac{s : +f\gamma, \alpha_1, \dots, \alpha_{i-1}, t : -f\delta, \alpha_{i+1}, \dots, \alpha_k}{s : \gamma, \alpha_1, \dots, \alpha_{i-1}, t : \delta, \alpha_{i+1}, \dots, \alpha_k} \text{move2: nonfinal move of licensee}$$

Notice that the domains of move1 and move2 are disjoint, so their union is a function.

structures $S(G) = \text{closure}(\text{Lex}, \{\text{merge}, \text{move}\})$

completed structures = expressions $w \cdot C$, for C the ‘‘start’’ category and any type $\cdot \in \{:, ::\}$

sentences $L(G) = \{w \mid w \cdot C \in S(G) \text{ for some } \cdot \in \{:, ::\}\}$, the strings of category C

The grammar rules are in this deductive format immediately specify a CKY-like method for recognizing MG languages. We need only recognize that the strings are represented by pairs of positions in the input, we add axioms for the lexical input sequence, $(0, 1) : w_1 \dots (0, n) : w_n$:

$$\frac{}{(i, j) : \gamma} \text{axioms: for } w \text{ in position } (i, j) \text{ where } w :: \gamma \in \text{Lex}, 0 \leq i \leq j \leq n$$

Note that empty lexical items go from i to $j = i$, for every $0 \leq i = j \leq n$, where n is the length of the string.

We make two additional, minor changes for efficiency. *First*, rather than imposing the SMC on the move step, we will not produce any result which has two $-f$ chains, for any f . And *second*, since the lexical/derived distinction only matters for items that have selector features, we do not represent this distinction on any other items.

This kind of MG recognition requires no more than $\mathcal{O}n^{2k+2}$ steps, where k is the number of licensors and n is the length of the input [8, 7]. Various MG recognizer implementations have been written, parsing MGs directly [25, 20, 6] or by translating them into MCFGs or closely related formalisms [5, 1, 15] but we describe a very direct and ‘naive’ approach adapted from Harkema [7], Kallmeyer [9, Fig.7.3] and Seki&al [23, pp.207-9]:

```

TOP-DOWN BACKTRACK CF RECOGNITION( $G, i$ )
0  agenda=chart=all results of axioms for  $i$ 
1  for  $j \in \{0, \dots, \text{len}(i)\}$ :
2      apply scan to elements up to  $j$ , putting new results into agenda,chart
3      while agenda  $\neq \epsilon$ :
4          apply merge,move, putting new results into agenda,chart
5      if  $S \in m[0][\text{len}(i)]$  then True else False

```

This algorithm allocates a matrix all at once, but then builds entries from left to right.

In our implementation, the agenda is a stack, as usual, while the storage of intermediate results is split into 2 parts:

- All (unary) move1 and move2 steps are applied as soon as each new item is generated, and are kept in a list for tracing purpose only.
- The pronounced part of each head is indexed by left and right edges, as usual in CKY, with the left,right edges of moving elements explicitly labeled.

The grammar mg0 from page 107 will be given in this form:

```

1  """  file: mg0.py
2      created: Thu Feb 28 09:17:43 PST 2013  E Stabler stabler@ucla.edu
3      I use (1,f) for select f, (-1,f) for category f
4          (2,f) for license f, (-2,f) for licensee f
5  """
6  mg0 = [[[]],[[1,'V'],(-1,'C')]],
7         ([[]],[[1,'V'],(2,'wh'),(-1,'C')]),
8         ([1,'the'],[[1,'N'],(-1,'D')]),
9         ([1,'which'],[[1,'N'],(-1,'D'),(-2,'wh')]),
10        ([1,'king'],[(-1,'N')]),
11        ([1,'queen'],[(-1,'N')]),
12        ([1,'wine'],[(-1,'N')]),
13        ([1,'beer'],[(-1,'N')]),
14        ([1,'drinks'],[[1,'D'],(1,'D'),(-1,'V')]),
15        ([1,'prefers'],[[1,'D'],(1,'D'),(-1,'V')]),
16        ([1,'knows'],[[1,'C'],(1,'D'),(-1,'V')]),
17        ([1,'says'],[[1,'C'],(1,'D'),(-1,'V')])
18  ]

```

A pretty printer displays this grammar a more readable format:

```

>>> from mg0 import *
>>> from mgcky import *
>>> prettyMG(mg0)
::=V C
::=V +wh C
the::=N D
which::=N D -wh
king::=N
queen::=N
wine::=N
beer::=N
drinks::=D =D V
prefers::=D =D V
knows::=C =D V
says::=C =D V

```

For use by the parser, we convert the list of items in the grammar to a python dictionary that associates each lexical item with all of its possible sets of features.

```

>>> dictOf(mg0)
{'the': [[(1, 'N'), (-1, 'D')], (2, 'wh')], 'king': [[(-1, 'N')], (1, 'D')], 'prefers': [[(1, 'D'), (1, 'D'), (-1, 'V')], (1, 'D')], 'says': [[(1, 'C'), (1, 'D'), (-1, 'V')], (1, 'D')], 'wine': [[(-1, 'N')], (1, 'D')], 'drinks': [[(1, 'D'), (1, 'D'), (-1, 'V')], (1, 'D')], 'knows': [[(1, 'C'), (1, 'D'), (-1, 'V')], (1, 'D')], '': [[(1, 'V'), (-1, 'C')], (1, 'V'), (2, 'wh'), (-1, 'C')], 'queen': [[(-1, 'N')], (1, 'D')], 'beer': [[(-1, 'N')]]}
>>> dictOf(mg0)[('the',)]
[[1, 'N'], (-1, 'D')]
>>> dictOf(mg0)[('which',)]
[[1, 'N'], (-1, 'D'), (-2, 'wh')]

```

We run the recognizer like this:

```

>>> recognize(mg0,['which','queen','prefers','the','wine'],'C')
--- scanning (0, 0) and then (0, 1) which
( 0 , 0 ) :: =V C
( 0 , 0 ) :: =V +wh C
( 0 , 1 ) :: =N D -wh
--- scanning (1, 1) and then (1, 2) queen
( 1 , 1 ) :: =V C
( 1 , 1 ) :: =V +wh C
( 1 , 2 ) :: N

```

```

( 0 , 2 ) : D -wh
--- scanning (2, 2) and then (2, 3) prefers
( 2 , 2 ) :: =V C
( 2 , 2 ) :: =V +wh C
( 2 , 3 ) :: =D =D V
( 2 , 3 ) : =D V (0,2):-wh
--- scanning (3, 3) and then (3, 4) the
( 3 , 3 ) :: =V C
( 3 , 3 ) :: =V +wh C
( 3 , 4 ) :: =N D
--- scanning (4, 4) and then (4, 5) wine
( 4 , 4 ) :: =V C
( 4 , 4 ) :: =V +wh C
( 4 , 5 ) :: N
( 3 , 5 ) : D
( 2 , 5 ) : =D V
( 2 , 5 ) : V (0,2):-wh
( 2 , 5 ) : C (0,2):-wh
( 2 , 5 ) : +wh C (0,2):-wh
( 0 , 5 ) : C
True
>>>

```

9.3 MCFGs

A context free rule like

$$S \rightarrow DP VP$$

can be regarded as an implication:

If string x is a DP and string y is a VP, then string xy is an S.

Or equivalently,

xy is an S if x is a DP and y is a VP.

We can write this in a ‘logical grammar’ notation as follows:

$$S(xy) \leftarrow DP(x) VP(y)$$

That is, if we interpret the arrow as “if-then”, rewrite rules have the arrow backwards. If we find the right side, then we have the left side. In this notation, a lexical rule

$$D \rightarrow \text{the}$$

is most naturally given by a logical rule that has no antecedent,

$$D(\text{the}) \leftarrow$$

which simply says that *the* is a string with category D. A logical notation like this has been used in various grammar formalisms.¹ Here we follow [13] in using it to define “multiple context free grammars” (MCFGs). An MCFG is a CFG that categorizes not just strings but also possibly k -tuples of strings up to some k . For example, the following grammar defines the xx copy language:

$$\begin{aligned}
S(xy) &\leftarrow T(x, y) \\
T(\epsilon, \epsilon) &\leftarrow \\
T(xz, yw) &\leftarrow A(x) A(y) T(z, w) \\
T(xz, yw) &\leftarrow B(x) B(y) T(z, w) \\
A(a) &\leftarrow \\
B(b) &\leftarrow
\end{aligned}$$

We can present a derivation of **abab** like this, where the leaves are all lexical items, and internal nodes are derived by the rules.

¹E.g. definite clause grammars [21], range concatenation grammars [3], literal movement grammars [4].

MCFG notation, the number of instances of `merge1` is finite. And similarly for `merge2`, `merge3`, `move1`, and `move2`. In a sense, every MG is an MCFG. The only difference is in the way the MG rules, in effect, quantify over categories, to define similar effects based on the first elements of the feature sequences. This ‘small’ difference actually has an exponential effect on succinctness, as pointed out in [27].

9.5 MG Earley-like recognition

Harkema’01 presents an Earley-like recognizer for MGs. But there is now a wide range of ideas for parsing MCFGs (or larger classes of grammars) proposed by Ljunglöf’04, Kanazawa’08, Kallmeyer&Maier’09 and Angelov’09. These can all be adapted to MGs.³ As in CFGs, these Earley-like recognizers in effect compute oracles so that they can avoid building constituents which could never be attached into the parse. We follow Ljunglöf’12 most closely here, adapting Ljunglöf’s python Earley-like MCFG parser to use MGs directly.⁴

The Earley parser computes an oracle for itself by predicting each constituent top-down before recognizing it. To use an MG top-down, we begin by predicting a completed derivation with start category *C*, and then the MG rules are applied in reverse, unchecking lexical feature from right to left. To facilitate this feature checking, it is convenient (and more succinct) to represent the lexicon with a graph.

⇒ more coming ⇐

References

- [1] ALBRO, D. M. An Earley-style parser for multiple context free grammars. UCLA, <http://www.linguistics.ucla.edu/people/grads/albro/earley.pdf>. Code: <http://www.linguistics.ucla.edu/stabler/coding.html>, 2002.
- [2] ANGELOV, K. Incremental parsing with parallel multiple context-free grammars. In *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics, EACL (2009)*, pp. 69–76.
- [3] BOULLIER, P. Range concatenation grammars. In *Proceedings of the 6th International Workshop on Parsing Technologies, IWPT’2000 (2000)*, pp. 53–64.
- [4] GROENINK, A. Literal movement grammars. In *Proceedings of the 7th Meeting of the European Association for Computational Linguistics (1995)*, pp. 90–97.
- [5] GUILLAUMIN, M. Conversions between mildly sensitive grammars. UCLA and École Normale Supérieure. <http://www.linguistics.ucla.edu/people/stabler/epssw.htm>, 2004.
- [6] HALE, J. *Grammar, Uncertainty, and Sentence Processing*. PhD thesis, Johns Hopkins University, 2003.
- [7] HARKEMA, H. A recognizer for minimalist grammars. In *Sixth International Workshop on Parsing Technologies, IWPT’00 (2000)*. <http://www.informatics.susx.ac.uk/research/groups/nlp/carroll/iwpt2000/after.html>.
- [8] HARKEMA, H. *Parsing Minimalist Languages*. PhD thesis, University of California, Los Angeles, 2001.
- [9] KALLMEYER, L. *Parsing Beyond Context-Free Grammars*. Springer, NY, 2010.
- [10] KALLMEYER, L., AND MAIER, W. An incremental Earley parser for simple range concatenation grammar. In *Proceedings of the 11th International Conference on Parsing Technologies (IWPT’09) (2009)*, Association for Computational Linguistics, pp. 61–64.
- [11] KANAZAWA, M. A prefix correct Earley recognizer for multiple context free grammars. In *Proceedings of the 9th International Workshop on Tree Adjoining Grammars and Related Formalisms (2008)*, pp. 49–56.
- [12] KANAZAWA, M., MICHAELIS, J., SALVATI, S., AND YOSHINAKA, R. Well-nestedness properly subsumes strict derivational minimalism. In *Logical Aspects of Computational Linguistics, LACL’11 (Berlin, 2011)*, S. Pogodalla and J.-P. Prost, Eds., LNCS/LNAI Volume 6736, Springer, pp. 112–128.
- [13] KANAZAWA, M., AND SALVATI, S. Generating control languages with abstract categorial grammars. In *Proceedings of the 12th Conference on Formal Grammar (FG’07) (Stanford, California, 2007)*, L. Kallmeyer, P. Monachesi, G. Penn, and G. Satta, Eds., CSLI Publications.
- [14] LJUNGLÖF, P. *Expressivity and Complexity of the Grammatical Framework*. PhD thesis, Göteborg University, 2004.

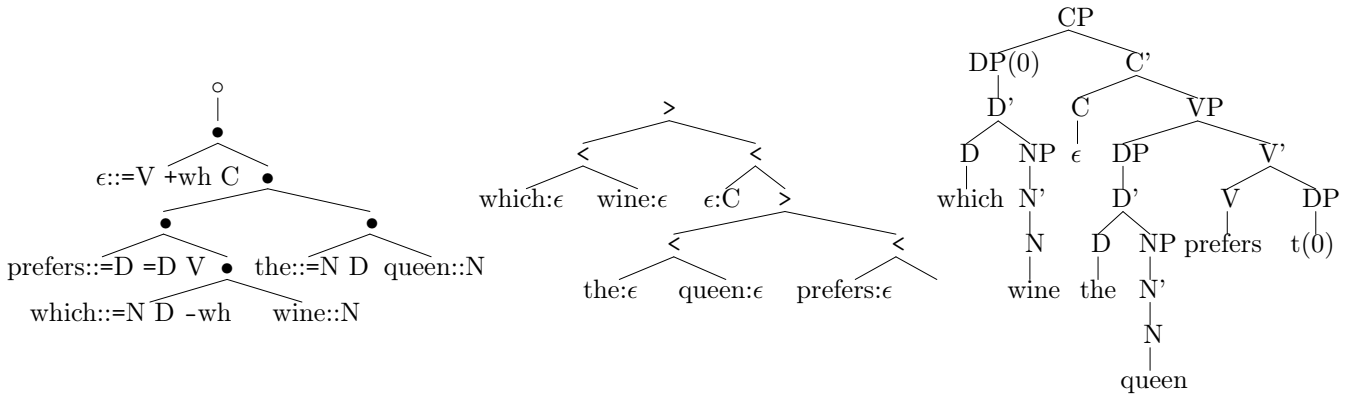
³Kallmeyer’10 surveys some of these Earley-like recognizers [9, §7]. The proposal of Kanazawa’08 is particularly interesting since it uses a natural ‘memoization’ strategy for the database-logic-based programming language datalog. It might be easy to explore this, using some of our MG analysis code, with a python implementation of datalog – see <https://sites.google.com/site/pydatalog/>.

⁴A python implementation of Ljunglöf’s parser is available at <https://github.com/heatherleaf/MCFParser.py>. Adapting Ljunglöf’s parser here sets the stage for the generalizations to parallel multiple context free grammars (PMCFGs) which we adapt to minimalist grammars with copying in §14.7 below.

- [15] LJUNGLÖF, P. Pure functional parsing: an advanced tutorial. Göteborg University, 2005.
- [16] LJUNGLÖF, P. Practical parsing of parallel multiple context-free grammars. In *TAG+11, 11th International Workshop on Tree Adjoining Grammar and Related Formalisms* (2012).
- [17] MICHAELIS, J. Derivational minimalism is mildly context-sensitive. In *Proceedings, Logical Aspects of Computational Linguistics, LACL'98* (NY, 1998), Springer, pp. 179–198.
- [18] MICHAELIS, J. *On Formal Properties of Minimalist Grammars*. PhD thesis, Universität Potsdam, 2001. *Linguistics in Potsdam 13*, Universitätsbibliothek, Potsdam, Germany.
- [19] MICHAELIS, J. Remarks on MCFGs in the light of minimalist grammars. In *Workshop on Multiple Context-Free Grammars and Related Formalisms, National Institute of Informatics, Tokyo, October 5-6* (2010).
- [20] NIYOGI, S. A minimalist implementation of verb subcategorization. In *Seventh International Workshop on Parsing Technologies, IWPT'01* (2001).
- [21] PEREIRA, F. C. N., AND WARREN, D. H. Definite clause grammars for natural language analysis. *Artificial Intelligence 13* (1980), 231–278.
- [22] POLLARD, C. *Generalized phrase structure grammars, head grammars and natural language*. PhD thesis, Stanford University, 1984.
- [23] SEKI, H., MATSUMURA, T., FUJII, M., AND KASAMI, T. On multiple context-free grammars. *Theoretical Computer Science 88* (1991), 191–229.
- [24] STABLER, E. P. Derivational minimalism. In *Logical Aspects of Computational Linguistics*, C. Retoré, Ed. Springer-Verlag (Lecture Notes in Computer Science 1328), NY, 1997, pp. 68–95.
- [25] STABLER, E. P. Minimalist grammars and recognition. In *Linguistic Form and its Computation*, C. Rohrer, A. Rosdeutscher, and H. Kamp, Eds. CSLI Publications, Stanford, California, 2001. (Presented at the SFB340 workshop at Bad Teinach, 1999).
- [26] STABLER, E. P. Recognizing head movement. In *Logical Aspects of Computational Linguistics*, P. de Groote, G. Morrill, and C. Retoré, Eds., Lecture Notes in Artificial Intelligence, No. 2099. Springer, NY, 2001, pp. 254–260.
- [27] STABLER, E. P. Two models of minimalist, incremental syntactic analysis. *Topics in Cognitive Science* (2013). Forthcoming.
- [28] STABLER, E. P., AND KEENAN, E. L. Structural similarity. *Theoretical Computer Science 293* (2003), 345–363.
- [29] VIJAY-SHANKER, K., AND WEIR, D. The use of shared forests in tree adjoining grammar parsing. In *Proceedings of the 6th Conference of the European Chapter of the Association for Computational Linguistics* (1993), pp. 384–393.

Chapter 10 Derivations, derived trees, and tree transductions

We can depict MG derivations with trees like the one on the left below, where \bullet represents a merge step and \circ represents a move step. The result of this 6 step derivation is the tree in the middle, but we usually represent the structure with the X-bar tree on the right:



It turns out that MG derivation trees – trees like the one on the left – are very simple. The set of derivation trees for any grammar is a “regular set of trees.” And from a derivation, it is very easy to derive a bare tree like the one in the middle or an X-bar tree like the one on the right. That is, the rather complex, non-CF structures in the middle and on the right are derived in two regular steps. First we build a well-formed derivation, and then we can (very efficiently!) “spell out” or transduce that derivation into a derived structure.¹ Let’s see how this works.

10.0 MG derivation tree languages are regular

10.0.1 Regular tree acceptors

We very briefly introduce bottom-up, finite state tree acceptors here. See for example [12] for a thorough introduction and many examples. Tree acceptors are usually defined by 4 parts, $M = (\Sigma, Q, F, \delta)$, where Σ is a ranked alphabet, Q is a set of states of rank 0 (i.e. leaves of the tree), $F \subseteq Q$ is the set of final states, and δ is a set of rules of the form

$$f(q_1, \dots, q_n) \rightarrow q$$

where $f \in \Sigma$ has rank n and $q_1, \dots, q_n, q \in Q$.

We define a derives relation \Rightarrow on trees in terms of the rewrite relation \rightarrow of acceptor M as follows.² The rewrite relation introduces states into the trees as new leaves.

Given a tree over ranked alphabet Σ , a context is a tree over $\Sigma \cup \{x\}$ where x has rank 0, $x \notin \Sigma$, and s has exactly one occurrence of x . For any context s over $\Sigma \cup \{x\}$ and any t over Σ , let $s\{x \mapsto t\}$ be the result of replacing x in s by tree t .

Then we say $t \vdash_M t'$ iff for some context s over $\Sigma \cup Q \cup \{x\}$, there is a rule $f(q_1, \dots, q_n) \rightarrow q$ in δ such that

$$\begin{aligned} t &= s\{x \mapsto f(q_0, \dots, q_n)\}, \text{ and} \\ t' &= s\{x \mapsto q\}. \end{aligned}$$

Informally, $t \vdash_M t'$ iff we derive t' from t by one rule application.

¹Cf. [28, 29, 27, 30]. Here we follow [23].

²Here we follow the rather elegant version of the standard definition given in [31].

Example. Consider the following grammar, where q_2 is the only final state:

$$\begin{aligned} c() &\rightarrow q_1 \\ b(q_1) &\rightarrow q_1 \\ a(q_1, q_1) &\rightarrow q_2 \end{aligned}$$

It is easy to see that this accepts the tree below left, by processing the tree from the bottom up:



Clearly this grammar also accepts:



and infinitely other trees, but does not accept:



or infinitely many other trees. Now let's consider MG derivation trees, with \bullet or \circ at the internal nodes and lexical items at the leaves.

10.0.2 MG derivations as regular tree languages

We can define a deterministic bottom-up finite state acceptor for MG derivation trees. Given any

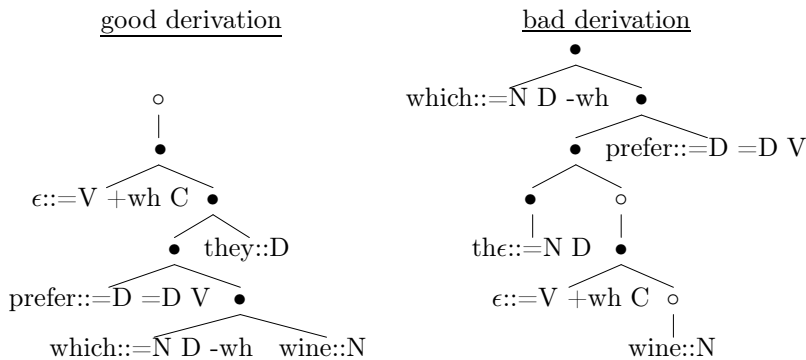
$$G = (\Sigma, F, Lex, C, \bullet, \circ),$$

consider all trees over the ranked alphabet containing \bullet^2, \circ^1 and ℓ^0 for $\ell \in Lex$.

Beginning with the lexical items, suppose we just merge things together arbitrarily, merging either two separate things (indicated by the binary symbol \bullet on the left below) or merging a structure with some constituent that the structure contains (indicated by \circ , on the right).



(Compare the suggestion in Chomsky'12 that merge simply takes any two elements x, y to form $\{x, y\}$, with interface constraints determining whether the resulting derivations are good.) Some trees built in this way are good derivations like the one on the left below. Others are not good, like the one on the right below.



It turns out that checking to see which tree forms a good derivation can be done extremely easily by a finite state device (here, we use a deterministic finite state bottom-up tree acceptor), and the mapping to output representations (‘derived trees’) is also finite state.

We will define a deterministic bottom-up tree acceptor that accepts exactly the complete derivation trees determined by G .

For any MG, the states Q can be represented as pairs (α, μ) where α is the sequence of head features α and μ is the sequence of movers (where each mover is itself a sequence of features). The SMC says that we cannot apply move to a +f structure in which there is more than one -f mover, and so we can simply refuse to create a set of movers from sequences with more than one -f. And the order of the movers does not matter, so μ can be regarded as a set. And the set F of final states will be the start category of the grammar, C .

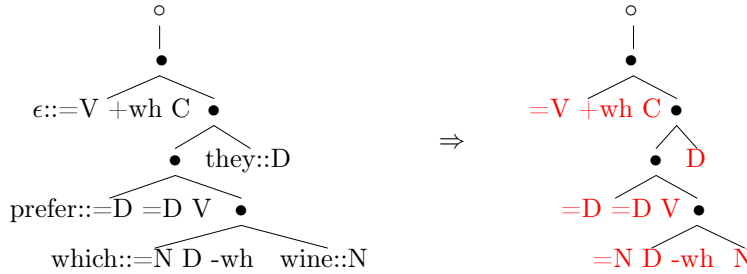
Then all instances of the following comprise the finitely many rules of the acceptor for any MG, where

- w is any sequence of vocabulary elements (in the lexicon)
- α, β are nonempty sequences of features
- μ_1, μ_2 are the sets of sequences of ‘moving features’

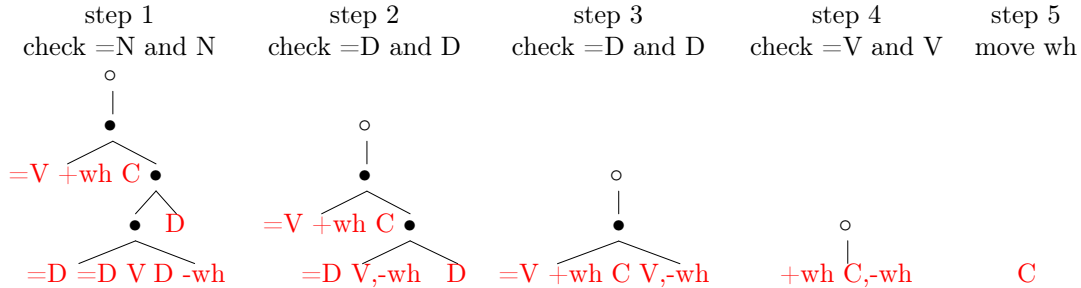
- | | | |
|------------|---|-------------------------------|
| (lex) | $w :: \alpha() \rightarrow \alpha$ | for each $w :: \alpha \in MG$ |
| (merge1,2) | $\bullet((=x\alpha, \mu_1), (x, \mu_2)) \rightarrow (\alpha, \mu_1 \cup \mu_2)$ | if SMC |
| (merge3) | $\bullet((=x\alpha), (\mu_1, x\beta, \mu_2)) \rightarrow (\alpha, \{\beta\} \cup \mu_1 \cup \mu_2)$ | if SMC |
| (move1) | $\circ((+x\alpha, \mu_1 \cup \{-x\})) \rightarrow (\alpha, \mu_1)$ | if SMC |
| (move2) | $\circ((+x\alpha, \mu_1 \cup \{-x\beta\})) \rightarrow (\alpha, \mu_1 \cup \{\beta\})$ | if SMC |

Applying SMC in merge, we block combining two constituents if they both have a -f moving element. And note that this acceptor is deterministic, because no 2 rules have the same left side.

So to check whether an MG derivation is good, our rules map each leaf, each lexical item to a state which is named by its sequence of features. We indicate acceptor states (the features) in red:³



Now instead of lexical items at the leaves, we have just feature sequences, which will be the ‘states’ of our finite state tree acceptor. The acceptor calculates the states for internal nodes by checking features of its daughters in the standard way for MGs [34], and the tree is a good derivation if at the end, at the root, we have just the single category feature C (or whatever category one assumes is the ‘start’ category). This is done in 5 trivial, deterministic steps:

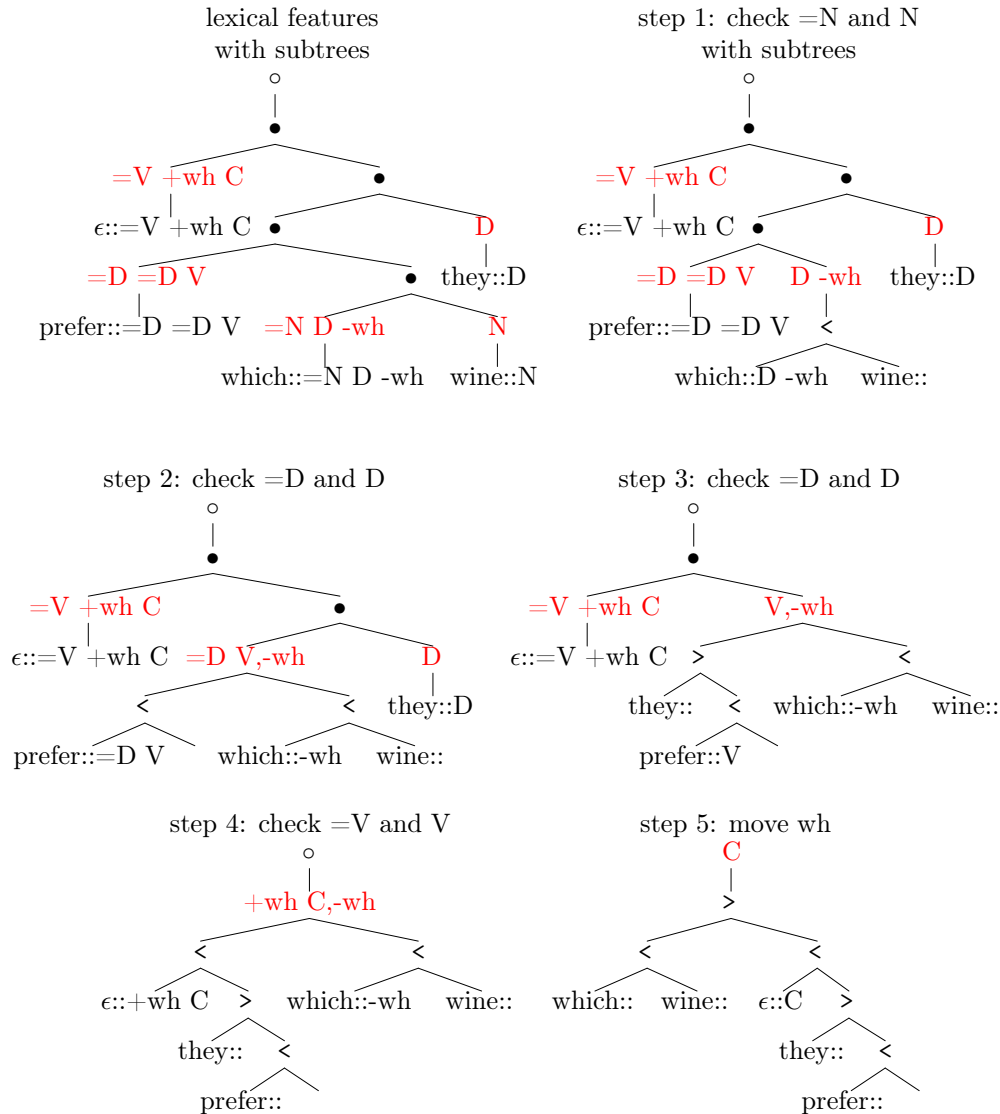


That these steps can be done by with a regular, deterministic bottom-up tree acceptor is pointed out in Koble&al’07,, but the essential insight about this structure was already implicit in Michaelis’98.

³This accepting derivation of the tree is taken from Appendix B of Stabler’13 [35].

10.1 From MG derivations to derived trees

Calculating a derived tree, with the moved constituents in their proper positions, adds very little extra effort. Kobele&al point out that it can be done by a deterministic, multi bottom-up tree transduction. Traversing the tree bottom-up as before, now we let each state have subtrees, and each step can apply a trivial assembly step to the subtrees of the states it is applying to. So now the first step replaces each leaf not just with a state (= the features), but with a state that keeps the original leaf as a subtree. And then, again, we take 5 steps, checking features as before but this time computing subtrees for each state. And again, we put the states in red:



Step 5 is the derived bare phrase structure representation of *which wine they prefer*. Getting X-bar representations instead is only slightly more difficult.

10.2 CKY-like MG parsing

This perspective shows how to convert our CKY recognizer into a parser: we need only collect the derivation trees from the CKY chart, and then we can map those derivation trees to derived trees in whatever format we prefer. To collect the derivation trees in CF CKY parsing, we began with the successful item, the S found between position 0 and the end of the sentence, and simply figured out how this could have been entered into the chart. But MG parsing requires more search (because of merge3!), and so it makes more sense to record our recognizer steps, so that we can simply trace our original steps backward. This is what we do in `mgckyp.py`: we record at each step what items the result comes from. In a good implementation, these previous steps are not complete copies of previous

steps, but pointers, and so the parsing overhead is not too heavy. Once we have the derivation tree, we have code to transduce it into a state tree that shows the acceptor states of the tree, or a bare tree, or an x-bar tree.

We have sessions like the following. The `parse` command in line 2 shows the chart and then returns `(True,trees)` if the chart is successful, as we see in line 49. The simple ‘user interface’ command `ui` used in line 50 prints a tree, and asks whether you want to look for another one. If not, we type `n` to return that tree, and then we can transduce it into various forms as we see here:

```

1  >>> from mgckyp import *
2  >>> parse(mg0,['the','queen','knows','which','beer','the','king','prefers'],'C')
3  --- scanning (0, 0) and then (0, 1) the
4  ( 0 , 0 ) :: =V C
5  ( 0 , 0 ) :: =V +wh C
6  ( 0 , 1 ) :: =N D
7  --- scanning (1, 1) and then (1, 2) queen
8  ( 1 , 1 ) :: =V C
9  ( 1 , 1 ) :: =V +wh C
10 ( 1 , 2 ) :: N
11 ( 0 , 2 ) : D
12 --- scanning (2, 2) and then (2, 3) knows
13 ( 2 , 2 ) :: =V C
14 ( 2 , 2 ) :: =V +wh C
15 ( 2 , 3 ) :: =C =D V
16 --- scanning (3, 3) and then (3, 4) which
17 ( 3 , 3 ) :: =V C
18 ( 3 , 3 ) :: =V +wh C
19 ( 3 , 4 ) :: =N D -wh
20 --- scanning (4, 4) and then (4, 5) beer
21 ( 4 , 4 ) :: =V C
22 ( 4 , 4 ) :: =V +wh C
23 ( 4 , 5 ) :: N
24 ( 3 , 5 ) : D -wh
25 --- scanning (5, 5) and then (5, 6) the
26 ( 5 , 5 ) :: =V C
27 ( 5 , 5 ) :: =V +wh C
28 ( 5 , 6 ) :: =N D
29 --- scanning (6, 6) and then (6, 7) king
30 ( 6 , 6 ) :: =V C
31 ( 6 , 6 ) :: =V +wh C
32 ( 6 , 7 ) :: N
33 ( 5 , 7 ) : D
34 --- scanning (7, 7) and then (7, 8) prefers
35 ( 7 , 7 ) :: =V C
36 ( 7 , 7 ) :: =V +wh C
37 ( 7 , 8 ) :: =D =D V
38 ( 7 , 8 ) : =D V (3,5):-wh
39 ( 5 , 8 ) : V (3,5):-wh
40 ( 5 , 8 ) : C (3,5):-wh
41 ( 5 , 8 ) : +wh C (3,5):-wh
42 ( 3 , 8 ) : C
43 ( 2 , 8 ) : =D V
44 ( 0 , 8 ) : V
45 ( 0 , 8 ) : C
46 --- scanning (8, 8)
47 ( 8 , 8 ) :: =V C
48 ( 8 , 8 ) :: =V +wh C
49 (True, [[['*'], ([], [(1, 'V'), (-1, 'C')])], ['*', ['*'], (['knows'], [(1, 'C'), (1, 'D'), (-1, 'V')])], ['o'], ['*'], ([],
50 >>> pptree(0,dt2t(ui(mg0,['the','queen','knows','which','beer','the','king','prefers'],'C'))))
51 ...
52 ['*'], ([], [(1, 'V'), (-1, 'C')])], ['*', ['*'], (['knows'], [(1, 'C'), (1, 'D'), (-1, 'V')])], ['o'], ['*'], ([], [(1, 'V')
53 another?] n
54 *
55     ::=V C
56     *
57         *
58             knows::=C =D V
59             o
60                 *
61                     ::=V +wh C
62                     *
63                         *
64                             prefers::=D =D V
65                             *
66                                 which::=N D -wh
67                                 beer::N
68                                 *

```

```

69             the ::= N D
70             king ::= N
71         *
72         the ::= N D
73         queen ::= N
74
75 >>> pptree(0,dt2xb(ui(mg0,['the','queen','knows','which','beer','the','king','prefers'],'C')))
76     ...
77     ['*', ([], [(1, 'V'), (-1, 'C')]), ['*', ['*', (['knows'], [(1, 'C'), (1, 'D'), (-1, 'V')])], ['o', ['*', ([], [(1, 'V'),
78     another? n
79     CP
80     C
81
82     []
83     VP
84     DP
85     D
86     the
87     NP
88     queen
89     V'
90     V
91     knows
92     CP
93     DP(0)
94     D
95     which
96     NP
97     beer
98     C'
99     C
100
101     []
102     VP
103     DP
104     D
105     the
106     NP
107     king
108     V'
109     V
110     prefers
111
112     ('DP(0)',)
113 >>>

```

With the NLTK tree display, we can look at these trees in the more readable graphical formats displayed in these notes.

