Computational Linguistics:
Structure and Processing

Edward P. Stabler
Lx185/208 draft: 2008-06-27 10:03

Contents

§0 Introduction

0.0 Questions

(0) **The basic question.** How do people use and acquire human languages?

(1) Traditional strategy: list what speakers of language X know

- squib, n. [skwIb]

  A common species of firework, in which the burning of the composition is usually terminated by a slight explosion.
  Something that fails ignominiously to satisfy the expectations aroused by it; an anti-climax, a disappointment.
  A short composition of a satirical and witty character; a lampoon.

  ⇒ In technical literature, a squib is just a short essay on a specific topic.

- (games)

  I'm a little _ _ _ _ _ _, short and stout. Carroll character: _ _ _ _ _

  "Is the computer going to be able to solve the clues involving puns and wordplay? I don't think so," Shortz wrote in an introduction to a volume of The New York Times daily crossword puzzles. He gave examples of clues he felt computers would miss, such as "Event That Produces Big Bucks" referring to "Rodeo," "Pumpkin-colored" translating as "Orange," or "It Might Have Quarters Downtown" meaning "Meter."

- (headlines) Enraged cow injures farmer with ax.

  (On the wrapper of my new socks:) Fits all sizes.

- (dialects) (Mos Def, "Tell the truth") Man u hear this bullshit they be talkin...

(2) **(Q)** How could any device acquire, recognize and produce a human language?

- distinguish the recognition problem (and what we know at a given time) from the learning problem
- Instead of trying to list lexicon and grammar, define their fundamental properties.

  Once we have an idea of what they are, we can aim to understand how they are acquired.

0.1 Computational arguments in linguistic theory

- 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. [92]

(3) ˈtɛlɛˌɡɹæf (in isolation, primary stress on 1st syllable, secondary on 3rd)

(4) ˌtɛlɛˈɡɹæf (in *telegraphic*, primary stress on 3rd syllable, secondary on 1st)

(5) tɛlˈɛˌɡɹæf (in *telegraphy*, primary stress on 2nd syllable)

- Chomsky and Halle 1968

  > Regular variations such as this are not matters for the lexicon, which should contain only idiosyncratic properties of items, properties not predictable by general rule. The lexical entry for *telegraph* must contain just enough information for the rules of English phonology to determine its phonetic form in each context; since the variation is fully determined, the lexical entry must contain no indication of the effect of context on the phonetic form. In fact, as we shall see, the lexical representation for the word *telegraph* should be (6), where each of the symbols *t,e,...* is to be understood as an informal abbreviation for a certain set of phonological categories (distinctive features):
  >
  > (6) +tele+græf+.
  >
  > Thus the lexical representation is abstract in a very clear sense; it relates to the signal only indirectly, through the medium of the rules of phonological interpretation that apply to it as determined by its intrinsic abstract representation and the surface structures in which it appears. [53, p.12]

- Kenstowicz 1994

  > According to Miller and Gildea (1987), in the course of normal development a child learns a vocabulary of some 80,000 lexical items. Many adults have lexicons of much greater size...While a groan and a giggle are readily distinguished from one another, no language encodes its vocabulary in such gross vocal wholes. Think of the cognitive hardware that would be required to perceive, learn, and put into operation 80,000 different gestures. A much more efficient system would stipulate a small number of basic atoms and some simple method for combining them to produce structured wholes. [147, p.13]

- Fromkin et al 2000

  > Minimally, a lexical entry will contain information that is sufficient to distinguish its surface realization from that of any other form which, in the judgement of the speaker, is realized distinctly in the same circumstances...The entries of *write* and *ride* must be distinct, since they are realized as [raɪt] and [raɪd] respectively under most circumstances. Therefore the voicing values in the final stops of *write* and *ride* will have to be part of the lexical entries, as it is these voicing values that cause the speaker to realize these two forms distinctly. [95, p.602]

§1 Time and sequence

1.0 Denoting linguistic objects

(0) Insert the minimum number of quotation marks to make these strings grammatical and sensible.

> The teacher's name is Ed.
>
> The is not always a determiner.
>
> I want to put hyphens between the words fish and and and and and chips in my fish and chips sign.

Then, here's a harder one (for this one, it helps to add commas too).

> Wouldn't the sentence I want to put a hyphen between the words fish and and and and and chips in my fish and chips sign have looked cleaner if quotation marks had been placed before fish and between fish and and and and and and and and and and and and and and and and and and and and chips as well as after chips?

We solve a problem like this every time we interpret a spoken sentence.

1.1 Order and sequence

(1) 1+1=2. Introducing structures and recursive definitions. (draw pictures)

```
(* we define a new type for natural numbers *)
type nn = Z | S of nn;;

(* e.g. *)
Z;;
S Z;;
S (S Z);;

let iszero x = match x with
    Z -> true
  | S _ -> false;;

let rec sum a b = match a with
    Z -> b
  | S x -> S (sum x b);;

(* e.g. *)
sum (S Z) (S Z);;
sum Z (S (S (S Z)));;
sum (S (S Z)) (S (S (S Z)));;
```

(2) 2*2=4.

```
let rec times a b = match a with
    Z -> Z
  | S x -> sum b (times x b);;

(* e.g. *)
times (S (S Z)) Z;;
times (S (S Z)) (S (S (S Z)));;
```

(3) Know what's happening. If you don't, you can print out clues.

```
let rec sumP a b = match a with
    Z -> begin print_endlineline "finally"; b; end
  | S x -> begin print_string "blip "; S (sumP x b); end;;

(* e.g. *)
sumP (S Z) (S Z);;
sumP (S (S Z)) (S (S (S Z)));;

let rec timesP a b = match a with
    Z -> begin print_string "ding "; Z; end
  | S x -> begin print_string "zzz "; sumP b (timesP x b); end;;

(* e.g. *)
timesP (S (S Z)) Z;;
timesP (S (S Z)) (S (S (S Z)));;
```

(4) Sequences. That is, lists. Instead of (1+1)=2 we have (append ["hi"] ["ho"]) = ["hi";"ho"]

```
(* we can define a new type for lists *)
type 'aa ls = Nil | Cons of 'a * 'aa ls;;

Nil;;
Cons ("sings", Nil);;
Cons ("Mary", Cons ("sings",Nil));;
Cons (1, Cons (0,Nil));;
Cons (1, Cons ("a",Nil));;

let rec append x y = match x with
    Nil -> y
  | Cons (h,t) -> Cons (h,append t y);;

(* e.g. *)
append (Cons ("how", Nil)) (Cons ("Mary", Cons ("sings", Nil)));;
append (Cons (1, Nil)) (Cons (1, Cons (0, Nil)));;

(append (Cons ("hi", Nil)) (Cons ("ho", Nil))) = (Cons ("hi", Cons ("ho",Nil)));;

let rec printStringList x = match x with
    Nil -> ()
  | Cons (h,t) -> begin print_string h; print_string " "; printStringList t; end;;

printStringList (Cons ("my", (Cons ("how", Cons ("Mary", Cons ("sings", Nil))))));;
```

(5) Lists in Ocaml. [] is Nil, and :: is Cons, but usually we don't show either one!

```
(* the OCaml notation for lists is easier to read *)
"sings"::[];;
"Mary"::"sings"::[];;
1::"sings"::[];;

List.mem "sings" ["Mary";"sings";"songs"];;
List.mem "song" ["Mary";"sings";"songs"];;

let rec member e list = match list with
    [] -> false
  | h::t -> if h=e then true else member e t;;

(* e.g. *)
member "sings" ["Mary";"sings";"songs"];;
member "song" ["Mary";"sings";"songs"];;

(* here is list "addition": appending *)
let rec append x y = match x with
    [] -> y
  | h::t -> h::(append t y);;

(* e.g. *)
append [] ["songs";"daily"];;
append ["Mary";"sings"] ["songs";"daily"];;

(append ["hi"] ["ho"]) = ["hi";"ho"];;
```

(6) Noam Chomsky, *Aspects of the Theory of Syntax* [45, p.8]:

> But the fundamental reason for this inadequacy of traditional grammars is a ... technical one. Although it was well understood that linguistic processes were in some sense "creative," the technical devices for expressing a system of recursive processes were simply not available until much more recently. In fact, a real understanding of how a language can (in Humboldt's words) "make infinite use of finite means" has developed only within the last thirty years, in the course of studies in the foundations of mathematics. Now that these insights are readily available it is possible to return to problems that were raised, but not solved, in traditional linguistic theory...

§2 Languages and grammars

2.0 The Chomsky hierarchy of rewrite grammars

(0) We assume our alphabets Σ are finite and nonempty.

(1) Σ^* is the set of all sequences of elements of Σ, including the empty sequence ϵ

(2) By a "language", we mean any subset of Σ^*.

(3) How many languages are there?

(4) A **rewrite grammar** $G = \langle Cat, \Sigma, \rightarrowtail, S\rangle$ has these parts:

Cat	finite set of Categories
Σ	finite nonempty set of vocabulary items disjoint from C
$\rightarrowtail \subseteq V^*CatV^* \times V^*$	a finite set of 'rules', where $V = Cat \cup \Sigma$
$S \in Cat$	start category

For each rewrite grammar, we define the relation $\Rightarrow \subseteq V^* \times V^*$ as follows: $x \Rightarrow z$ iff there are $t, u, v, w \in V^*$ such that $x = tuv$, $u \rightarrowtail w$, and $z = twv$.

(5) We let $\Rightarrow^*$ be the reflexive, transitive closure of $\Rightarrow$.

(6) The *language* defined by the grammar $L(G) = \{x \in \Sigma^* |\ S \Rightarrow^* x\}$.

(7) How many grammars are there?

(8) Most languages do not have grammars.

(9) Chomsky hierarchy:

a. A **context sensitive (CS) grammar** is rewrite grammar in which all the rules have one of the following forms:

$uAv \rightarrowtail uwv$	where $A \in Cat,\ u, v, w \in V^*, w \in V^+$
$S \rightarrowtail \epsilon$	if S is not on the right side of any rule.

b. A **context free (CF) grammar** is a rewrite grammar in which all the rules have the following form:

$$A \rightarrowtail u \qquad \text{where } C \in Cat,\ u \in V^*.$$

c. A **(right linear) regular (Reg) grammar** is a rewrite grammar in which all the rules have one of the following forms:

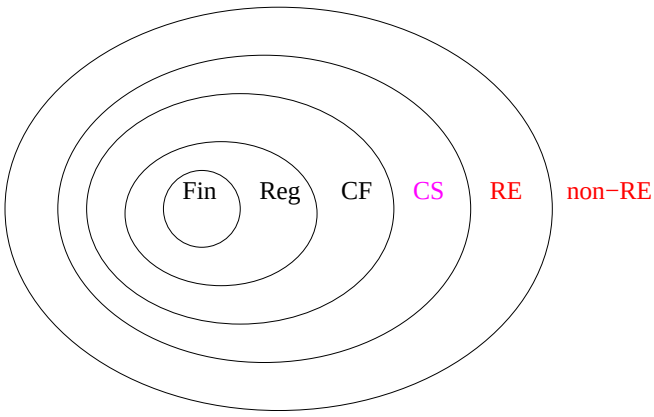
$C \rightarrowtail \epsilon$	where $C \in Cat$ and ϵ is the empty sequence
$C \rightarrowtail a\ D$	where $C, D \in Cat$ and $a \in \Sigma$.

d. A **list grammar** is a rewrite grammar in which all the rules have the following form:

$$C \rightarrowtail u \qquad \text{where } C \in Cat \text{ and } u \in \Sigma^*.$$

(10) List grammars define *finite languages* (Fin).