10.3 Ambiguity and more complex examples

Now that we have a parser which can display our trees, let's look at ambiguity, and then at a couple of other linguistic proposals which involve (mainly) phrasal movement.

10.3.1 Ambiguous strings from unambiguous lexical sequences

We have already seen that MGs can be regarded as a (exponentially more succinct) notation for MCFGs, where an MCFG differs from a CFG only in the way the strings are manipulated. So if we don't manipulate the strings, every MCFG is just a CFG. That is, the yields of the derivation trees of MGs are context free languages. We are now in a position to notice something else about them.

Unless we have 'precedence rules' of some kind to determine the preferred reading, the logical formula $\neg p \wedge q$ is ambiguous. And obviously the ambiguity can matter – one reading entails $\neg p$, but the other one doesn't!

One way to eliminate the ambiguity is with parentheses,

$$(\neg p) \wedge q \quad \text{vs.} \quad \neg(p \wedge q).$$

Another way is to use Polish notation,

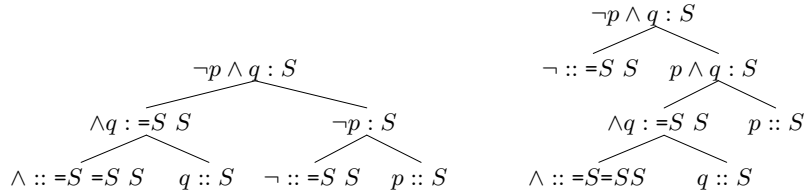
$$\wedge \neg pq \quad \text{vs.} \quad \neg \wedge pq.$$

The parenthesis notation is more commonly used, but the Polish notation is formally more elegant.⁴

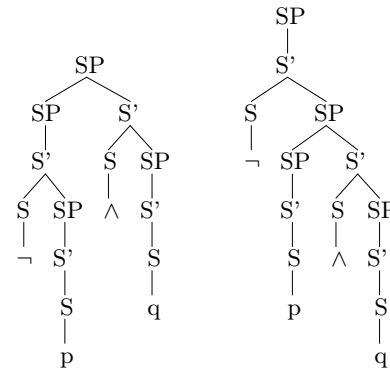
Now consider the minimalist grammar **mgpc**:

$$\begin{array}{lll}
 p :: S & q :: S & r :: S \\
 \neg :: =S S & \vee :: =S =S S & \wedge :: =S =S S
 \end{array}$$

This grammar has ambiguous expressions, since we have the following two different two derivations of $\neg p \wedge q$:



These correspond to trees that we might depict with X-bar structure in the following way:



While these examples show that **mgpc** has ambiguous expressions, compare the derivation trees. Notice that the yields of the two simple derivation trees shown above (not the X-bar structures, but the derivation trees) are not the same. The two yields are, respectively,

$$\begin{array}{lll}
 \wedge :: =S=SS & q :: S & \neg :: =SS & p :: S \\
 \neg :: =SS & \wedge :: =S=SS & q :: S & p :: S
 \end{array}$$

In fact, not only this grammar, but every minimalist grammar is unambiguous in this sense [16]. Each sequence of lexical items has at most one derivation. These sequences are, in effect, Polish notation for the sentence, one that completely determines the whole derivation. Notice that if we leave out the features from the lexical sequences above, we have exactly the standard Polish notation:

$$\begin{array}{l}
 \wedge q \neg p \\
 \neg \wedge qp
 \end{array}$$

The lexical sequences that are the yields of any MG derivation trees form an unambiguous context free language.

10.3.2 Promotion analyses of relative clauses

In Ling 1, relative clauses are often described as questions used as modifiers. For example, in

The farmer [who chased the cat] saw the dog The cat [of which I have spoken often] chased the rat

it seems the answers to *who chased the cat* are supposed to help us understand which farmer you might be referring to But that perspective is not really right, as we notice immediately from the fact that perfect questions like these cannot be relative clauses (even ignoring the subject-auxiliary inversion, which seems to happen only in matrix clauses):

⁴Why isn't the formally elegant notation more common? It is harder to read! (why?) Shoenfield's logic text actually adopts the prefix notation for the language officially, but presents things in the infix, parenthesized notation for us to read. Cf. Hawkins' idea that we like to keep arguments near their operators.

which farmer chased the cat? but not *the man which farmer chased the cat
 of which cat have you spoken? but not *the cat of which cat you have spoken

It seems that a more natural perspective is one in which the head noun must be “raised” or “promoted” out of the question in order to become a relative clause:

farmer [which ___ chased the cat]?
 cat [of which ___ you have spoken]?

Let’s see how this might work.

Kayne [20, §8] proposes something rough like this. Promotion analyses were independently proposed much earlier because of an apparent similarity between relative clauses and certain kinds of focus constructions [32, 36, 1, 3]:⁵

- a. This is the cat that chased the rat
- b. It’s the cat that chased the rat
- a. * That’s the rat that this is the cat that chased
- b. * It’s that rat that this is the cat that chased
- a. Sun gaya wa yaron (Hausa)
 PERF.3PL tell IOBJ child
 ‘they told the child’
- b. yaron da suka gaya wa
 child REL 3PL tell IOBJ
 ‘the child that they told’
- c. yaron ne suka gaya wa
 child FOCUS 3PL tell IOBJ
 ‘it’s the child that they told’
- a. nag-dala ang babayi sang bata (Ilonggo)
 AGT-bring TOPIC woman OBJ child
 ‘the woman brought a child’
- b. babanyi nga nag-dala sang bata
 woman REL AGT-bring OBJ child
 ‘a woman that brought a child’
- c. ang babanyi nga nag-dala sang bata
 TOPIC woman REL AGT-bring OBJ child
 ‘it’s the woman that brought a child’

The suggestion is that in all of these constructions, the focused noun raises to a prominent position in the clause. In the relative clauses, the clause with the raised noun is the sister of the determiner; in the clefts, the clause is the sister of the copula. We could assume that these focused elements land in separate focus projections, but for the moment let’s assume that they get pulled up to the CP.

Kayne assumes that the relative pronoun also originates in the same projection as the promoted head, so we get analyses with the structure:

1. The hammer_{*i*} [which *t_j*]_{*j*} [*t_j* broke *t_h*]_{*k*} [the window]_{*h*} *t_k*
2. The window_{*i*} [which *t_j*]_{*j*} [the hammer]_{*h*} [*t_h* broke *t_j*]_{*k*} *t_k*

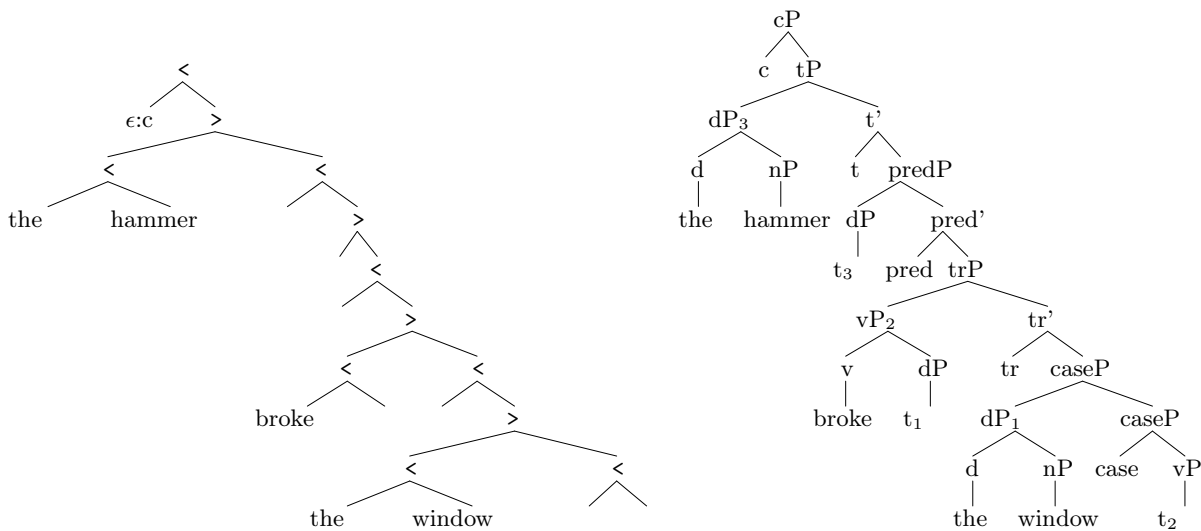
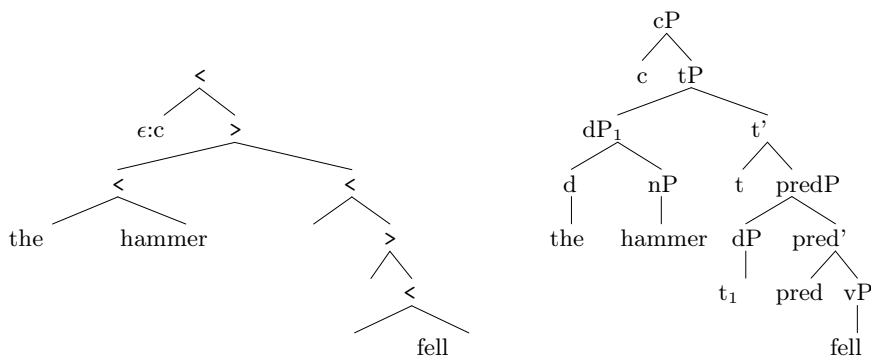
We can obtain this kind of analysis by allowing noun heads of relative clauses to be focused, entering the derivation with some kind of focus feature *-f*.

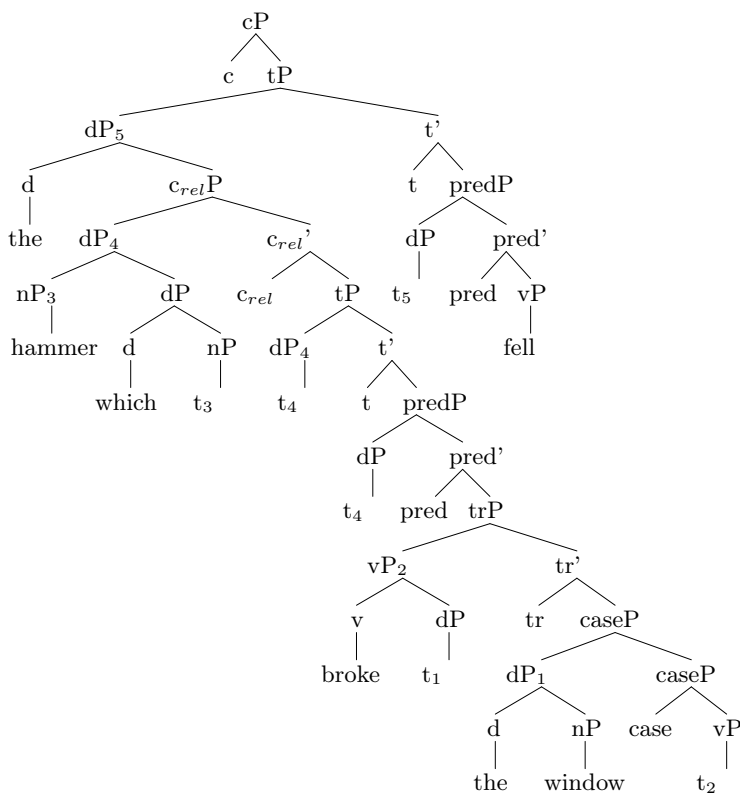
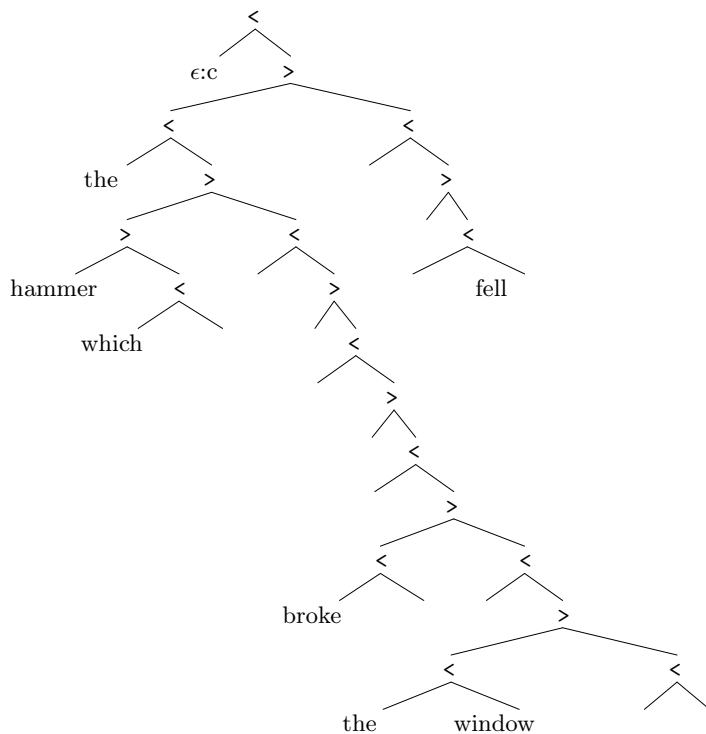
⁵Some recent studies suggest that relative clauses in English and some other languages may have both promotion structures as well as others [4, 19, 17].

$\epsilon ::= t \ c$	$\epsilon ::= t \ +wh_{rel} \ c_{rel}$
$\epsilon ::= pred \ +case \ t$	
$the ::= n \ d \ -case \ the$	$which ::= n \ +f \ d \ -case \ -wh_{rel}$
$hammer ::= n$	$the ::= c_{rel} \ d \ -case$
$window ::= n$	$hammer ::= n \ -f$
$\epsilon ::= v \ +case \ case$	$window ::= n \ -f$
$\epsilon ::= tr \ =d \ pred$	
$\epsilon ::= case \ +tr \ tr$	
$broke ::= d \ v \ -tr$	

NB: focused lexical items in the second column.

This allows us to derive (showing bare tree on left, x-bar tree on right):





Buell [7] shows that Kayne’s analysis of relative clauses does not extend easily to Swahili. In Swahili, it is common to separate the NP head of the relative clause from the relative pronoun -cho:

- Hiki ni kitabu ni- li- cho- ki- soma
 7.this cop 7.book 1s.subj- past- 7.o.relpro 7.obj- read
 ‘This is the book that I read’

- Hiki ni kitabu ni- ki- soma -cho
7.this cop 7.book 1s.subj- 7.obj- read -7.o.relpro read
'This is the book that I read'

Other critiques of Kayne are presented in [6] and some of them are answered in [3] and elsewhere...

10.3.3 Smuggling

John_i seems to Mary [*t_i* to be nice]

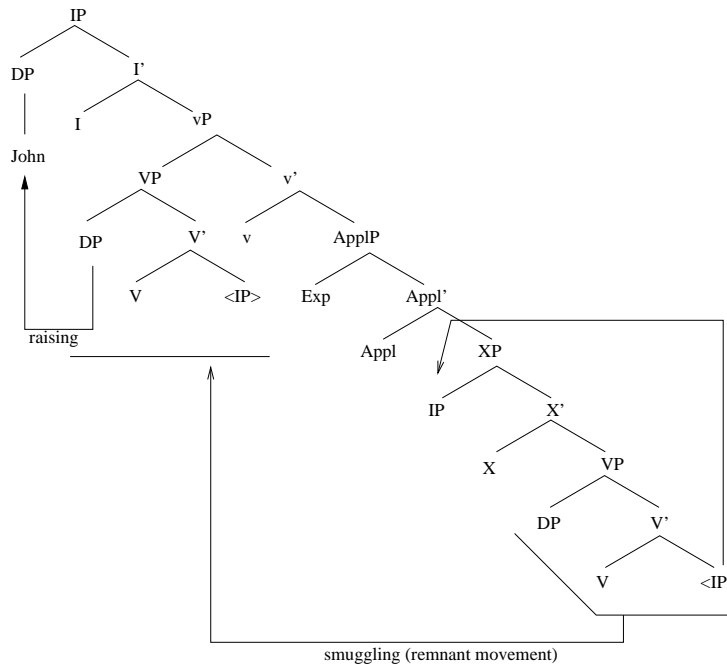
[_{IP}John [_{I'}I [_{VP}[_{VP}<John>] [_{V'}seem <IP>]]] [_{v'}V [_{AppIP}Mary [_{AppI'}Appl [_{XP}IP [_{X'}X <VP>]]]

John gets to matrix subject from VP, after that VP has moved to spec,vP

Chris Collins provides a step-by-step analysis of raising constructions [11]. (Note that he adopts a convention of leaving off closing right brackets when no confusion can result.) I will not discuss the motivation for this analysis – see Collins’ paper – but he has his eye on the c-command worries raised by examples like the following:

- *John seems to her_i to like Mary_i
- The dog seems to [every boy]_i to like all of his_i toys

These issues have been discussed by many people [8, 2, 21, 5]. For the moment, I am interested in the mechanics of the particular derivation Collins proposes.



(In transcribing this tree from [11, p.295], I corrected what I take to be an error in the published version: I put an X as the left child of X' where the published version has XP.)

1. Merge(John, nice) = [_{AdjP}John nice]
2. Merge(be, AdjP) = [_{VP} be [_{AdjP}John nice]
3. Merge(to, VP) = [_{IP} to [_{VP} be [_{AdjP}John nice]
4. Move(IP[John]) = [_{I'P}John [_{I'} to [_{VP} be [_{AdjP}<John> nice]
5. Merge(seem,IP) = [_{VP}seem [_{IP}John [_{I'} to [_{VP} be [_{AdjP}<John> nice]
6. Move(John,VP) = [_{VP}John [_{VP}seem [_{IP}<John> [_{I'} to [_{VP} be [_{AdjP}<John> nice]
7. Merge(X,VP) = [_{XP}X [_{VP}John [_{VP}seem [_{IP}<John> [_{I'} to [_{VP} be [_{AdjP}<John> nice]

8. $\text{Move}(\text{XP}[\text{IP}]) = [\text{XP} [\text{IP}(\text{John}) \text{ to be nice}] [\text{VP} \text{John} [\text{VP} \text{seem} \langle \text{IP} \rangle]]$
9. $\text{Merge}(\text{Appl}, \text{XP}) = [\text{AppIP} \text{Appl} [\text{XP} [\text{IP} \langle \text{John} \rangle \text{ to be nice}] [\text{VP} \text{John} [\text{VP} \text{seem} \langle \text{IP} \rangle]]]$
10. $\text{Merge}(\text{Mary}, \text{AppIP}) =$
 $[\text{AppIP} \text{Mary} [\text{AppI}' \text{Appl} [\text{XP} [\text{IP} \langle \text{John} \rangle \text{ to be nice}] [\text{VP} \text{John} [\text{VP} \text{seem} \langle \text{IP} \rangle]]]]]$

** whoops, this last step is trouble! **

You should now be able to assign features to lexical items in such a way as to make these steps happen.

Collins is especially interested in the movement of the VP, followed by the extraction of John from that VP. He calls this ‘smuggling’. We can allow that part of the smuggling to happen. . . But we cannot allow this last step, step 10, because neither *Mary* nor *John* has had its case checked in the last structure shown in this slide, and our grammars will not allow a derivation to complete once there are 2 -k phrases in it. – This is analogous to the problem Chomsky wrestles with in [9, Chapter 4], where (even with head movement etc, which we have not introduced yet) he wants to extraction of a DP from a VP that has both a subject and object DP in it. The system we have introduced here will not allow this, unless the DPs move for different reasons.

A different idea has been suggested by Dave Schueler (p.c.), who writes:

It seems that while [MGs] don’t allow, assuming the SMC, anything capturing the true spirit of Collins’s smuggling idea, they do allow a certain type of "featural smuggling", since the SMC only disallows the first features of any two items being the same. That is, at a certain point in the derivation, you could have a: :D -k and b: :D -x -k, with b lower (merged earlier) than a. Then, a head could check -k on a, then another head could check -x on b, and still another could later check -k on b, with b thus moving “over” a.

One way in which this doesn’t capture the spirit of Collins’s idea is that it doesn’t require a literal smuggling movement; no larger structure containing b has to move over a before b moves out of that structure. Instead, b can stay low until the +x head is merged, then move there, then up to check -k. However, the movement to check +x could be considered the smuggling step in a way.

In the case of passive, and perhaps all linguistically relevant uses, it violates the spirit of Collins another way; it requires a lexical difference between D’s which will be passive subjects and those that don’t. However, this might be part of the large general property of MGs that they require more features than people doing informal work notice.

Can you implement this idea? (Possible squib topic!)

References

- [1] ÁFARLI, T. A. A promotion analysis of restrictive relative clauses. *The Linguistic Review* 11 (1994), 81–100.
- [2] BARSS, A., AND LASNIK, H. A note on anaphora and double objects. *Linguistic Inquiry* 17 (1986), 347–354.
- [3] BHATT, R. Adjectival modifiers and the raising analysis of relative clauses. In *Proceedings of the North Eastern Linguistic Society, NELS 30* (1999).
- [4] BHATT, R. The raising analysis of relative clauses: Evidence from adjectival modification. *Natural Language Semantics* 10 (2002), 43–90.
- [5] BOECKX, C. Conflicting c-command requirements. *Studia Linguistica* 53 (1999), 227–250.
- [6] BORSLEY, R. D. Relative clauses and the theory of phrase structure. *Linguistic Inquiry* 28 (1997), 629–647.
- [7] BUELL, L. Swahili relative clauses. UCLA M.A. thesis, 2000.
- [8] CHOMSKY, N. *Knowledge of Language*. Praeger, NY, 1986.
- [9] CHOMSKY, N. *The Minimalist Program*. MIT Press, Cambridge, Massachusetts, 1995.
- [10] CHOMSKY, N. Problems of projection. *Lingua forthcoming* (2012).
- [11] COLLINS, C. A smuggling approach to raising in English. *Linguistic Inquiry* 36, 2 (2005), 289–298.
- [12] COMON, H., DAUCHET, M., GILLERON, R., JACQUEMARD, F., LUGIEZ, D., TISON, S., AND TOMMASI, M. Tree automata techniques and applications. <http://www.grappa.univ-lille3.fr/tata>, 1997.
- [13] ENGELFRIET, J., LILIN, E., AND MALETTI, A. Composition and decomposition of extended multi bottom-up tree transducers. *Acta Informatica* 46, 8 (2009), 561–590.

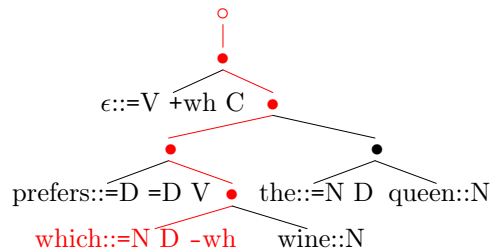
- [14] FÜLÖP, Z., KÜHNEMANN, A., AND VOGLER, H. A bottom-up characterization of deterministic top-down tree transducers with regular look-ahead. *Information Processing Letters* 91 (2004), 57–67.
- [15] GRAF, T. Closure properties of minimalist derivation tree languages. In *Logical Aspects of Computational Linguistics, LACL'11* (2011).
- [16] HALE, J., AND STABLER, E. P. Strict deterministic aspects of minimalist grammars. In *Logical Aspects of Computational Linguistics, LACL'05*, Lecture Notes in Artificial Intelligence LNCS-3492. Springer, NY, 2005, pp. 162–176.
- [17] HARRIS, J. Interpreting raising and matching analyses of relative clauses. In *27th West Coast Conference on Formal Linguistics, WCCFL XXVII* (2008).
- [18] HAWKINS, J. A. Why are categories adjacent? *Journal of Linguistics* 27 (2001), 1–34.
- [19] HULSEY, S., AND SAUERLAND, U. Sorting out relative clauses. *Natural Language Semantics* 14 (2006), 111–137.
- [20] KAYNE, R. S. *The Antisymmetry of Syntax*. MIT Press, Cambridge, Massachusetts, 1994.
- [21] KITAHARA, H. *Elementary Operations and Optimal Derivations*. MIT Press, Cambridge, Massachusetts, 1997.
- [22] KOBELE, G. M. Minimalist tree languages are closed under intersection with recognizable tree languages. In *Logical Aspects of Computational Linguistics, LACL'11* (2011), S. Pogodalla and J.-P. Prost, Eds., pp. 129–144.
- [23] KOBELE, G. M., RETORÉ, C., AND SALVATI, S. An automata-theoretic approach to minimalism. In *Model Theoretic Syntax at 10. ESSLLI'07 Workshop Proceedings* (2007), J. Rogers and S. Kepsers, Eds.
- [24] MALETTI, A. Every sensible extended top-down tree transducer is a multi bottom-up tree transducer. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (Montréal, Canada, June 2012), Association for Computational Linguistics, pp. 263–273.
- [25] MALETTI, A., GRAEHL, J., HOPKINS, M., AND KNIGHT, K. The power of extended top-down tree transducers. *SIAM Journal on Computing* 39, 2 (2009), 410–430.
- [26] MICHAELIS, J. Derivational minimalism is mildly context-sensitive. In *Proceedings, Logical Aspects of Computational Linguistics, LACL'98* (NY, 1998), Springer, pp. 179–198.
- [27] MICHAELIS, J., MÖNNICH, U., AND MORAWIETZ, F. Derivational minimalism in two regular and logical steps. In *Proceedings of the 5th International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+5)* (2000), pp. 163–170.
- [28] MÖNNICH, U. TAGs M-constructed. In *TAG+ Workshop, Institute for Research in Cognitive Science, University of Pennsylvania* (1998).
- [29] MÖNNICH, U. Well-nested tree languages and attributed tree transducers. In *The 10th International Conference on Tree Adjoining Grammars and Related Formalisms TAG+10* (2010).
- [30] MORAWIETZ, F. *Two Step Approaches to Natural Language Formalisms*. de Gruyter, Berlin, 2003.
- [31] MOSS, L., AND TIEDE, H.-J. Applications of modal logic in linguistics. In *Handbook of Modal Logic*, P. Blackburn, J. van Benthem, and F. Wolter, Eds., Studies in Logic and Practical Reasoning, Volume 3. Elsevier, Amsterdam, 2007, pp. 1031–1076.
- [32] SCHACHTER, P. Focus and relativization. *Language* 49, 1 (1973), 19–46.
- [33] SHOENFIELD, J. R. *Mathematical Logic*. Addison-Wesley, Menlo Park, California, 1967.
- [34] STABLER, E. P. Computational perspectives on minimalism. In *Oxford Handbook of Linguistic Minimalism*, C. Boeckx, Ed. Oxford University Press, Oxford, 2011, pp. 617–641.
- [35] STABLER, E. P. Two models of minimalist, incremental syntactic analysis. *Topics in Cognitive Science* (2013). Forthcoming.
- [36] VERGNAUD, J.-R. *Dépendances et Niveaux de Représentation en Syntaxe*. PhD thesis, Université de Paris VII, 1982.

Chapter 11 MG GLC beam recognition

A number of different ‘all paths at once’ strategies are defined in [1], but there has been relatively little work on ‘one path at a time’ strategies. Some first steps were taken in [7, 2, 8, 9], but the topic deserves more attention, since psychologists often propose models of this kind, and it is likely to reveal interesting new perspectives on the grammar. With memoization, the various ‘one path at a time’ strategies can be efficient, but some of them may be usable with appropriate probabilistic beam strategies, even without memoization [4].

11.1 Top-down MG beam recognition

A top-down MG beam recognizer is easy to define and implement. What we want to do is parse the derivation tree (not the derived tree!), but we want to parse that tree in the order of left to right in the derived tree. So for example, in the derived tree for *which wine the queen prefers*, the first word *which* is at the bottom of the tree, and is not shown in leftmost position, but of course it is leftmost. So the “left branch” of the tree is really the branch to the leftmost word, shown in red here:



Left-mostness is determined by the derivation steps, of course, and so we can mark the linear order of each node with an index, as it is created top-down.¹ We index each node as we create it, and put these predicted elements into a priority queue that sorts by least index, that is by left-mostness, rather than keeping the predictions in a stack as done in CF recognizers. Recall from page 45 that we are already using a priority queue to keep the parses sorted from most probable to least. Now each parse uses (not a stack but) a priority queue to keep the predicted elements sorted from leftmost to rightmost.

With this idea, to recognize the 12-node derivation tree above, we begin as usual by predicting C. We use the single dot to indicate that the predicted C can be either lexical (::) or derived (:), and we put the predicted categories in [square brackets]:

which wine the queen prefers, p·C]. (0) start

The first step expands that prediction with a move step, to get the following:

which wine the queen prefers, [:+wh C₁, -wh₀]. (1) expand: unmove

This indicates that the +wh C head with index 1 will follow the moving -wh phrase which has index 0. (Note that we use parentheses here for grouping the predicted constituents, not to indicate completed elements.) The second step expands that prediction with a merge-complement:

which wine the queen prefers, [:V₁₁, -wh₀] [::=V +wh C₁₀] (2) expand: unmerge

Note that the mover has not been passed to the complement V. The head $::=V +wh C_{10}$ will precede the head of the complement $:V_{11}, -wh_0$, as indicated by the indices 10 and 11 indicate, but the constituent containing the -wh

¹This idea was suggested most directly in work by Mainguy [3], but indices have been used in similar ways in earlier work too [8].

mover is still placed first because it has the 0 index – it will precede all of the other elements. Expanding the leftmost constituent again with a merge-specifier step we get:

$$\text{which wine the queen prefers, } [:=D \text{ V}_{111}, -\text{wh}_0] [:=V +\text{wh} \text{ C}_{10}] [:D_{110}] \quad (3) \text{ expand: unmerge}$$

Note that the mover has not been passed to the specifier D, but stays with the head. Again we expand the leftmost element to merge the -wh object, removing it from the verbal projection, and so the sorted order is now:

$$\text{which wine the queen prefers, } [:D -\text{wh}_0] [:=V +\text{wh} \text{ C}_{10}] [:D_{110}] [:=D =D \text{ V}_{111}] \quad (4) \text{ expand: unmerge}$$

Again expanding the leftmost element we get:

$$\text{which wine the queen prefers, } [:=N \text{ D} -\text{wh}_{00}] [:=N_{01}] [:=V +\text{wh} \text{ C}_{10}] [:D_{110}] [:=D =D \text{ V}_{111}] \quad (5) \text{ expand: unmerge}$$

This leftmost element is the category of the first word so we scan to get:

$$\text{wine the queen prefers, } [:=N_{01}] [:=V +\text{wh} \text{ C}_{10}] [:D_{110}] [:=D =D \text{ V}_{111}] \quad (6) \text{ scan}$$

At this point we have completed the left branch that we marked in red, with the other branches attaching to the red branch waiting now as predictions. So now, scanning again,

$$\text{the queen prefers, } [:=V +\text{wh} \text{ C}_{10}] [:D_{110}] [:=D =D \text{ V}_{111}] \quad (7) \text{ scan}$$

We have reached the empty complementizer, which is scanned without changing the input:

$$\text{the queen prefers, } [:D_{110}] [:=D =D \text{ V}_{111}] \quad (8) \text{ scan}$$

At this point, we expand with a merge step

$$\text{the queen prefers, } [:=N \text{ D}_{1100}] [:=N_{1101}] [:=D =D \text{ V}_{111}] \quad (9) \text{ unmerge}$$

Scanning:

$$\text{queen prefers, } [:=N_{1101}] [:=D =D \text{ V}_{111}] \quad (10) \text{ scan}$$

Scanning:

$$\text{prefers, } [:=D =D \text{ V}_{111}] \quad (11) \text{ scan}$$

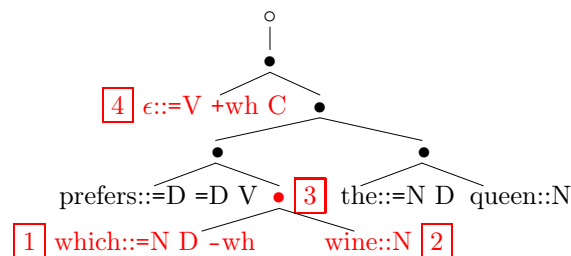
And finally, the last scan to success:

$$\epsilon, \epsilon \quad (12) \text{ scan}$$

It is quite easy to implement this calculation. It is natural to use the graphical representation of the lexicon that is described in §9.5, and the search can be a beam. This recognition algorithm is presented in detail in Appendix B of Stabler’13, and is implemented in `mgtdb.py`. It is extended to a parser in `mgtdbp.py`.²

11.2 Bottom-up MG beam recognition

A bottom-up left-to-right MG recognizer should perform each merge step immediately after the merging elements have been built, and move steps should be performed immediately after the move-triggering element has been built. Considering the same derivation again, repeated here with the first bottom-up steps numbered in red.



²Implementations of this parser in various languages are available at <https://github.com/epstabler/mgtdb/wiki>.

These are exactly the first steps of the correct parse in the order that they are taken by the CKY parser. To handle non-adjacent parts of constituents, let's use positions for the moment, as done in CKY. So then, stepping through the correct recognition of the derivation, the steps are the following. I will put the predicted elements in [square brackets] and the completed elements in (parentheses).

which wine the queen prefers,	[·C _{0,5}]	(0) start
wine the queen prefers,	(::=N D -wh _{0,1}) [·C _{0,5}]	(1) shift-reduce
the queen prefers,	(::N _{1,2}) (::=N D -wh _{0,1}) [·C _{0,5}]	(2) shift-reduce
the queen prefers,	(:D -wh _{0,2}) [·C _{0,5}]	(3) reduce
the queen prefers,	(::=V +wh C _{2,2}) (:D -wh _{0,2}) [·C _{0,5}]	(4) reduce
queen prefers,	(::=N D _{2,3}) (::=V +wh C _{2,2}) (:D -wh _{0,2}) [·C _{0,5}]	(5) shift-reduce
prefers,	(::N _{3,4}) (::=N D _{2,3}) (::=V +wh C _{2,2}) (:D -wh _{0,2}) [·C _{0,5}]	(6) shift-reduce
prefers,	(:D _{2,4}) (::=V +wh C _{2,2}) (:D -wh _{0,2}) [·C _{0,5}]	(7) reduce
ε,	(::=D =D V _{4,5}) (:D _{2,4}) (::=V +wh C _{2,2}) (:D -wh _{0,2}) [·C _{0,5}]	(8) shift-reduce
ε,	(:=D V _{4,5,-wh_{0,2}}) (:D _{2,4}) (::=V +wh C _{2,2}) [·C _{0,5}]	(9) reduce
ε,	(:V _{2,5,-wh_{0,2}}) (::=V +wh C _{2,2}) [·C _{0,5}]	(10) reduce
ε,	(:+wh C _{2,5,-wh_{0,2}}) [·C _{0,5}]	(11) reduce
ε,	ε	(12) reduce-complete

These steps are taken by our incremental CKY recognizer. They could also be taken by a beam parser, but I have not implemented this.

11.3 Left-corner MG beam recognition

A left-corner recognizer is bottom-up on left corners and top-down on their sisters (if any). ‘Left’ in the term ‘left corner’ refers to pronounced order.

which wine the queen prefers,	[·C _{0,5}]	(0) start
wine the queen prefers,	(::=N D -wh _{0,1}) [·C _{0,5}]	(1) shift
wine the queen prefers,	[N _{1,i}] (:D -wh _{0,i}) [·C _{0,5}]	(2) lc-reduce
the queen prefers,	(:D -wh _{0,2}) [·C _{0,5}]	(3) scan
the queen prefers,	(::=D =D V _{i,j}] (:D V _{i,j} , -wh _{0,2}) [·C _{0,5}]	(4) lc-reduce
the queen prefers,	(::=V +wh C _{2,2}] (::=D =D V _{i,j}] (:D V _{i,j} , -wh _{0,2}) [·C _{0,5}]	(5) lc-reduce ε
the queen prefers,	[:V _{2,k,-wh_{ℓ,2}] (:+wh C_{2,k,-wh_{ℓ,2}] (::=D =D V_{i,j}] (:D V_{i,j}, -wh_{0,2}) [·C_{0,5}]}}	(6) lc-reduce
queen prefers,	(::=N D _{2,3}] [:V _{2,k,-wh_{ℓ,2}] (:+wh C_{2,k,-wh_{ℓ,2}] (::=D =D V_{i,j}] (:D V_{i,j}, -wh_{0,2}) [·C_{0,5}]}}	(7) shift
queen prefers,	(::N _{3,m}] (:D _{2,m}] [:V _{2,k,-wh_{ℓ,2}] (:+wh C_{2,k,-wh_{ℓ,2}] (::=D =D V_{i,j}] (:D V_{i,j}, -wh_{0,2}) [·C_{0,5}]}}	(8) lc-reduce
prefers,	(:D _{2,4}] [:V _{2,k,-wh_{ℓ,2}] (:+wh C_{2,k,-wh_{ℓ,2}] (::=D =D V_{i,j}] (:D V_{i,j}, -wh_{0,2}) [·C_{0,5}]}}	(9) scan
prefers,	(::=D =D V _{4,j}] (:+wh C _{2,j,-wh_{0,2}) [·C_{0,5}]}	(10) lc-reduce-complete
prefers,	(:+wh C _{2,5,-wh_{0,2}}) [·C _{0,5}]	(11) scan
ε,	ε	(12) lc-reduce-complete

As far as I know, this has never been carefully defined or implemented.

11.4 GLC MG beam recognition

The full range of GLC methods has not been investigated.

References

- [1] HARKEMA, H. *Parsing Minimalist Languages*. PhD thesis, University of California, Los Angeles, 2001.
- [2] JOSHI, A. K. Processing crossed and nested dependencies: An automaton perspective on the psycholinguistic results. *Language and Cognitive Processes* 5, 1 (1990), 1–27.
- [3] MAINGUY, T. A probabilistic top-down parser for minimalist grammars. <http://arxiv.org/abs/1010.1826v1>, 2010.
- [4] ROARK, B. Probabilistic top-down parsing and language modeling. *Computational Linguistics* 27, 2 (2001), 249–276.
- [5] SEKI, H., MATSUMURA, T., FUJII, M., AND KASAMI, T. On multiple context-free grammars. *Theoretical Computer Science* 88 (1991), 191–229.
- [6] STABLER, E. P. Two models of minimalist, incremental syntactic analysis. *Topics in Cognitive Science* (2013). Forthcoming.

- [7] VIJAYASHANKER, K. *A Study of Tree Adjoining Languages*. PhD thesis, University of Pennsylvania, 1987.
- [8] VILLEMONTÉ DE LA CLERGERIE, E. Parsing MCS languages with thread automata. In *Proceedings, 6th International Workshop on Tree Adjoining Grammars and Related Frameworks, TAG+6* (2002).
- [9] WARTENA, C. *Storage Structures and Conditions on Movement in Natural Language Syntax*. PhD thesis, Universität Potsdam, 1999.

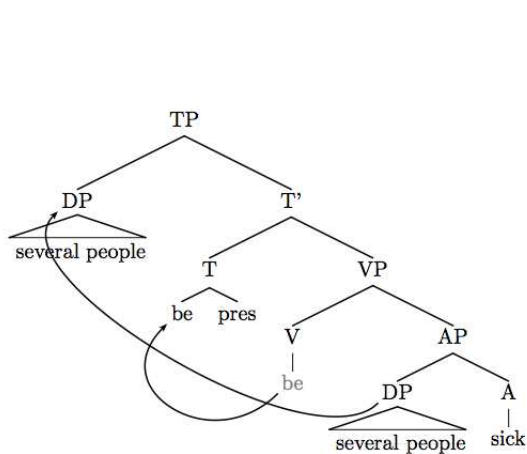
Chapter 12 MG elaborated

Let's extend MGs to implement a grammar similar to the one developed in the Koopman&al'13 text (ISAT), which is used here at UCLA.

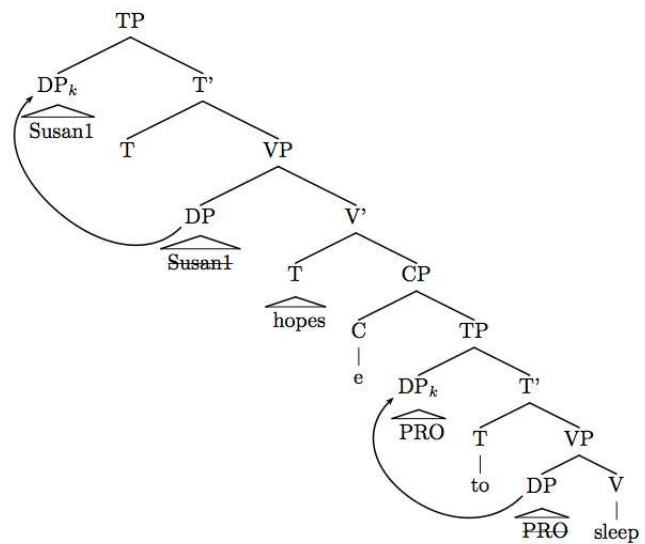
12.1 Persistent features

12.1.1 EPP and Attract closest

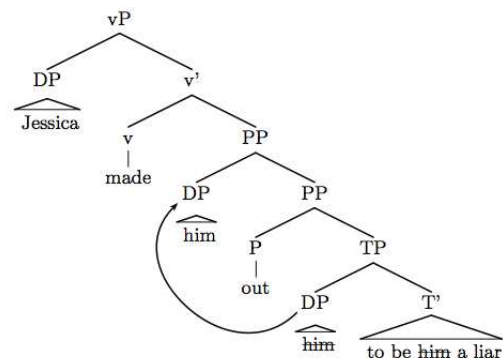
In some linguistic proposals, the trigger for movement to subject position in simple English sentences is called the "extended projection principle." In [14, §8.5], for example, this requirement is encoded by giving the T (tense) head a feature $epp_{DP/CP}$ which triggers the movement of a DP. (When T is finite, the DP must be nominative, but we leave aside case marking for the moment.) Some of the examples we see in the ISAT text are these:



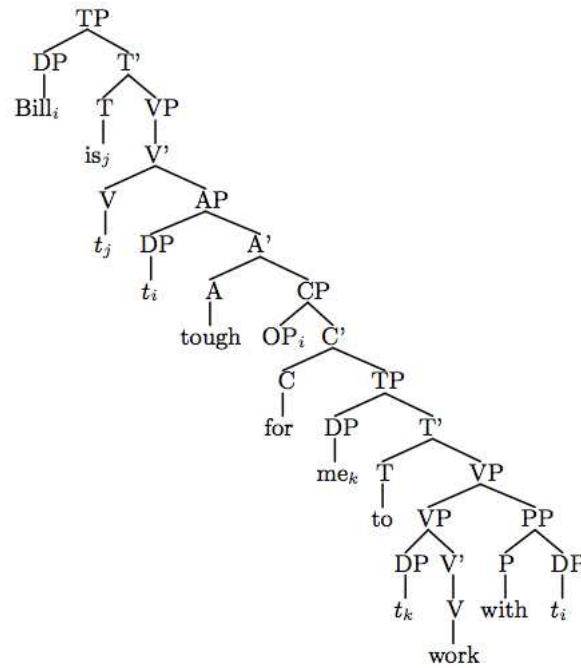
ISAT: p243



ISAT: p269



ISAT: p403



ISAT: p454

The problem with implementing these analyses in MGs is that the subject's categorial features are deleted when they are selected by the VP. But this requires only a simple extension of MGs. Let's assume that, whenever D (or C) is selected, in addition to deleting the D (C), another item is produced with the feature -D (-C).¹ We implement this idea by adding this instance of merge, which is exactly like the previous one except that instead of deleting the categorial feature f , it makes $-f$ available for licensing:

$$\text{merge}_{SC}(t_1[=f], t_2[f]) = \begin{cases} \left\langle \begin{array}{l} < \\ t_1 \quad t_2[-f] \end{array} \right. & \text{if } t_1 \text{ has exactly 1 node} \\ \left\langle \begin{array}{l} > \\ t_2[-f] \quad t_1 \end{array} \right. & \text{otherwise} \end{cases}$$

But then, how do we make sure, in a transitive clause, that it is the subject rather than the object that raises to spec,TP? One idea is that the object cannot move to Spec,TP because the subject is closer [14, p.332]:

Attract Closest: Only the closest potential candidate can move to an attracting head (which selects for it)

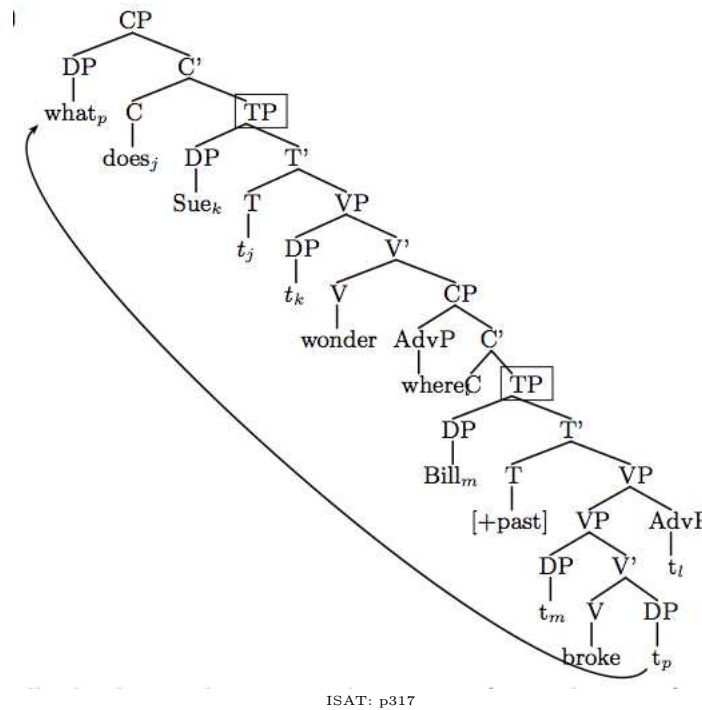
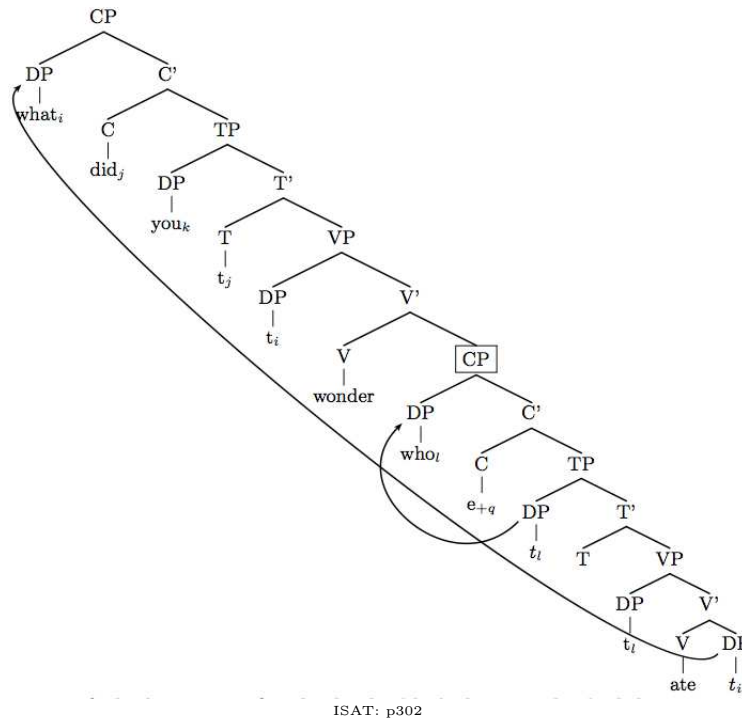
We implement this restriction by blocking merge rules that apply to $t[=f]$ and $t'[f]$ when eight tree has a $-f$ mover. We add this SMC-like restriction to all merge steps: a selector $=X$ and selectee X cannot merge if either has a $-X$ mover. With this approach, an EPP feature is simply a $+f$ where f is categorial, and we can simply represent the lexical entry for past tense this way:

$$\begin{aligned} -\text{ed}::=V +D T \\ -\text{ed}::=V +C T \end{aligned}$$

12.1.2 Successive movements

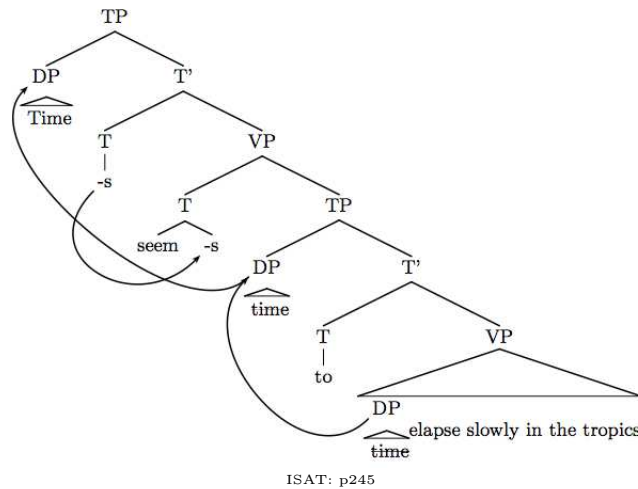
The text also allows $-wh$ features to persist in successive wh -movements like these:

¹We can assume that every categorial feature X can, in principle, be attracted by epp_X , but obviously we only need to record those values $-X$ for which epp_X features exist in the grammar.



If the first movement deleted the -wh, the second one would not be possible, so we add to our rules the possibility of optionally leaving the -wh undeleted in a movement step.

A similar thing can happen with raising, as we see in examples like this:



This can be allowed with the adjustment mentioned in the previous section: we optionally do not delete the -f feature in each movement.

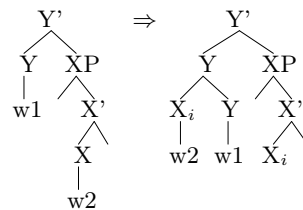
For these kinds of ‘feature persistence’, we add an instance of move that does not delete the licensee feature:

$$\text{move}_{SC}(t_1[+f]) = t_2[-f] \text{ if (SMC) only one head has -f as its first feature}$$

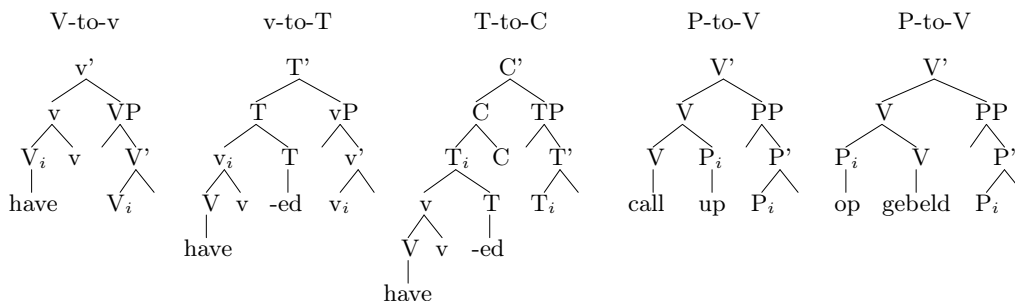
This rule is added to the original rules, so, in effect, the added -f features are optional. These elaborations of MGs are rather minor.

12.2 Head movement and affix hopping

The movement allowed in MGs is phrasal: that is, whenever a head is moved, the phrases it selects move too (unless they have already moved away). But in the 1980’s, many linguists assumed that movement could move just a head. And in most of these proposals, the assumed head movement is ‘local’ in the sense that a head moves up to the head selecting its phrase (or else, in affix hopping, the selecting head moves down to the head of the selected phrase):



There are many examples in the literature. The movements involved in ‘subject-auxiliary inversion’ (V to v, v to T, T to C), and ‘preposition incorporation’ (P to V) are just a few of them:



Many alternatives have been considered, but head movement remains popular. Roberts’10 says

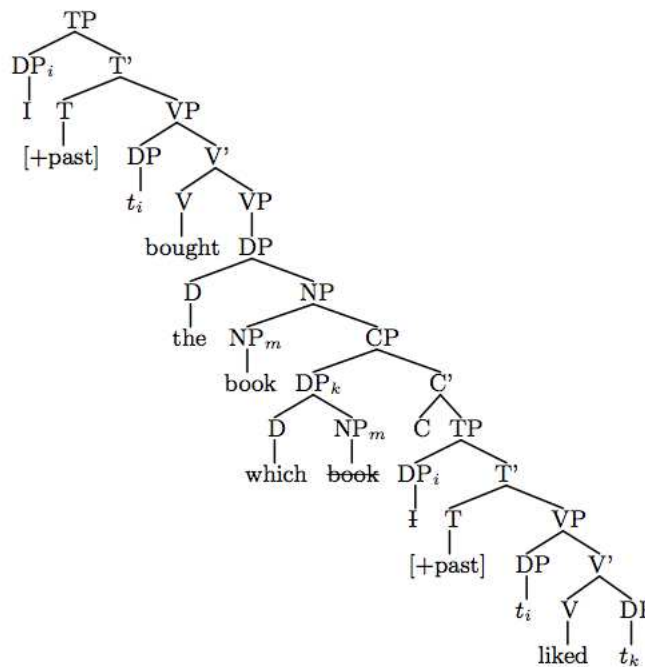
... the variety of head movement I will argue for is indistinguishable from XP-movement in all respects except the irreducible one that this is the case where Move applies to a possibly non-maximal, certainly minimal, element.

The problem is that many things go along with the simple ‘irreducible’!² The simple fact is that head movement moves a head without any of its arguments or adjuncts. More complicated is the fact that a moved head does not become a specifier, but the selecting head incorporates the selected head to become complex. And if we allow affix hopping, as ISAT does, then the selected head can incorporate the selecting head too! Here let’s consider how we could adapt MGs, in the most straightforward way possible, to allow this.³

$$\text{mergeHM}(t_1[=>\mathbf{f}], t_2[\mathbf{f}]) = \begin{array}{c} < \\ \swarrow \quad \searrow \\ \text{hd}_2 t_1 \quad t_2(\text{hd}_2 \mapsto \epsilon) \end{array} \quad \text{if } t_1 \text{ has exactly 1 node}$$

$$\text{mergeAH}(t_1[<=\mathbf{f}], t_2[\mathbf{f}]) = \begin{array}{c} < \\ \swarrow \quad \searrow \\ t_1(\text{hd}_1 \mapsto \epsilon) \quad t_2(\text{hd}_2 \mapsto \text{hd}_2 \text{hd}_1) \end{array} \quad \text{if } t_1 \text{ has exactly 1 node}$$

12.3 Adjunction

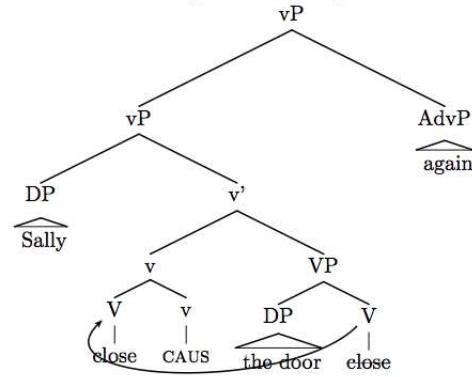


ISAT: p451

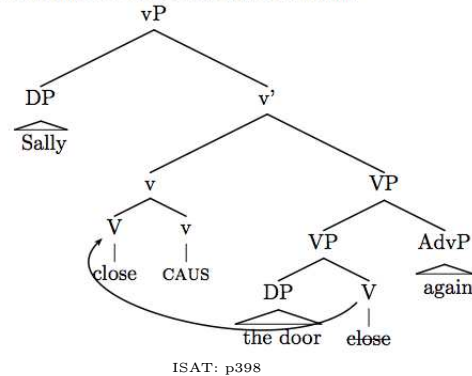
²For critical assessments of Roberts’ proposal see [17, 22].

³Here we basically follow [33]. Various other ideas are reviewed in Stabler’03.

(108) Structure for the Repetitive Reading



(109) Structure for the Restitutive Reading



ISAT: p398

$$\text{adjoin}(t_1[f], t_2[g\delta]) = \begin{cases} \begin{array}{c} < \\ t_1 \quad t_2[g] \end{array} & \text{if } t_1 \text{ if } f \in \text{left-adjoiners}(g) \\ \begin{array}{c} > \\ t_2[g] \quad t_1 \end{array} & \text{if } t_1 \text{ if } f \in \text{right-adjoiners}(g) \end{cases}$$

A similar idea is proposed by Frey and Gärtner [6].⁴

12.4 Subjacency

Ross'67 made famous the idea that movement could be stated as a general and simple rule if it is constrained to avoid unwanted applications like these:⁵

XXX

We have imposed the 'shortest move constraint' (SMC). Salvati'11 shows that without that constraint, MGs can define much more complex, intractable languages.

ISAT (§10.5.3) adds this restriction on movement:

⁴The 'Maryland' tradition in syntax suggests that adjunction should be simpler than merge (see, for example, [2, §3.4]), an idea which has been formalized in a rather elegant way by Hunter [13].

⁵Whether these restrictions should be part of the grammar is controversial. The idea that constraints on movement are extragrammatical has been of interest especially in alternatives to Chomskian syntax that want to dispense with constraints or with both constraints and movement. For example, in recent discussions of combinatory categorial grammar, Steedman, in [37, p.49] and [36], for example, suggests that island effects may be extragrammatical, due to performance factors. The categorial approach need not take that view, though. Moortgat, Morrill and others have explored enriching the types of categorial grammars to encode island environments [21, 19]. Vermaat, Retoré, Lecomte, and others have shown that type-logical grammars can mimic Chomskian analyses [39, 40, 15]. Morrill and colleagues provides an elegant categorial approach to discontinuity in which such constraints could be imposed [20]. Recently, Szabolcsi and others have argued that (at least some) constraints on movement are really semantic [], and Sag and others have argued that (at least some) constraints can be regarded as (gradient) memory restrictions [28, 31, 11], while

(Subjacency) Movement cannot cross two bounding nodes, where the bounding nodes are TPs and CPs that are not complements of V.

As discussed for example by Sportiche'81, the bounding nodes in different languages could vary.

12.5 CKY parsing: the tuple-based representation

The CKY-like methods extend easily to MGs with head-movement, adjunction, and persistent features with the operations we have defined, but now we have many more cases to cover. And we impose this extended version of the SMC:

(SMCx1) Merge cannot apply to a $=f$ selector and f category if either constituent has a $-f$ mover, or if both constituents have a $-g$ mover for any g .

(SMCx2) Move cannot apply to a $+f$ constituent unless it has exactly one $-f$ subtree.

Furthermore:

(Subjacency) Nothing can move across two bounding nodes, where a bounding node is a TP or CP that is not a complement of V. This is enforced by associating each mover with an integer $k \in \{0, 1, 2\}$, beginning with $k = 0$ and then:

- T or C cannot merge with X if any mover of X already has a mover with integer 2. Otherwise, when T or C merges with X every mover of X is incremented.
- When V merges with T or C, each mover is decremented.

(Adjunct Island Constraint) Nothing can move out of an adjunct. I.e., the adjunct rules do not apply if the adjunct has any movers.

$$\frac{(\epsilon, s, \epsilon) :: =f\gamma \quad (t_s, t_h, t_c) \cdot f, \alpha_1, \dots, \alpha_k}{(\epsilon, s, t_s t_h t_c) : \gamma, \alpha_1, \dots, \alpha_k} \text{merge1: lexical item selects a non-mover as complement}$$

$$\frac{(\epsilon, s, \epsilon) :: =f\gamma \quad (t_s, t_h, t_c) \cdot f, \alpha_1, \dots, \alpha_k}{(\epsilon, s, \epsilon) : \gamma, t_s t_h t_c : -f, \alpha_1, \dots, \alpha_k} \text{merge1}_{\text{EPP}}: \text{lex item selects EPP mover}$$

$$\frac{(\epsilon, s, \epsilon) :: =>f\gamma \quad (t_s, t_h, t_c) \cdot f, \alpha_1, \dots, \alpha_k}{(\epsilon, t_h s, t_s t_c) : \gamma, \alpha_1, \dots, \alpha_k} \text{merge1}_{\text{hm}}: \text{lex item selects head-moving non-mover}$$

$$\frac{(\epsilon, s, \epsilon) :: =>f\gamma \quad (t_s, t_h, t_c) \cdot f, \alpha_1, \dots, \alpha_k}{(\epsilon, t_h s, \epsilon) : \gamma, t_s t_c : -f, \alpha_1, \dots, \alpha_k} \text{merge1}_{\text{hm,sc}}: \text{lex item selects head-moving SC mover}$$

$$\frac{(\epsilon, s, \epsilon) :: f=>\gamma \quad (t_s, t_h, t_c) \cdot f, \alpha_1, \dots, \alpha_k}{(\epsilon, \epsilon, t_s t_h t_c) : \gamma, \alpha_1, \dots, \alpha_k} \text{merge1}_{\text{hop}}: \text{lex item selects and hops to non-mover}$$

$$\frac{(\epsilon, s, \epsilon) :: f=>\gamma \quad (t_s, t_h, t_c) \cdot f, \alpha_1, \dots, \alpha_k}{(\epsilon, \epsilon, \epsilon) : \gamma, t_s t_h t_c : -f, \alpha_1, \dots, \alpha_k} \text{merge1}_{\text{hop,sc}}: \text{lexical item selects and hops to SC mover}$$

$$\frac{(s_s, s_h, s_c) :: =f\gamma, \alpha_1, \dots, \alpha_k \quad (t_s, t_h, t_c) \cdot f, \iota_1, \dots, \iota_l}{(t_s t_h t_c s_s, s_h, s_c) : \gamma, \alpha_1, \dots, \alpha_k, \iota_1, \dots, \iota_l} \text{merge2: derived item selects a non-mover as specifier}$$

$$\frac{(s_s, s_h, s_c) :: =f\gamma, \alpha_1, \dots, \alpha_k \quad (t_s, t_h, t_c) \cdot f, \iota_1, \dots, \iota_l}{(s_s, s_h, s_c) : \gamma, t_s t_h t_c : -f, \alpha_1, \dots, \alpha_k, \iota_1, \dots, \iota_l} \text{merge2}_{\text{sc}}: \text{derived item selects SC mover as specifier}$$

$$\frac{(s_s, s_h, s_c) \cdot =f\gamma, \alpha_1, \dots, \alpha_k \quad (t_s, t_h, t_c) \cdot f\delta, \iota_1, \dots, \iota_l}{(s_s, s_h, s_c) : \gamma, t_s t_h t_c : \delta, \alpha_1, \dots, \alpha_k, \iota_1, \dots, \iota_l} \text{merge3: any item selects a mover}$$

$$\frac{(s_s, s_h, s_c) \cdot =f\gamma, \alpha_1, \dots, \alpha_k \quad (t_s, t_h, t_c) \cdot f\delta, \iota_1, \dots, \iota_l}{(s_s, s_h, s_c) : \gamma, t_s t_h t_c : \neg f\delta, \alpha_1, \dots, \alpha_k, \iota_1, \dots, \iota_l} \text{merge3}_{\text{sc}}: \text{any item selects SC multiple mover}$$

$$\frac{(s_s, s_h, s_c) : +f\gamma, \alpha_1, \dots, \alpha_{i-1}, t : -f, \alpha_{i+1}, \dots, \alpha_k}{(ts_s, s_h, s_c) : \gamma, \alpha_1, \dots, \alpha_{i-1}, \alpha_{i+1}, \dots, \alpha_k} \text{move1: final move of licensee}$$

$$\frac{(s_s, s_h, s_c) : +f\gamma, \alpha_1, \dots, \alpha_{i-1}, t : -f, \alpha_{i+1}, \dots, \alpha_k}{(s_s, s_h, s_c) : \gamma, \alpha_1, \dots, \alpha_{i-1}, t : -f, \alpha_{i+1}, \dots, \alpha_k} \text{move1}_{sc}: \text{nonfinal move of SC mover}$$

$$\frac{(s_s, s_h, s_c) : +f\gamma, \alpha_1, \dots, \alpha_{i-1}, t : -f\delta, \alpha_{i+1}, \dots, \alpha_k}{(s_s, s_h, s_c) : \gamma, \alpha_1, \dots, \alpha_{i-1}, t : \delta, \alpha_{i+1}, \dots, \alpha_k} \text{move2: nonfinal move of licensee}$$

$$\frac{(s_s, s_h, s_c) : +f\gamma, \alpha_1, \dots, \alpha_{i-1}, t : -f\delta, \alpha_{i+1}, \dots, \alpha_k}{(s_s, s_h, s_c) : \gamma, \alpha_1, \dots, \alpha_{i-1}, t : -f\delta, \alpha_{i+1}, \dots, \alpha_k} \text{move2}_{sc}: \text{nonfinal move of SC multiple licensee}$$

$$\frac{(s_s, s_h, s_c) \cdot f \quad (t_s, t_h, t_c) \cdot g\gamma, \alpha_1, \dots, \alpha_k}{(s_s s_h s_c t_s, t_h, t_c) : g\gamma, \alpha_1, \dots, \alpha_k} \text{adjoinL: if } f \in \text{left-adjoiners}(g)$$

$$\frac{(s_s, s_h, s_c) \cdot f \quad (t_s, t_h, t_c) \cdot g\gamma, \alpha_1, \dots, \alpha_k}{(t_s, t_h, t_c s_s s_h s_c) : g\gamma, \alpha_1, \dots, \alpha_k} \text{adjoinR: if } f \in \text{right-adjoiners}(g)$$

$$\frac{(s_s, s_h, s_c) \cdot f\delta \quad (t_s, t_h, t_c) \cdot g\gamma, \alpha_1, \dots, \alpha_k}{(t_s, t_h, t_c) : g\gamma, s_s s_h s_c : \delta, \alpha_1, \dots, \alpha_k} \text{adjoin3: if } f \in \text{right-adjoiners}(g) \cup \text{left-adjoiners}(g)$$

There are so many similarities among these rules, one suspects that a much simpler grammar would work as well or better.

Since each constituent is now indexed by 3 strings (6 integer positions), the python CKY implementation uses not a matrix but a dictionary that maps 6-tuples of integers to the list of feature sequences at that position, together with their movers. Note that, for a 39 word sentence we build a 40^2 matrix, and looping through the upper half of a $40^2 = 1600$ cell matrix is not too bad. But the upper half of a $40^6 = 4,096,000,000$ matrix takes significantly longer to examine! Fortunately, those matrices are usually very sparse, and so instead of looping through all the (mainly empty) cells in a huge matrix, we loop through the actual items listed in the dictionary.⁶ Otherwise, the implementation is essentially similar to the one described in §10.2 on page 120.