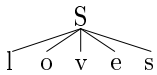
Otherwise, We say that a language L is an α language iff it is defined by some α grammar, for any of the grammar types α



(11) **Example.** The language $\{love, loves\}$.

$$S \rightarrowtail love \qquad\qquad S \rightarrowtail loves.$$

List grammar derivations can be depicted by *phrase structure trees*:

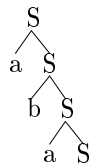


A finite list of dictionary entries (either their spellings, or their phonetic transcriptions), is another, much bigger list language.

(12) **Example.** Language $\{a, b\}^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$:

$$S \rightarrowtail \epsilon \qquad\qquad \begin{array}{l} S \rightarrowtail aS \\ S \rightarrowtail bS. \end{array}$$

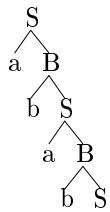
Right linear grammar derivations can also be depicted by *phrase structure trees*:



Notice that when $S \rightarrowtail \epsilon$ in the tree, interpreting that as the expansion of S to the empty sequence, I simply show the S with nothing below it.

(13) **Example.** The following 3 rule right linear grammar grammar defines $(ab)^* = \{\epsilon, ab, abab, ababab, \ldots\}$:

$$S \rightarrowtail \epsilon \qquad\qquad \begin{array}{l} S \rightarrowtail aB \\ B \rightarrowtail bS. \end{array}$$



It turns out that any language definable by a right linear grammars can also be defined by a grammar that branches only on the left edge, but we will only need the right branching versions here.

2.1 Languages, grammars and finite automata

(14) A *finite automaton* $A = \langle Q, \Sigma, \delta, I, F \rangle$ has these parts:

$$\begin{array}{rl} Q & \text{finite set of states} \\ \Sigma & \text{finite nonempty set of vocabulary items} \\ \delta : (Q \times \Sigma) \to \wp(Q) & \text{the 'transitions'} \\ I \subseteq Q & \text{initial states} \\ F \subseteq Q & \text{final states} \end{array}$$

We extend δ to $\delta' : (Q \times \Sigma^*) \to \wp(Q)$ as follows:

$$\begin{array}{rl} \delta'(q, \epsilon) = \{q\} & \text{for all } q \in Q \\ \delta'(q, aw) = \{\delta(q', w) | q' \in \delta(q, a)\} & \text{for all } q \in Q, a \in \Sigma, w \in \Sigma^* \end{array}$$

When $q' \in \delta'(q, w)$ we often say that w *labels a path from* q *to* q'. We extend δ' to a function $\delta'' : (\wp(Q) \times \Sigma^*) \to \wp(Q)$ as follows:

$$\delta''(S, w) = \bigcup\{\delta'(q, w) |\ q \in S\} \quad \text{for all } S \in \wp(Q)$$

Then we simply call all these functions δ, letting context of use (i.e. the types of arguments) make clear which function we mean.

The language accepted by A, $L(A) = \{w |\ \delta(I, w) \cap F \neq \emptyset\}$.

We say that A is *deterministic* if it has at most one initial state, and if the sets of "next-states" in the range of δ never have more than one element.

A *trim* if every state in on some path from an initial state to a final state.

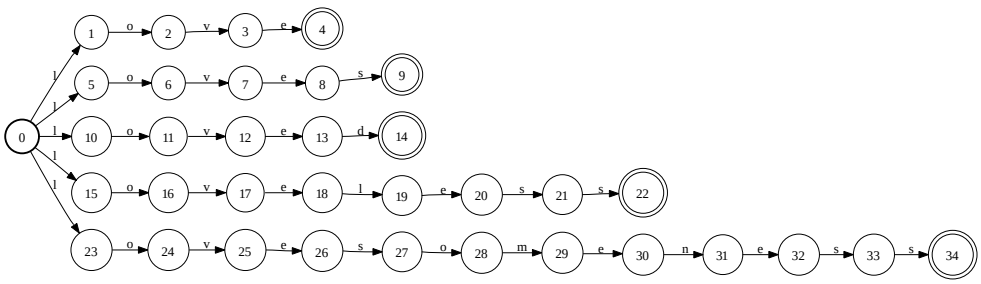
A is *cyclic* if there is some path containing more than one occurrence of a state; otherwise, A is *acyclic*.

(15) **Thm.** For every right linear grammar G, there is a finite automaton A, such that $L(G) = L(A)$.

(16) The language

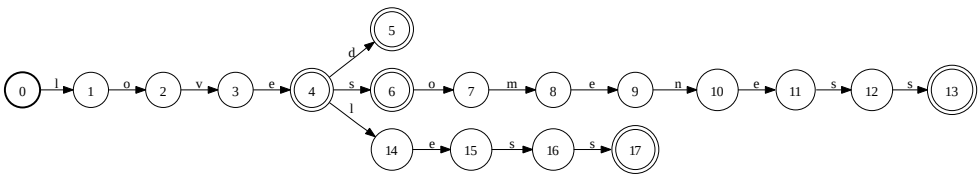
$$L = \{love, loves, loved, loveless, lovesomeness\}$$

is accepted by the following automaton, where we show the initial state 0 with a bold circle, and the final states with double circles:

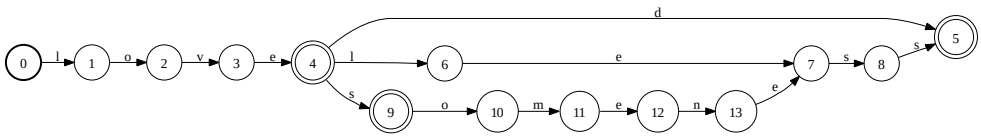


This one might be called the *list automaton*, since every element of the finite language is given as a separate sequence. Every list automaton is deterministic.

(17) The *prefix tree automaton*, $PT(L)$, in which the states are the prefixes p of the language, and for any input symbol a, if pa is a prefix of some string in the language, $\delta(p, a) = \{pa\}$. The final states are the words of the languages. So for our example, the $PT(L)$ is this:



(18) The *canonical acceptor* A_L, the minimal deterministic acceptor for L,



(19) **Even when it is theoretically possible to treat a problem with a weak machine, we can sometimes get a much more elegant, or much smaller, or much more efficient treatment with a more powerful device.**

(20) Basic properties

a. Regular languages are closed under union.

b. Regular languages are closed under intersection.

c. Regular languages are closed under concatenation.

d. Regular languages are closed under complements.

(21) For any language L and any $x, y \in \Sigma^*$ define

$$x \equiv_L y \text{ iff for all } z \in \Sigma^*,\ xz \in L \text{ iff } yz \in L.$$

This is sometimes called the *Nerode equivalence relation for L*.

(22) For any $x \in \Sigma^*$, the "block" of elements equivalent to x is $B(x, \equiv_L) = \{y |\ y \equiv_{\pi_L} x\}$.

(23) $\pi_L = \{B(x, \equiv_L) |\ x \in \Sigma^*\}$ is a partition of Σ^*.

(24) Thm. If $w \in L$ and $B(w, \pi_L) = B(w', \pi_L)$ then $w' \in L$.

(25) Thm. If $a \in \Sigma$ and $B(w, \pi_L) = B(w', \pi_L)$ then $B(wa, \pi_L) = B(w'a, \pi_L)$.

Proof: Assume $a \in \Sigma$, $w \in \Sigma^*$ and $B(w, \pi_L) = B(w', \pi_L)$. By definition, for any $x \in \Sigma^*$,
$wx \in L$ iff $w'x \in L$ So let $x = az$:
$w(az) \in L$ iff $w'(az) \in L$ But then
$(wa)z \in L$ iff $(w'a)z \in L$ and so $B(wa, \pi_L) = B(w'a, \pi_L)$.

□

(26) **Thm.** (Myhill-Nerode Theorem) For any language L, the number of blocks in π_L is finite iff L is regular.

Proof: ($\Leftarrow$) Since every regular language is accepted by a finite acceptor $A = \langle Q, \Sigma, \delta, I, F\rangle$, assume $L = L(A)$. Define a partition π_A by

$$x \equiv_{\pi_A} y \text{ just in case } \delta(I, x) = \delta(I, y).$$

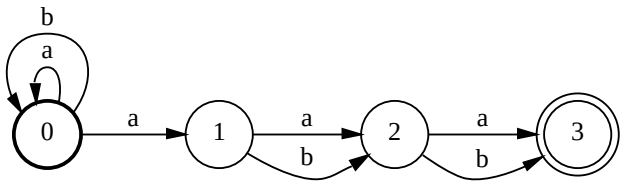
Since the values of δ are subsets of Q, the number of blocks in π_A cannot be larger than $2^{|Q|}$. But if $B(x, \pi_A) = B(y, \pi_A)$ then for all z, $B(xz, \pi_A) = B(yz, \pi_A)$, and so $xz \in L$ iff $yz \in L$. Hence, by the definition of the Nerode equivalence relation, if $B(x, \pi_A) = B(y, \pi_A)$ then $B(x, \pi_L) = B(y, \pi_L)$. It follows that π_L has no more states than π_A, and hence π_L has finitely many blocks.
($\Rightarrow$) Assume π_L has finitely many blocks. We define *the canonical acceptor for L*, A_L. We let the blocks of π_L themselves be the states of the automaton, $Q = \{B(w, \pi_L) |\ w \in \text{Prefix}(L)\}$. So, by assumption, Q is finite. Let

$$\delta(B(w, \pi_L), a) = \{B(wa, \pi_L)\} \quad \text{whenever } w, wa \in \text{Prefix}(L),$$
$$F = \{B(w, \pi_L) |\ w \in L\}, \text{ and}$$
$$I = \{B(\epsilon, \pi_L)\}.$$

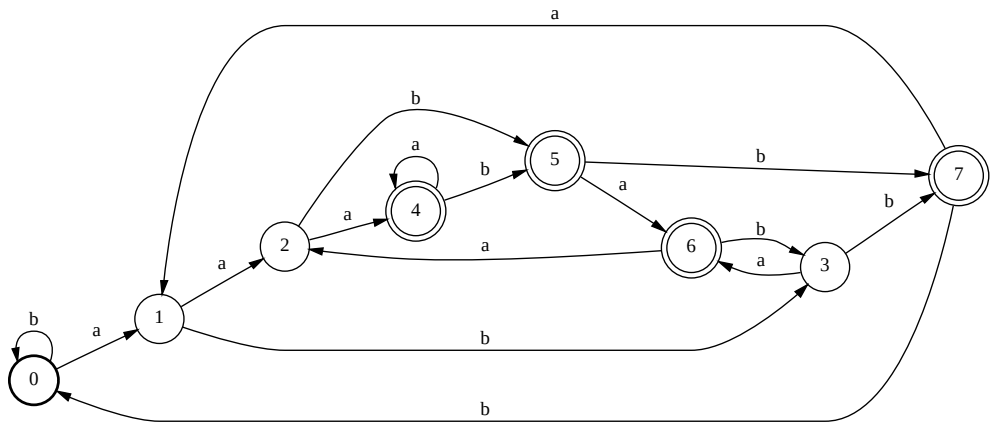
Now it is clear that $A_L = \langle Q, \Sigma, \delta, I, F\rangle$ is deterministic. Furthermore, this automaton accepts L, since by definition $w \in L(A_L)$ iff $B(w, \pi_L) \in F$ iff $w \in L$. □

(27) Thm. $L = \{a^n b^n |\ n \in \mathbb{N}\}$ is not regular.