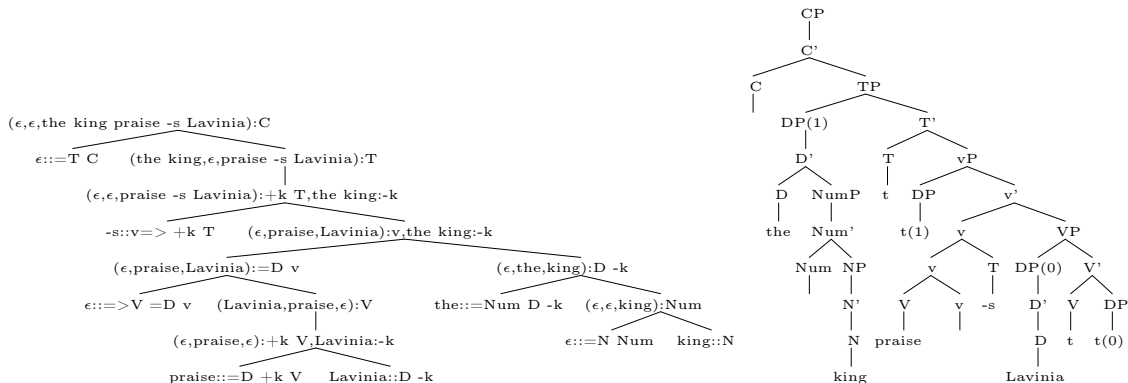
12.6 A simple English

We have enough machinery in place to handle quite a broad range of syntactic structures in a conventional, Chomskian fashion. It is worth a brief digression to see how some of the basics might get treated in this framework, and this will provide some valuable practice for later.

According to a simple, traditional analysis, transitive verb phrases are formed from two projections, vP and VP, where the lower VP selects the object. To achieve this in MGs, we let **transitive verbs** have lexical items requiring object selection and case assignment, like this:

$$\text{praise}::=D \ V \quad \epsilon::=>V =D \ v$$

Here we see that the V selects a DP and then moves it to assign (accusative) case, forming a VP. This VP is then selected by v and the head V is left adjoined to the head v by head movement, and then the subject (the “external argument”) of the verb is selected.

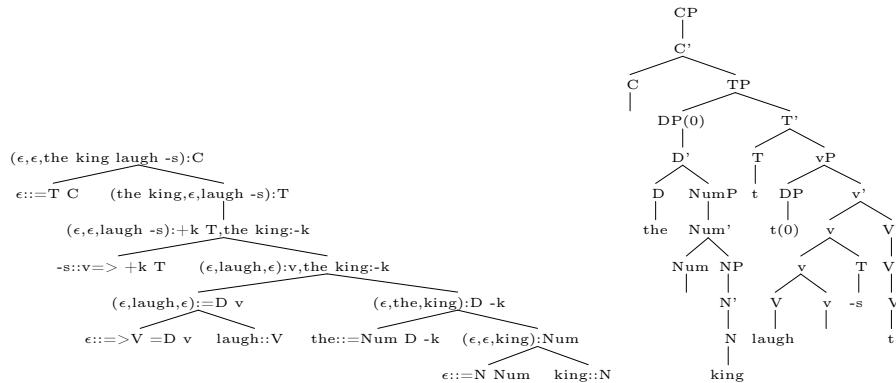


⁶Note that looping through almost all cells is only necessary for merge3 and for the new cases of merge_{hm} and merge_{hop}. For the non-head-moving instances of merge1 and merge2, we know where to look, and we loop through those possibilities.

In contrast, an **intransitive verb** has a simpler lexical entry like this:

laugh::V

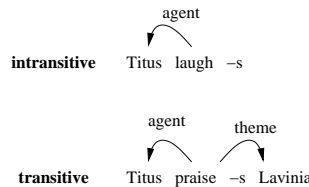
This verb selects no object and assigns no case, but it combines with *v* to get its subject in the usual way.



Of course, some verbs like *eat* can occur in both transitive and intransitive forms, so verbs like this have two lexical entries:

eat::V eat::=D +k V.

Considering what each V and its associated *v* selects, we can see that they are the semantic arguments. So the familiar semantic relations are being mirrored by selection steps in the derivation:



Throughout this section, we will aim to have derivations that mirror semantic relations in this way.

12.6.1 CP-selecting verbs and nouns

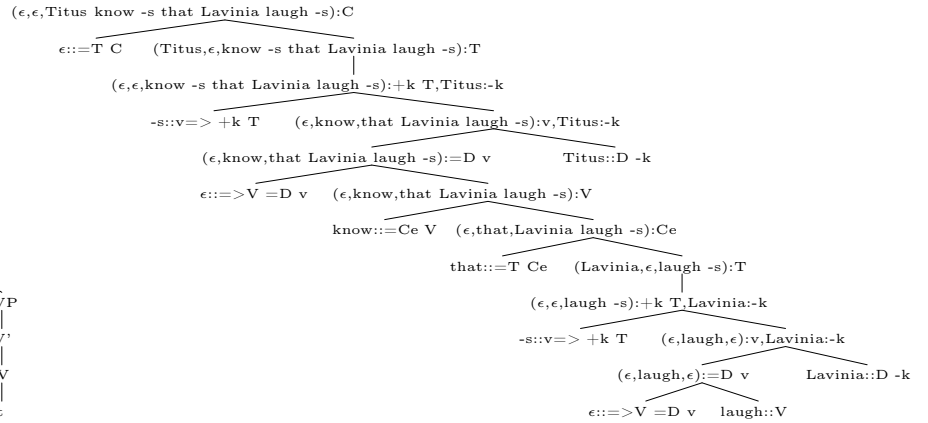
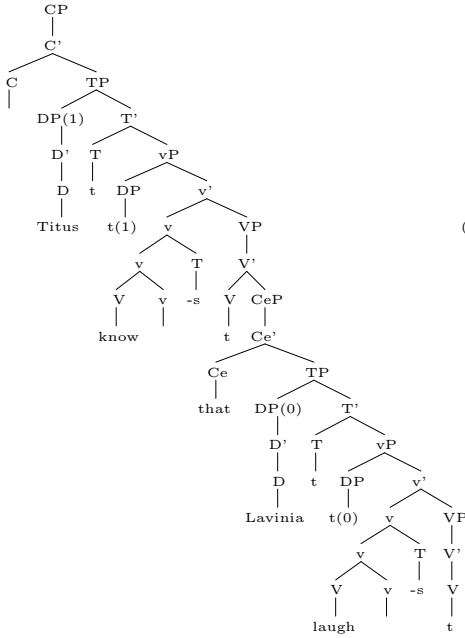
It is easy to add verbs that select categories other than DP. For example, some verbs select full clauses as their complements. It is commonly observed that matrix clauses have an empty complementizer while embedded clauses can begin with *that*, and verbs vary in the kinds of clauses they allow:

- (0) * That Titus laughs
- (1) Titus thinks that Lavinia laughs
- (2) * Titus thinks which king Lavinia praises
- (3) * Titus wonders that Lavinia laughs
- (4) Titus wonders which king Lavinia praises

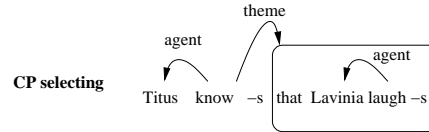
Verbs like *know* select both kinds of complements and can also occur in transitive and intransitive constructions. We can get these distinctions with lexical entries like this:

that::=T Ce	ε::=T Ce	whether::=T Cwh	
ε::=T +wh Cwh	ε::=>T +wh Cwh		
know::=Ce V	know::=Cwh V	know::=D +k V	know::V
doubt::=Ce V	doubt::=Cwh V	doubt::V	
think::=Ce V		think::V	
	wonder::=Cwh V	wonder::V	

With these lexical entries we obtain derivations like this (showing a conventional depiction on the left and the actual derivation tree on the right):



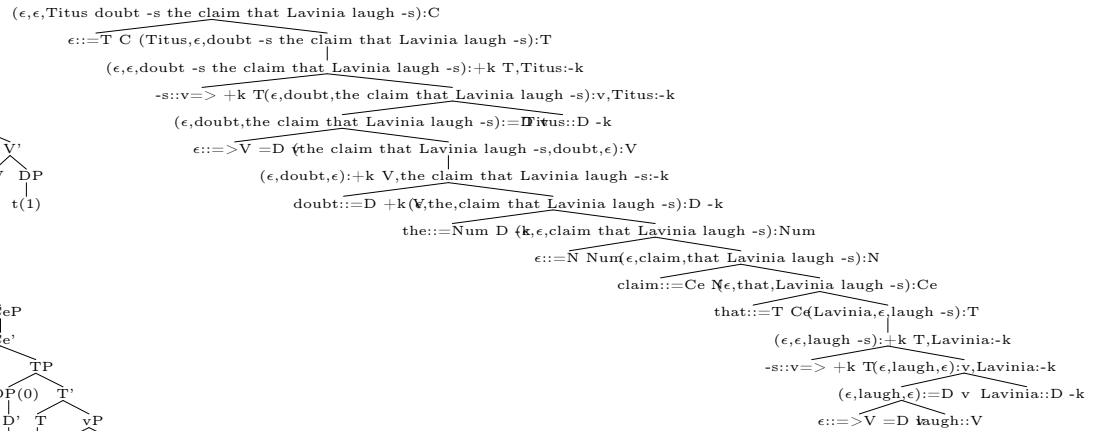
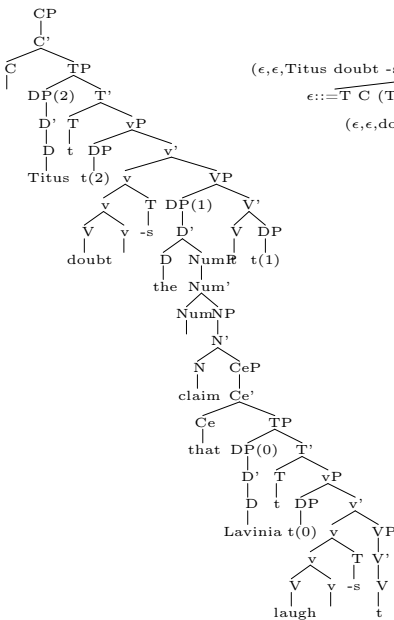
Semantically, the picture corresponds to the derivation as desired:



We can also add nouns that select clausal complements:

claim::=Ce N proposition::=Ce N

With these lexical entries we get trees like this:



12.6.2 TP-selecting raising verbs

The selection relation corresponds to the semantic relation of taking an argument. In some sentences with more than one verb, we find that not all the verbs take the same number of arguments. We notice for example that

auxiliaries select VPs but do not take their own subjects or objects. A more interesting situation arises with the so-called “raising” verbs, which select clausal complements but do not take their own subjects or objects. In this case, since the main clause tense must license case, a lower subject can move to the higher clause.

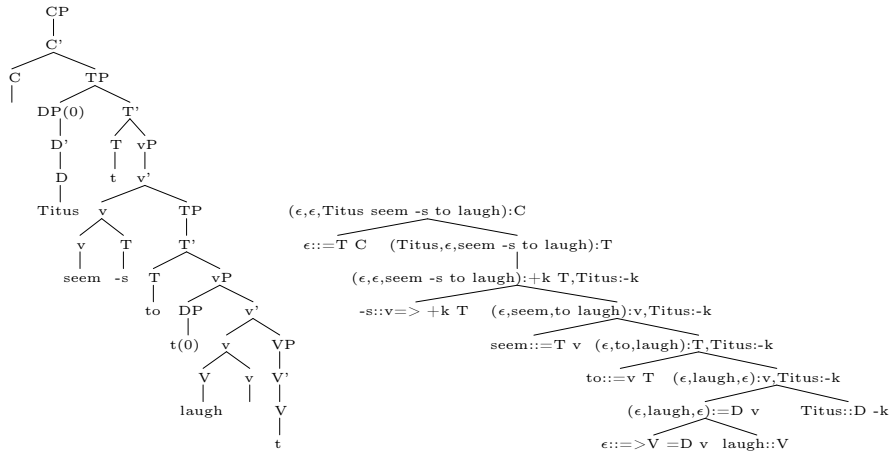
A simple version of this idea is implemented by the following lexical item for the raising verb *seem*

$$\text{seem} ::= T \ v$$

and by the following lexical items for the infinitival *to*:

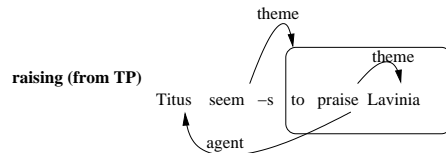
$$\text{to} ::= v \ T \quad \text{to} ::= \text{Have} \ T \quad \text{to} ::= \text{Be} \ T$$

With these lexical entries, we get derivations like this:

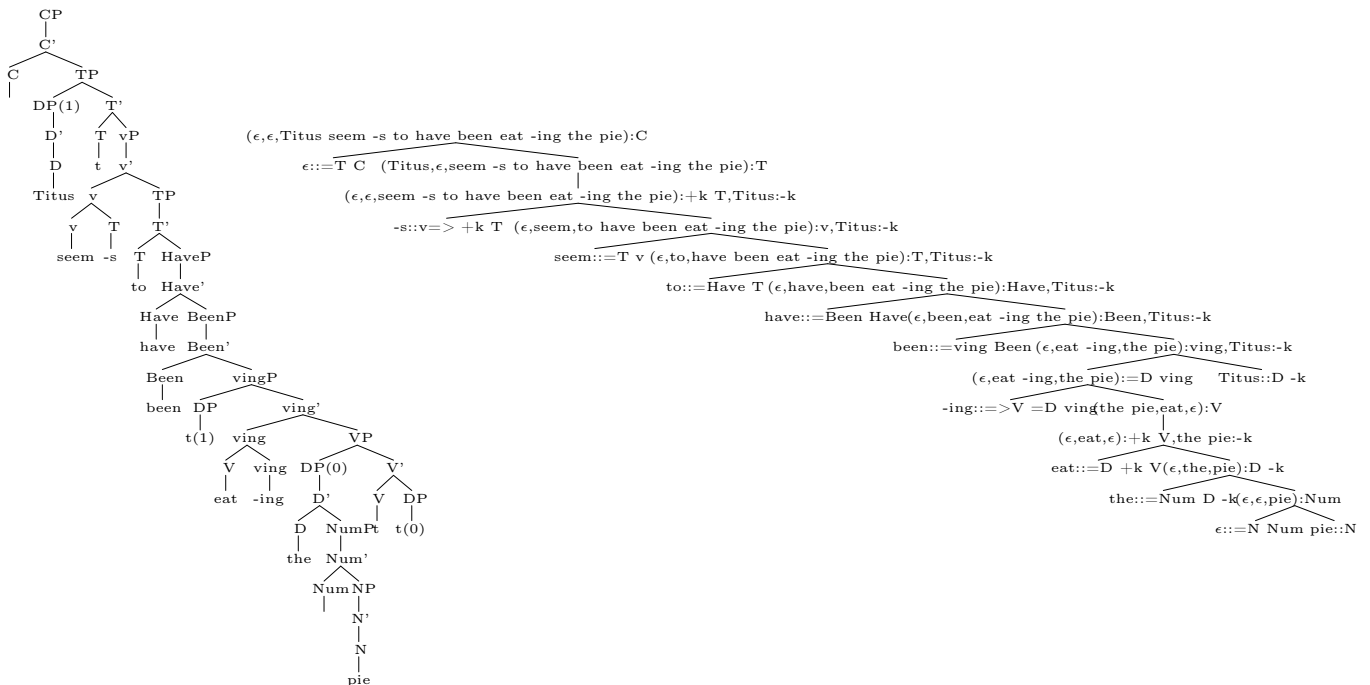


Notice that the subject of *laugh* cannot get case in the infinitival clause, so it moves to the higher clause. In this kind of construction, the main clause subject is not selected by the main clause verb!

Semantically, the picture corresponds to the derivation as desired:



Notice that the infinitival *to* can occur with *have*, *be* or a main verb, but not with a modal:

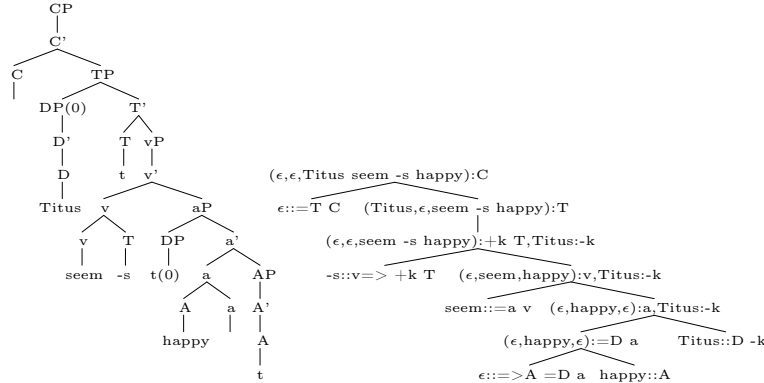


12.6.3 AP-selecting raising verbs

A similar pattern of semantic relations occurs in constructions like this:

Titus seems happy

In this example, Titus is not the ‘agent’ of seeming, but rather the ‘experiencer’ of the happiness, so again it is natural to assume that *Titus* is the subject of *happy*, raising to the main clause for case. We can assume that adjective phrase structure is similar to verb phrase structure, with the possibility of subjects and complements, to get constructions like this:



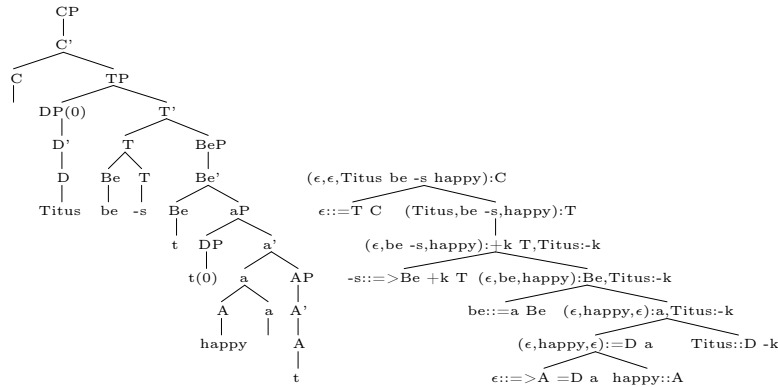
We obtain this derivation with these lexical items:

$\epsilon::=>A =D a.$ $black::A$ $white::A$
 $happy::A$ $unhappy::A$
 $seem::=a v$

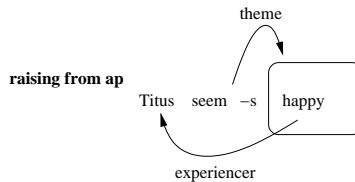
The verb *be* needs a similar lexical entry

$be::=a Be$

to allow for structures like this:



Semantically, the picture corresponds to the derivation as desired:



12.6.4 AP small clause selecting verbs, raising to object

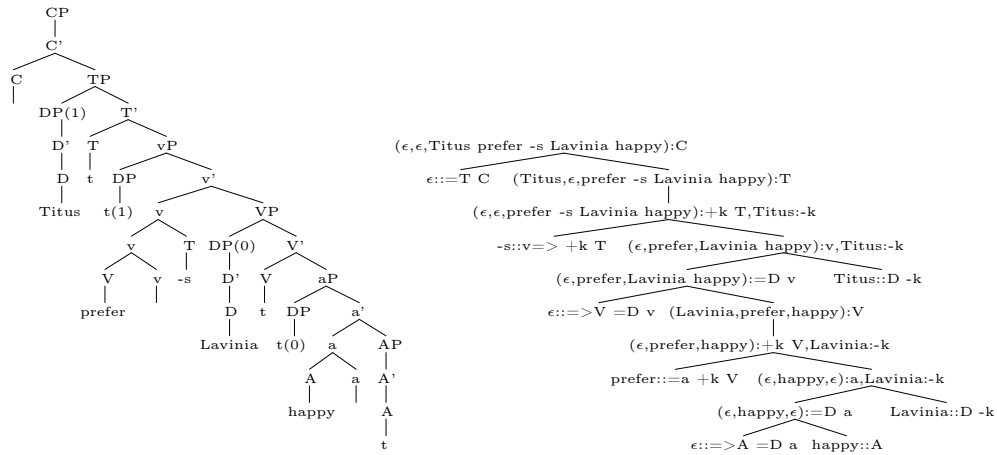
We get some confirmation for the analyses above from so-called ‘small clause’ constructions like:

Titus considers Lavinia happy
 He prefers his coffee black
 He prefers his shirts white

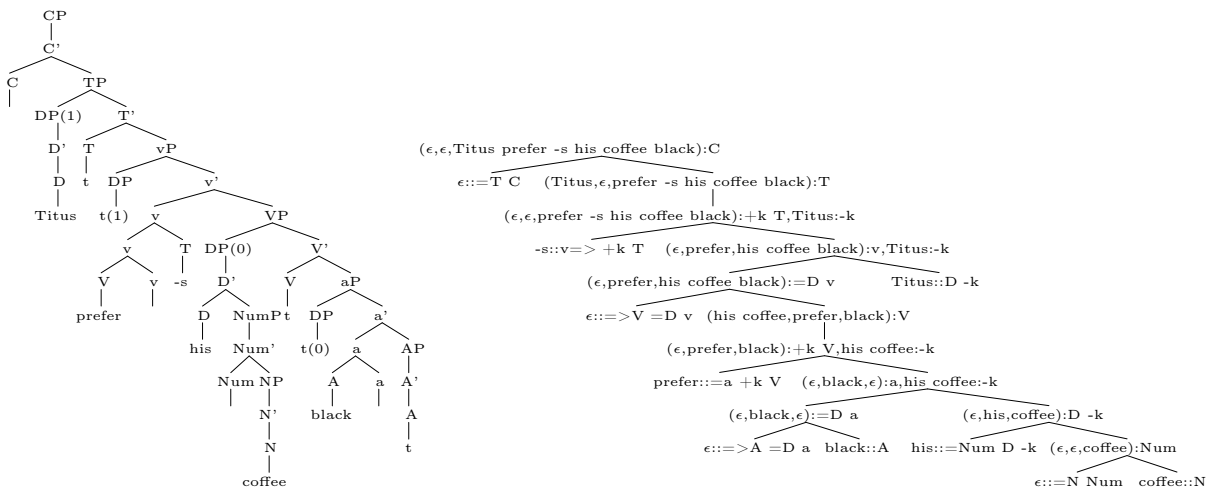
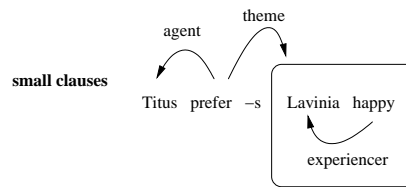
The trick is to allow for the embedded object to get case. One hypothesis is that this object gets case from the governing verb. A simple version of this idea is implemented by the following lexical items:

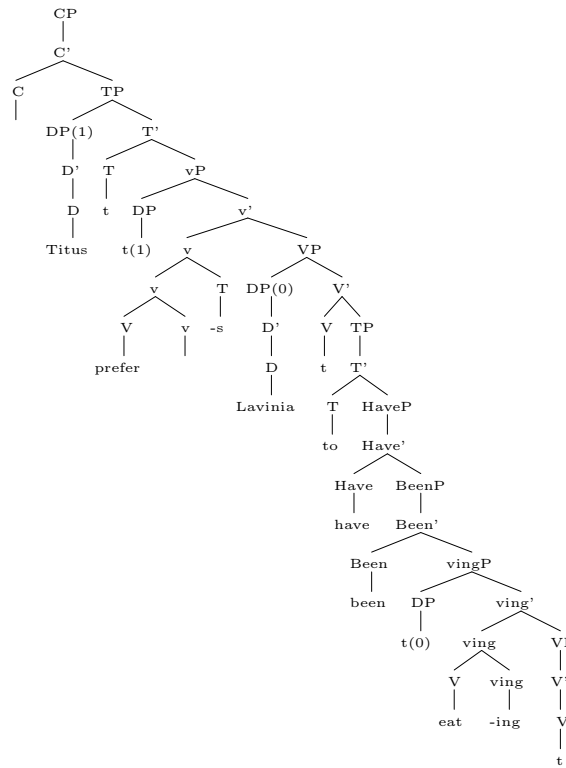
prefer::=a +k V prefer::=T +k V
 consider::=a +k V consider::=T +k V

With these lexical items, we get derivations like this:



Semantically, the picture corresponds to the derivation as desired:





12.6.5 PP-selecting verbs, adjectives and nouns

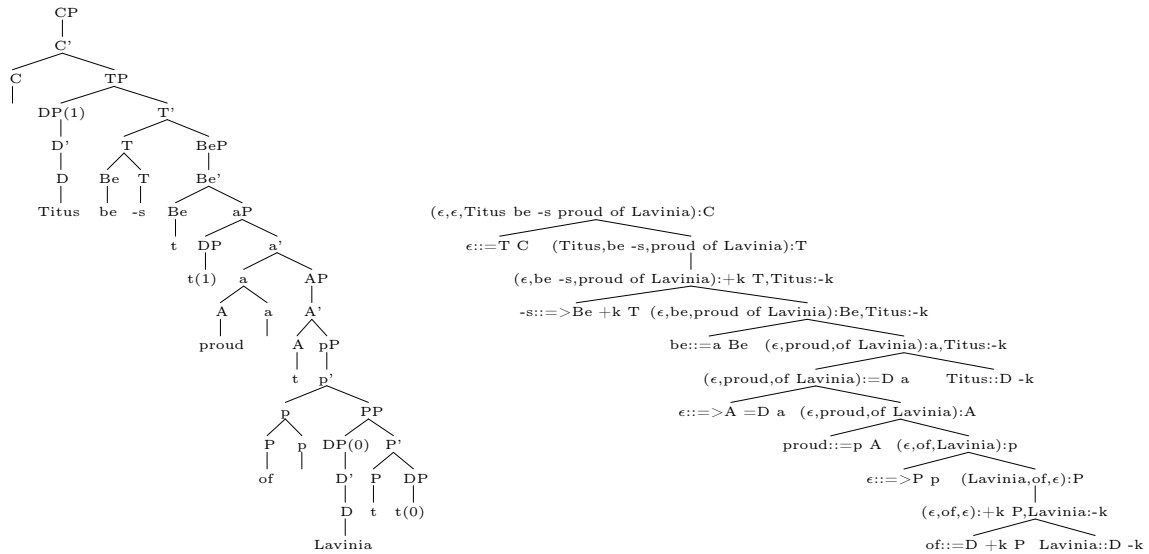
We have seen adjective phrases with subjects, so we should at least take a quick look at adjective phrases with complements. We first consider examples like this:

Titus is proud of Lavinia
 Titus is proud about it

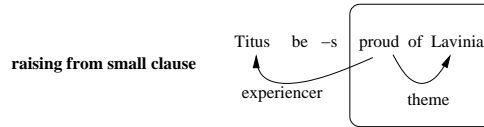
We adopt lexical items which make prepositional items similar to verb phrases, with a “little” p and a “big” P:

proud::=p A proud::A proud::=T a
 ε::=>P p
 of::=D +k P about::=D +k P

With these lexical items we get derivations like this:



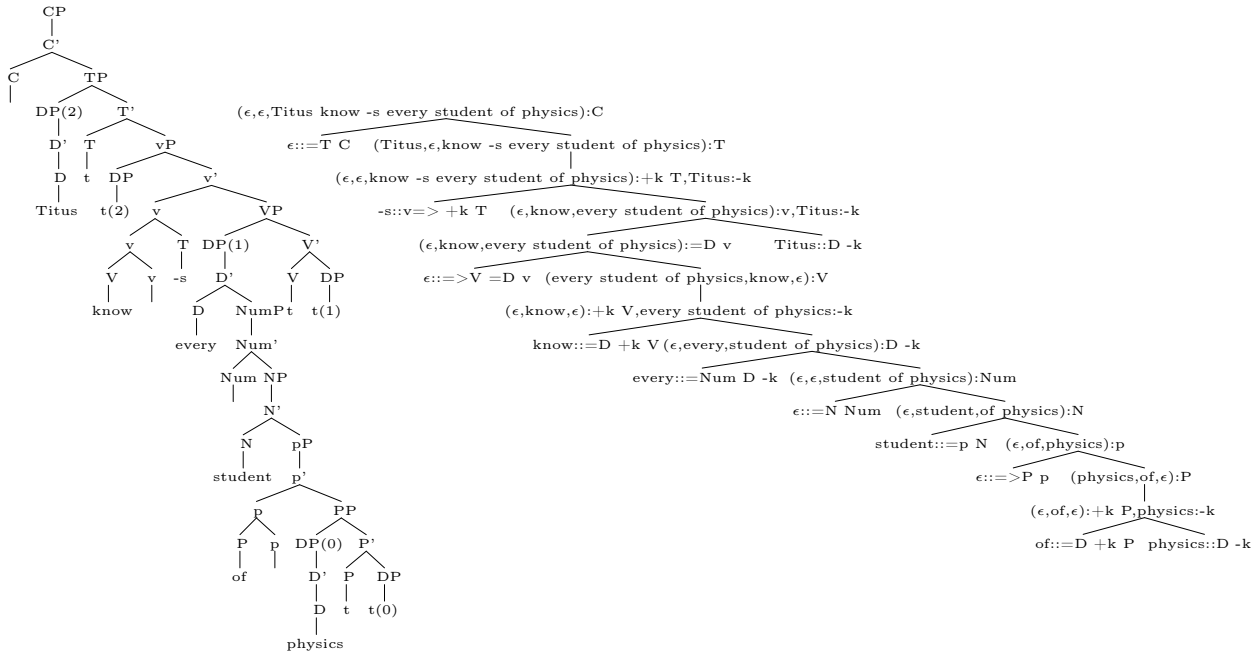
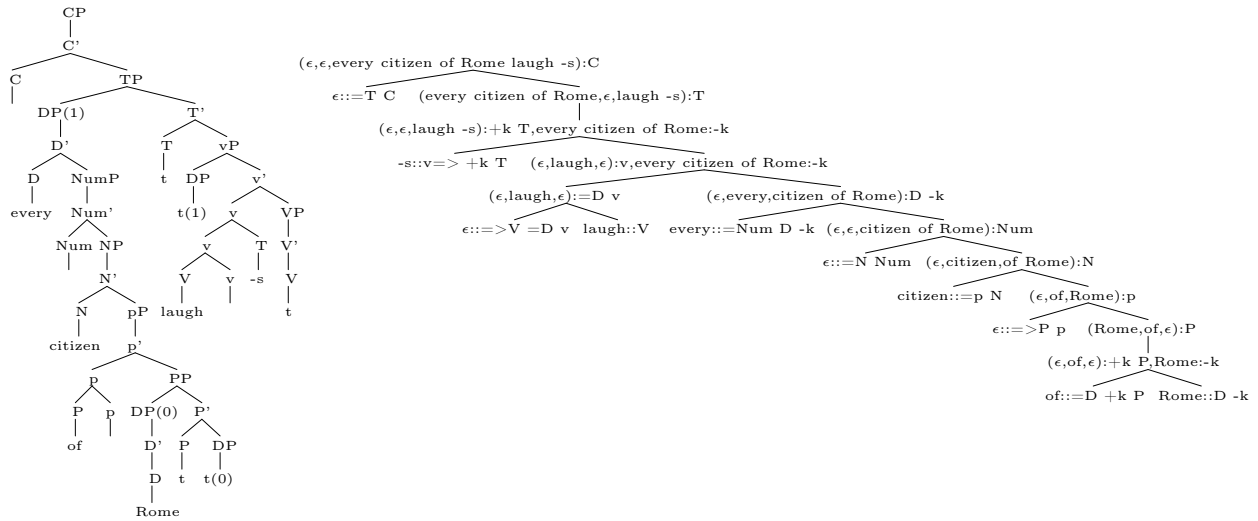
Semantically, the picture corresponds to the derivation as desired:



Similarly, we allow certain nouns to have PP complements, when they specify the object of an action or some other similarly constitutive relation:

student::=p N student::N physics::D -k
 citizen::=p N citizen::N Rome::D -k

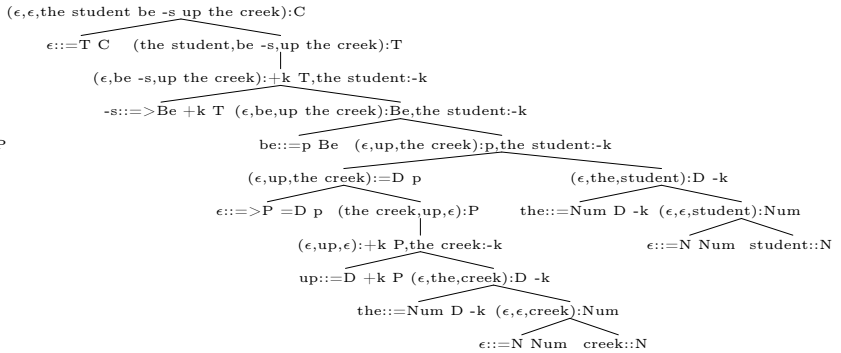
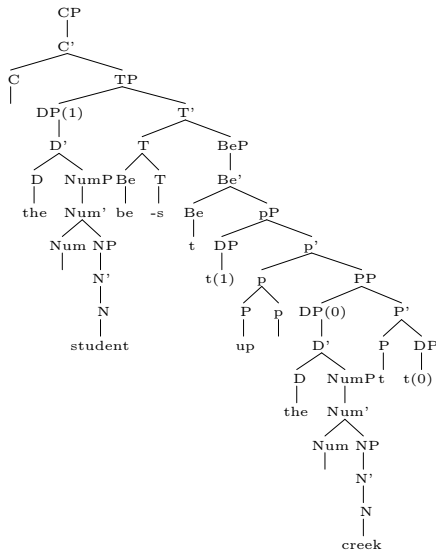
to get constructions like this:



If we add lexical items like the following:

be::=p Be seem::=p v
 $\epsilon::=>P =D p$ up::=D +k P
 creek::N

then we get derivations like this:



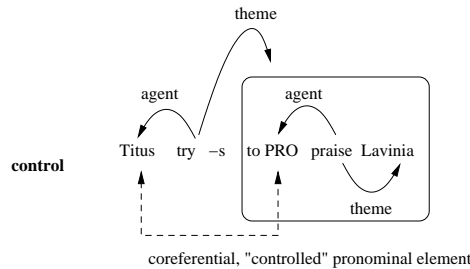
12.6.6 Control verbs

There is another pattern of semantic relations that is actually more common than the raising verb pattern: namely, when a main clause has a verb selecting the main subject, and the embedded clause has no pronounced subject, with the embedded subject understood to be the same as the main clause subject:

Titus wants to eat
Titus tries to eat

One proposal for these constructions is that the embedded subjects in these sentences is an empty (i.e. unpronounced) pronoun which must be “controlled” by the subject in the sense of being coreferential. (For historical reasons, these verbs are sometimes also called “equi verbs.”)