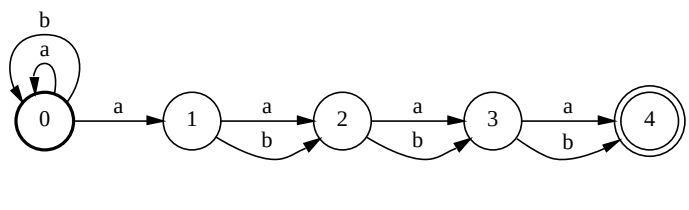
(28) The automata shown above are all deterministic. The following automaton is not:

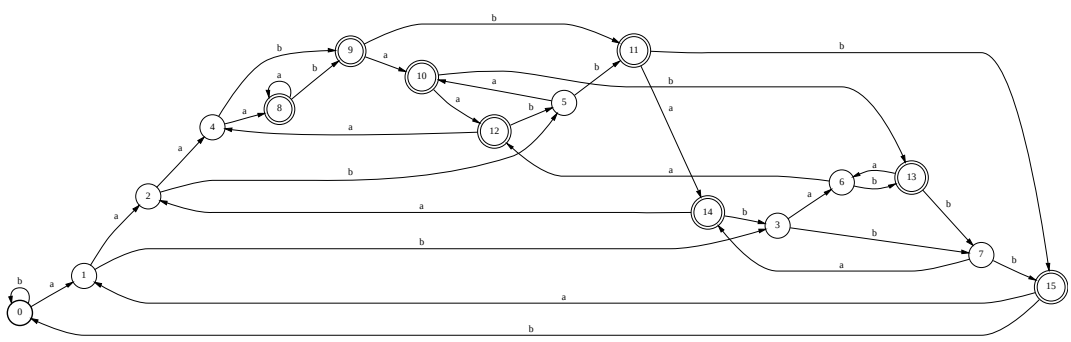


The corresponding minimal deterministic automaton is this one:



Adding one state to the nondeterministic automaton, we find that its minimal deterministic equivalent doubles in size:





(29) There are n-state automata A such that the smallest DFA accepting $L(A)$ has at least $2^n - 1$ states.

2.2 Finite transducers

(30) A finite transducer $A = (Q, \Sigma, \Delta, \delta, \mu, I, F)$ has these parts:

$$\begin{array}{rl} Q & \text{finite set of states} \\ \Sigma & \text{finite nonempty set of input vocabulary items} \\ \Delta & \text{finite nonempty set of output vocabulary items} \\ \delta : (Q \times \Sigma) \to \wp(Q \times \Delta^*) & \text{the 'transitions'} \\ I \subseteq Q & \text{initial states} \\ F \subseteq Q & \text{final states} \end{array}$$

We extend δ to $\delta' : (Q \times \Sigma^*) \to \wp(Q \times \Delta^*)$,

$$\delta'(q, i) = \{(q', o) | \quad \begin{array}{l} i \text{ labels a path from } q \text{ to } q' \\ \text{which is labelled with output sequence } o. \end{array}$$

Then we extend δ' to a function $\delta'' : (\wp(Q) \times \Sigma^*) \to \wp(Q)$ as follows:

$$\delta''(S, w) = \bigcup\{\delta'(q, w) | \ q \in S\} \quad \text{for all } S \in \wp(Q)$$

Then we simply call all these functions δ, letting context of use (i.e. the types of arguments) make clear which function we mean.

A finite transducer is *deterministic* if it has at most one initial state, and if the sets of "next-states" in the range of δ never have more than one element.

(31) **Basic properties**

a. The domains and ranges of finite transductions are regular.

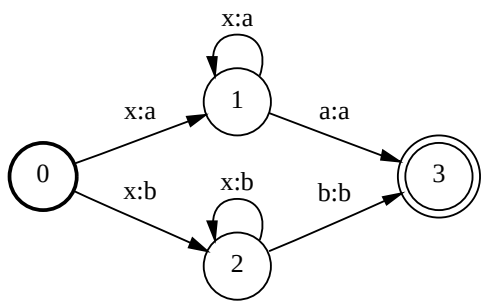
b. Finite transductions are closed under union.

c. Finite transductions are not closed under intersection.

Proof sketch: This is easily established by noting that we can define a transduction from a^n to $b^n c^*$ and a transduction from a^n to $b^* c^n$, but the intersection of these relations maps a^n to $b^n c^n$, which cannot be defined by a finite machine. □

d. Some finite transductions are essentially nondeterministic.

Proof sketch: The following transducer has no deterministic equivalent. Given strings $x^n a$ or $x^n b$, the machine cannot deterministically decide whether to start emitting a's or b's. Of course, some transducers can be determinized – see e.g. [225, §7.9] for an algorithm that applies in some special cases.



□

e. Finite transductions closed under intersecting their domains with regular languages.

Proof sketch: Given a finite state transducer T and a finite state machine A, we can easily construct the finite state transducer which defines the restriction of the transduction of T to the intersection $Dom(T) \cap A$.

Given $T = \langle Q_1, \Sigma, \Sigma_2, \delta_1, I_1, F_1 \rangle$ and $A = \langle Q_2, \Sigma, \delta_2, I_2, F_2 \rangle$,

define $T' = \langle Q_1 \times Q_2, \Sigma, \Sigma_2, \delta, I_1 \times I_2, F_1 \times F_2 \rangle$, where for all $a \in \Sigma, b \in \Sigma_2, q_1, r_1 \in Q_1, q_2, r_2 \in Q_2$,

$$([q_1, q_2], a, b, [r_1, r_2]) \in \delta \text{ iff } (q_1, a, b, r_1) \in \delta_1 \text{ and } (q_2, a, r_2) \in \delta_2.$$

This result has important practical applications and so it is presented in, for example, [226, §1.3.7]. □

f. Finite transductions closed under inverses.

Proof sketch: We simply interchange the input and output symbols labeling each transition. □

g. Finite transductions closed under compositions.

Proof sketch: Given $T = \langle Q_1, \Sigma_1, \Sigma_2, \delta_1, I_1, F_1 \rangle$ and
$A = \langle Q_2, \Sigma_2, \Sigma_3, \delta_2, I_2, F_2 \rangle$,
define $T' = \langle Q_1 \times Q_2, \Sigma, \Sigma_2, \delta, I_1 \times I_2, F_1 \times F_2 \rangle$, where for all $a \in \Sigma_1, b \in \Sigma_2, c \in \Sigma_3, q_1, r_1 \in Q_1, q_2, r_2 \in Q_2$,

$$([q_1, q_2], a, c, [r_1, r_2]) \in \delta \text{ iff } (q_1, a, b, r_1) \in \delta_1 \text{ and } (q_2, b, c, r_2) \in \delta_2.$$

□

(32) Some approaches provide an account of English sentences like

Maria eats tortillas
Maria is eating tortillas
Maria will have been eating tortillas

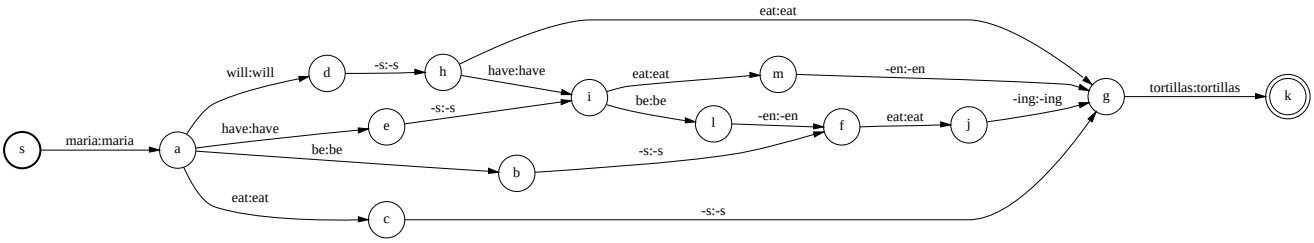
according to which syntax places the verbal inflections next to the verbs, like this,

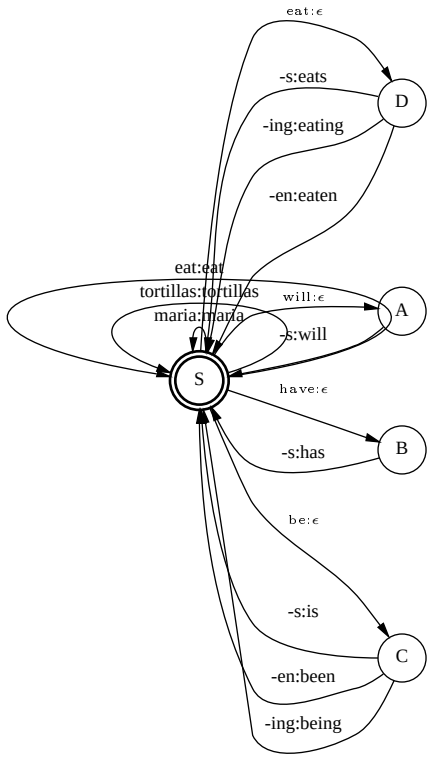
Maria eat -s tortillas
Maria be -s eating tortillas
Maria will -s have -ϵ be -en eat -ing tortillas

The surface forms are then defined by a separate step which combines stems and affixes, sometimes yielding an 'irregular' form.

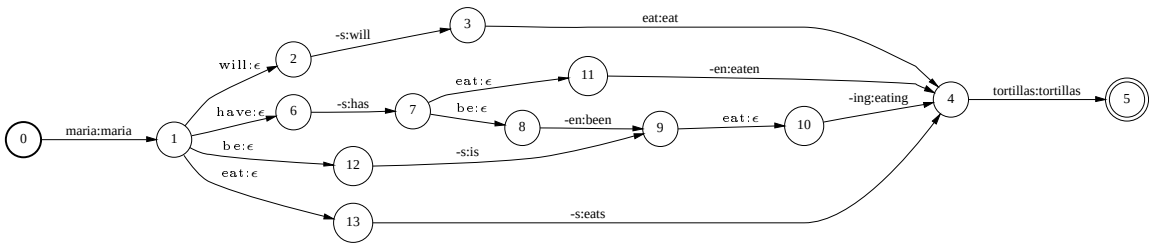
Anticipating later developments, if the syntax were given by a regular grammar, and the morphology were given by a transducer, the two could be composed into a single device.

To see how this works, Let's take a simple finite state grammar, and convert it into a finite state transducer which is the identity map on every string. And simple "spell-out" rules can be represented by another finite transduction:





Now we can compose these two transductions, so that there are not two steps (syntax + spell-out) but only one, using the definition of composition given above:



This composed machine has the same number of states as the first, "syntactic" processor, but in principle it could need a much larger number of states, up to the product of the states of the two original machines. In this case, it could be much more economical to refrain from composing the two machines, and simply "coroutine" them, i.e. step through both machines at once while processing the input.