The idea is that we have a semantic pattern here like this:

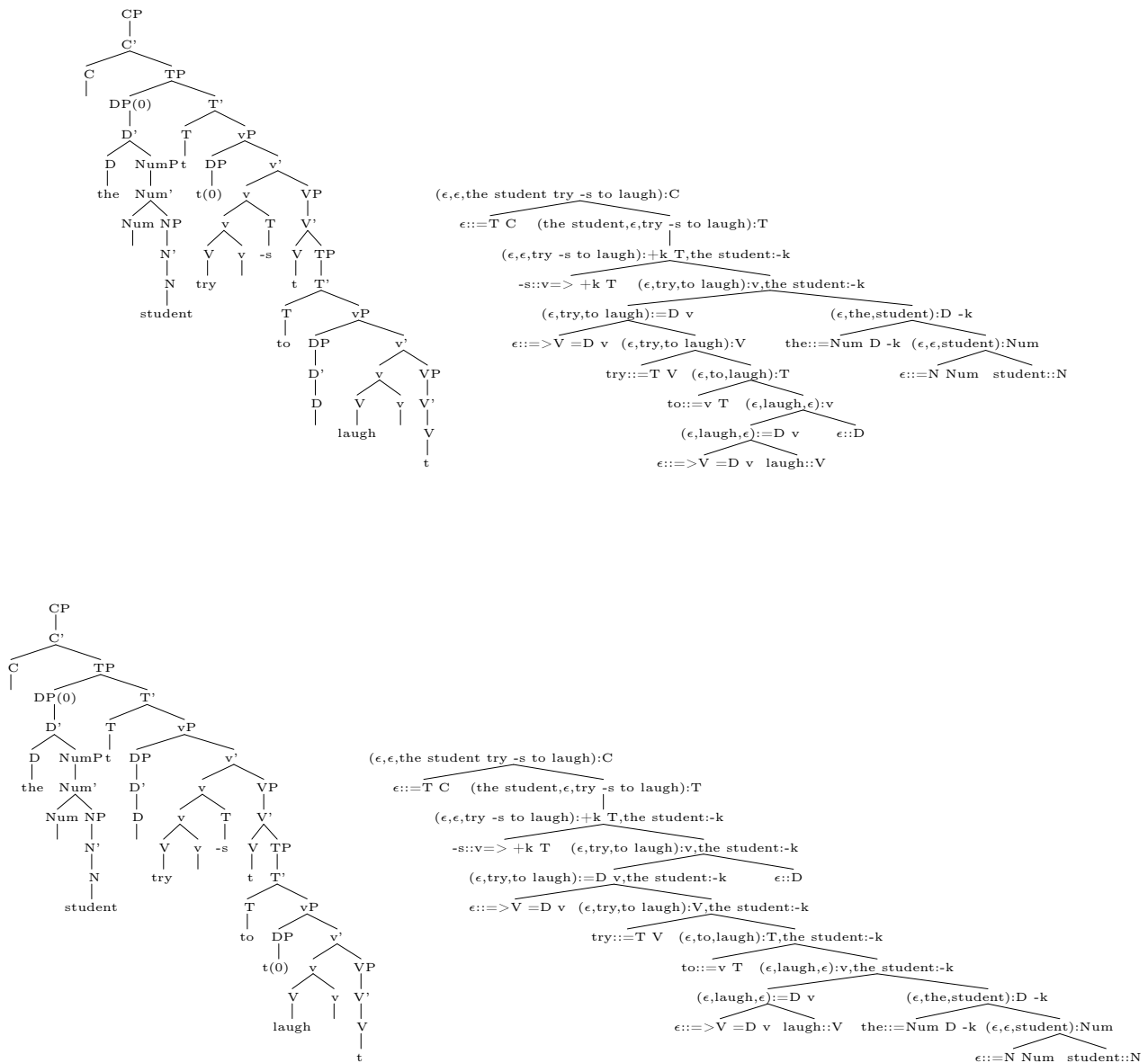


We almost succeed in getting a simple version of this proposal with just the following lexical items:

try::=T V want::=T V want::=T +k V
ε::D

Notice that the features of *try* are rather like a control verb’s features, except that it does not assign case to the embedded object. Since the embedded object cannot get case from the infinitival either, we need to use the empty determiner provided here because this lexical item does not need case.

The problem with this simple proposal is that the empty D is allowed to appear in either of two positions. The first of the following trees is the one we want, but the lexical items allow the second one too:



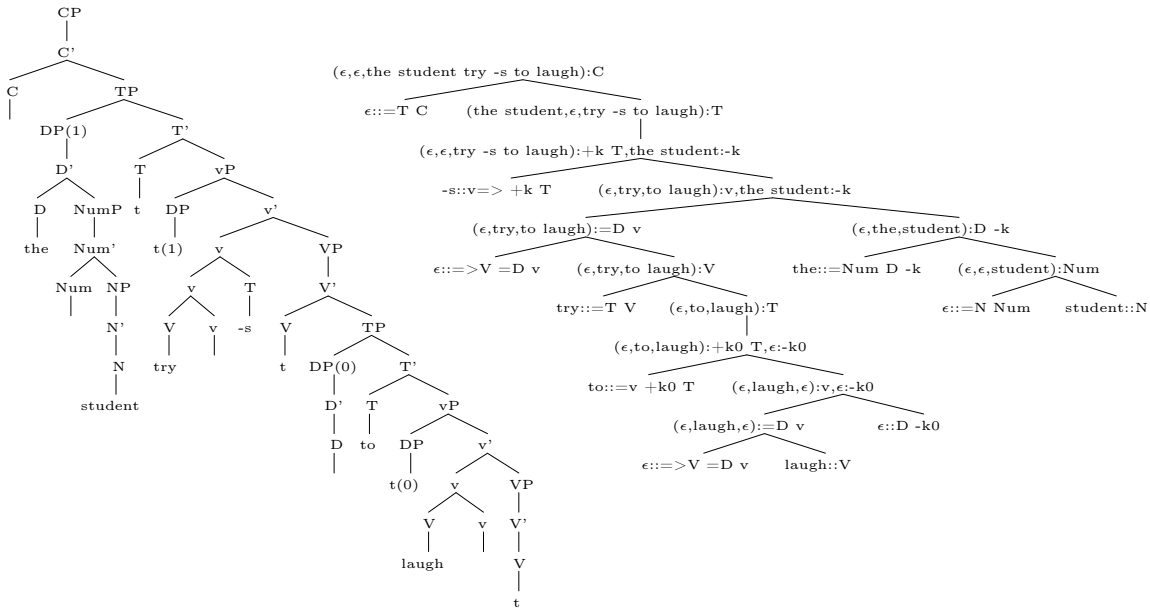
This second derivation is kind of wierd – it does not correspond to the semantic relations we wanted. How can we rule it out?

One idea is that this empty pronoun (sometimes called PRO) actually requires some kind of feature checking relation with the infinitive tense. Sometimes the relevant feature is called “null case” [4, 41, 16]. (In fact, the proper account of control constructions is still controversial – cf., for example, Hornstein, 1999.)

A simple version of this proposal is to use a new feature k0 for “null case,” in lexical items like these:

ε:: D -k0
 to::=v +k0 T to::=Have +k0 T to::=Be +k0 T

With these we derive just one analysis for *the student try -s to laugh*:



Notice how this corresponds to the semantic relations diagrammed on the previous page.

12.6.7 Modifiers as adjuncts

We allow PP complements of N, but traditional transformational grammar also allows PPs to adjoin on the right of an NP to yield expressions like

student [from Paris]
 student [from Paris] [in the classroom]
 student [from Paris] [in the classroom] [by the blackboard].

Adjective phrases can also modify a NP, typically adjoining to the left in English:

[Norwegian] student
 [young] [Norwegian] student
 [very enthusiastic] [young] [Norwegian] student.

And of course both can occur:

[very enthusiastic] [young] [Norwegian] student [from Paris] [in the classroom] [by the blackboard].

Unlike selection, this process seems optional in almost all cases, and there does not seem to be any fixed bounds on the number of possible modifiers, so it is widely (but by no means universally) thought that the mechanisms and structures of modifier attachment are fundamentally unlike complement attachment. Our mechanisms for adjunction allow that.

To indicate that APs can left adjoin to NP, and PPs and CPs (relative clauses) can right adjoin to NP, we use the notation:

leftAdjoiner[N]=[a]
 rightAdjoiner[N]=[p,Cwh]

Similarly for verb modifiers, as in *Titus loudly laughs* or *Titus laughs loudly* or *Titus laughs in the castle*:

leftAdjoiner[v]=[Adv]
 rightAdjoiner[v]=[Adv,P]

For adjective modifiers like *very* or *extremely*, in the category deg(ree), as in *Titus is very happy*:

leftAdjoiner[a]=[Deg]

Adverbs can modify prepositions, as in *Titus is completely up the creek*:

$$\text{leftAdjoiner}[P]=[\text{Adv}]$$

The category num can be modified by qu(antity) expressions like many,few,little,1,2,3,... as in *the 1 place to go is the cemetery, the little water in the canteen was not enough, the many activities include hiking and swimming.*

$$\text{leftAdjoiner}[\text{Num}]=[\text{Qu}]$$

Determiners can be modified on the left by *only, even* which we give the category emph(atic), and on the right by CPs (appositive relative clauses as in *Titus, who is the king, laughs*). We would also like to DP adjuncts of DP. These are the appositives, as in *Titus, the king, laughs*. We can get these things with

$$\begin{aligned} \text{leftAdjoiner}[D]&=[\text{Emph}] \\ \text{rightAdjoiner}[D]&=[\text{Cwh},D] \end{aligned}$$

12.7 Some additional extensions

Although our grammar has many mechanisms, some linguists think that it does not have enough. Here we describe some additional ideas that could easily be added, without changing the basic properties of the grammars.

12.7.1 Left-merge and right-merge (parameterized?)

12.7.2 ϕ feature and agreement marking

Cf., e.g. Onambele'12 on agreement in the Bantu language Ewondo.

12.7.3 Multiple wh-movement

Gärtner and Michaelis [10]

12.7.4 Late adjunction

Frey and Gärtner'02 mention that there are various arguments for 'late adjunction' in the Chomskian tradition, and this idea has been developed by Gaertner and Michaelis'03.

Exercises

1. **Topicalization.** The grammar of this section allows wh-movement to form questions, but it does not allow topicalization, which we see in examples like this:

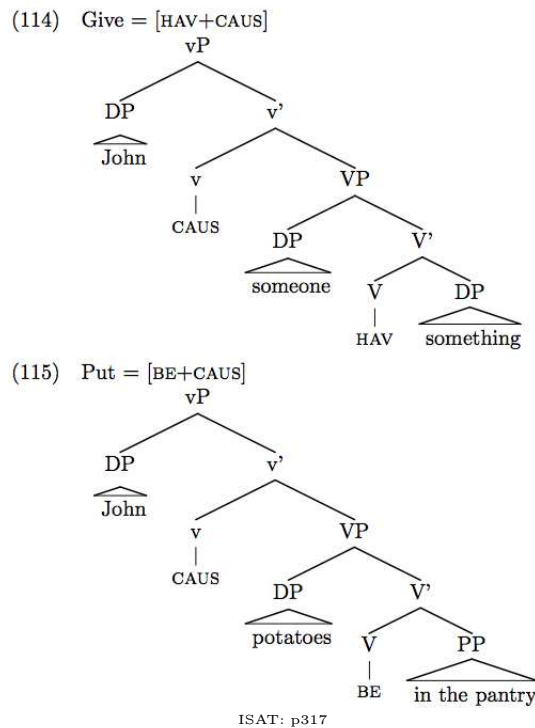
Lavinia, Titus praise -s
The king, Titus want -s to praise

One idea is that the lexicon includes in addition to DPs like *Lavinia*, a -topic version of this DP, which moves to a +topic specifier of CP. Since any DP can be -topic, though, it might be better to simply let the topic position be +D, as we did for movement to subject position in §12.1.1. Can you extend the grammar to get these topicalized constructions in this way? If so, present (i) a complete derivation and (ii) a brief linguistic assessment of this approach. If not, (i) explain why -D does not work and instead present a complete derivation using -topic, and (ii) write a brief linguistic assessment of this approach.

2. **Put and give:** We did not consider verbs like *put* which require two ‘internal’ arguments:

the cook put -s the pie in the oven
* the cook put -s the pie
* the cook put -s in the oven
the cook gave me some pies
the cook gave some pies to me
? the cook gave some pies

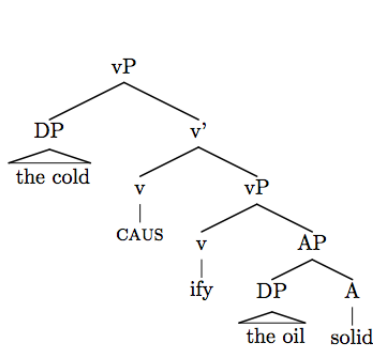
One idea about these constructions is mentioned in ISAT:



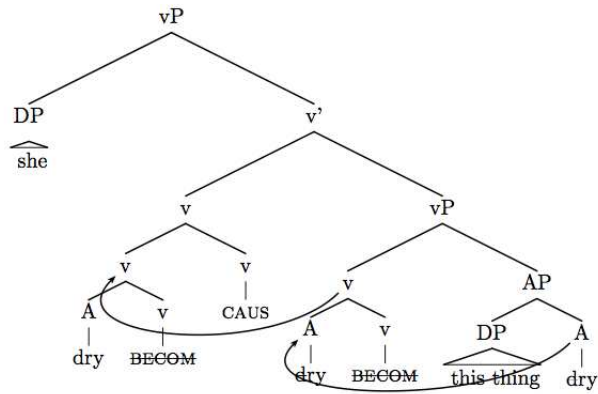
Notice that HAV can move up to CAUS by head movement, producing a [HAV CAUSE] complex which could be pronounced *put*. And BE can move up to CAUS to produce a [CAUS BE] complex which could be pronounced *give*. Based on this idea, present complete MG derivations that are as close to these proposals as possible, one derivation for each of these examples (here the places where the HAV, BE originated are marked with ___):

the cook [HAV CAUS] -0 me ___ some pies.
the cook [BE CAUS] -s the pie ___ in the oven

3. **Morphology.** ISAT extends the previous idea to morphology too, as we see in these examples:



ISAT: p381



ISAT: p382

No head movement is shown on the left, but clearly we could move *solid* to *ify* and then move *solid -ify* to produce a [*solid ify CAUS*] complex, in a way similar to the formation of [*dry BECOM CAUS*] on the right. Present complete MG derivations that are as close to these proposals as possible, one derivation for each of these examples:

the cold [*solid -ify CAUS*] the oil
 she [*dry BECOM CAUS*] this thing

4. **Copy raising.** Some English dialects (like mine) allow raising verbs to appear with certain finite clauses, as in these examples [1, 26]:

John seems like he wants to work
 Emintrude looks like the cat has got her tongue
 Mary appears as if she has seen a ghost

Extend the grammar to get at least the first of these sentences, presenting a complete derivation, and then write a brief linguistic assessment of your approach.

5. **ECM constructions.** Consider sentences like this:

Mary believes John to be in the room

It is as if *John* gets its case as the object of *believes*, but originates in the embedded infinitival. Write lexical items which will allow this kind of analysis, and show the completed derivation.

Optional extra step: Collins [5, pp.96-104] uses this example in his argument for asymmetric feature checking and particularly for Chomsky's [3] story about \pm interpretable features. Assess these arguments.

6. **Head vs. phrasal movement.** Dave Schueler (p.c.) points out that our formalization of head movement in the configuration of selection is similar to a suggestion made by Pesetsky & Torrego [24]:

(5a) What did Mary buy?

In (5a), [a feature] μ T on C is attracting a feature of its own complement – a constituent with which C has just merged. If the entire complement of C were to be copied as Spec,CP, C would, in effect, be merging with the same constituent twice. We suggest that it is precisely in these circumstances that the head of the complement, rather than the complement itself, is copied. In the present context, this suggestion is speculative, but it is in fact the flip side of a more familiar generalization: the Head Movement Constraint of Travis (1984). Travis's condition states that head movement is always movement from a complement to the nearest head. Our condition dictates that movement from a complement to the nearest head is always realized as head movement. We may call the two together the "Head Movement Generalization":

(13) Head Movement Generalization

Suppose a head H attracts a feature of XP as part of a movement operation.

- (i) If XP is the complement of H, copy the head of XP into the local domain of H.
- (ii) Otherwise, copy XP into the local domain of H.

Describe some cases where the Head Movement Generalization would not be followed in an MG. (If you paid attention in the section above, you have not far to look for some first examples.) Do actual constructions in human languages that really look like they call for such a thing?

References

- [1] ASUDEH, A., AND TOIVONEN, I. Perception and uniqueness: Evidence from English and Swedish copy raising. University of Canterbury, New Zealand, 2004.
- [2] BOECKX, C. *Understanding Minimalist Syntax: Lessons from Locality in Long-distance Dependencies*. Blackwell, Oxford, 2008.
- [3] CHOMSKY, N. *The Minimalist Program*. MIT Press, Cambridge, Massachusetts, 1995.
- [4] CHOMSKY, N., AND LASNIK, H. Principles and parameters theory. In *Syntax: An international handbook of contemporary research*, J. Jacobs, A. von Stechow, W. Sternfeld, and T. Vennemann, Eds. de Gruyter, Berlin, 1993. Reprinted in Noam Chomsky, *The Minimalist Program*. MIT Press, 1995.
- [5] COLLINS, C. *Local Economy*. MIT Press, Cambridge, Massachusetts, 1997.
- [6] FREY, W., AND GÄRTNER, H.-M. On the treatment of scrambling and adjunction in minimalist grammars. In *Proceedings, Formal Grammar'02* (Trento, 2002).
- [7] GÄRTNER, H.-M., AND MICHAELIS, J. A note on the complexity of constraint interaction. In *Logical Aspects of Computational Linguistics, LACL'05*, Lecture Notes in Artificial Intelligence LNCS-3492. Springer, NY, 2005, pp. 114–130.
- [8] GÄRTNER, H.-M., AND MICHAELIS, J. Some remarks on locality conditions and minimalist grammars. Symposium on Interfaces and Recursion, Centre for General Linguistics, Typology and Universals Research (ZAS), Berlin, 2005.
- [9] GÄRTNER, H.-M., AND MICHAELIS, J. A note on countercyclicity and minimalist grammars. In *Proceedings of the 8th Conference on Formal Grammar*, G. Penn, Ed. CSLI Publications, Stanford, CA, 2008.
- [10] GÄRTNER, H.-M., AND MICHAELIS, J. On the treatment of multiple wh-interrogatives in minimalist grammars. Zentrum für Allgemeine Sprachwissenschaft, Universität Bielefeld, 2010.
- [11] HOFMEISTER, P., CASASANTO, L. S., AND SAG, I. A. How do individual cognitive differences relate to acceptability judgments?: A reply to sprouse, wagers, and phillips. *Language* 88, 2 (2012), 390–400.
- [12] HORNSTEIN, N. Movement and control. *Linguistic Inquiry* 30 (1999), 69–96.
- [13] HUNTER, T. Insertion minimalist grammars: Eliminating redundancies between merge and move. In *The Mathematics of Language*, M. Kanazawa, A. Kornai, M. Kracht, and H. Seki, Eds., vol. 6878 of *Lecture Notes in Computer Science*. Springer, 2011, pp. 90–107.
- [14] KOOPMAN, H., SPORTICHE, D., AND STABLER, E. *An Introduction to Syntactic Analysis and Theory*. Blackwell, Oxford, 2013.
- [15] LECOMTE, A., AND RETORÉ, C. Towards a minimal logic for minimalist grammars. In *Proceedings, Formal Grammar'99* (Utrecht, 1999).
- [16] MARTIN, R. *A Minimalist Theory of PRO and Control*. PhD thesis, University of Connecticut, Storrs, 1996.
- [17] MATUSHANSKY, O. Review of ian roberts, *agreement and head movement: clitics, incorporation, and defective goals*. *Journal of Linguistics* 47, 2 (2011), 538–545.
- [18] MICHAELIS, J. *On Formal Properties of Minimalist Grammars*. PhD thesis, Universität Potsdam, 2001. *Linguistics in Potsdam 13*, Universitätsbibliothek, Potsdam, Germany.
- [19] MOORTGAT, M. Symmetric categorial grammar. *Journal of Philosophical Logic* 8, 6 (2009), 681–710.
- [20] MORRILL, G., AND VALENTÍN, O. Generalized discontinuity. In *Revised Selected Papers of Formal Grammar*, P. de Groote and M.-J. Nederhof, Eds., Lecture Notes in Computer Science LNCS 7395. Springer, Berlin, 2012, pp. 146–161.
- [21] MORRILL, G. V. *Type-logical Grammar: Categorical Logic of Signs*. Kluwer, Dordrecht, 1994.
- [22] O'FLYNN, K. Head movement in minimalist syntax. UCLA Master's thesis, 2011.
- [23] ONAMBELE MANGA, C. L. *Vers une Grammaire Minimaliste de Certains Aspects Syntactiques de la Langue Ewondo*. PhD thesis, Université Paris 8, 2012.

- [24] PESETSKY, D., AND TORREGO, E. T-to-C movement: Causes and consequences. In *Ken Hale: A Life in Language*, M. Kenstowicz, Ed. MIT Press, Cambridge, Massachusetts, 2001.
- [25] ROBERTS, I. G. *Agreement and Head Movement: Clitics, Incorporation, and Defective Goals*. MIT Press, Cambridge, Massachusetts, 2010.
- [26] ROGERS, A. Three kinds of physical perception verbs. In *Proceedings of the 7th Regional Meeting of the Chicago Linguistic Society (1971)*, pp. 206–222.
- [27] ROSS, J. R. *Constraints on Variables in Syntax*. PhD thesis, Massachusetts Institute of Technology, 1967.
- [28] SAG, I., HOFMEISTER, P., AND SNIDER, N. Processing complexity in subadjacency violations: the complex noun phrase constraint. In *Proceedings of 43rd Regional Meeting of the Chicago Linguistics Society (2007)*.
- [29] SALVATI, S. Minimalist grammars in the light of logic. In *Logic and Grammar: Essays Dedicated to Alain Lecomte on the Occasion of His 60th Birthday*, S. Pogodalla, M. Quatrini, and C. Retoré, Eds., no. 6700 in Lecture Notes in Computer Science. Springer, 2011.
- [30] SPORTICHE, D. Bounding nodes in French. *The Linguistic Review* 1 (1981), 219–246.
- [31] SPROUSE, J., WAGERS, M., AND PHILLIPS, C. Working memory capacity and island effects: A reminder of the issues and the facts. *Language* 88, 2 (2012), 401–407.
- [32] STABLER, E. P. Derivational minimalism. In *Logical Aspects of Computational Linguistics*, C. Retoré, Ed. Springer-Verlag (Lecture Notes in Computer Science 1328), NY, 1997, pp. 68–95.
- [33] STABLER, E. P. Recognizing head movement. In *Logical Aspects of Computational Linguistics*, P. de Groote, G. Morrill, and C. Retoré, Eds., Lecture Notes in Artificial Intelligence, No. 2099. Springer, NY, 2001, pp. 254–260.
- [34] STABLER, E. P. Comparing 3 perspectives on head movement. In *From Head Movement and Syntactic Theory, UCLA/Potsdam Working Papers in Linguistics*, A. Mahajan, Ed. UCLA, 2003, pp. 178–198. Available at <http://www.humnet.ucla.edu/humnet/linguistics/people/stabler>.
- [35] STABLER, E. P. Computational perspectives on minimalism. In *Oxford Handbook of Linguistic Minimalism*, C. Boeckx, Ed. Oxford University Press, Oxford, 2011, pp. 617–641.
- [36] STEEDMAN, M., AND BALDRIDGE, J. Combinatory categorial grammar. In *Non-Transformational Syntax: Formal and Explicit Models of Grammar*, R. Borsley and K. Börjars, Eds. Wiley-Blackwell, Boston, 2011.
- [37] STEEDMAN, M. J. *The Syntactic Process*. MIT Press, Cambridge, Massachusetts, 2000.
- [38] TRAVIS, L. *Parameters and effects of word order variation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1984.
- [39] VERMAAT, W. The minimalist move operation in a deductive perspective. *Research on Language and Computation* 2, 1 (2004), 69–85.
- [40] VERMAAT, W. *The Logic of Variation: A cross-linguistic account of wh-question formation*. PhD thesis, Utrecht University, 2006.
- [41] WATANABE, A. *Case Absorption and Wh Agreement*. Kluwer, Dordrecht, 1993.

Chapter 13 MG simplified

Let's see whether we can get approximately the same coverage as the grammar of the previous chapter with a grammar of similar complexity but with mechanisms that are much simpler, using this idea:

- There is just one operation for building structure, still pronounced 'merge' and written \bullet but slightly different from anything we have considered earlier.

Once this one idea is properly introduced, we have these results:

- All features are persistent, there if you need them more than once.
- Feature ordering in lexical items determines cartography and blocks improper movement.
- Every category is a phase.
- Head movement effects can be obtained (or at least some of them).
- Agreement effects can be obtained (or at least some of them).

The changes sound huge, but really they are not. In effect, this chapter brings insights of §10 to the fore to find a simpler perspective, without changing anything more in mainstream grammar than necessary. The conclusion reviews a number of outstanding problems, some of which look serious, but we can speculate optimistically that perhaps we are getting closer to the truth than has been possible before.

13.1 One mode of combination

We use the single mode of combination suggested by Chomsky, subsuming merge and move by relaxing the requirement that each tree be independently derived. But we can keep the grammar formalism clear and precise by moving some of the complexity into a syntactic valuation function $\langle\langle\cdot\rangle\rangle$ and into the definition of a checking relation \times . We can then show that it is (possibly more succinct, but) weakly equivalent to our earlier one. So the proposal is that all syntactic complexes are built by the operation,

$$\bullet(t_1, t_2) = \{t_1, t_2\} \quad \text{if } \langle\langle t_1 \rangle\rangle \times \langle\langle t_2 \rangle\rangle \text{ is defined.}$$

As will become clear in a moment, $\langle\langle t_1 \rangle\rangle \times \langle\langle t_2 \rangle\rangle$ is not defined when $t_1 = t_2$, so the value of $\bullet(t_1, t_2)$ is always a set with two elements.¹ The reason for the syntactic denotation function $\langle\langle\cdot\rangle\rangle$ is that the question of whether $\bullet(t_1, t_2)$ is defined typically depends on much less than the whole of the trees t_1, t_2 . The function $\langle\langle\cdot\rangle\rangle$ specifies just those properties of syntactic objects which could be relevant to whether \bullet is defined. It is similar to a 'labeling' function [1]. The relation \times , pronounced 'checks', is not symmetric, and so the merge relation \bullet is not symmetric either. In a set $\{t_1, t_2\} = \{t_2, t_1\}$, the values $\langle\langle t_1 \rangle\rangle$ and $\langle\langle t_2 \rangle\rangle$ make immediately clear whether either checks the other, and hence also whether the set $\{t_1, t_2\}$ is built by the grammar.

The pronounced value of any syntactic structure t is given by a phonetic denotation function $\langle\langle t \rangle\rangle$. And the interpreted value of any syntactic structure is given by a semantic denotation function $\llbracket t \rrbracket$.

Obviously, the aim is not to shift complexity from one part of the grammar to another, but rather to keep the whole grammar as simple as possible by factoring relevantly different aspects and defining each with exactly those properties that are essential, no more and no less.

13.2 Persistent features

The idea introduced for EPP in §12.1.1 on page 135 seems widely applicable, and so we will keep it in the simplified system, but the implementation can be adjusted as follows.

XXX

¹This is not a crucial property though. We consider an argument for relaxing it in §13.7 below.

13.3 Feature ordering, cartography, and improper movement

13.4 Every category is a phase

13.5 Head movement subsumed

There are two ways to get the effects of head movement using only phrasal movement. One is to move the arguments of the head out of its projection, so that the phrasal projection of the head contains only the head. One problem with this strategy is to explain why, if the arguments are moving, they often seem to stay in their original order. Another problem is that the constituencies do not seem right. Another way to get the effects of head movement is to put less in the phrasal projection. I have not seen this explicitly proposed in the literature as a replacement for head movement, but it is implicit in the kind of morphology that we see in some “nanosyntax” proposals (syntax defined over sub-lexical elements) and in exercises 2,3 of the previous chapter, on page 154. It is the idea pursued here. In effect, we use movement as a diagnostic of phrase-hood, and ‘rolling up’ derivations to get verb clusters of various kinds as suggested by Koopman, Kayne and others.

13.6 Agreement subsumed

13.7 Adjunction again

We can have a DP adjunct of DP, so should the relation • allow some reflexive instances?

13.8 An even simpler English

To illustrate how everything works together, let’s attempt to present a grammar of English that is similar to the simple English of §12.6, but now using only ★.

⇒ more coming ⇐

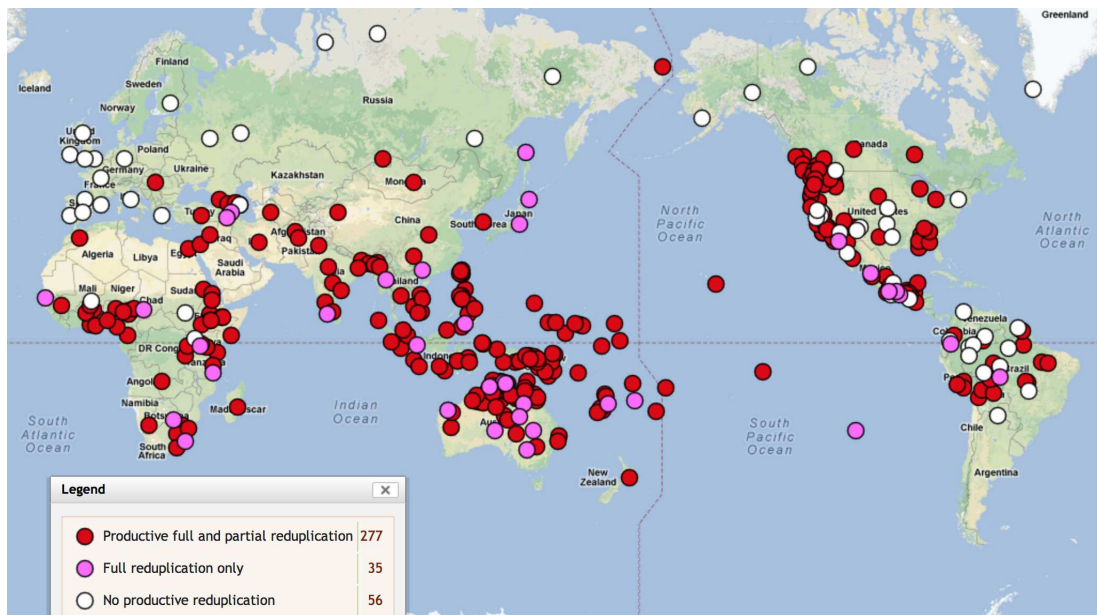
13.9 Implementation

References

- [1] COLLINS, C. Eliminating labels. In *Derivation and Explanation*, S. D. Epstein and D. Seeley, Eds. Blackwell, Oxford, 2002.
- [2] KAYNE, R. S. *The Antisymmetry of Syntax*. MIT Press, Cambridge, Massachusetts, 1994.
- [3] KAYNE, R. S. Antisymmetry and the lexicon. Manuscript, <http://ling.auf.net/lingbuzz>, 2007.
- [4] KOOPMAN, H., AND SZABOLCSI, A. *Verbal Complexes*. MIT Press, Cambridge, Massachusetts, 2000.
- [5] SEKI, H., MATSUMURA, T., FUJII, M., AND KASAMI, T. On multiple context-free grammars. *Theoretical Computer Science* 88 (1991), 191–229.
- [6] STABLER, E. P. Varieties of crossing dependencies: Structure dependence and mild context sensitivity. *Cognitive Science* 93, 5 (2004), 699–720.

Chapter 14 MG with copying

The World Atlas of Linguistic Structures shows that reduplication of phonetic material within a word is found in a large proportion of the world's languages [30]:

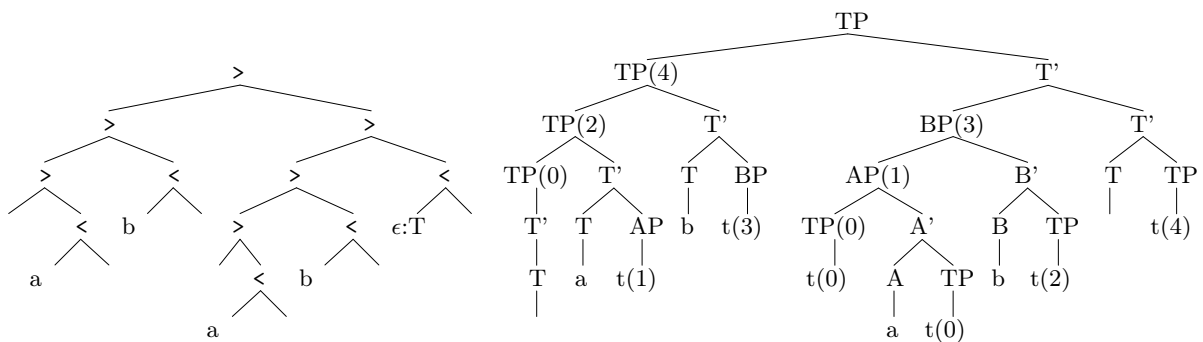


European, Siberian, and Inuit languages appear to be exceptional in having rather little reduplication. In the world's languages, we also find copying of full words and sometimes larger phrases in the syntax of many languages. This chapter briefly considers arguments for various kinds of syntactic reduplication, and then explores how MGs can be extended to allow this.