(33) **Components of grammar which the linguist separates need not be separated in the model of language performance.**

§3 Dictionaries and the lexicon

(0) From the online notes (refresh your online notes frequently!):

```
let rec naiveReverse list = match list with
    [] -> []
  | x::xs -> append (naiveReverse xs) [x];;
```

But this definition does more work than it should. A better idea is this one:

```
let rec reverse2 input tmp = match input with
    [] -> tmp
  | x::xs -> reverse2 xs (x::tmp);;
```

Since we will usually be starting with the second argument empty, we can define:

```
let reverse input = reverse2 input [];;
```

(1) From the online notes:

```
let rec explode2 string i n =
  if i>=n then [] else string.[i]::explode2 string (i+1) n;;

let explode string = explode2 string 0 (String.length string);;

(* e.g.*)
# explode "abc";;
- : char list = ['a'; 'b'; 'c']
```

(2) Mitton's publicly available phonetic dictionary is adapted from the Oxford Advanced Learner's Dictionary of Current English (OALDCE), and provides a kind of phonetic transcription for British English. (Tony Robinson's beep is a bigger English phonetic dictionary but that one has only spelling and pronunciation, while the Mitton dictionary has syntactic information too.)

```
<TITLE TYPE="main"> Oxford advanced learner's dictionary of current English   :  expanded "computer usable" version /  compiled by Roger Mitton</TITLE>
<TITLE TYPE=SUB>A machine readable edition</TITLE>
<RESPSTMT>
<RESP>unspecified</RESP>
<NAME>Hornby, Albert Sydney</NAME>
</RESPSTMT>
<RESPSTMT>
<RESP>unspecified</RESP>
<NAME>Cowie, Anthony Paul</NAME>
</RESPSTMT>
<RESPSTMT>
<RESP>unspecified</RESP>
<NAME>Lewis, John Windsor,  1926-</NAME>
</RESPSTMT>
...
'em                     @m                      Qx$                     1
'neath                  niT                     T-$                     1
'shun                   SVn                     W-$                     1
'twas                   tw0z                    Gf$                     1
'tween                  twin                    Pu$,T-$                 1
'tween-decks            'twin-deks              Pu$                     2
'twere                  tw3R                    Gf$                     1
'twill                  twIl                    Gf$                     1
'twixt                  twIkst                  T-$                     1
'twould                 twUd                    Gf$                     1
```

70646 entries, 9117095 bytes, ⟨orthography,phonetic,listOfPOS,sylls+listOfVpats⟩

I replaced every double quote " in the dictionary by \", and then made the whole thing into 27 arrays of 4-tuples of strings, like this, in mitton.ml:

```
let mitton_A = [|
("'em","@m","Qx$","1");
("'neath","niT","T-$","1");
("'shun","SVn","W-$","1");
("'twas","twOz","Gf$","1");
("'tween","twin","Pu$,T-$","1");
("'tween-decks","'twin-deks","Pu$","2");
("'twere","tw3R","Gf$","1");
("'twill","twIl","Gf$","1");
("'twixt","twIkst","T-$","1");
("'twould","twUd","Gf$","1");
("'un","@n","Qx$","1");
("A","eI","Ki$","1");
("A's","eIz","Kj$","1");
...
("zoned","z@Und","Hc%,Hd%","16A");
("zones","z@Unz","Ha%,Kj%","16A");
("zoning","'z@UnIN","Hb%,L0%","26A");
("zoo","zu","K6%","1");
("zoological",",zu@'l0dZIkl","0A%","5");
("zoologist","zu'Ol@dZIst","K6%","4");
("zoologists","zu'Ol@dZIsts","Kj%","4");
("zoology","zu'Ol@dZI","L0%","4");
("zoom","zum","I0%,L0%","12A,2C");
("zoomed","zumd","Ic%,Id%","12A,2C");
("zooming","'zumIN","Ib%","22A,2C");
("zooms","zumz","Ia%","12A,2C");
("zoophyte","'z@U@faIt","K6$","3");
("zoophytes","'z@U@faIts","Kj$","3");
("zoos","zuz","Kj%","1");
("zoot suit","'zut sut","K6$","2");
("zoot suits","'zut suts","Kj$","2");
("zucchini","zU'kinI","M9%","3");
("zucchinis","zU'kinIz","Kj$","3")
|];;
```

mitton2.ml is the same, but keeps only the first 2 fields.

column 2: phonetic transcription

Mitton	IPA	example	here	Mitton	IPA	example	here
i	i	bead	0	N	ŋ	sing	22
I	ɪ	bid	1	T	θ	thin	23
e	ɛ	bed	2	D	ð	then	24
&	æ	bad	3	S	ʃ	shed	25
A	a	bard	4	Z	ʒ	beige	26
0(zero)	ɒ	cod	5	O	ɛə	cord	27
U	ʊ	good	6	u	u	food	28
p	p		7	t	t		29
k	k		8	b	b		30
d	d		9	g	g		31
V	ʌ		10	m	m		32
n	n		11	f	f		33
v	v		12	s	s		34
z	z		13	3	ɜ	bird	35
r	r		14	l	l		36
w	w		15	h	h		37
j	j		16	@	ə	about	38
eI	eɪ	day	17	@U	oʊ	go	39
aI	aɪ	eye	18	aU	aʊ	cow	40
oI	oɪ	boy	19	I@	ɪə	beer	41
e@	ɛə	bare	20	U@	ʊə	tour	42
R	ɹ	far	21				
'	(primary stress)		43	,	(secondary stress)		44
+	(break)		45	- (or space)	(break)		46

(3) Ocaml arrays take less space in memory than lists, so the mitton dictionary has been represented as an array. With big arrays, it is important to avoid creating multiple copies, so we access their elements using the integer position of their elements, with a function similar to the `List.nth` function for lists.

The mitton dictionary in `mitton2.ml` is given in 27 arrays. The first for entries – mainly capitalized items – that occur before those beginning with 'a', then an array of the 'a'-words, then the 'b'-words, and so on. I named the arrays `mitton_A, mitton_a, mitton_b,...mitton_z`.

(4) For example, suppose we want to write a function that will make a list of all the elements of an array. (There is a built-in function that does this too, `Array.to_list`, but we will write our own.)

We don't want to do this with any huge arrays, but consider the array that has just the last 6 entries of mitton_e:

```
let testArray = [|
    ("eyewash","'aIw0S");
    ("eyewitness","'aIwItn@s");
    ("eyewitnesses","'aIwItn@sIz");
    ("eying","'aIIN");
    ("eyrie","'e@rI");
    ("eyries","'e@rIz");
    ("eyry","'e@rI");
|];;
```

Suppose that we want a list of the first elements of an array like this. We can calculate the list with this function:

```
let testArray = [|
    ("eyewash","'aIw0S");
    ("eyewitness","'aIwItn@s");
    ("eyewitnesses","'aIwItn@sIz");
    ("eying","'aIIN");
    ("eyrie","'e@rI");
    ("eyries","'e@rIz");
    ("eyry","'e@rI");
|];;

(* calculate the list of elements from position i to position n *)
let rec listFromArray2 a i n =
  if i>=n then []
  else (fst a.(i))::listFromArray2 a (i+1) n;;

let listFromArray a = listFromArray2 a 0 (Array.length a);;

(* e.g. *)
listFromArray testArray;;
```

(5) **Exercise.** Write a function that will take an array a like the previous one, and produce a list of the characters in the first elements of a.

```
charListsFromArray1 testArray;;
- : char list list =
[['a'; 'z'; 'a'; 'l'; 'e'; 'a']; ['a'; 'z'; 'a'; 'l'; 'e'; 'a'; 's'];
 ['a'; 'z'; 'i'; 'm'; 'u'; 't'; 'h'];
 ['a'; 'z'; 'i'; 'm'; 'u'; 't'; 'h'; 's']; ['a'; 'z'; 'u'; 'r'; 'e'];
 ['a'; 'z'; 'u'; 'r'; 'e'; 's']]
```

(6) To draw my beautiful finite state machines, I use two freely distributed packages from AT&T: Graphviz, and Fsmtools. Let's use Graphviz first. (On windows, download the "static" version.)

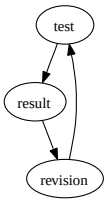
Test your installation by creating a file called `test.dot` like this:

```
digraph G {
        "test"->"result";
        "result"->"revision";
        "revision"->"test";
}
```

Then, in a terminal window, you should be able to execute:

```
dot -Tgif test.dot > dottest.gif
```

You should be able to view this gif file to see:



(7) Suppose that we wanted to use Ocaml to generate a file like `test.dot`. How could we do that? The following code does the trick:

```
let testOutput () =
  let out = open_out "test2.dot" in
    begin
      output_string out "digraph G {
        \"test\"->\"result\";
        \"result\"->\"revision\";
        \"revision\"->\"test\";
}
";
        close_out out;
    end;;
```

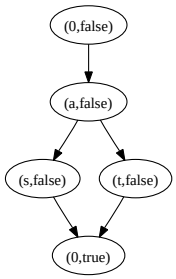
Another useful command, with a format familiar from other programming languages, is

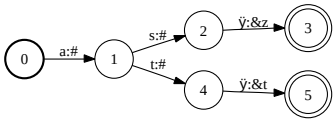
```
Printf.fprintf out "(%c,%b,%i,%s)\n" 'a' true 3 "hey";
```

(8) **Exercise.** Consider this tiny prefix tree:

```
Pt ('\255', [], false,
    [Pt ('a', [], false,
         [Pt ('s', [], false,
              [Pt ('\255', ['&'; 'z'], true, [])]);
          Pt ('t', [], false,
              [Pt ('\255', ['&'; 't'], true, [])])])])
```

Write a function that will take the tree as argument to produce a dot file for the graph on the left. We showed this same tree in the notes as the machine on the left.





That is, for each node in the prefix tree `Pt(inchar,outchars,final,subtrees)`, draw a link from (inchar,final) to each of the nodes it dominates. With this strategy, the graph on the left is not a tree, even though the machine on the right is.)

§4 Creating and minimizing lexical transducers

4.0 creating a prefix tree transducer from an array of pairs

Suppose we want to add an input-output pair to a tree if it doesn't have it, and otherwise leave the tree unchanged. This is quite easy to do.

```
(* here, ensureIn and ensureDown are mutually recursive, so we use "and" *)
let rec ptEnsure input output =
  (* ensureIn: string -> pt list -> input char list -> pt list *)
  let rec ensureIn output trees list = match list with
      [] -> ensureMember (Pt('\255',(explode output),true,[])) trees
    | c::cs -> ensureDown c cs output trees
  (* ensureDown: char -> char list -> string -> pt list -> pt list *)
  and ensureDown c cs output list = match list with
      [] -> [Pt(c,[],false,(ensureIn output [] cs))]
    | Pt (i,o,f,subtrees)::trees ->
        if compare c i = 0 then Pt(i,o,f,(ensureIn output subtrees cs))::trees
        else if compare c i < 0 then (* c comes before i *)
          Pt(c,[],false,(ensureIn output [] cs))::Pt (i,o,f,subtrees)::trees
        else (* c comes after i *)
          (Pt(i,o,f,subtrees))::(ensureDown c cs output trees)
  (* ptEnsure: char list -> string -> pt -> pt *)
  in function
    Pt (i,o,f,subtrees) -> Pt(i,o,f,(ensureIn output subtrees input));;
```

Notice that this definition uses two functions which are "mutually recursive": each calls the other, recursively. We can now build up a prefix tree transducer, one pair at a time:

```
# ptEnsure ['a'] "b" (Pt('\255',[],false,[]));;
- : char prefixTree =
Pt ('\255', [], false, [Pt ('a', [], false, [Pt ('\255', ['b'], true, [])])])

# ptEnsure ['c'] "d" (ptEnsure ['a'] "b" (Pt('\255',[],false,[]))) ;;
- : char prefixTree =
Pt ('\255', [], false,
 [Pt ('a', [], false, [Pt ('\255', ['b'], true, [])]);
  Pt ('c', [], false, [Pt ('\255', ['d'], true, [])])])
```