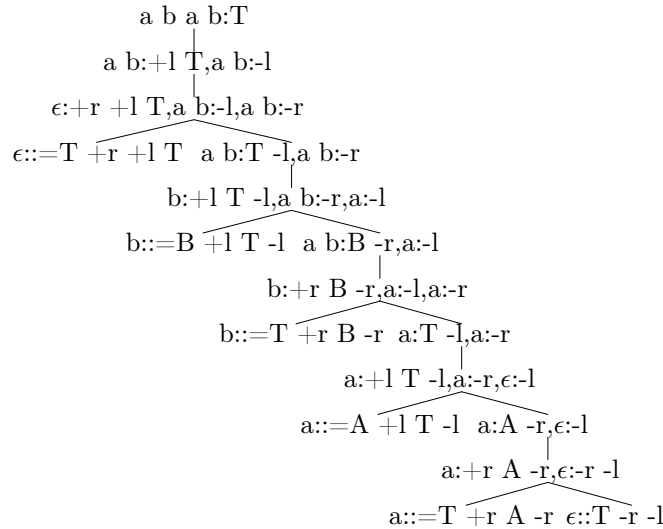
14.1 Earlier analysis, rejected

MGs can define copying, as we saw in this simple MG for $L_{xx} = \{xx \mid x \in \{a, b\}^+\}$ discussed earlier:

$$\begin{array}{ll}
 a ::= A +l T -l & b ::= B +l T -l \\
 a ::= T +r A -r & b ::= T +r B -r \\
 \epsilon ::= T +r +l T & \epsilon ::= T -r -l
 \end{array}$$



Those trees for *abab* are derived by this derivation:

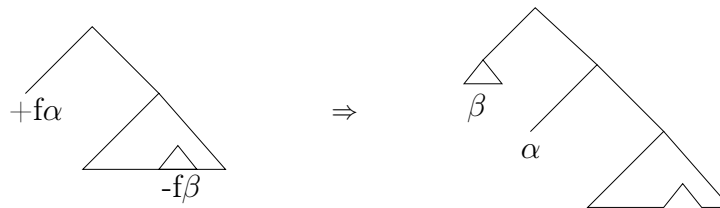


Note that this approach requires that every copiable element have its own category, instead of copying the derivation introduces pairs of identical elements, and every copied element moves. This is not a believable as a model of syntactic reduplication in human languages: it misses the simple copying generalization.

14.2 Copying in syntax

Surprisingly, it is quite common to find reduplication proposed among syntactic mechanisms on grounds completely independent of any apparent copying in pronounced forms. In particular, many linguists think that the movement operation involves some kind of copying.

In the tree-based definition of MGs, the move operation can be drawn like this:

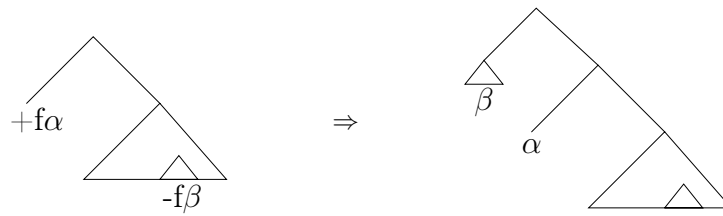


That is, when the the head of a tree has the features $+f\alpha$, and some subtree has the features $-f\beta$, the subtree is taken from its original position (i.e. it is deleted there) and move to specifier position (i.e. it is copied there), cancelling the features. This is like the **trace theory** of movement: the moved constituent is replaced by something that does not have structure and is not pronounced. One unsatisfying fact about this operation, and something that distinguishes it from merge, is that we make a change inside of a structure that is already built.¹ When we ‘flatten’ the move rule, that property goes away – we no longer need to change anything in already-built structures, because we never put the $-f\beta$ subtree into the original structure. But many linguists think that this flattening is the wrong idea,² or at least an importantly different one.

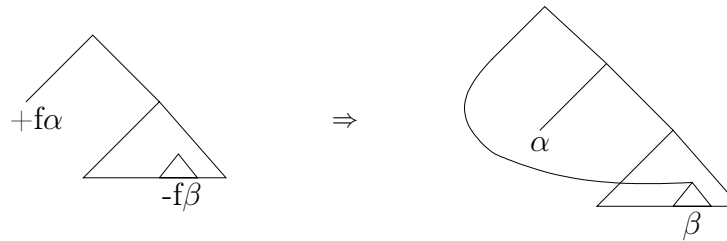
More popular than our MG move operation are two other ideas about movement. The **copy theory** says that when a phrase moves, the original stays where it is but is not pronounced. If we use strike-out to indicate parts of the structure that are not pronounced, this idea about move can be depicted like this:

¹For this reason, Chomsky says that it violates the simplicity consideration he calls ‘no tampering’. If the trace is not simply a deletion site but has an index or something like that, then the trace theory also violates the ‘inclusiveness’ which suggests that everything should come from the lexicon; nothing should be inserted by a syntactic operation.

²In the case of MGs, we can prove that flattening does not eliminate anything crucial, but perhaps the formulation of the true story about constraints on movement in human language will require the full trees, or perhaps there is late adjunction which requires unboundedly large trees, or something!



A different idea, the **multidominance theory**, says that when a phrase moves, what really happens is that one of the constituents already attached in the structure gets attached again, so that it now has two mothers (or more if it moves multiple times). This idea can be depicted like this:



One argument for the copy or multidominance theory is that, after movement, some processes seem to act like the moved constituent is still in its original position. Most notably and obviously: at least sometimes, it seems that the moved constituent is interpreted as if it is still in its original position, as in the (apparently) simple case of topicalization of the object of a verb, in which the fronted element is of course still the object of the verb:

- (0) Mary, I like ___

This argument is not decisive, since we can define the interpretation function to do this even with the MG approach.

Another argument in favor of copy or multidominance theory comes from the idea that, at least sometimes, a moved element is pronounced (or partially pronounced) in more than one of its positions. For example, this kind of analysis has been proposed for verbal clefts in some languages. In §8 of ISAT, English clefts are introduced,

- (1) It is Mary (that) he likes ___ the most
 (2) It is after class (that) I can most easily meet with you ___

Here something moves, and nothing is repeated. But in the the African Kru language, Vata, Hilda Koopman'84,'97 noticed that in verbal clefts, verbal elements are repeated:

- (3) pā ñ ká mǝ pá ā
 throw you will it throw Q
 'Are you going to throw it?' [throw as opposed to roll]
 (4) pā í ká mǝ pá
 throw I will it throw
 'I will throw it'

Koopman 1984 proposed that these were instances of movement of the V to C, rather like the subject-auxiliary inversion in English polarity questions. Kandybowicz 2008 observes something similar in another Niger-Congo language, Nupe:

- (5) Bi-ba Musa à ba nakàn o
 cut Musa FUT cut meat FOC
 'It is cutting that Musa will do to the meat [as opposed to say, cooking]'

And this example is similar, from the Niger-Congo language Fongbe, reported by Lefebvre,

- (6) Lón wè súnù ó Lón
 jump it is man D jump
 'It is jump that the man did [not e.g. run away]'

Many linguists have observed what looks like copying in syntax, in many languages, copying verbs, wh-words, clitic pronouns, and more.³

We observed on page 4 that copy constructions in English seem to exist only in rather peripheral parts of the grammar [23]:

(NP-or-no-NP)	Linguistics or no linguistics	(NP-shmNP)	Linguistics shminguistics
(a-NP-is-a-NP-is-a-NP)	A dog is a dog is a dog	(CP or CP?)	Is she beautiful or is she beautiful?

These are not central in English grammar, and may not share all the properties of the better-integrated reduplication constructions mentioned above, but we might nevertheless wonder how normal speakers of English compute their analyses! We need some way to compute an appropriate analysis of

Linguistics exam or no linguistics exam, I am going to the party!

In the copy theory, we could say that, in these cases, more than one of the moved copies is pronounced. And in the multidominance theory, we would say that the phrase can be pronounced under more than one of its mothers. But in MGs, we have no copying mechanism, and because there is only one moved phrase, of course it is never in more than one place at a time! The MG approach to copying on page 161 requires pairs of identical elements, distinguished by category, to be introduced, instead of copying them.

14.3 MGCs

Greg Kobele’06 shows that it is not hard to add copy-movement to MGs. We can add a ‘copy’ diacritic to our movement triggers, so instead of just triggers $+f$ we can also have copy-move triggers $+f̂$ which are used by this new case of move:

$$\text{move}(t_1[+\hat{f}]) = t_2 \begin{matrix} > \\ / \quad \backslash \\ t_1 \{ t_2[-f] \mapsto t_2 \} \end{matrix}$$

(The flattened version of this approach is also straightforward. The ‘persistence’ of the string is treating in a way analogous to the persistence of features: merge3 optionally both launches and copies the moving string in case the trigger for move is a copy-move trigger.)

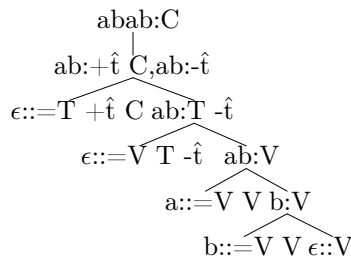
Example. With this new copy rule, we can define the xx copy language much more easily than we could before! Consider this grammar:

$$\begin{aligned} \epsilon &::= V & \epsilon &::= V \ T \ -t \\ a &::= V \ V & \epsilon &::= T \ +\hat{t} \ C \\ b &::= V \ V \end{aligned}$$

In this grammar, the 3 lexical items on the left allow us to define strings of category V, and that set is

$$\Sigma^* = \{a, b\}^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}.$$

The first lexical item on the the top right selects any V to form a T which needs to be licensed in a $+t$ position. The last lexical item forms a C by selecting T and then copy-moving the $-t$.



Let’s briefly consider a few more natural examples. . .

³Cf. Abels 2001; Bošković 2001; Bobaljik 1995, 2002; Brody 1995; Franks 1998; Groat and O’Neil 1996; Grohmann 2003; Hiraiwa 2005; Hornstein 2001; Landau 2006; Lefebvre 1992; Lidz and Idsardi 1998; ManasterRamer 1986; Nkemnji 1995; Nunes 1995, 1999, 2004; Pesetsky 1997, 1998; Richards 1997; Runner 1998; Stjepanovic 2003; Wilder 1995; and others.

14.4 Example: MGC for verbal clefts

Consider this grammar:

$$\begin{array}{ll} \epsilon::=V =D =D v & \epsilon::=T +\hat{c} C \\ \text{throw}::V -c & \\ \text{you}::D & \text{it}::D \end{array}$$

Then we have:

$$\begin{array}{c} \text{throw you will it throw:C} \\ | \\ \text{you will it throw:+}\hat{c} C,\text{throw:-}\hat{c} \\ \epsilon::=T +\hat{c} C \text{ you will it throw:T,throw:-}\hat{c} \\ | \\ \text{will it throw:+}D,\text{you:-}D,\text{throw:-}\hat{c} \\ | \\ \text{will}::=v +D T \text{ it throw:v,you:-}D,\text{throw:-}\hat{c} \\ | \\ \text{you}::D \text{ it throw}::=D v,\text{throw:-}\hat{c} \\ | \\ \text{it}::D \text{ throw}::=D =D v,\text{throw:-}\hat{c} \\ | \\ \epsilon::=V =D =D \text{ throw}::V -c \end{array}$$

14.5 Example: MGC for X-or-no-X

We have not worried yet about coordinate structures. They have many special properties [!] But for the sake of discussion, let's adopt this simple analysis for the moment:

$$\begin{array}{ll} \text{fight}::N & \text{or}::=N =N N \\ \text{fight}::N & \end{array}$$

So we have

$$\Rightarrow \text{PIC} \Leftarrow$$

To get a few more complex NPs, let's add also these simplistic rules for noun compounds and NPs with PP modifiers:⁴

$$\begin{array}{ll} \text{linguistics}::N & \epsilon::=N =N N \\ \text{exam}::N & \\ \epsilon::=N D & \text{from}::=D P \\ \epsilon::=P =N N & \text{about}::=D P \end{array}$$

$$\Rightarrow \text{PIC} \Leftarrow$$

The key ingredient for NP-or-no-NP is something like this, assuming that the categorial feature N can persist as a movement feature -N,⁵

$$\text{or no}::=N +\hat{N} \text{ OrNo.}$$

Finally, to form our examples we can ignore the derivation of the rest of the sentence

$$\text{let's party:T}$$

With these lexical items we have:

$$\Rightarrow \text{PIC} \Leftarrow$$

⁴We discussed various ideas about adjunction in §12.6.7 and §13.7. There are (no surprise) many ideas about noun compounds in the literature too. Cf. e.g. [29] for a very different idea from the simple one presented here.

⁵This idea was introduced for EPP effects in §12.1.1 on page 135, and adopted in the more minimal account again in §13.2 on page 159.

14.6 MGC languages are not mildly context sensitive

Joshi’s hypothesis that human languages are weakly and strongly *mildly context sensitive* MCS, mentioned on page 93, has one component – ‘constant growth’ – which might seem surprising. Recall that a class of languages is MCS if, and only if,

- It properly include the context free languages
- Every language can be parsed in polynomial time
- The class includes some (but not all) crossing dependencies
- The languages have the constant growth property: for each MCS language, there is a finite constant k such that if a string s has length i and some string s' has length greater than $i + k$, then there is at least one string of intermediate length between them.

Let’s consider this last condition more carefully. Suppose we try to design a language with big gaps between string lengths. We could have a lexical item which is big, and a construction that doubles or triples that length:

$$\begin{array}{ll} \epsilon ::= X X & \epsilon ::= C \\ a a a a a a a a a a a a : X & \epsilon ::= X C \end{array}$$

Obviously this grammar generates $\{a^{10^x} | x \geq 0\}$, which satisfies the finite copying condition with $k = 10$. We could try a little harder, like this:

$$\begin{array}{ll} \epsilon ::= X X & \epsilon ::= C \\ a a a a a a a a a a a a : X & \epsilon ::= Y C \\ \epsilon ::= X =X =X =X =X =X =X =X =X =X Y & \\ \epsilon ::= Y =Y Y & \end{array}$$

Obviously this grammar generates $\{a^{100^x} | x \geq 0\}$, which satisfies the finite copying condition with $k = 100$.

How could we possibly violate the finite copying condition? We need a different kind of mechanism for combining constituents, and we have just defined such a mechanism! Consider this grammar:

$$a : X \text{ -f} \quad \epsilon ::= X \text{ +f } X \text{ -f} \quad \epsilon ::= X \text{ +f } C$$

This grammar provides, for example, these derivations:

$$\Rightarrow \text{PIC} \Leftarrow$$

It is not hard to see that this grammar with copying movements derives the language $\{a^{2^n} | n \geq 0\}$. It is obvious that this language violates the finite copying condition: considering the strings of the language in order of length, the differences in lengths increase exponentially without bound:⁶

$$\begin{array}{ll} L = \{ & a^{2^0}, \quad a^{2^1}, \quad a^{2^2}, \quad a^{2^3}, \quad a^{2^4}, \quad \dots \} \\ = \{ & a, \quad aa, \quad aaaa, \quad aaaaaaaaa, \quad aaaaaaaaaaaaaaaaa, \quad \dots \} \\ k = & 1, \quad 2, \quad 4, \quad 8, \quad \dots \end{array}$$

Obviously, this kind of growth occurs when we can have copies of things with copies in them. Does that ever happen in natural language? There is some evidence that the most plausible grammars are powerful enough to allow it, and some controversy about whether we actually see it in forms that are small enough for us to have clear judgements (or show other evidence of analyzing as such).

14.7 Recognizing, parsing MGCs

. We saw in §9.3 on page 113 that MGs are a very succinct notation for a certain kind of MCFGs. In the same way, MGCs are a succinct notation for a certain kind of PMCFGs, where that stands for the ‘Parallel Multiple Context Free Grammars’ defined by Seki&al’91. Seki&al observed that PMCFGs can be efficiently parsed with a CKY-like method.

⁶Michaelis and Kracht’96 prove that this language cannot be defined by any MG using Parikh’s theorem. MG definable languages are ‘semilinear’, but this one is not.

14.7.1 Earley-like parsing for MGCs

We first develop an idea based on the work of Ljunglöf'12 and Angelov'09, based on the first steps which were taken in §AngelovSec1.

⇒ XX ⇐

14.7.2 A puzzle: Late adjunction in MGCs

14.8 Reflections on the MCS hypothesis

Since the empirical arguments for copying in human language seem extremely good, why does Joshi'85 propose the MCS hypothesis that excludes grammars that copy? First, note that one could say that the shift from MGs to MGCs is not really a change in syntax: the way the syntactic derivation is calculated is the same, with only a change in the spellout function. But even so, the MCS hypothesis deliberately excludes grammars with copying functions, since these grammars can violate the finite copying condition. Another idea is that perhaps the copying allowed by MGCs is much more liberal than necessary: in particular, it allows copies of copies, without bound, and perhaps this really goes beyond anything in human grammars. I think this is Joshi's view, and certainly it is harder to argue against. Some considerations favoring copies of constituents with copies in them were very briefly suggested here, but the matter deserves more careful consideration.

References

- [1] ABELS, K. The predicate cleft construction in Russian. In *Formal Approaches to Slavic Linguistics 9*, S. Franks, T. H. King, and M. Yadroff, Eds. Michigan Slavic Publications, Ann Arbor, Michigan, 2001, pp. 1–19.
- [2] ANGELOV, K. Incremental parsing with parallel multiple context-free grammars. In *Proceedings of the 13th Conference of the European Chapter of the Association for Computational Linguistics, EACL (2009)*, pp. 69–76.
- [3] BOBALJIK, J. D. *Morphosyntax: The Syntax of Verbal Inflection*. PhD thesis, Massachusetts Institute of Technology, 1995.
- [4] BOBALJIK, J. D. A-chains at the interfaces: Copies, agreement and covert movement. *Natural Language and Linguistic Theory 20* (2002), 197–267.
- [5] BOŠKOVIĆ, Ž. *On the Nature of the Syntax-Phonology Interface: Cliticization and Related Phenomena*. Elsevier, Oxford, 2001.
- [6] BRODY, M. *Lexico-Logical Form: A Radically Minimalist Theory*. MIT Press, Cambridge, Massachusetts, 1995.
- [7] BURDEN, H. *Implementations of Parsing Algorithms for Linear Multiple Context Free Grammars*. PhD thesis, Göteborg University, 2005.
- [8] FRANKS, S. Clitics in Slavic. In *Comparative Slavic Morphosyntax Workshop (1998)*. <http://www.indiana.edu/~slavconf/linguistics/index.html>.
- [9] GROAT, E., AND O'NEIL, J. Spell-out at the LF interface: Achieving a unified computational system in the minimalist framework. In *Minimal Ideas: Syntactic Studies in the Minimalist Framework*, W. Abraham, S. D. Epstein, H. Thráinsson, and C. J.-W. Zwart, Eds. John Benjamins, Philadelphia, 1996.
- [10] GROHMANN, K. K. *Prolific Domains: On the Anti-Locality of Movement*. *Linguistik Aktuell/Linguistics Today 66*. John Benjamins, Amsterdam, 2003.
- [11] HIRAIWA, K. *Dimensions of Symmetry in Syntax: Agreement and Clausal Architecture*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 2005.
- [12] HORNSTEIN, N. *Move! A Minimalist Theory of Construal*. Blackwell, Oxford, 2001.
- [13] KANDYBOWICZ, J. On fusion and multiple copy spell-out. In *The Copy Theory of Movement on the PF Side*. John Benjamins, Amsterdam, 2007, pp. 119–150.
- [14] KANDYBOWICZ, J. *The Grammar of Repetition: Nupe Grammar at the Syntax-Phonology Interface*. John Benjamins, Philadelphia, 2008.
- [15] KOBELE, G. M. *Generating Copies: An Investigation into Structural Identity in Language and Grammar*. PhD thesis, UCLA, 2006.
- [16] KOOPMAN, H. *The Syntax of Verbs: From Verb Movement Rules in the Kru Languages to Universal Grammar*. Foris, Dordrecht, 1984.

- [17] KOOPMAN, H. Unifying predicate cleft constructions. In *Proceedings of the 23rd Annual Meeting of the Berkeley Linguistics Society, Special Session on Syntax and Semantics in Africa* (1997), K. Moore, Ed. Reprinted in H. J. Koopman, ed., *The Syntax of Specifiers and Heads*, NY: Routledge, pp. 366-382.
- [18] LANDAU, I. Chain resolution in Hebrew v(p)-fronting. *Syntax* 9, 1 (2006), 32–66.
- [19] LEFEBVRE, C. Toward a typology of predicate cleft languages. *Journal of West African Languages* 22, 1 (1992), 53–61.
- [20] LIDZ, J., AND IDSARDI, W. Chains and phonological form. In *Proceedings of the 22nd Annual Penn Linguistics Colloquium, Working Papers in Linguistics* 5.1. University of Pennsylvania, 1998, pp. 109–125.
- [21] LJUNGLÖF, P. A polynomial time extension of parallel multiple context-free grammar. In *Proceedings, Logical Aspects of Computational Linguistics, LACL'05* (2005), pp. 177–188.
- [22] LJUNGLÖF, P. Practical parsing of parallel multiple context-free grammars. In *TAG+11, 11th International Workshop on Tree Adjoining Grammar and Related Formalisms* (2012).
- [23] MANASTER-RAMER, A. Copying in natural languages, context freeness, and queue grammars. In *Proceedings of the 1986 Meeting of the Association for Computational Linguistics* (1986).
- [24] MICHAELIS, J., AND KRACHT, M. Semilinearity as a syntactic invariant. In *Logical Aspects of Computational Linguistics* (NY, 1997), C. Retoré, Ed., Springer-Verlag (Lecture Notes in Computer Science 1328), pp. 37–40.
- [25] NKEMNJI, M. A. *Heavy Pied-Piping in Nweh*. PhD thesis, University of California, Los Angeles, 1995.
- [26] NUNES, J. *Linearization of Chains and Sideward Movement*. Cambridge, Massachusetts, MIT Press, 2004.
- [27] PESETSKY, D. Some optimality principles of sentence pronunciation. In *Is the Best Good Enough? Optimality and Competition in Syntax*, P. Barbosa, D. Fox, P. Hagstrom, M. McGinnis, and D. Pesetsky, Eds. MIT Press, Cambridge, Massachusetts, 1998, pp. 337–383.
- [28] RICHARDS, N. *What Moves Where When in Which Language?* PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1997.
- [29] ROEPER, T., AND SNYDER, W. Language learnability and the forms of recursion. In *UG and external systems*, A. M. Di Sciullo and R. Delmonte, Eds. John Benjamins, Amsterdam, 2005, pp. 155–169.
- [30] RUBINO, C. Reduplication. In *The World Atlas of Language Structures*, M. Haspelmath, M. S. Dryer, D. Gil, and B. Comrie, Eds. Oxford University Press, Oxford, 2005.
- [31] RUNNER, J. *Noun Phrase Licensing*. Garland, NY, 1998.
- [32] SEKI, H., MATSUMURA, T., FUJII, M., AND KASAMI, T. On multiple context-free grammars. *Theoretical Computer Science* 88 (1991), 191–229.
- [33] STABLER, E. P. Varieties of crossing dependencies: Structure dependence and mild context sensitivity. *Cognitive Science* 93, 5 (2004), 699–720.
- [34] STJEPANOVIC, S. A word-order paradox resolved by copy deletion at PF. *Linguistic Variation Yearbook* 3 (2003), 139–177.
- [35] WILDER, C. Rightward movement as leftward deletion. In *On Extraction and Extraposition in German*, *Linguistik Aktuell/Linguistics Today* 11. John Benjamins, Amsterdam, 1995, pp. 273–309.

Part III

Interfaces, origins, summary

Chapter 15 Below syntax: Morphology, phonology

A wide range of languages can be parsed efficiently – the whole, vast class of languages definable by minimalist grammars with copying. We would like to achieve not just feasibility here, but feasibility in the sense of near linear time on easily understood utterances of normal fluent speech. But we have been considering the parsing problem as if it applies fundamentally to sequences of printed words. Obviously, in ordinary conversation we deal with (some analysis of) acoustic input, and in reading we deal with (some analysis of) visual input. How close to the acoustic and visual interfaces will the syntax take us, and what are the inputs at those most superficial levels? Are there other kinds of grammars (morphology, phonology, phonetics) at those levels?

15.1 Phonology: What is it?

We used systems of rewriting rules, particularly as formulated in SPE, to give concreteness to our work and to the paper. However, we continually sought solutions in terms of algebraic abstractions of sufficiently high level to free them from any necessary attachment to that or any other specific theory. . . From a practical point of view, the result of the work reported here has been a set of powerful and sometimes quite complex tools for compiling phonological grammars in a variety of formalisms into a single representation, namely a finite-state transducer. – Ronald Kaplan & Martin Kay [7]

OT raises a particularly interesting theoretical question in this context: it allows the specification of a ranking among the constraints and allows lower ranked constraints to be violated in order for higher ranked constraints to be satisfied. We . . . study the formal properties of one particular case of this general formalization in which the mapping from input to possible output forms, GEN, is representable as a finite state transducer, and where each constraint is represented by means of some total function from strings to non-negative integers, with the requirement that the inverse image of every integer be a regular set. These two formal assumptions are sufficiently generous to allow us to capture most of the current phonological analyses within the OT framework that have been presented in the literature. We prove that the generative capacity of the resulting system does not exceed that of the class of finite state transducers precisely when each constraint has a finite co-domain. . .

– Robert Frank & Giorgio Satta [4]

15.1.1 Preliminaries

Let's consider the following English and American sounds (plus a few diacritics), listed here in the standard IPA notation, Mitton's ascii notation, and the 'Arpabet' ascii notation of CMU and TIMIT.¹

¹The standard NLTK installation includes a sample from the TIMIT corpus – a selection of sentences spoken and transcribed in different dialects of English. For the CMU pronouncing dictionary: <http://svn.code.sf.net/p/cmuspinyin/code/trunk/cmudict/>, and Hayes has a corrected version of the CMU dictionary here: <http://www.linguistics.ucla.edu/people/hayes/251English/>.

IPA	Mitton	ARPA	example	IPA	Mitton	ARPA	example
i	i	IY	see	æ	&	AE	cat
æ	&	AE	abnormal	ɪ	I	IH	hip
ɛ	e	EH	bed	ə	@	AH	about
u	u	UW	food	ʌ	V	AH	bud
ʊ	U	UH	foot	ɐ	0	AA	cot,pond
ɜ	3	ER	fur	a	A	AA	father
ɔ	O	AO	caught	eɪ	eI	EY	day
aɪ	aI	AY	my	oɪ	oI	OY	boy
aʊ	aU	AW	cow	ɪə	I@		beer
eə	e@		hair	ʊə	U@		pure,moor
oʊ	@U	OW	go	p	p	P	pi
t	t	T	tie	k	k	K	kick
b	b	B	by	d	d	D	die
g	g	G	guy	f	f	F	fie
v	v	V	vie	s	s	S	sigh
z	z	Z	zoo	ʃ	S	SH	shoe
θ	T	TH	thin	ð	D	DH	then
ʒ	Z	ZH	beige	ɹ	r	R	red
h	h	HH	head	j	j	Y	yet
ɹ	R	R	far	l	l	L	lie
w	w	W	wed	m	m	M	meet
n	n	N	neat	ŋ	N	NG	sing
ɟʃ	dZ	JH	joke	tʃ	tS	CH	choke
r	d	D	muddy	ˈ	ˈ	1	primary stress

The table above is generated by `ipa.py`, which is also used to define some basic conversions between notations. The Mitton [10] dictionary is in `mitton.py` and the CMU dictionary with Hayes' corrections is in `cmuHayes.py`. These dictionaries will be useful for practicing with our phonology.

There are a number of free finite state analysis tools,² but it is valuable to write some of your own too, to understand how they work.

Let's look at simple python implementations for nondeterministic finite automata with ϵ transitions.

15.1.2 SPE-like rules for alternations

The question of whether nasalization is phonetic, not phonological, is addressed in [2]

15.1.3 Rules for phonotactics

15.1.4 OT phonology

15.1.5 Unconditioned variation, and reduction in fluent speech

The king prefers a nice day / an ice day

Phonetic and acoustic reductions in fluent speech have [3]

15.1.6 Orthography: graphemes, unconditioned variation

A straightforward approach to using machine learning methods to identify grapheme/phoneme alignment is proposed in [8].³

One source of unconditioned variation, mentioned in Zuraw's handout, comes from the wide range of scripts and typefaces. Another source of unconditioned variation is spelling errors. OCR technologies have grappled with both of these for many years, but there seems to be rather little psycholinguistic research on these factors in reading.

²See the free finite state tools available from the Xerox project <http://www.stanford.edu/~laurik/fsmbook/home.html> and from AT&T <http://www2.research.att.com/~fsmtools/fsm/>.

³And, remarkably, this idea was awarded a patent in 2011: <http://www.google.com/patents/US7991615>.

15.2 Morphology in the syntax? Zero-level + spellout

Morphology, we argue, may be reduced entirely to the function that spells out the syntactic tree by choosing and inserting phonologically contentful lexical items. – Patrik Bye & Peter Svenonius [1]

15.3 Phonetics in the syntax?

[2] [9] [11]

Hayes et al'02[6, 5]

15.4 Orthography and reading

Linguists focus mainly on spoken language, but obviously the human ability to record language for visual recognition has transformed our society in so many ways that it becomes very hard for us in the 21st century to imagine life and culture without written records of any kind. It seems, at least upon first reflection, that reading requires training of a kind that is not necessary for the acquisition of spoken or signed language. For spoken language, it is apparently enough for a child to be immersed in any approximately normal community of speakers; no formal schooling, no regimented and tested memorization of words is required. Sign languages show that the distinction of reading and writing is not the visual medium; the schooling may be required because written characters are typically recognized quietly, privately.

References

- [1] BYE, P., AND SVENONIUS, P. Non-concatenative morphology as epiphenomenon. In *The Morphology and Phonology of Exponence: The State of the Art*, J. Trommer, Ed. Oxford University Press, Oxford, 2012, pp. 427–495.
- [2] COHN, A. Nasalization in english: Phonology or phonetics. *Phonology* 10, 1 (1993), 43–81.
- [3] DENG, L., YU, D., AND ACERO, A. A quantitative model for formant dynamics and contextually assimilated reduction in fluent speech. In *Proceedings of the International Conference on Spoken Language Processing, International Speech Communication Association* (2004).
- [4] FRANK, R., AND SATTA, G. Optimality theory and the generative complexity of constraint violability. *Computational Linguistics* 24 (1998), 307–315.
- [5] HAYES, B. Phonetically-driven phonology: The role of optimality theory and inductive grounding. In *Functionalism and formalism in linguistics*, M. Darnell, E. Moravcsik, M. Noonan, F. J. Newmeyer, and K. M. Wheatley, Eds. John Benjamins, Amsterdam, 1999, pp. 243–285.
- [6] HAYES, B., KIRCHNER, R., AND STERIADE, D., Eds. *The Phonetic Bases of Markedness*. Cambridge University Press, NY, 2002.
- [7] KAPLAN, R., AND KAY, M. Regular models of phonological rule systems. *Computational Linguistics* 20 (1994), 331–378.
- [8] LI, X., GUNAWARDANA, A., AND ACERO, A. Adapting grapheme-to-phoneme conversion for name recognition. In *IEEE Workshop on Automatic Speech Recognition And Understanding* (2007).
- [9] LIN, Y. *Learning Features and Segments from Waveforms*. PhD thesis, University of California, Los Angeles, 2005.
- [10] MITTON, R. *Oxford Advanced Learner’s Dictionary of Current English: expanded ‘computer usable’ version*. 1992.
- [11] YU, K. *Learning tones from the speech signal*. PhD thesis, UCLA, 2011.

Chapter 16 Above syntax: Natural logic, discourse dynamics

We have noted that minimalist grammars are designed to produce meaningful expressions by the arrangement of meaningful parts of those expressions. This is the idea of compositionality, which is suggested by these famous lines from Frege 1923:

It is astonishing what language can do. With a few syllables it can express an incalculable number of thoughts, so that even a thought grasped by a terrestrial being for the very first time can be put into a form of words which will be understood by someone to whom the thought is entirely new. This would be impossible, were we not able to distinguish parts in the thought corresponding to the parts of a sentence, so that the structure of the sentence serves as an image of the structure of the thought. [12]

The point of parsing is to identify the interpreted elements and their manner of assembly. This allows the meaning of complexes to be calculated from the meanings of their parts, but this a “god’s eye” perspective: given the mappings from words to their referents (usually understood to be elements of types built from e and t, or e and t and worlds and times), the semantics shows how to compute a mapping from sentences to truth values (or to functions for worlds to truth values or to situations). If we regard the mind of the language user as a computational engine, though, often we are dealing with expressions whose meanings we know very little about, and everything that happens is just further computation on the structural analysis. That is, the parse is the basis of further inferences about what the speaker intended:

... the picture of meaning to be developed here is inspired by Wittgenstein’s idea that the meaning of a word is constituted from its use ... Thus the meaning of the sentence does not have to be *worked out* on the basis of what is known about how it is constructed; for that knowledge by itself constitutes the sentence’s meaning... then compositionality is a trivial consequence of what we mean by “understanding” in connection with complex sentences. (Horwich [24, pp.3,9])

PF and LF constitute the ‘interface’ between language and other cognitive systems, yielding direct representations of sound, on the one hand, and meaning on the other as language and other systems interact, including perceptual and production systems, conceptual and pragmatic systems. (Chomsky [5, p.68])

The output of the sentence comprehension system... provides a domain for such further transformations as logical and inductive inferences, comparison with information in memory, comparison with information available from other perceptual channels, etc...[These] extra-linguistic transformations are defined directly over the grammatical form of the sentence, roughly, over its syntactic structural description (which, of course, includes a specification of its lexical items). (Fodor et al. [9])

We will aim here to get an understanding of human sentence understanding that is analogous to our understanding of how a simple device like a calculator works, in two complementary parts: a semantics that tells us what is represented, and an inference system that tells us how we reason with those representations. Following linguistic tradition, we focus on the semantics first, but with an eye on capturing the inferences that normal human speakers find natural.

Very often, structural ambiguities in morpheme sequences corresponds to a semantic ambiguity; two different derivations of the same sequence of words often have two different interpretations. (In many cases, context may make clear which of those two interpretations is the “intended” one, as we will discuss in the next section.) So we do not interpret the morpheme sequences directly. Rather, we interpret derivations. So the structure of the model is this [38, 23, 45, 27, and many others]:

syntax:	$(\text{Lexicon}, F)$
language:	$L = \text{closure}(\text{Lexicon}, F)$, with derivations Γ
(partial) semantics:	$\llbracket \cdot \rrbracket : \Gamma \rightarrow M$
synonymy:	$a \equiv_{\mu} b$ iff $a, b \in \text{domain}(\llbracket \cdot \rrbracket)$ and $\llbracket a \rrbracket = \llbracket b \rrbracket$.
μ is compositional	iff $f(a_1, \dots, a_n) \equiv_{\mu} f(b_1, \dots, b_n)$ whenever $a_i \equiv_{\mu} b_i$ (all $f \in F$)

In this chapter we provide a glimpse of compositional semantics of MG derivations, which will set the stage for studying inferences that are sound or probable. The topics briefly reviewed in this chapter could easily fill a whole book, and more! Not only do we need to find a way to frame “the whole of human knowledge,” as Tarski says, in order to provide an account of how we express and reason about anything you could have a conversation about, but really, getting this right is the whole point of trying to parse the syntax appropriately. In this section, we go quickly but provide references to more thorough accounts.