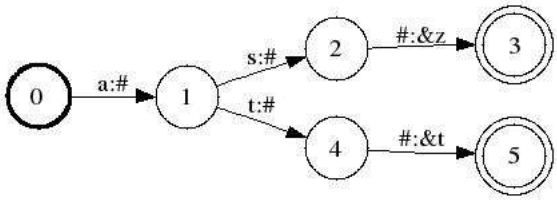
Now it is a simple matter to map an array of pairs of strings from our dictionary `mitton2.ml` to a prefix tree:

```
# let mittonPairEnsure (inputString,outputString) tree =
  ptEnsure (explode inputString) outputString tree;;
val mittonPairEnsure : string * string -> char prefixTree -> char prefixTree = <fun>

# let array2pt mittonArray = Array.fold_right mittonPairEnsure mittonArray (Pt('\255',[],false,[]));;
val array2pt : (string * string) array -> char prefixTree = <fun>

# let mitton_eg2 = [|
    ("are","AR");
    ("area","'e@rI@");
|];;
val mitton_eg2 : (string * string) array = [|("are", "AR"); ("area", "'e@rI@")|]
```

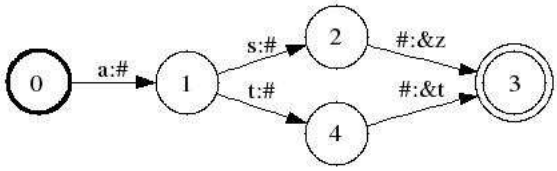
```
# let ptt2 = array2pt mitton_eg2;;
val ptt2 : char prefixTree =
  Pt ('\255', [], false,
   [Pt ('a', [], false,
     [Pt ('r', [], false,
       [Pt ('e', [], false,
         [Pt ('a', [], false,
           [Pt ('\255', ['\'; 'e'; '@'; 'r'; 'I'; '@'], true, [])]);
          Pt ('\255', ['A'; 'R'], true, [])])])])])
```

4.1 minimization overview

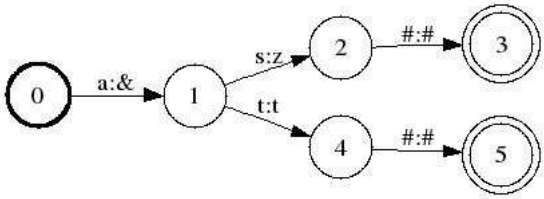
(0) sequential prefix tree



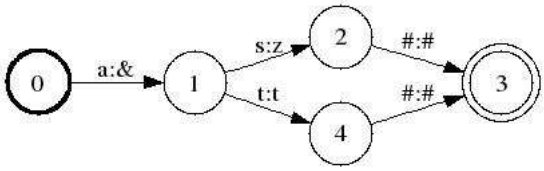
(2') first try, merge equiv states



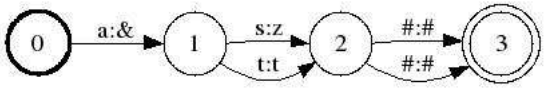
(1) second try, (step i: first, push output up)



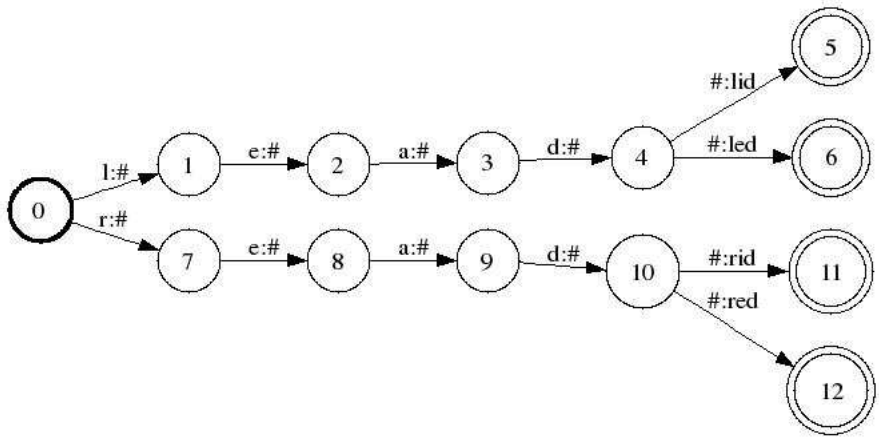
(2) then (step ii: merge equiv states)



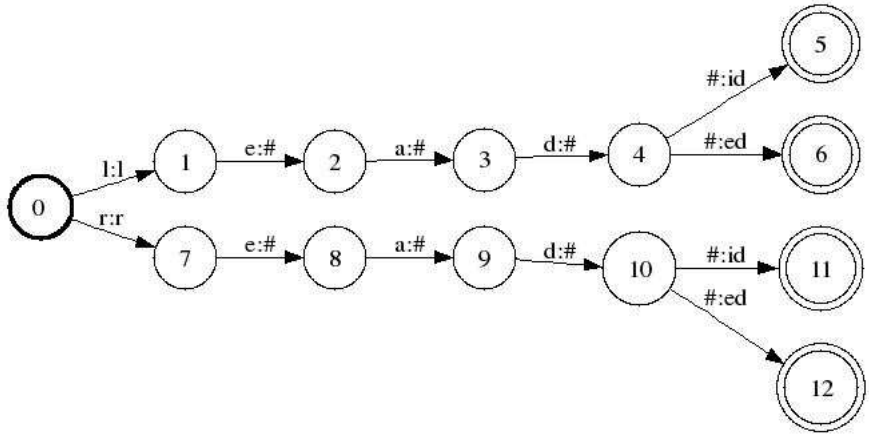
(3) (merge equiv states, continued)



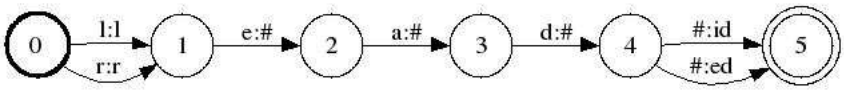
(4) prefix tree



(5) (step i: push output up)



(6) (step ii: merge equiv states)

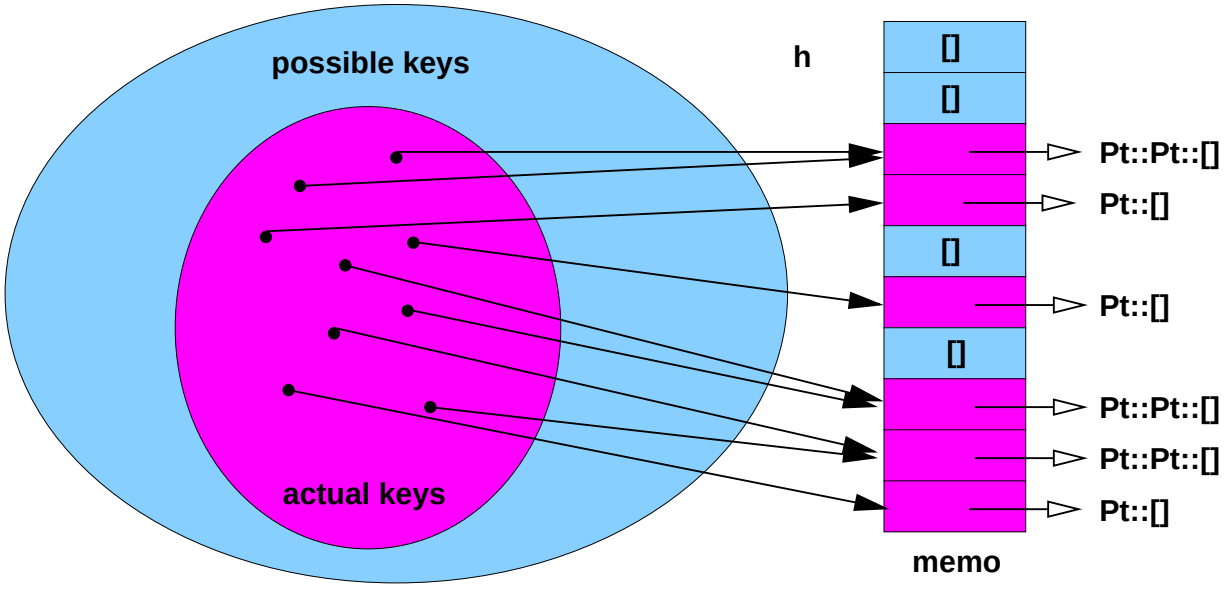


4.2 pushing the output up

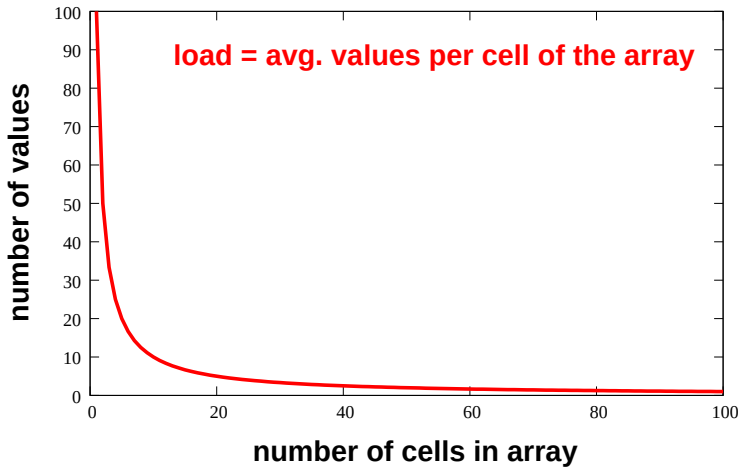
```
(*  file: dictMin0.ml
 creator: E Stabler
    date: 2007-01-21 08:41:38 PDT

  The functions in this file move output as high as possible in the transducer.
  It begins with the prefix tree, which should be stored in:  mittonPt.bin
*)

(* setup *)
type 'aa prefixTree = Pt of 'a * 'aa list * bool * 'a prefixTree list;;

let inputChannel = open_in "mittonPt.bin";;
let bigPt = input_value inputChannel;;
close_in_noerr inputChannel;;

(** minimization **)
(** maxPrefix: 'a list -> 'a list -> 'a list * 'a list * 'a list ***)
(* given list1 list2, return (maximum prefix,remainder1,remainder 2) *)
let rec maxPrefix = function
    ([],list2) -> ([],[],list2)
  | (list1,[]) -> ([],list1,[])
  | ((x::xs),(y::ys)) -> if (compare x y)=0
    then let (p,r1,r2)=(maxPrefix (xs,ys)) in (x::p,r1,r2)
    else ([],(x::xs),(y::ys));;

(* tests *)
maxPrefix ([0;1;2],[0;1]);;
maxPrefix ([0;1;2],[0;1;2;3;4]);;
maxPrefix ([0;1;2],[0;1;2]);;

(** nextElement: 'a -> 'a list list -> bool * 'a list list ***)
(* given e lists, if e begins every list, return (true,remainders) else (false,_) *)
let rec nextElement e = function
    [] -> (true,[])
  | []::_ -> (false,[])
  | (x::xs)::lists ->
      if x=e then let (yes,rem)=(nextElement e lists) in (yes,xs::rem)
      else (false,[]);;

(** maxPrefixLists 'a list list -> 'a list * 'a list list ***)
(* given lists, return (maximum prefix of lists,list of remainders) *)
let rec maxPrefixLists = function
    [] -> ([],[])
  | []::lists -> ([],[]::lists)
  | (x::xs)::lists -> let (yes,rem)=(nextElement x lists) in
      if yes then let (prefix,quotients)=(maxPrefixLists (xs::rem)) in
        ((x::prefix),quotients)
      else ([],(x::xs)::lists);;

(* tests *)
maxPrefixLists [[0;1;2];[0;1];[0;1;2]];;
maxPrefixLists [[0;1;2;3];[0;1;2];[0;1;2;3;4]];;
maxPrefixLists [[0;1;2;3;4]];;

(** pseudoDet prefixTree -> prefixTree **)
(** given a prefixTree, moveOutput as high as possible **)
(* i.e. change each output to the quotient of the original output with the common prefix *)

let rec outputsOf = function
    [] -> []
  | Pt (_,o,_,_)::ts -> o::(outputsOf ts);;

(* tests *)
outputsOf [];;
outputsOf [Pt ('\000', ['&'; 'z'z'], true, [])];;
outputsOf [Pt ('\000', ['&'; 'z'z'], true, []); Pt ('\000', ['&'; 't't'], true, [])];;

let rec placeOutputs = function
```

```
    ([],[]) -> []
  | (o::os,(Pt (i,_,f,ts)::trees)) ->
      Pt (i,o,f,ts)::placeOutputs (os,trees)
  | _ -> raise (failwith "placeOutputs error");;

(* tests *)
placeOutputs ([[]],[Pt ('\000', ['&'; 't't'], true, [Pt ('\000', ['&'; 't't'], true, [])])]);;
placeOutputs ([['x'x']],[Pt ('\000', ['&'; 't't'], true, [])]);;
placeOutputs ([['x'x'];['y'y']],[Pt ('\000', ['&'; 'z'z'], true, []); Pt ('\000', ['&'; 't't'], true, [])]);;

(* given tree, push output of daughters and return new tree *)
let rec pushOutput (Pt (i,o,f,ts)) =
  let newTs=(List.map pushOutput ts) in
  let (prefix,remainders)=(maxPrefixLists (outputsOf newTs)) in
    Pt (i,o@prefix,f,(placeOutputs (remainders,newTs)));;

(* tests *)
let asAt=
  Pt ('\255', [], false,
   [Pt ('a'a', [], false,
     [
       Pt ('s's', [], false, [Pt ('\255', ['&'; 'z'z'], true, [])]);
       Pt ('t't', [], false, [Pt ('\255', ['&'; 't't'], true, [])]);
     ])]);;
pushOutput asAt;;

(* in transducer prefixTrees, push output but leave root alone *)
let ptPushOutput (Pt (i,o,f,ts)) = Pt (i,o,f,(List.map pushOutput ts));;

(* tests *)
ptPushOutput asAt;;

let mitton_eg3 = [|
    ("lead","led");
    ("lead","lid");
    ("read","red");
    ("read","rid");
|];;

let leadRead = Pt ('\255', [], false,
   [Pt ('l'l', [], false,
     [Pt ('e'e', [], false,
       [Pt ('a'a', [], false,
         [Pt ('d'd', [], false,
           [Pt ('\255', ['l'l'; 'i'i'; 'd'd'], true, []);
            Pt ('\255', ['l'l'; 'e'e'; 'd'd'], true, [])])])])])]);
    Pt ('r'r', [], false,
     [Pt ('e'e', [], false,
       [Pt ('a'a', [], false,
         [Pt ('d'd', [], false,
           [Pt ('\255', ['r'r'; 'i'i'; 'd'd'], true, []);
            Pt ('\255', ['r'r'; 'e'e'; 'd'd'], true, [])])])])])]);;

ptPushOutput leadRead;;

let bigPt2 = ptPushOutput bigPt;;

let outputChannel = open_out "mittonPtPseudoDet.bin" in
  output_value outputChannel bigPt2;
  close_out_noerr outputChannel;;
```

4.3 storage in a 'chart'



Hash function h maps keys to locations in the memo array




```
(*  file: dictMin1.ml
 creator: E Stabler
    date: 2007-01-21 15:41:38 PDT
 purpose: this file includes the code I used to introduce hash tables
*)

(* ensureMember e l adds e to end of list l unless it's already in the list *)
let rec ensureMember e = function
    [] -> [e]
  | x::xs -> if e=x then x::xs else x::(ensureMember e xs);;

let explode word =
  let rec explode2 i word tmp =
    if i<0 then tmp else explode2 (i-1) word (word.[i]::tmp)
  in explode2 (String.length word - 1) word [];;

type 'aa prefixTree = Pt of 'a * 'aa list * bool * 'a prefixTree list;;

(* for tests *)
let emptypt = Pt ('\255', [], false, []);;

let asAt=
  Pt ('\255', [], false,
```

```
  [Pt ('a'a', [], false,
    [
      Pt ('s's', [], false, [Pt ('\255', ['&'; 'z'z'], true, [])]);
      Pt ('t't', [], false, [Pt ('\255', ['&'; 't't'], true, [])]);
    ])]);;

let leadRead = Pt ('\255', [], false,
  [Pt ('l'l', [], false,
    [Pt ('e'e', [], false,
      [Pt ('a'a', [], false,
        [Pt ('d'd', [], false,
          [Pt ('\255', ['l'l'; 'i'i'; 'd'd'], true, []);
           Pt ('\255', ['l'l'; 'e'e'; 'd'd'], true, [])])])])])]);
   Pt ('r'r', [], false,
    [Pt ('e'e', [], false,
      [Pt ('a'a', [], false,
        [Pt ('d'd', [], false,
          [Pt ('\255', ['r'r'; 'i'i'; 'd'd'], true, []);
           Pt ('\255', ['r'r'; 'e'e'; 'd'd'], true, [])])])])])]);;

(* hash table setup *)
let hashSize = 9689;;

type bucket = char prefixTree list;;
type memo = bucket array;;
let memo = Array.create hashSize [];;

(* hash function definitions *)
let rec codePt (Pt (i,o,f,subtrees)) =
  let codeInput c = Char.code c in
  let codeOutput cs = List.fold_right (function x -> function y -> (Char.code x) + y) cs 0 in
  let codeBool x = if x then 1 else 0 in
  let codeSubtrees ts = List.fold_right (function x -> function y -> (codePt x) + y) ts 0 in
    ((codeInput i) + (codeOutput o) + (codeBool f) + (codeSubtrees subtrees));;

let rec hash pt = (codePt pt) mod hashSize;;

(* tests *)
codePt emptypt;;  (* - : int = 255 *)
codePt asAt;;     (* - : int = 1409 *)
codePt leadRead;; (* - : int = 3353 *)

hash emptypt;;    (* - : int = 255 *)
hash asAt;;       (* - : int = 1409 *)
hash leadRead;;   (* - : int = 3353 *)

(* WARNING: for some reason, tuareg C-XC-e does not recognize the next 2 defs: copy them manually *)

(* like ensureMember, but returns (list,val) where val=true iff returned list has new element *)
let rec ensureMemberValue e = function
      [] -> (e::[],true)
    | x::xs -> if e=x then (x::xs,false) else let (list,value) = ensureMemberValue e xs in (x::list,value);;

let hashEnsure x array =
  let bucket = hash x in
  let contents = array.(bucket) in
  let (newList,value) = ensureMemberValue x contents in
    if value then array.(bucket) <- newList else ();;

let hashMember x array = List.mem x array.(hash x);;

(* tests *)
hashEnsure emptypt memo;;
hashEnsure asAt memo;;

hashMember emptypt memo;;
hashMember asAt memo;;
hashMember leadRead memo;;

let rec hashSubtrees (Pt (i,o,f,subtrees)) array =
  List.iter (function x -> hashSubtrees x array) subtrees;
```

```
  hashEnsure (Pt (i,o,f,subtrees)) array;;

let numberOfElementsInArray array =
  Array.fold_right (function x -> (function y -> (List.length x) + y)) array 0;;

let fetch (bucket,pos) = List.nth memo.(bucket) pos;;

(* tests *)
numberOfElementsInArray memo;;

(* big test: hash the whole pseudodeterminized mitton tree *)
let memo = Array.create hashSize [];;  (* create fresh version *)
let inputChannel = open_in "mittonPtPseudoDet.bin";;
let pseudoDet = input_value inputChannel;;
close_in_noerr inputChannel;;

hashSubtrees pseudoDet memo;;

numberOfElementsInArray memo;;
```

4.4 merging equivalent states

```
(*  file: dictMin2.ml
 creator: E Stabler
    date: 2007-01-21 16:41:38 PDT
 purpose: this file includes all the code I used to produced the dag
   version of the Mitton dictionary. It begins with the
   pseudodeterminized tree, which should be stored in: mittonPtPseudoDet.bin
*)

(* ensureMember e l adds e to end of list l unless it's already in the list *)
let rec ensureMember e = function
    [] -> [e]
  | x::xs -> if e=x then x::xs else x::(ensureMember e xs);;

let explode word =
  let rec explode2 i word tmp =
    if i<0 then tmp else explode2 (i-1) word (word.[i]::tmp)
  in explode2 (String.length word - 1) word [];;

type 'aa prefixTree = Pt of 'a * 'aa list * bool * 'a prefixTree list;;

let inputChannel = open_in "mittonPtPseudoDet.bin";;
let pseudoDet = input_value inputChannel;;
close_in_noerr inputChannel;;

(*** an 'a subdag has type ('a * 'a list * bool * (int*int) list ***)
(*** a dag will be given by an array of subdags ****)

type 'aa subdag = 'a * 'aa list * bool * (int*int) list;;

let hashSize = 9689;;

type bucket = char subdag list;;
type memo = bucket array;;
let memo = Array.create hashSize [];;

let codeSubdag (i,o,f,addresses) =
  let codeInput c = Char.code c in
  let codeOutput cs = List.fold_right (function x -> function y -> (Char.code x) + y) cs 0 in
  let codeBool x = if x then 1 else 0 in
  let codeAddresses ts = List.fold_right (function (x,y) -> function z -> x + y + z) ts 0 in
    ((codeInput i) + (codeOutput o) + (codeBool f) + (codeAddresses addresses));;

let rec hash subdag = (codeSubdag subdag) mod hashSize;;

(* the "address" of an element in the chart is given by the pair of integers (cell, position) *)

(* WARNING: tuareg C-xC-e has trouble with the next 2 functions, so I copy them manually! *)
```

```
(* like ensureMemberValue, but returns (list,val,pos) where pos is the position of element *)
let rec ensureMemberValPos e = function
     [] -> (e::[],true,0)
   | x::xs -> if e=x then (x::xs,false,0) else let (list,value,pos) = ensureMemberValPos e xs in (x::list,value,pos+1);;

(* like the earlier definition, but this time return the address of the element *)
let hashEnsure x array =
  let bucket = hash x in
  let contents = array.(bucket) in
  let (newList,value,position) = ensureMemberValPos x contents in
    if value then array.(bucket) <- newList else ();  (bucket,position);;

(* we traverse a tree and produce a dag, represented as a chart of chart addresses *)
(* tree2hashedDag pt -> array -> dag *)
let rec tree2hashedDag (Pt (i,o,f,subtrees)) array =
  let subdags = List.map (function x -> tree2hashedDag x array) subtrees in
    hashEnsure (i,o,f,subdags) array;;

let dag = tree2hashedDag pseudoDet memo;;   (* val dag : int * int = (8721, 0) *)

let outputChannel = open_out "mittonDag.bin";;
output_value outputChannel memo;;
close_out_noerr outputChannel;;

(* since fetch uses the memo array, redefine it after memo is updated *)
let fetch (bucket,pos) = List.nth memo.(bucket) pos;;

(********* TRANSDUCE **************)
(* transDown: char -> pt list -> (output*nextPtList) *)
let rec transDownDag c = function
     [] -> raise (failwith "input not accepted")
   | dag::dags -> let (i,o,f,subdags)=fetch dag in
       if compare c i = 0 then  (o, subdags)
       else if compare c i < 0 then raise (failwith "input not accepted")
       else transDownDag c dags;;

let rec dagTrans input address =
  (* dagTrans: dag list -> char list -> char list *)
  let rec transDagList dags = function
     [] -> let (out,nextDagList) = transDownDag '\255' dags in out
   | c::cs -> let (out,nextDagList) = transDownDag c dags in out@transDagList nextDagList cs
  (* dagTrans: char list -> dag -> char list *)
  in let (_,_,_,subdags) = (fetch address)
  in transDagList subdags input;;

(********* END TRANSDUCE **************)

dagTrans (explode "we") (8721, 0);;
dagTrans (explode "have") (8721, 0);;
dagTrans (explode "compressed") (8721, 0);;
dagTrans (explode "the") (8721, 0);;
dagTrans (explode "prefix") (8721, 0);;
dagTrans (explode "tree") (8721, 0);;
dagTrans (explode "substantially") (8721, 0);;

dagTrans (explode "blogger") (8721, 0);;
```