16.1 A natural logic for deduction

Syntactic properties are... “local”... which is to say that they are constituted entirely by what parts a representation has and how those parts are arranged... But though it’s true that the syntax of a representation is a local property in that sense, it’s also true that the syntax of a representation determines certain of its relations to other representations. Syntax, as it were, faces inward and outward at the same time. – Jerry Fodor [10, p.20]

I will use the term ‘natural logic’ to refer to theories about human-like inferences over the sorts of structures that humans use in reasoning, including, among other things, the syntactic structures of spoken and gestured languages.

16.1.1 Order-based deduction

Keenan and Faltz [26] give the following examples of an important semantic relation that any linguistic theory should provide an account of:

<u>John is a linguist and Mary is a biologist</u>	<u>walk and talk</u>
John is a linguist	walk
<u>tall and handsome</u>	<u>some but not all</u>
handsome	some
<u>John is crying</u>	<u>sing loudly</u>
John is crying or laughing	sing

The relation exemplified by all of these examples is clearly relevant to inference: the first example is an inference of the usual sort, and the other examples can be embedded in certain sentential contexts to make inferences. So what relation do we see in all these examples? It is the Boolean order \leq . In each of these examples, the denotation of the expression above the line is less than or equal to (\leq) the denotation of the example below the line, where \leq is the standard Boolean order. We will define this standard order \leq and then, later, show how it plays a role in inference in different kinds of logic and then in a human-like language.

We assume that there are exactly two truth values $\{0, 1\}$ and that they are ordered: $0 \leq 1$. This famous ordered set is sometimes called **2** or **t** or, in OCaml, **bool**. In OCaml, you can of course check that **false** \leq **true** evaluates to **true**, as expected! The \leq function is polymorphic, as is the Boolean \leq . The extension of \leq to the whole infinite range of Boolean types is easy to define:

- (0) Where the truth values $t = \{0, 1\}$, and e is another type, that is, any nonempty set, $\text{TH}(e, t) = \text{closure}(\{e, t\}, F)$ where F is the function mapping any sets s, t to the type $s \rightarrow t$, which we interpret as they set of total functions from s to t .
- (1) The set of Boolean types is the smallest set such that **t** is Boolean, and if type x is Boolean, so is $y \rightarrow x$ for any type y .
- (2) For any Boolean type s in $\text{TH}(e, t)$ and any $x, y \in s$, $x \leq_s y$ iff

$$\begin{cases} s = e \text{ and } x = y, \text{ or} \\ s = t \text{ and either } x = 0 \text{ or } y = 1, \text{ or} \\ s = [t \rightarrow u] \text{ and } \forall z \in t, x(z) \leq_u y(z) \end{cases}$$

As will gradually become clear in the next sections, this is the relation we see in all the examples from Keenan and Faltz mentioned above.

A **logic** usually has 3 parts: a syntax, a semantics, and an inference system defined on the syntax. Human language can be regarded as a logic, where the inference system is the logic of discourse (or maybe, the various kinds of logic of the various kinds of discourses). We practice with some simple systems first. I love these lines that Alfred Tarski wrote in 1945 for the preface of his *Introduction to Logic* [42], when he was at the University of California, Berkeley. He says that the first edition of the book was intended to present “a clear idea of that powerful trend of contemporary thought which is concentrated about modern logic”:

This trend arose originally from the somewhat limited task of stabilizing the foundations of mathematics. In its present phase, however, it has much wider aims. For it seeks to create a unified conceptual apparatus which would supply a common basis for the whole of human knowledge. Furthermore, it tends to perfect and sharpen the deductive method, which in some sciences is regarded as the sole means of establishing truths, and indeed in every domain of intellectual activity is at least an indispensable auxiliary tool for deriving conclusions from accepted assumptions.

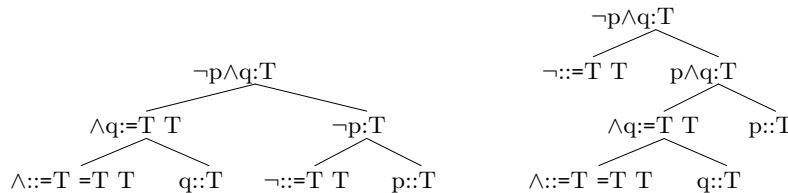
Our goals here are similar, in the sense that we aim to say something about what we can express in our language. But rather than providing a foundation for correct deductive inference, our goal is simply to model what deductive and other inferential steps people make in ordinary language use, as for example in the understanding of normal fluent speech.

16.1.2 Propositional calculus

Syntax. Consider again the formulation of the propositional calculus with a minimalist grammar (MG) containing 6 lexical items, from §??, repeated here:

1	$p::T$	2	$q::T$	3	$r::T$
4	$\neg::=T\ T$	6	$\wedge::=T\ =T\ T$	7	$\supset::=T\ =T\ T$
5	$\vee::=T\ =T\ T$				

With this grammar, the string $\neg p \wedge q::T$ is structurally ambiguous, with these two derivations, in which \neg combines with different elements:



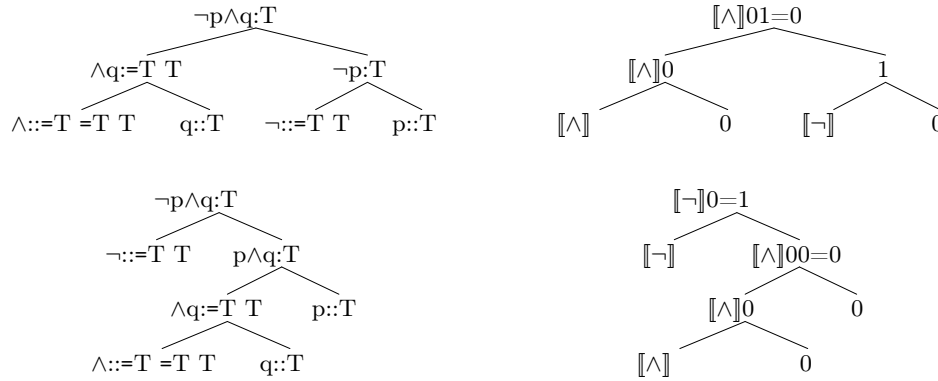
Specifying the lexical items by their number, the derivation trees have the lexical yields is 6241 and 4621, respectively. In a more readable notation, the lexical yields are $\wedge q \neg p$ and $\neg \wedge qp$. Unlike the pronounced ‘string language’ of this grammar, the yields of the derivations are unambiguous. As noted in §??, for any MG grammar, the yields of the derivations form an unambiguous context free language, so we can give any MG semantics on derivations simply by referring to these sequences. **Notice** that with this grammar, the sentence usually written $p \supset q$, that is, “p implies q,” has the derivation $\supset qp$. This is because the “object” q of the operator \supset is given as the first argument, even though it follows the “subject” p.

Semantics. Here we interpret derivation trees, but we use the abbreviated notations just introduced. For example, $\neg \wedge qp$ is unambiguously $\neg((\wedge q)p)$. Let’s write $f : t$ for a function f of type t, and remember the notational convention that a type $x \rightarrow y \rightarrow z$ associates to the right, so it is $x \rightarrow (y \rightarrow z)$. Then an interpretation $\llbracket \cdot \rrbracket$ of the propositional calculus given above is a function such that

- i. $\llbracket p \rrbracket : t$. That is, $\llbracket p \rrbracket \in \{0, 1\}$.
- ii. $\llbracket q \rrbracket : t$. That is, $\llbracket q \rrbracket \in \{0, 1\}$.
- iii. $\llbracket r \rrbracket : t$. That is, $\llbracket r \rrbracket \in \{0, 1\}$.
- iv. $\llbracket \neg \rrbracket : t \rightarrow t$, where $\llbracket \neg \rrbracket x = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases}$

- v. $\llbracket \vee \rrbracket : t \rightarrow t \rightarrow t$, where $\llbracket \vee \rrbracket xy = \begin{cases} 0 & \text{if } x = y = 0 \\ 1 & \text{otherwise} \end{cases}$
- vi. $\llbracket \wedge \rrbracket : t \rightarrow t \rightarrow t$, where $\llbracket \vee \rrbracket xy = \begin{cases} 1 & \text{if } x = y = 1 \\ 0 & \text{otherwise} \end{cases}$
- vii. $\llbracket \supset \rrbracket : t \rightarrow t \rightarrow t$, where $\llbracket \supset \rrbracket xy = \begin{cases} 0 & \text{if } x = 1, y = 0 \\ 1 & \text{otherwise} \end{cases}$
- viii. $\llbracket (x y) \rrbracket = \llbracket x \rrbracket \llbracket y \rrbracket$. That is, the denotation of x is applied to the denotation of y .

Notice that the definition of $\llbracket \cdot \rrbracket$ has 1 clause for each of the 7 lexical items, and then one recursive case. It is easy to see that $\llbracket \wedge \rrbracket \leq \llbracket \vee \rrbracket$ and $\llbracket \supset \rrbracket \leq \llbracket \vee \rrbracket$, but neither of $\llbracket \wedge \rrbracket$ and $\llbracket \supset \rrbracket$ is less than or equal to the other. Applying this definition to our examples above, when $\llbracket p \rrbracket = \llbracket q \rrbracket = 0$, we see that the two derivations have different truth values:



16.1.3 Polarity and reasoning

Standard introductory presentations of deductive reasoning for the propositional calculus introduce rules like the following. These rules are adapted from Parsons (2009), but note that we are stating them here over the prefix notation that our derivations use, and so the arguments of \supset are in a nonstandard order: read $\supset qp$ as “ q , if p ,” which is equivalent to “if p then q .”

$\frac{\supset qp \quad p}{q} \text{ modus ponens}$	$\frac{\supset qp \quad \neg q}{p} \text{ modus tollens}$	$\frac{\begin{matrix} \mathfrak{P} \\ \dots \\ q \end{matrix}}{\supset pq} \text{ conditional}$
$\frac{\begin{matrix} \mathfrak{P} \\ \dots \\ q \\ \neg q \end{matrix}}{\neg p} \text{ neg}$	$\frac{p}{\neg \neg p} \text{ double neg}$	$\frac{\neg \neg p}{p} \text{ double neg}$
$\frac{\neg p \quad \wedge pq}{p} \text{ s}$	$\frac{\wedge pq}{q} \text{ s}$	$\frac{p \quad q}{\wedge pq} \text{ adj}$
$\frac{p}{\vee pq} \text{ add}$	$\frac{q}{\vee pq} \text{ add}$	$\frac{\vee pq \quad \neg q}{p} \text{ mpt} \quad \frac{\vee pq \quad \neg p}{q} \text{ mpt}$

Rules of this sort have been very carefully studied! The natural, easy inferences from Keenan and Faltz that we mentioned at the outset seem to overlap with these; there are details to spell out, but it looks like the first example from Keenan and Faltz, on page 178, is an instance of rule *s*, and some of their other examples look closely related. However, these standard rules for the propositional calculus do not highlight the fundamental property that all the Keenan and Faltz examples have in common. And some things that seem obvious, like $\vee p \neg p$ have slightly tricky proofs, while other things that seem very non-obvious are not hard to show, like $\supset \supset r \supset p \supset r \supset qp$, that is, (if (if p then q) then r) then (if (if p then r) then r). The standard inference rules for the propositional calculus do not immediately give us a good model of the inferences that people find easy and natural (no surprise!). The project of designing logics which are a better fit with reasoning that people find easy is sometimes called “natural logic” [4, 35, 37, ?, 14, 3, 31].

Can we use an order-based inference to show $\vee p \neg p$? It is easy to see that, if $p = 0$ then $p \leq \vee p \neg p$, and if $p = 1$ then $p \leq \vee p \neg p$, so \leq holds in all cases. Can we use this idea? First, we can notice that it is not always safe to

replace a subformula by a greater one. For example, it's not generally true that $\neg p \leq \neg \vee p \neg p!$ We say that the occurrence of p in $\neg p$ **negative polarity**, because \neg is a decreasing operator. We can define these notions this way:

- (3) For any $f \in [T \rightarrow U]$ in $\text{TH}(e, t)$, f is **increasing**, $f \uparrow$, iff whenever $x \leq_T y$, $fx \leq_U fy$, and f is **decreasing**, $f \downarrow$, iff whenever $x \leq_T y$, $fx \geq_U fy$.

Clearly, $\llbracket \neg \rrbracket$ is decreasing, since decreasing p in $\neg p$ will increase $\llbracket \neg p \rrbracket$. Notice that \supset is increasing in its first argument (the argument usually written after the \supset symbol) and decreasing in its second argument. This follows from the definition of $\llbracket \supset \rrbracket$: decreasing the value of p in $\supset qp$ will result in either no change or an increase in $\llbracket \supset qp \rrbracket$.

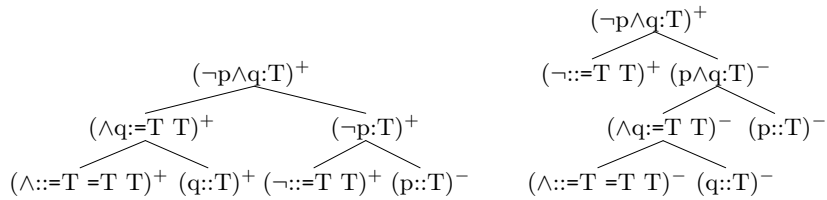
The other idea we need is the **polarity** of a (sub)derivation.

- (4) z **has positive polarity** or, more simply, **is positive** in z , z^+ in z .

$$z \text{ is positive in } (xy), M^+ \text{ in } (xy) \text{ if } \begin{cases} z^+ \text{ in } x, \text{ or} \\ z^+ \text{ in } y, \text{ and } \llbracket x \rrbracket \uparrow, \text{ or} \\ z^- \text{ in } y, \text{ and } \llbracket x \rrbracket \downarrow \end{cases}$$

$$z \text{ is negative in } (xy), z^- \text{ in } (xy), \text{ if } \begin{cases} z^- \text{ in } x, \text{ or} \\ z^- \text{ in } y, \text{ and } \llbracket x \rrbracket \uparrow, \text{ or} \\ z^+ \text{ in } y, \text{ and } \llbracket x \rrbracket \downarrow \end{cases}$$

Now we can label the subtrees of a derivation tree to indicate whether they appear in a positive (+) or negative (-) context:



Or, for short:

$$(\vee^+ (\neg^+ p^-) q^+)^+ \quad (\neg^+ ((\vee^- p^-) q^-)^-)^+$$

As expected, the only negative polarities are in the scope of \neg , which differs in the 2 trees. Notice that when a derivation gets merged with \neg , all its polarities flip!

Let's write $x[z]$ to mean that z is a particular occurrence of derivation z in derivation x . Now it is easy to show that if an occurrence of z is positive in x , that is, $x[z^+]$, and if $y \leq z$, then $x[z^+] \leq x[y^+]$. And we have the reverse situation when $x[z^-]$. That is, we have these inference rules:

$$\frac{x[z^+]}{x[y^+]} \text{ if } z \leq y \qquad \frac{x[z^-]}{x[y^-]} \text{ if } y \leq z$$

That is, when x occurs with positive polarity, it can be "increased", preserving truth. And when x occurs with negative polarity, it can be "decreased", preserving truth. This is called **polarity-based** or **monotonicity-based** reasoning.

16.1.4 Relation calculus

Unary relations of type $e \rightarrow t$ are usually called properties. And often when we speak of relations we mean binary relations, things of type $e \rightarrow e \rightarrow t$. Much of our ordinary conversations involve unary and binary relations, and simple relations among them. So it is natural to begin there, with the sorts of 'syllogistic' reasoning Aristotle noticed.

In 'generalized quantifier theory' and standard 'extensional' approaches to semantics, we observe these two basic things:

- **Almost every semantic domain allows the Boolean operations: meet, join, complement.**

	I paid every student and some teacher
DP combinations	You should pay John or Mary
	I paid not John but Mary
D combinations	some but not all students are curious
A combinations	I saw tall but not short giraffes
vP combinations	I walk to work and take the bus to the beach
...	...

- **In many contexts, simple determiners like *every* and *some* express relations between properties.** Treating properties like $\llbracket \text{student} \rrbracket$ as having the type $e \rightarrow t$ (or equivalently, as sets of things), we can treat $\llbracket \text{every student} \rrbracket$ as mapping properties to truth values, $(e \rightarrow t) \rightarrow t$ (or equivalently, as sets of properties). Then the denotation of $\llbracket \text{every} \rrbracket$ has the type $(e \rightarrow t) \rightarrow (e \rightarrow t) \rightarrow t$ (or equivalently, as sets of pairs of properties.) In fact, simple most determiners don't care what the properties are, but just about the size of the overlaps between the properties; that is, an expression of the form $Q_A B$, like *every student sings*, can be regarded as expressing a claim about the relation between the sizes of the sets $|A - B|$ and $|A \cap B|$. The proposition *every student sings* says that overlap of $\llbracket \text{student} \rrbracket$ and the complement of $\llbracket \text{sings} \rrbracket$ is empty. In general,

$\text{some}_A B$	$ A \cap B < 0$	$\text{every}_A B$	$ A - B = 0$
$\text{no}_A B$	$ A \cap B = 0$	$\text{at most } N_A B$	$ A \cap B \leq N$
$N_A B = \text{at least } N_A B$	$ A \cap B \geq N$	$\text{most}_A B$	$ A - B > A \cap B $
$\text{more than } N_A B$	$ A \cap B < N$	$\text{fewer than } N_A B$	$ A \cap B > N$
$\text{exactly } N_A B$	$ A \cap B = N$	$\text{the } N_A B$	$ A - B = 0 \& A \cap B = N$
$\text{all but } N_A B$	$ A - B = N$	$N \text{ out of } M_A B$	$ A - B = A \cap B \frac{M-N}{M}$
$\text{between } N \text{ and } M_A B$	$N \leq A \cap B \leq M$	$\text{every third}_A B$	$ A - B \geq 2 * A \cap B $
$\text{finitely many}_A B$	$ A \cap B > \aleph_0$	$\text{infinitely many}_A B$	$ A \cap B \geq \aleph_0$

With relations and these (higher order) relations over relations, many simple propositions can be expressed. The definitions above establish that some determiners are increasing or decreasing in their first or second arguments. For example, $\llbracket \text{every} \rrbracket$ is decreasing in its first argument but increasing in its second argument, which means that in any sentence, like *Every student sings*, you can decrease $\llbracket \text{student} \rrbracket$ to $\llbracket \text{Canadian student} \rrbracket$ or increase $\llbracket \text{sings} \rrbracket$ to $\llbracket \text{sings or dances} \rrbracket$, preserving truth. Marking the polarities of expressions here (as detailed for example in [3]):

$$\frac{((\text{Every}^+ \text{ student}^-)^+ \text{ sings}^+)^+}{\text{Every Canadian student sings}} \quad \frac{((\text{Every}^+ \text{ student}^-)^+ \text{ sings}^+)^+}{\text{Every student sings or dances}}$$

The determiner *no*, on the other hand, is decreasing in both arguments, which means both arguments can be decreased preserving truth:

$$\frac{(\text{No}^+ \text{ student}^-)^+ \text{ sings}^-}{\text{No Canadian student sings}} \quad \frac{(\text{No}^+ \text{ student}^-)^+ \text{ sings}^-}{\text{No student sings and dances}}$$

Determiners like *exactly*, on the other hand, are neither increasing nor decreasing, and so reasoning with them sometimes requires more than just the recognition of \leq relations.

Reasoning with binary relations has a certain beauty too. Tarski says,¹

... the calculus of relations has an intrinsic charm and beauty which makes it a source of intellectual delight to all who become acquainted with it. [41, p.89]

Tarski also introduces it in his 1946 logic text for beginners [42], and he works on it with Givant in *Set Theory without Variables* [43]. Using \circ to represent the composition of relations, and $^{-1}$ to represent inverses as usual:

first order predicate calculus	relation calculus
$\forall x \forall y (R(x, y) \supset R(y, x))$	$R \subseteq R^{-1}$
$\forall x \forall y \forall z ((R(x, y) \wedge R(y, z)) \supset R(x, z))$	$R \circ R \subseteq R$
$\forall x \forall y \forall z ((R(x, y) \wedge R(x, z)) \supset \exists w (R(y, w) \wedge R(z, w)))$	$R^{-1} \circ R \subseteq R \circ R^{-1}$

¹This passage is noted by Marx [29] in a recent proposal about extensions of the relation calculus. Marx suggests that the general preference for first order predicate calculus over the relation calculus may be analogous to the general preference for QWERTY keyboards over alternatives. Our examples of relation calculus formulas, in the table just below, are also taken from this paper.

16.1.5 First order and higher order predicate calculus

The standard first order predicate calculus adds variables to the relation calculus. A variable x is interpreted not as naming anything in type e or type t or anything built from them, but something new. The denotation of a (first order) variable x is usually taken to be a function from assignments to individuals, so an assignment is a function of type $G : \text{Var} \rightarrow e$, and for a first order variable x , $\llbracket x \rrbracket$ has type $g \rightarrow e$. To make the rest of the semantics fit together properly, there are a couple of strategies, but one is to let the type of $\llbracket Px \rrbracket$ be not t but $g \rightarrow t$, for unary predicate P . So then $\llbracket P \rrbracket$ has type $(g \rightarrow e) \rightarrow (g \rightarrow t)$. In short, we relativize all the standard denotation types to assignments.

XXX MORE COMING XXX

16.1.6 English fragments

It is (not necessary but) common to assume that English expressions should be treated as having variable-like elements too.² For example, in the sentence *Mary knows who you like*, the verb *knows* takes the CP complement *who you like*, and the CP is usually regarded as *who* moving to the specifier position of *you like*. So really, that phrase *you like* is naturally regarded as having a variable in it, *you like x_{wh}* , where the variable gets bound by *who*. Let's quickly review how this kind of semantics could be provided for little fragments of English like the ones we formulated in §???. (Cf. Kobele's [28].)

Following an idea from Kobele, let's let the type of basic entities be not e but E , and the type of truth values be not t but T . Then we can redefine e and t as types that are relativized to assignments, as suggested in §16.1.5 just above. But instead of introducing an infinite set of variables, let's just treat the licensees of each grammar as variables. In §?? and §?? we treat linguistic expressions that have moving elements as tuples of categorized expressions, so we treat their semantic values as tuples as well. We use these basic types:

entities	E	
truth values	T	$= \{0, 1\}$
assignments	G	$= F \rightarrow E$
individuals	e	$= G \rightarrow E$
0-ary relations, propositions	t	$= G \rightarrow T$
1-ary relations, properties	$e \rightarrow t$	
2-ary relations, binary relations	$e \rightarrow e \rightarrow t$	
	\dots	
	GQs	$(e \rightarrow t) \rightarrow t$
	Dets	$(e \rightarrow t) \rightarrow (e \rightarrow t) \rightarrow t$
basic 0-ary relations T		
basic 1-ary relations $E \rightarrow T$		
basic 2-ary relations $E \rightarrow E \rightarrow T$		
\dots		

Note that, using the definition of Boolean types given in (1) on page 178 above, every type listed above is Boolean except E , G , e . The Boolean types all have a natural partial order \leq , with \wedge, \vee, \neg .

(5) Semantic values f, g combine according to their types, as follows:

$$f + g = \begin{cases} f(g) & \text{if } f : a \rightarrow b \text{ and } g : b \\ g(f) & \text{otherwise, if } g : a \rightarrow b \text{ and } f : b \\ f \wedge g & \text{otherwise, if } f, g \text{ have the same Boolean type} \end{cases}$$

(6) For any $g, h \in G$, let $g \approx_i h$ iff $\forall j \neq i, g(j) = h(j)$.

$$\text{For any } g \in G, a \in E, \text{ let } g^{[i:=a]}(x) \begin{cases} a & \text{if } x = i \\ g(x) & \text{otherwise.} \end{cases}$$

²This is standard practice in logic, and certainly it is coherent and well-understood, but some linguists have misgivings about it. See for example [2].

- (7) To reduce parens, as usual, the function space operator associates to the right

$$a \rightarrow b \rightarrow c = a \rightarrow (b \rightarrow c)$$

while application associates to the left

$$f(a)(b) = (f(a))(b).$$

- (8) Let
- μ
- assign basic denotations to determiners, nouns and verbs as follows:

$$\begin{array}{ll} \text{Sue}::D \mapsto \mathbf{sue} : E & \text{Joe}::D \mapsto \mathbf{joe} : E \\ \text{some}::=N D \mapsto \mathbf{some} : (E \rightarrow T) \rightarrow \langle 1 \rangle & \text{every}::=N D \mapsto \mathbf{every} : (E \rightarrow T) \rightarrow \langle 1 \rangle \\ \text{car}::N \mapsto \mathbf{car} : E \rightarrow T & \text{truck}::N \mapsto \mathbf{truck} : E \rightarrow T \\ \text{passed}::=D =D V \mapsto \mathbf{passed} : E \rightarrow E \rightarrow T & \text{hit}::=D =D V \mapsto \mathbf{hit} : E \rightarrow E \rightarrow T \end{array}$$

where for any P, R in $E \rightarrow T$,

$$\begin{array}{l} \mathbf{some}(P)(R) = 1 \text{ iff } \exists x \in E, P(x) = R(x) = 1 \\ \mathbf{every}(P)(R) = 1 \text{ iff } \forall x \in E, \text{ if } P(x) = 1 \text{ then } R(x) = 1. \end{array}$$

These values of $D(P)(R)$ for $R \in E \rightarrow T$ are then extended to predicates R of higher arity by ‘lifting’ into the higher types in the standard way [25, 21], so that, for example, for any $a \in E$ and $R^2 \in E \rightarrow E \rightarrow T$,

$$\begin{array}{l} \mathbf{some}(P)(R^2)(a) = 1 \text{ iff } \exists x \in E, P(x) = R^2(x)(a) = 1 \\ \mathbf{every}(P)(R^2)(a) = 1 \text{ iff } \forall x \in E, \text{ if } P(x) = 1 \text{ then } R^2(x)(a) = 1. \end{array}$$

The lift of a quantifier Q like $\mathbf{some}(P)$ can be computed by applying the combinator

$$\lambda Q^{b \rightarrow c}. \lambda R^{E \rightarrow b}. \lambda y^E. Q(\lambda x. R(x)(y)).$$

Letting $\langle 1 \rangle$ be the polymorphic type of the quantifiers obtained by iterated applications of this combinator, our determiners have the type $(E \rightarrow T) \rightarrow \langle 1 \rangle$.

- (9) In terms of
- μ
- we now define
- $\llbracket \cdot \rrbracket$
- as follows:

$$\llbracket \text{em1}(a_0, (b_0, b_1, \dots, b_k)) \rrbracket = (\llbracket a_0 \rrbracket + \llbracket b_0 \rrbracket, \llbracket b_1 \rrbracket, \dots, \llbracket b_k \rrbracket)$$

$$\llbracket \text{em2}((a_0, a_1, \dots, a_k), (b_0, b_1, \dots, b_l)) \rrbracket = (\llbracket a_0 \rrbracket + \llbracket b_0 \rrbracket, \llbracket a_1 \rrbracket, \dots, \llbracket a_k \rrbracket, \llbracket b_1 \rrbracket, \dots, \llbracket b_l \rrbracket).$$

Recalling that em3 applies to a second argument $t \cdot f\delta, \iota_1, \dots, \iota_l$ where $\delta \neq \epsilon$, when the first feature of δ is $-f$,

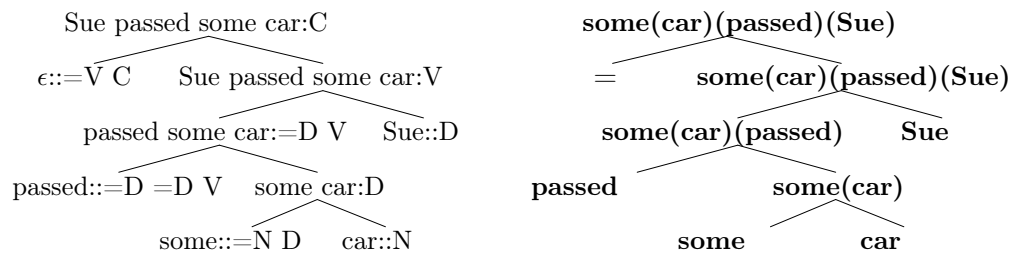
$$\llbracket \text{em3}((a_0, \dots, a_k), (b_0, \dots, b_l)) \rrbracket = (\llbracket a_0 \rrbracket(x_f), \llbracket a_1 \rrbracket, \dots, \llbracket a_k \rrbracket, \llbracket b_0 \rrbracket, \dots, \llbracket b_l \rrbracket).$$

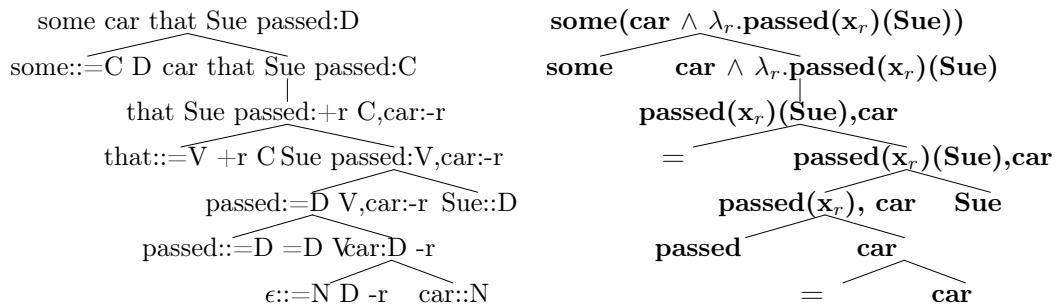
$$\llbracket \text{im1}(a_0, \dots, a_k) \rrbracket = (\llbracket a_i \rrbracket \lambda_f \llbracket a_0 \rrbracket, \dots, \llbracket a_{i-1} \rrbracket, \llbracket a_{i+1} \rrbracket, \dots, \llbracket a_k \rrbracket)$$

Recalling that im2 applies to an argument $s : +f\gamma, a_1, \dots, a_{i-1}, t : -f\delta, a_{i+1}, \dots, a_k$ where $\delta \neq \epsilon$, when the first feature of δ is $-g$,

$$\llbracket \text{im2}(a_0, a_1, \dots, a_k) \rrbracket = (\llbracket a_0 \rrbracket^{f \rightarrow g}, \llbracket a_1 \rrbracket, \dots, \llbracket a_k \rrbracket)$$

- (10)
- Examples.**





16.1.7 Polarity sensitivity

Many human languages have “negative polarity items” and other sorts of “polarity sensitive” expressions. Haspelmath [20] finds various sorts of sensitivity to negative elements in English, French, German, Russian, Latin, Hindi, Kazakh, Yakut, Swahili, Hausa, Chinese, Quechua, and many other languages. In English, for example, we have:

- (11) * John visited anyone. (with no special stress on anyone)
- (12) John didn’t visit anyone.
- (13) * Mary visited any children
- (14) No one visited any children
- (15) * It’s true that Mary visited any children
- (16) It’s not true that Mary visited any children
- (17) * Everyone thinks that Mary visited any children
- (18) No one thinks that Mary visited any children
- (19) * I know that John visited anyone.
- (20) I doubt that John visited anyone.

These examples might invite the idea that the various forms of *any NP* must have negative polarity. But that would be a mistake, since (at least some) native English speakers accept:

- (21) It’s not true that John visited anyone.
- (22) It’s not true that noone visited any children.
- (23) It’s not true that noone thinks that Mary visited any children
- (24) It’s not true that I doubt that John visited anyone.

So what Ladusaw proposes is not that *any NP* needs to appear in a negative polarity position, but rather that it needs to appear in the scope of a decreasing operator, or equivalently, that it needs to be contained in a phrase that has negative polarity. Whether the position of *any NP* itself has negative polarity is irrelevant. Ladusaw’s proposal fits the data above, but the conditions on *any NP* are actually quite tricky [15]. Even Ladusaw’s proposal is disappointing for the logician, and the tricky cases even more so, since it means that the syntax is not really marking the negative polarity positions. It comes very close, though, in many simpler sentences with one negation and limited embedding.

16.1.8 Human polarity reasoning

The examples of polarity-based inferences given in the introduction are rather simple, but they extend to more complex cases. Consider this argument from Sommers [39] and Purdy [35]:

Some horses are faster than some dogs. All dogs are faster than some men. (Implicit assumption: faster and its converse are transitive.) Therefore, some horses are faster than some men.