4.5 exercises

1. Write a program that will compute the number of times each input character appars in a prefix tree
2. Get `mitton2_r.ml` from the website (see the "code-snippets" page)
3. Use `array2pt` to produce a prefix tree from it (you can copy and paste this from the web page)
4. Use the function from exercise 1 to compute how many times each character appears in the `mitton_r` prefix tree, and display the result in a readable form.

§5 First glimpse of phonotactics

- The Chomsky hierarchy can be defined by the kinds of rules allowed in grammmars

  (and in many other equivalent ways)

- With standard assumptions about English grammar, English is not regular

  (because the Nerode equivalence relation in English has infinitely many blocks)

- Some finite languages (like the lexicon) are <u>not</u> best described by a list!

  There are regularities in the lexicon that, of course, the list representation does not exploit.

  Our transducers can use these, so they are not only much more compact than lists but also (as we'll see in detail soon) easier to use.

- With a good lookup method (like 'hashing'), very many things can be looked up quickly (in 'constant time')

- Simply compressing the dictionary transducer with the Shützenberger-Mohri algorithm does things like sharing the `s` in (spellings) `hooks` and `crooks` or (pronunciations) `hUks` and `krUks`. In fact, it shares `Uks` in these two entries. Will it also share the `s` in these entries with the one in (spelling) `miss` or (pronunciation) `mIs`? (no)

5.0 Functions that count

You should be getting used to counting functions like these:

```
let rec printStringList list = match list with
    [] -> print_string "\n"
  | x::xs -> begin print_string x; printStringList xs; end;;

printStringList ["hello";"world"];;

let rec ensureMember e list = match list with
    [] -> [e]
  | x::xs -> if x=e then x::xs else x::(ensureMember e xs);;

ensureMember 0 [];;
ensureMember 0 [1;0];;
ensureMember 0 [1;2];;

let rec collectElements list hypothesis = match list with
    [] -> hypothesis
  | x::xs -> collectElements xs (ensureMember x hypothesis);;

collectElements [3;0;1;2;2;1;0;3] [];;

let rec addMember e list = match list with
    [] -> [(e,1)]
  | (x,c)::xs -> if x=e then (x,c+1)::xs else (x,c)::(addMember e xs);;

addMember 3 [];;
let h1 = addMember 3 [];;
let h2 = addMember 1 h1;;
let h3 = addMember 2 h2;;
let h4 = addMember 3 h3;;

let rec countElements list hypothesis = match list with
    [] -> hypothesis
  | x::xs -> countElements xs (addMember x hypothesis);;

countElements [3;1;2;3];;
```

Using our prefix trees, the counting functions are very similar:

```
type 'a prefixTree = Pt of 'a * 'a list * bool * 'a prefixTree list;;

let t1 =
  Pt ('\255', [], false,
   [Pt ('a', [], false,
     [Pt ('s', [], false, [Pt ('\255', ['&'; 'z'], true, [])]);
      Pt ('t', [], false, [Pt ('\255', ['&'; 't'], true, [])])])]);;

let rec numberOfNodesInTree tree =
  let rec numberOfNodesInTrees list = match list with
      [] -> 0
    | t::ts -> (numberOfNodesInTree t)+(numberOfNodesInTrees ts)
  in match tree with
    Pt (_,_,_,subtrees) -> (numberOfNodesInTrees subtrees)+1 ;;

numberOfNodesInTree t1;;

let rec printInputsInTree tree =
  let rec printInputsInTrees list = match list with
      [] -> ()
    | t::ts -> begin printInputsInTree t; printInputsInTrees ts; end
  in match tree with
    Pt (i,_,_,subtrees) -> begin print_char i; printInputsInTrees subtrees; end;;

printInputsInTree t1;;

let fixUnprintable c = if c='\255' then '0' else c;;

let rec printInputsInTree2 tree =
  let rec printInputsInTrees2 list = match list with
      [] -> ()
    | t::ts -> begin printInputsInTree2 t; printInputsInTrees2 ts; end
  in match tree with
    Pt (i,_,_,subtrees) -> begin print_char (fixUnprintable i); printInputsInTrees2 subtrees; end;;

printInputsInTree2 t1;;

let countOnlyPrintable c = if c='\255' then 0 else 1;;

let rec countInputsInTree tree =
  let rec countInputsInTrees list = match list with
      [] -> 0
    | t::ts -> (countInputsInTree t)+(countInputsInTrees ts)
  in match tree with
    Pt (i,_,_,subtrees) -> (countOnlyPrintable i)+(countInputsInTrees subtrees);;

countInputsInTree t1;;

let rec collectInputsInTree tree hypothesis =
  let rec collectInputsInTrees list hypothesis = match list with
      [] -> hypothesis
    | t::ts -> collectInputsInTrees ts (collectInputsInTree t hypothesis)
  in match tree with
    Pt (i,_,_,subtrees) -> if i='\255'
    then collectInputsInTrees subtrees hypothesis
    else collectInputsInTrees subtrees (ensureMember i hypothesis);;

collectInputsInTree t1 [];;
```

5.1 Exercises

These exercises are all easy, but if you decide to undertake any of them, check with me first. I can help you figure out what to do, and locate other resources which might be helpful. In many of these exercises, you could use either the spellings or the phonetic transcriptions. . . if your main interests are linguistic, I recommend using the phonetics!

(0) How do the (spelling or phonetic) character counts in the

i. list automaton for Mitton

compare with the character counts for

ii. the prefix automaton, and

iii. the minimized transducer?

We saw the numbers of characters decrease from (i) to (ii) to (iii), but do the shapes of the distributions change? We can get a first visual indication simply by plotting the counts for (i), (ii), and (iii) for the characters in 'rank order'.

(1) How do the character counts for (i), (ii), (iii) compare to character counts from a natural corpus? A plot will reveal some interesting things.

(2) Using the extra columns of the dictionary,

- what is the proportion of nouns? How does this compare to the proportion in natural discourse [124]?
- what is the proportion of 1(2,3,4,5)-syllable words in the dictionary? in normal discourse?

(3) Prepare another dictionary with the same methods we used for Mitton, and compare the results.

- E.g. prepare a dictionary transducer using Mitton's columns 2 and 3, where the elements of column 3 are not given treated as a sequence, but as a set.
- Or find a dictionary of your favorite other language, and calculate (i), (ii), and (iii).

(4)

- Compute the phone transition frequencies in the Mitton dictionary ('bigrams').
- Compare these with the phoneme transition frequencies in a natural corpus by, first, looking up the phoneme sequence for each word in the dictionary (deciding on a simple policy in cases of ambiguity), and second, counting the bigrams in these.
- Plot these two frequencies by their rank-order in Mitton, or by their rank order in the natural text. What do you notice in these results?
- Computing the probability of each word as the product of the relative frequencies of its bigrams, are the most frequent words the most probable?

(5) Let's say that the *edit distance* between two words is given by the number of letters (or phones, if you use the pronunciation) where they differ.

- Compute the edit distances between every pair of words
- How many words differ in just one character (or phone)?
- Use a cluster plot to look at the results (or at some small subset of them).
- As a linguist, define a better measure of edit distance (based on features) and repeat the classification.
- As a psycholinguist, how could you define a better measure of edit distance, based on perceptual similarity?
- As an engineer, to process typed documents, how could you find a better measure of edit distance, based on probability of typing and spelling errors?

5.2 Syllables

(6) A famous quote from Leonard Bloomfield 1933:

> In any succession of sounds, some strike the ear more forcibly than others: differences of *sonority* play a great part in the transition effects of vowels and vowel-like sounds... In any succession of phonemes there will thus be an up-and-down of sonority... Evidently some of the phonemes are more sonorous than the phonemes (or the silence) which immediately precede or follow... Any such phoneme is a *crest of sonority* or a *syllabic*; the other phonemes are *non-syllabic*... An utterance is said to have as many *syllables* (or *natural syllables*) as it has syllabics. The ups and downs of *syllabification* play an important part in the phonetic structure of all languages. [22, p120]

(7) It is traditionally assumed that a syllable is formed from zero or more consonants, followed by a vowel, and ending with a shorter sequence of zero or more consonants (but we will see this is an approximation). The consonants before the vowel, the vowel, and the consonants after the vowel are called the onset, the nucleus and the coda, respectively, with the nucleus as the only obligatory part.

(8) So where is this account of syllables in the Chomsky hierarchy?

(9) It is not controversial whether the syllable is needed for defining phone distributions [251], but it is often used, and is even more often used in definitions of stress placement, so we consider it here.

(10) The possible onsets in English are restricted. (They are restricted in every language, but the exact restrictions vary.) The restrictions on onsets are complex, but we can get some idea of the situation with the usual simplified story, which goes something like this. In English, any single consonant is a possible onset. Only certain 2-consonant onsets are possible. The ones that occur most often in common English words are given by the 32 +'s in this table:

	w	j	r	l	m	n	p	t	k
p		+	+	+					
t	+	+	+						
k	+	+	+	+					
b		+	+	+					
d	+		+						
g	+	+	+	+					
f		+	+	+					
T	+		+						
S			+						
s	+			+	+	+	+	+	+

This chart misses a few words with unusual sounds (borrowings from other languages, etc.). For example, *sphere* begins with the unusual onset [sf]. The number of 3-consonant onsets is much smaller. The familiar ones are these 9:

	w	j	r	l	m	n
sp		+	+	+		
st		+	+			
sk	+	+	+	+		

Again, certain other onsets appear in words borrowed from other languages.

(11) *Sonority Principle:* onsets usually rise in sonority towards the nucleus, and codas fall in sonority away from the nucleus.
Compare * *rtag*/*trag*, * *gatr*/*gart*

(12) Given a list of possible onsets like the one above, a simple traditional procedure which divides most English words into syllables is this:

1. Mark each +vocalic phone (vowels and syllabic liquids) as a nucleus.
2. Then, take the longest sequence of consonants preceding each nucleus which can form a possible onset to be an onset for the following nucleus.
3. Take all remaining consonants to be codas of the preceding nuclei.