Extending our MG to get sentences like these, we expect this reasoning could be formalized roughly like this:

- i. (faster (some dogs)) (some horses)
- ii. (faster (some men)) (all dogs)

Note that (faster (some men)) denotes things that are faster than some men, so, by ii and polarity:

- iii. all (faster (some men)) dogs

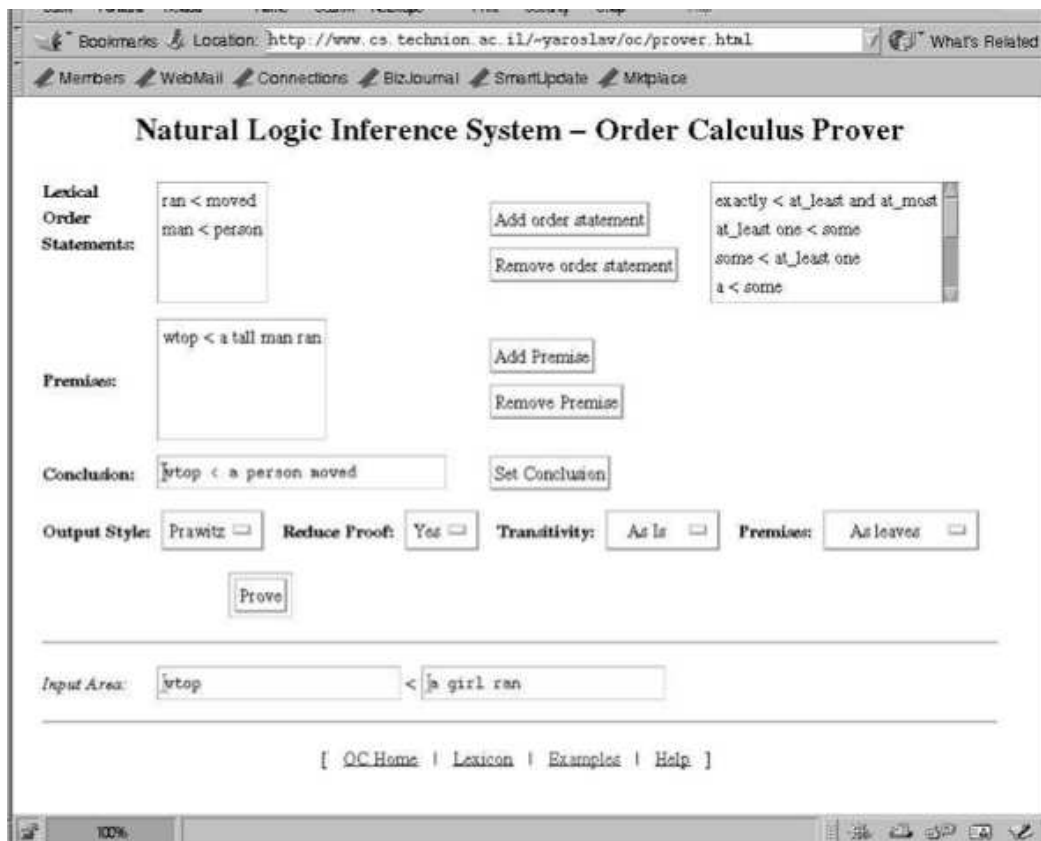
So by i, iii and polarity:

- iv. (faster (some (faster (some men)))) (some horses)

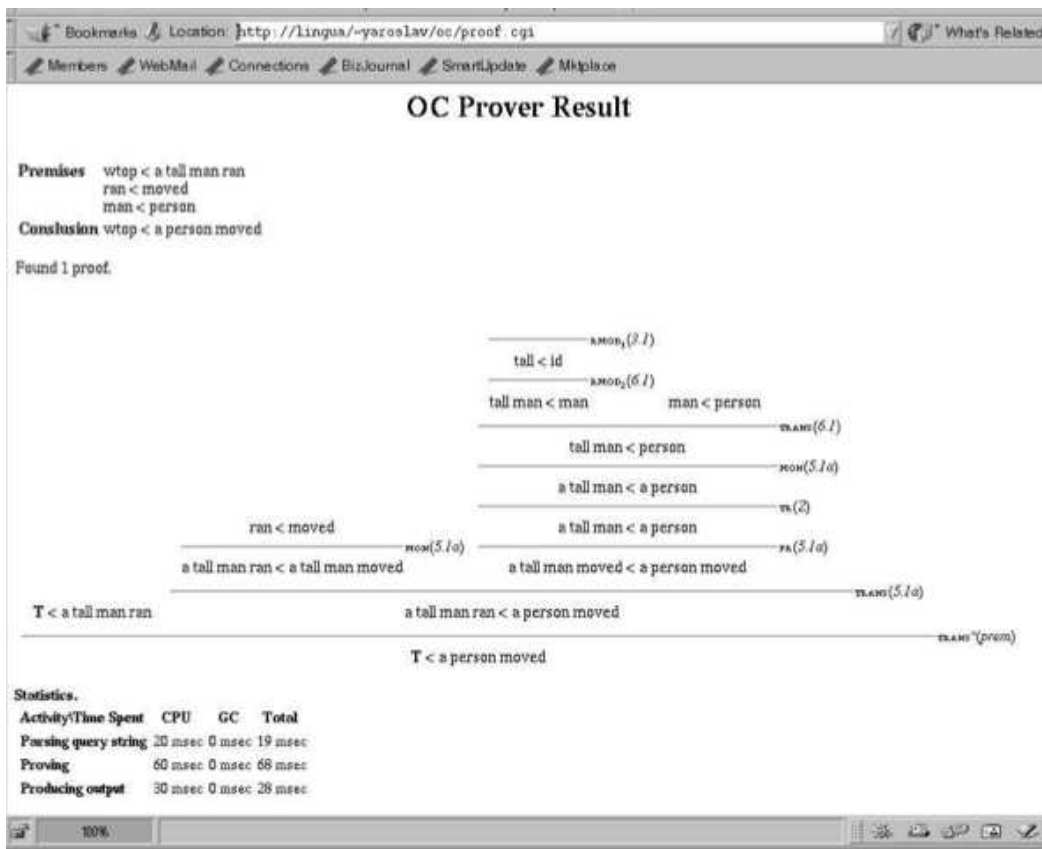
So by the transitivity of *faster*:

- v. (faster (some men)) (some horses)

Yaroslav Fyodorov describes a polarity-based proof construction tool in his thesis [13].³



³The pictured theorem prover which was available at <http://www.cs.technion.ac.il/~yaroslav/> is no longer available.

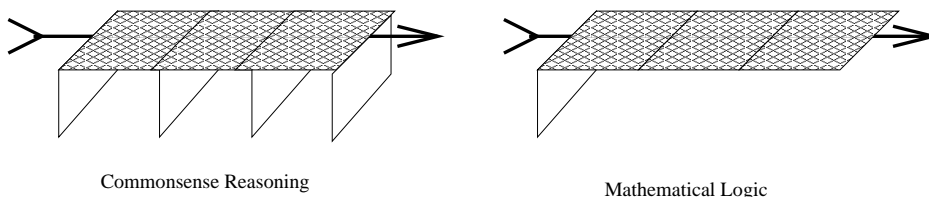


Considering the reasoning about the fast horses, notice that although we may be able to capture that argument in just a few steps, it takes some thought. The conclusion is not completely obvious (unless, perhaps, you have been practicing with puzzles like this!). So a proof-stepper is often needed if the reasoning involves more than a few steps.

Minsky makes a natural suggestion about commonsense reasoning [30, pp193,189]:

As scientists we like to make our theories as delicate and fragile as possible. We like to arrange things so that if the slightest thing goes wrong, everything will collapse at once! . . .

Here's one way to contrast logical reasoning and ordinary thinking. Both build chainlike connections between ideas. . .



Logic demands just one support for every link, a single, flawless deduction. Common sense asks, at every step, if all of what we've found so far is in accord with everyday experience. No sensible person ever trusts a long, thin chain of reasoning. In real life, when we listen to an argument, we do not merely check each separate step; we look to see if what has been described so far seems plausible. We look for other evidence beyond the reasons in that argument.

16.2 ‘Bridging inference’: probabilities, Bayesian nets, prototypes, etc.

Setting aside whether our inferences are shallow, the inferences made in conversation typically involve non-deductive reasoning, ‘guesswork’. In effect, when trying to understand a discourse, we ask ourselves, why would the speaker be saying this? The answers we find do not follow deductively from premises we are certain about, but rather are tentative and depend on contextual support. Using the terms of the quote on page 178 at the beginning of section 16.1: the crucial inferences in understanding typical conversation are *not local*.

In the class discussion of deductive reasoning, I uttered the sentence from Keenan&Faltz’85 several times:

John is a linguist.

In some cases I was naming the sentence, so what I really said on those occasions is better given by the name, the DP used to mention the sentence,

‘John is a linguist’

And on other occasions, I used the sentence but was ‘speaking hypothetically’. It was obvious to everyone that I was never asserting that anyone was a linguist. How was this so obvious?

XXX MORE COMING XXX

16.2.1 Inference to best explanation as probabilistic reasoning

XXX MORE COMING XXX

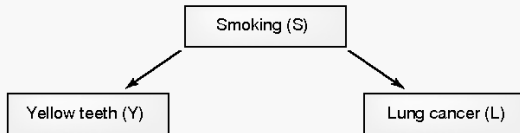
Probabilistic reasoning is unlike deductive reasoning in some fundamental respects. In particular, deductive consequence is transitive, but if p probabilistically confirms q and q probabilistically confirms r , it does not follow that p probabilistically confirms r . It is worth exploring such differences a little bit before we develop the probabilistic alternative.

XXX MORE COMING XXX

16.2.2 Bayesian nets for probabilistic reasoning

One idea is to limit the range of probabilistic dependencies that we keep track of. This is the essential idea behind Minsky's "scripts" and "frames," but I think we find a much more robust and flexible development in graphical, network models of belief. Suppose for example that all the propositions whose truth is supported by p are on a branch of a tree dominated by p . Then, at least in certain special conditions, it is possible to quickly propagate evidential adjustments through all the relevant propositions. This is the basic idea behind Bayesian nets [34, 6].

Box 1. Bayes nets, the Markov assumption and conditional independence



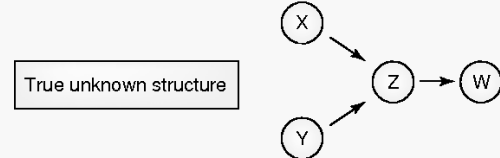
TRENDS in Cognitive Sciences

The graph above represents the claim that smoking is a cause of yellowed teeth and lung cancer, but that lung cancer does not cause yellowed teeth and yellowed teeth do not cause lung cancer. It also represents claims about the conditional probability relations among the three variables: for all values of Y , S and L (for example, all combinations of present or absent)

$$\Pr(Y, S, L) = \Pr(Y|L, S) \cdot \Pr(L|S) \cdot \Pr(S) = \Pr(Y|S) \cdot \Pr(L|S) \cdot \Pr(S)$$

where $\Pr(Y = \text{present} | L = \text{absent}, S = \text{present})$, for example, represents the probability of yellowed teeth among smokers without lung cancer. The first equality is necessarily true, but the second is an assumption, the Markov factorization, which says that the joint distribution of all variables is equal to a product of the conditional distributions of each variable on its parents in the graph. The Markov factorization is equivalent, in this example, to the claim that $\Pr(Y|S, L) = \Pr(Y|S)$.

Box 3. Comparing Bayesian learning and constraint-based learning of Bayes nets



TRENDS in Cognitive Sciences

Bayesian learning

- (1) Prior probability distribution $\Pr(G; \theta)$ over all directed acyclic graphs G and probability distributions θ on the variables (vertices in G), with a Markov factorization for G .
- (2) Likelihood function $L(D; G, \theta)$ giving the probability of the observations D conditional on the truth of G , θ .
- (3) Compute the probability of any graph G conditional on the data by using Bayes Theorem and integrating over θ

$$\Pr(G|D) = \frac{\int \Pr(G; \theta) L(D; G, \theta) d\theta}{\Pr(D)}$$

- (4) Find the graphs G such that for all other graphs G^* , $\Pr(G|D) \geq \Pr(G^*|D)$

Figures from Glymour's 2003 survey article [17, 16]

Note that these nets limit the inferential dependencies with a graph that is assumed to be acyclic, which may be realistic when the dependencies are "causal" and not self-reinforcing, but otherwise seems unrealistic. (Cf. Fodor's critique of these approaches in [11, §4].) Nevertheless, it may be a useful starting point.

16.3 Discourse dynamics

[40, 22, 44, 48]

Exercises:

References

- [1] ACKERMAN, N. L., FREER, C. E., AND ROY, D. M. On the computability of conditional probability. *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011* (2011).
- [2] BARKER, C., AND JACOBSON, P. Introduction: Direct compositionality. In *Direct Compositionality*, C. Barker and P. Jacobson, Eds. Oxford University Press, NY, 2007.
- [3] BERNARDI, R. *Reasoning with Polarity in Categorical Type Logic*. PhD thesis, University of Utrecht, Utrecht, 2002.
- [4] BRAINE, M. The ‘natural logic’ approach to reasoning. In *Reasoning, Necessity and Logic*, W. Overton, Ed. Erlbaum, Hillsdale, NJ, 1990.
- [5] CHOMSKY, N. *Knowledge of Language*. Praeger, NY, 1986.
- [6] DARWICHE, A., AND GINSBERG, M. L. A symbolic generalization of probability theory. In *Proceedings of the Tenth National Conference on Artificial Intelligence* (Menlo Park, California, 1992), P. Rosenbloom and P. Szolovits, Eds., AAAI Press, pp. 622–627.
- [7] DOUVEN, I. Further results on the intransitivity of evidential support. *Review of Symbolic Logic* 4, 4 (2011), 487–497.
- [8] FALK, R. Probabilistic reasoning is not logical. *Mathematics Magazine* 81, 4 (2008), 268–275.
- [9] FODOR, J., GARRETT, M., WALKER, E., AND PARKES, C. Against definitions. *Cognition* 8 (1980), 263–367.
- [10] FODOR, J. A. *The Mind Doesn’t Work That Way: The Scope and Limits of Cognitive Psychology*. Bradford, Cambridge, Massachusetts, 2001.
- [11] FODOR, J. A. *LOT2: The Language of Thought Revisited*. Clarendon Press, NY, 2008.
- [12] FREGE, G. Gedankengefüge. *Beträge zur Philosophie des deutschen Idealismus* 3 (1923), 36–51. Translated and reprinted as ‘Compound thoughts’ in *Mind* 72(285): 1–17, 1963.
- [13] FYODOROV, Y. *Implementing and Extending Natural Logic*. PhD thesis, Israel Institute of Technology, 2002. <http://www.cs.technion.ac.il/~yaroslav/>.
- [14] FYODOROV, Y., WINTER, Y., AND FRANCEZ, N. Order-based inference in natural logic. *Research on Language and Computation* (2003). Forthcoming.
- [15] GIANNAKIDOU, A. ‘only’, emotive factive verbs, and the dual nature of polarity dependency. *Language* 83, 3 (2006), 575–603.
- [16] GLYMOUR, C. Android epistemology for babies: Reflections on words, thoughts and theories. *Synthese* 122 (2000), 53–68.
- [17] GLYMOUR, C. Learning, prediction and causal Bayes Nets. *Trends in Cognitive Science* 7, 1 (2003), 43–47.
- [18] GOODMAN, N., MANSINGHKA, V., ROY, D., BONAWITZ, K., AND TARLOW, D. Church: a language for generative models. In *Proceedings of the Twenty-Fourth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-08)* (Corvallis, Oregon, 2008), AUAI Press, pp. 220–229.
- [19] HAENNI, R., ROMELJN, J.-W., WHEELER, G., AND WILLIAMSON, J. *Probabilistic Logics and Probabilistic Networks*. Springer, NY, 2011.
- [20] HASPELMATH, M. *Indefinite pronouns*. Clarendon, Oxford, 1997.
- [21] HENDRIKS, H. *Studied Flexibility: Categories and types in syntax and semantics*. PhD thesis, Universiteit van Amsterdam, 1993.
- [22] HOBBS, J. R. *Discourse and Inference*. in preparation, ISI/University of Southern California, 2004.
- [23] HODGES, W. Formal features of compositionality. *Journal of Logic, Language and Information* 10 (2001), 7–28.
- [24] HORWICH, P. *Meaning*. Oxford University Press, Oxford, 1998.
- [25] KEENAN, E. L. Further beyond the Frege boundary. In *Quantifiers, Logic, and Language*, J. van der Does and J. van Eijck, Eds. CSLI Publications, Amsterdam, 1996.
- [26] KEENAN, E. L., AND FALTZ, L. M. *Boolean Semantics for Natural Language*. Reidel, Dordrecht, 1985.
- [27] KEENAN, E. L., AND STABLER, E. P. *Bare Grammar: Lectures on Linguistic Invariants*. CSLI Publications, Stanford, California, 2003.
- [28] KOBELE, G. M. Inverse linking via function composition. *Natural Language Semantics* 18 (2010), 183–196.
- [29] MARX, M. Relation algebra with binders. *Journal of Language and Computation* 11, 5 (2000), 691–700.
- [30] MINSKY, M. L. *The Society of Mind*. Simon and Schuster, NY, 1988.

- [31] MOSS, L. Natural language, natural logic, natural deduction. *Forthcoming* (2004). Indiana University.
- [32] PARIS, J. B. *The Uncertain Reasoner's Companion*. Cambridge University Press, NY, 1994.
- [33] PARSONS, T. *An Introduction to Symbolic Logic*. UCLA Lecture Notes, 2009.
- [34] PEARL, J. *Probabilistic Reasoning in Intelligent Systems: Patterns of Plausible Inference*. Morgan Kaufmann, San Francisco, 1988.
- [35] PURDY, W. C. A logic for natural language. *Notre Dame Journal of Formal Logic* 32 (1991), 409–425.
- [36] ROY, D. *Computability, Inference and Modeling in Probabilistic Programming*. PhD thesis, Massachusetts Institute of Technology, 2010.
- [37] SANCHEZ-VALENCIA, V. *Studies on Natural Logic and Categorical Grammar*. PhD thesis, University of Amsterdam, Amsterdam, 1991.
- [38] SMULLYAN, R. M. *Theory of Formal Systems*. Princeton University Press, Princeton, 1961.
- [39] SOMMERS, F. The calculus of terms. *Mind* 79 (1987), 1–39.
- [40] STEDE, M., AND SCHMITZ, B. Discourse particles and discourse functions. *Machine Translation* 15, 1-2 (2000), 125–147.
- [41] TARSKI, A. On the calculus of relations. *Journal of Symbolic Logic* 6 (1941), 73–89.
- [42] TARSKI, A. *Introduction to logic and to the methodology of deductive sciences*. Oxford University Press, NY, 1946.
- [43] TARSKI, A., AND GIVANT, S. *A formalization of set theory without variables*. American Mathematical Society, Providence, Rhode Island, 1987.
- [44] TOBOADA, M. Discourse markers as signals (or not) of rhetorical relations. *Journal of Pragmatics* 38, 4 (2006), 567–592.
- [45] WESTERSTÄHL, D. On the compositional extension problem. *Journal of Philosophical Logic* 33, 6 (2004), 549–582.
- [46] WILLIAMSON, J. An objective Bayesian account of confirmation. In *Explanation, Prediction, and Confirmation*, D. Dieks, W. J. Gonzalez, and S. Hartmann, Eds. Springer, NY, 2011, pp. 53–81.
- [47] WITTGENSTEIN, L. *Philosophical Investigations*. MacMillan, NY, 1958. This edition published in 1970.
- [48] WOLF, F., AND GIBSON, E. Representing discourse coherence: A corpus-based study. *Computational Linguistics* 31, 2 (2005), 249–287.

Chapter 17 Around syntax: Selecting the best parse

A wealth of experimental findings document that the structural and semantic analyses that comprehenders assign at the point of ambiguity, mid-sentence, are determined by these lexical factors, including the probability that a given verb takes particular complements, as well as the semantic fit of constituents into the intended roles assigned by the verb ... The probabilistic recovery of structure is sensitive to other contingencies as well. In particular, the referential implications of these representations are computed in real-time and serve as an important top-down constraint ... In a similar interactive fashion, prosodic evidence is also weighed by the listener in real time... The picture emerging from these data is one in which the recognition of the word within a sentence automatically triggers linguistic representations at multiple levels. This triggering is probabilistic in nature: Using all evidence in hand, a listener is engaged in a kind of guessing-game in which the linguistic procedures that gave rise to the utterance are recovered. Referential implications are also computed and, when possible, used to constrain the listener's syntactic hypotheses. Finally, local ambiguity is the norm in real-time language comprehension. It is close to impossible to find an utterance of even 10 words in length, and of modest conceptual content, that cannot be interpreted in more than one way at some point during its hearing. Computational linguists recognized this as soon as they started to implement parsers designed to handle natural text ... The overall implication is that as a constant matter in the course of understanding, the listener must rapidly evaluate competing analyses at one or more levels of representation, choosing among them as they arise, in response to differences in interpretive accessibility at each such level. the brain processes it. – John C. Trueswell & Lila R. Gleitman [6]

17.1 Discourse cues for parse selection and understanding

[3]
[1]
[5]

17.2 Prosody and other phonetic cues

We advocate an approach to prosody that tries to reconcile the existence of grammatical constraints with the enormous variability in pronunciations of a sentence. . . . the listener assumes that a speaker using short prosodic phrases (lots of prosodic boundaries) will not omit a prosodic boundary at the largest syntactic break in the utterance for no reason. . . Prosody is therefore central to understanding spoken language, and we speculate that it might supply the basic skeleton that allows us to hold an auditory linguistic sequence in memory while the brain processes it.

– Lyn Frazier, Katy Carlson & Charles Clifton Jr. [2]

17.3 Grammatical illusions

[4]

Exercises:

References

- [1] ASHER, N., AND LASCARIDES, A. *Logics of Conversation*. Cambridge University Press, NY, 1997.
- [2] FRAZIER, L., CARLSON, K., AND CLIFTON JR, C. Prosodic phrasing is central to language comprehension. *Trends in Cognitive Sciences* 10, 6 (206), 244–249.
- [3] HOBBS, J. R. *Discourse and Inference*. in preparation, ISI/University of Southern California, 2004.
- [4] PHILLIPS, C., WAGERS, M., AND LAU, E. Grammatical illusions and selective fallibility in real-time language comprehension. *Language and Linguistics Compass forthcoming* (2010).
- [5] SINGER, M. Inference processing in discourse comprehension. In *Oxford Handbook of Psycholinguistics*, G. Gaskell, Ed. Oxford University Press, Oxford, 2007, pp. 343–359.
- [6] TRUESWELL, J., AND GLEITMAN, L. Learning to parse and its implications for language acquisition. In *Oxford Handbook of Psycholinguistics*, G. Gaskell, Ed. Oxford University Press, Oxford, 2007, pp. 343–359.

Chapter 18 Before syntax: Learning the language

*So is the calculus something that we adopt arbitrarily?
No more so than the fear of fire, or the fear of a raging man coming at us.*
– Ludwig Wittgenstein [4, I.#68]

18.1 Exact learning

18.2 Probabilistic learning

References

- [1] CLARK, A., AND YOSHINAKA, R. Beyond semilinearity: Distributional learning of parallel multiple context-free grammars. In *Proceedings of the 11th International Conference on Grammatical Inference, ICGI 2012* (2012).
- [2] FORSTER, M. *Wittgenstein on the Arbitrariness of Grammar*. Princeton University Press, Princeton, New Jersey, 2004.
- [3] KATO, Y., SEKI, H., AND KASAMI, T. Stochastic multiple context-free grammar for RNA pseudoknot modeling. In *Proceedings of the Eighth International Workshop on Tree Adjoining Grammar and Related Formalisms, TAG+’06* (Morristown, New Jersey, USA, 2006), Association for Computational Linguistics, pp. 57–64.
- [4] WITTGENSTEIN, L. *Philosophical Grammar*. University of California Press, Los Angeles, 1979. Notes written 1930-1934. Trans. Anthony Kenny.
- [5] YOSHINAKA, R. Efficient learning of multiple context-free languages with multidimensional substitutability from positive data. *Theoretical Computer Science* 412, 19 (2011), 1821–1831.

Chapter 19 Summary and key open problems

The most important question for a scientist is: what are the key open problems? All the work in these notes aims to get us to the next important questions. The trick to making progress in understanding things is to find aspects of the domain which are not just significant but ones that are likely to yield to study. The latter desiderata is very much harder than the former: many questions we are interested in are ones that we do not know how to gain insight into. So it is clear that this section reflects the personal perspective of the author even more than the preceding material does. Let's try to briefly summarize the main results of all the work reviewed earlier, with the goal of showing how they bring us to the next important, feasible questions.

19.1 The situation in syntax and phonology

One striking feature of these lecture notes is that no grammar of any particular language is developed to any extent. Before explaining why that might be the case, let's consider a striking feature of the field: It is not easy to find a significant body of material that is accepted by all or even most reasonable and well-informed linguists. Several handbooks of linguistics have been published recently, and I cannot help but be appalled at how insubstantial the chapters on syntax are. There are also recent handbooks devoted entirely to syntax, and large multi-volume surveys. These present a great wealth of information about a rather wide range of languages, but it is hard to find in them a substantial unifying vision of what human syntax is. The lack of consensus is widely noted, and leads some to despair about the field, but that reaction is in appropriate! In a recent paper, Steedman and Baldrige'11 ask "Why are there so many theories of grammar around these days?" and immediately answer: "Sometimes the similarities are disguised by the level of detail at which the grammar is presented." That leaves us the problem of discerning common assumptions lurking behind irrelevant notational details, and this requires appropriately careful comparison of theoretical mechanisms. This is more abstract than is usually required for good linguistic work, and so the comparisons remain largely undone, except in the slightly more abstract and more formal kind of work that we see, for example, in the traditions of categorial grammar and tree adjoining grammar. That work has revealed some substantial points of consensus. Joshi's "mildly context sensitive hypothesis", which we have discussed, is an attempt to formulate a substantial claim that is relatively theory neutral. Similarly Kaplan and Kay'94, mention as a great virtue of their finite state perspective on phonology that it can remain constant "even under radical changes in the theory."

This is the rational strategy, I think: use formal representations with well-understood connections to a reasonable range of mainstream theories.

with commitments that are either explicitly acknowledged as preliminary simplifications or

We can consider proposals that apply to a wide range of grammars, a range that can reasonably be assumed to include the kinds of grammars realized in human speakers. This project rests on work on the details, but abstracts away from detailed to proposals to make much weaker, much more plausible proposals.

XXXX

My favorite example of getting lost in the details is the emphasis on traces in assessments of Chomskian syntax.
XXX

19.2 The situation in parsing

Parsing as possible: [1]

Best idea: Some version of GLC MGC

Search: Adaptive as Hale suggests

19.3 The situation above and below parsing: Phonetics and discourse

While phonology looks to be of a piece with syntax, phonetics is different.

Similarly, discourse.. XX

19.4 The situation around parsing: Defining ‘best parse’

19.5 The situation before parsing: acquisition

References

- [1] FODOR, J. A. *The Mind Doesn't Work That Way: The Scope and Limits of Cognitive Psychology*. Bradford, Cambridge, Massachusetts, 2001.
- [2] KAPLAN, R., AND KAY, M. Regular models of phonological rule systems. *Computational Linguistics* 20 (1994), 331–378.
- [3] STEEDMAN, M., AND BALDRIDGE, J. Combinatory categorial grammar. In *Non-Transformational Syntax: Formal and Explicit Models of Grammar*, R. Borsley and K. Börjars, Eds. Wiley-Blackwell, Boston, 2011.

Index

- =, python assignment, 5
- ==, python equality, 5
- \vdash , derives in logic, 2
- \models , models in logic, 2
- 2-register machines, 2

- A* search, 50
- A-movement, 36
- abacus, 2
- Abels, Klaus, 164
- Abney, Steven P., 73
- adjunction, 139, 152, 165
 - late, 153, 167
- agreement, 36
 - in MGs, 153
- Aho, Alfred V., 63, 86
- Altmann, Gerry T. M., 64
- Angelov, Krasimir, 167
- arc-eager GLC, 67, 73
- Aristotle, syllogistic, 181
- Asher, Nicholas, 193

- Baldrige, Jason, 197
- Bayesian networks, 189
- Berwick, Robert C., 1, 63
- Bever, Thomas G., 34
- binding, 36
- Boëthius, Anicius Manlius Severinus, 3
- Bošković, Željko, 164
- Bobaljik, Jonathan David, 164
- breadth-first search, 50
- Brody, Michael, 164
- Brosgol, Benjamin Michael, 73
- Buell, Leston, 126
- Bye, Patrik, 173

- Carlson, Katy, 193
- categorial grammar, 140
- center-embedding, 34
- Chomsky normal form CFGs, 83
- Chomsky, Noam, 34, 96, 162, 177
- Church, Alonzo, 2
- Church-Turing hypothesis, 2
- CKY recognizer
 - for CFGs, 83
 - for MGs, 110
 - for MGs with head movement, 141
- cleft construction, 163

- click studies, 34
- Clifton, Charles, Jr., 50, 193
- complement, in MG structures, 96
- completeness
 - in logic, 2
 - in recognition, parsing, 2
- constant growth, 93, 166
- control, 150
- copy theory of movement, 162
- copying, see reduplication
- Cuetos, Fernando, 64
- Culicover, Peter, 4
- cycles, in a CFG, 83

- datalog, 115
- Demers, Alan J., 73, 75
- discourse, 190
 - cues for parse selection, 193
- dynamic programming, 86
- dynamic semantics, 190

- Earley recognizer
 - for CFGs, 87
 - for MGCs, 167
 - for MGs, 115
- Euclid of Alexandria, 5
- Euclid's algorithm for gcd, 5

- Faltz, Leonard M., 178
- feature checking, in minimalist grammar, 96
- Fodor, Janet, 64
- Fodor, Janet Dean, 177
- Fodor, Jerry A., 1, 34, 177, 189
- frame, Minsky's, 189
- Frank, Robert, 171
- Franks, Steven, 164
- Frazier, Lyn, 50, 64, 193
- Frege, Gottlob, 177
- Frey, Werner, 140, 153

- Gärtner, Hans-Martin, 140, 153
- garden path, 64
- garden path effects, 30
- Garrett, Merrill F., 34, 177
- generalized left corner, GLC, 73
- Gibson, Edward, 64
- Givant, Steven, 182
- Gleitman, Lila R., 193

- global ambiguity, 64
- Glymour, Clark, 189
- Graf, Thomas, 1, 34
- grammatical illusions, 193
- Groat, Erich, 164
- Grohmann, Kleantes K., 164

- Hale, John, 50, 197
- Halle, Morris, 34
- Harkema, Henk, 111, 115
- Harnish, Robert M., 1
- Hayes, Bruce, 174
- head movement, 36
- heap, 45
- Hemforth, Barbara, 64, 71
- Hiraiwa, Ken, 164
- Hobbs, Jerry R., 193
- Hopcroft, John E., 33
- Hornstein, Norbert, 164
- Horwich, Paul, 177
- Hunter, Tim, 140

- Idsardi, William, 164
- inclusiveness condition, 162

- Johnson, Mark, 73
- Joshi, Aravind, 1, 93, 166, 197

- k-best parsing, 45
- Kallmeyer, Laura, 111, 115
- Kanazawa, Makoto, 115
- Kandybowicz, Jason, 163
- Kaplan, Ronald, 34, 171, 197
- Kay, Martin, 34, 171, 197
- Kayne, Richard S., 160
- Keenan, Edward L., 1, 178
- Khoussainov, Bakhadyr, 33
- Kleene's theorem, 33
- Knuth, Donald E., 63
- Kobele, Gregory M., 1, 34, 164
- Konieczny, Lars, 64
- Koopman, Hilda, 1, 160, 163
- Kracht, Marcus, 1, 166

- Landau, Idan, 164
- Lascarides, Alex, 193
- late adjunction, see adjunction
- late closure, 64
- lattice, of GLC recognizers, 75
- Lecomte, Alain, 140
- Lefebvre, Claire, 164
- licensees, in MG, 95, 110
- licensors, in MG, 95, 110
- Lidz, Jeffrey, 164
- Lin, Ying, 174
- Ljunglöf, Peter, 115, 167
- local ambiguity, 63
- logic
 - completeness, 2
 - defined, 2
 - soundness, 2
- logic, definition, 179

- Maier, Wolfgang, 115
- Manaster-Ramer, Alexis, 164
- Manning, Christopher D., 71
- Marcus, Mitchell P., 63
- Marx, Maarten, 182
- Mateescu, Alexandru, 101
- MCFG, multiple context free grammar, 113
- memoization, 87
- MGC, minimalist grammar with copying, 164
- Michaelis, Jens, 1, 114, 153, 166
- Miller, George M., 34
- Mitchell, Don C., 64
- model, in logic, 2
- Montague, Richard, 2
- Moortgat, Michael, 140
- Morrill, Glyn, 140
- Moschovakis, Yiannis N., 2
- multidominance theory of movement, 163
- multiple wh-movement, 153

- natural logic, 178, 180
- negative polarity items, 185
- Nerode, Anil, 33
- Nerode-Myhill theorem, 33
- Nkemnji, Michael A., 164
- no tampering condition, 162
- non-canonical parsing, 63
- noun compound, 165
- Nozohoor-Farshi, Rahman, 63
- Nunes, Jairo, 164

- O'Neil, John, 164
- Onambele Manga, Christophe L., 153
- optimality theory, OT, 34

- parallel implementation, 50
- parameters
 - directionality, 153
- Parkes, C.H., 177
- parser
 - completeness, 2
 - defined, 2
 - soundness, 2
- Pereira, Fernando, 64
- Pesetsky, David, 164
- Pickering, Martin J., 64
- PMCFG, parallel multiple context free grammar, 166
- polarity-based deduction, 181
- Pollard, Carl, 114
- predicate calculus, 183
- prefix property, 87
- priority queue, 45

- probabilistic inference, 188
- propositional calculus, 179
- prosody
 - cues for parse selection, 193
- pydatalog, 115
- Pythagoras of Samos, 3
- python, 2
 - introduced, 4
 - NLTK library, 27

- raising to object, 146
- raising to subject, 144
- reanalysis, in psycholinguistics, 50, 87
- recognizer
 - completeness, 2
 - defined, 2
 - soundness, 2
- reduplication
 - in English, 4
 - syntactic, 161
- Reisch, Gregor, 3
- relation calculus, 181
- Retoré, Christian, 140
- Richards, Norvin, 164
- Ritchie, Graeme, 73
- Ross, John R., 140
- Runner, Jeffrey, 164

- Sag, Ivan, 140
- Sakarovitch, Jacques, 33
- Salomaa, Arto, 33, 101
- Salvati, Sylvain, 140
- Satta, Giorgio, 171
- Schueler, Dave, 128, 155
- Schütze, Hinrich, 71
- Seki, Hiroyuki, 111, 114
- selector features, in MG, 95, 110
- shortest move condition, SMC, 96, 140
- Singer, Murray, 193
- slash dependencies, 109
- SMC, see shortest move condition
- Sommers, Fred, 185
- soundness
 - in logic, 2
 - in recognition, parsing, 2
- specifier, in MG structures, 96
- Sportiche, Dominique, 1, 141
- Staub, Adrian, 50
- Steedman, Mark, 64, 140, 197
- Stjepanovic, Sandra, 164
- subjacency, 141
- Svenonius, Peter, 173
- Swahili, relative clauses, 126
- Szabolcsi, Anna, 140
- Szymanski, Thomas G., 63

- Tarski, Alfred, 179, 182

- tense
 - in English, 4
 - logic, 4
- Thatcher, James W., 34
- trace theory of movment, 162
- Trueswell, John C., 193
- Turing, Alan M., 2
- type-logical grammar, 140

- Ullman, Jeffrey D., 33, 63, 86

- van Gompel, Roger P. G., 64
- verbal cleft, 163
- Vermaat, Willemijn, 140
- Vijay-Shanker, K., 109

- Walker, E.C.T., 177
- Weinberg, Amy S., 63
- Weir, David, 109
- Wilder, Chris, 164
- Williams, John H., 63
- Wittgenstein, Ludwig, 177, 195

- Yu, Kristine M., 1, 174