For obvious reasons, the idea behind this last step is sometimes called "onsets before codas" or "maximize onsets" (This tendency may be due to the perceptual cues needed to recognize consonants, and the fact that final consonants are often unreleased [192, 250].)

(13) The simplistic rules (1-3) work properly for many words (try *matron, atlas, enigma*), but they do not seem to provide quite the right account of words like *apple* or *gummy*. The first syllable of *apple* is stressed, and it sounds like it should include the consonant. Cases like these are called "ambisyllabic:" a consonant is ambisyllabic if it is part of a (permissible) onset but immediately follows a stressed lax (-ATR) vowel.

(14) **Using rules (1-3)** (ignoring ambisyllabicity etc)

These steps are not too hard to compute – they can be computed in a number of steps that is a linear function of input length.

But the rules presume that we have the whole word to work on, which precludes incremental syllabification, as each syllable is being heard/read/produced, and obviously requires more and more memory without bound as word length increases. (And this is seriously problematic if, as discussed later, we wanted to use syllabification as a clue about word boundaries.) If we mark all vowels, left to right, then we have to "go back" to mark onsets, and then back again to mark codas.

Is there a way to compute the same results without unbounded memory demands and without all this back-and-forth? Yes.

(15) Is the relation between a word and its syllabified form a finite transduction, when this relation is defined by the 3 rules above? yes

Let's define a finite, cyclic transducer that computes the same result. To keep things simple, lets map each word into a sequence in which syllables are separated by the period '.', from which we can easily find the onsets (the consonants preceding the vowel) and the codas (consonants following the vowel).

5.3 Morphophonology

(16) Claim: English pluralization is naturally represented as a finite transduction.

```
("snack","sn&k");
("snacks","sn&ks");
("snag","sn&g");
("snags","sn&gz");
```

Can we separate the pluralization option? Yes.

(17) Pluralization options can be "compiled", that is, we can compose the lexicon with the plural transducer.

(18) Some derivational suffixes can combine only to roots [80]:

-an -ian	changes N to N	librari-an, Darwin-ian
	changes N to A	reptil-ian
-age	changes V to N	steer-age
	changes N to N	orphan-age
-al	changes V to N	betray-al
-ant	changes V to N	defend-ant
	changes V to A	defi-ant
-ance	changes V to N	annoy-ance
-ate	changes N to V	origin-ate
-ed	changes N to A	money-ed
-ful	changes N to A	peace-ful
	changes V to A	forget-ful
-hood	changes N to N	neighbor-hood
-ify	changes N to V	class-ify
	changes A to V	intens-ify
-ish	changes N to A	boy-ish
-ism	changes N to N	Reagan-ism
-ist	changes N to N	art-ist
-ive	changes V to A	restrict-ive
-ize	changes N to V	symbol-ize

-ly	changes A to A	dead-ly
-ly	changes N to A	ghost-ly
-ment	changes V to N	establish-ment
-ory	changes V to A	advis-ory
-ous	changes N to A	spac-eous
-y	changes A to N	honest-y
-y	changes V to N	assembl-y

(19) Some suffixes can combine with a root, or a root+affix:

-y	changes N to N	robber-y
-y	changes N to A	snow-y, ic-y, wit-ty, slim-y
-ary	changes N-ion to N	revolut-ion-ary
-ary	changes N-ion to A	revolut-ion-ary, legend-ary
-er	changes N-ion to N	vacat-ion-er, prison-er
-ic	changes N-ist to A	modern-ist-ic, metall-ic
-(at)ory	changes V-ify to A	class-ifi-catory, advis-ory

(20) Some suffixes combine with a specific range of suffixed items

-al	changes N to A allows -ion, -ment, -or	natur-al
-ion	changes V to N allows -ize, -ify, -ate	rebell-ion
-ity	changes A to N allows -ive, -ic, -al, -an, -ous, -able	profan-ity
-ism	changes A to N allows -ive, -ic, -al, -an	modern-ism
-ist	changes A to N allows -ive, -ic, -al, -an	formal-ist
-ize	changes A to V allows -ive, -ic, -al, -an	special-ize

(21) The coincidence between syntax and morphology extends to "subcategorization" as well. In the class of verbs, we can see that at least some of the subcategories of verbs with distinctive behaviors correspond to subcategories that allow particular kinds of affixes. For example, *-ify* and *-ize* combine with N or A to form V: *class-ify, intens-ify, special-ize, modern-ize, formal-ize, union-ize*, but now we can notice something more: the verbs they form are all transitive:

i. a. The agency class-ified the documents
   b. *The agency class-ified

ii. a. The activists union-ized the teachers
   b. *The activists union-ized (no good if you mean they unionized the teachers)

iii. a. The war intens-ified the poverty
   b. *The war intens-ified (no good if you mean it intensified the poverty)

(22) Another suffix *-able* combines with many transitive verbs but not with most verbs that only select an object:

i. a. Titus manages the project (transitive verb)
   b. This project is manag-able

ii. a. The sun shines (intransitive verb)
   b. *The sun is shin-able

iii. a. The train arrived ("unaccusative" verb)
   b. * The train is arriv-able

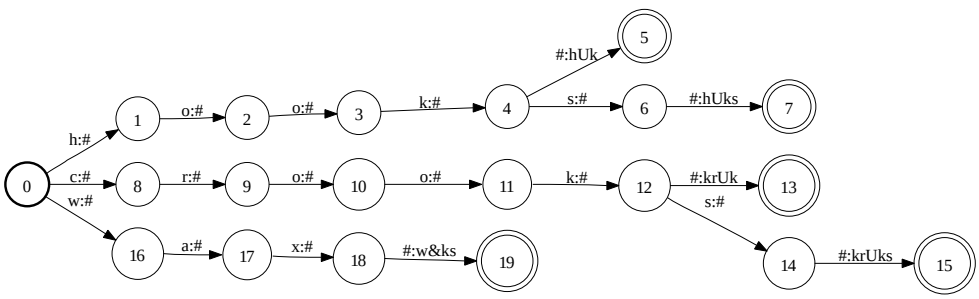
§6 Representing the lexicon

6.0 lexical representation in dictionaries

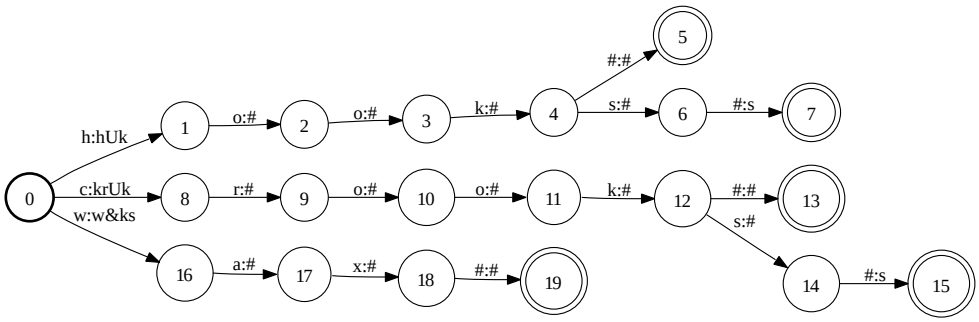
```
("hook","hUk");
("hooks","hUks");

("crook","krUk");
("crooks","krUks");

("woman","'wUm@n");
("women","'wImIn");

("child","tSaIld");
("children","'tSIldr@n");

("shelf","Self");
("shelves","Selvz");

("mouse","maUs");
("mice","maIs");

("foot","fUt");
("feet","fit");

("fish","fIS");

("miss","mIs");
("mix","mIks");
("wax","w&ks");
("whacks","w&ks");
```

6.1 Computer science approach to lexical representation

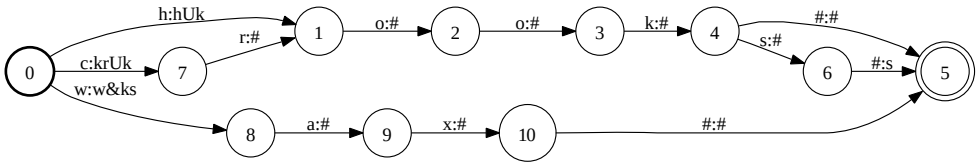
- let's get that redundancy out! Gzip? No. (why not?) Let's use Shützenberger-Mohri:
- First, compute a finitely subsequential prefix tree:



- Then push the output towards the root:



- Then share subtrees:

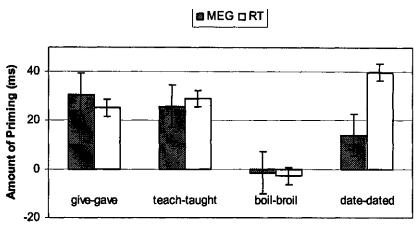


This is the minimal transducer (least number of states) that defines the relation we started with.

6.2 Linguistic approach to lexical representation

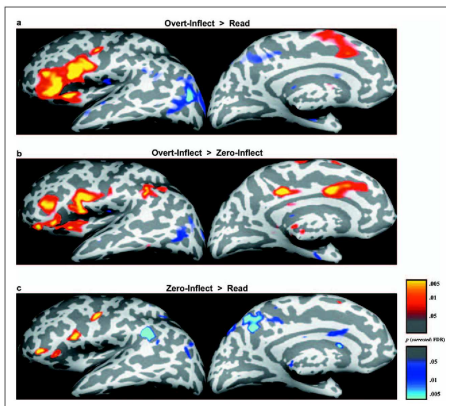
- First, the traditional dictionary has the wrong stuff in it! If we list morphemes (semantic atoms), and morph rules, the rest can be calculated
- Even for irregular inflections, there is evidence that both the root and the affixes are recognized.

MEG, rt



[252, p.87]

fMRI



[228, p.12]

- for many non-words, English speakers propose same plurals:

(wug,wVg)	→	(wugs,wVgz)	
(wuk,wVk)	→	(wuks,wVkz)	[102]

Lx1: Regular English pluralization:

add [s] after ps=[p,k,T,t,f,h];

add [@z] after zs=[z,s,S,Z];

add [z] after gs=[d,n,r,w,R,N,D,b,g,m,l,i,I,e,&,A,0,U,V,eI,aI,oI,e@,O,u,3,@,@U,aU,I@,U@]

- 'dual route' models [205]: represent exceptions separately
  - ⋆ If you see more than one Mickey Mouse at Disneyland, you did not see Mickey Mice.
- In other languages, even regular inflection is often not 'concatenative'

- ⋆ Tagalog: *takbuh*(run) → *tumakbuh*(ran), *lakad*(walk) → *lumakad*(walked) [277]
- ⋆ Papago: *ñeid*(see) → *ñei*(perf), *hi:nk*(bark) → *hi:n*(perf) [275]

- Stress is regular too, so maybe so maybe the kinds of considerations mentioned here so maybe Chomsky&Halle are right that it shouldn't be in the lexicon (except when it's exceptional)? [These considerations are very different from the ones Chomsky&Halle mention!. . . we'll discuss their arguments later.] To handle stress, many proposal assume that (i) syllable structure is given, and (ii) roots and affixes are distinguished.
- **Summary, tentative ideas**
  - ⋆ syllables are not marked in lex, but computed
  - ⋆ regular affixation is not marked in lex, but computed
  - ⋆ irregular affixes may just be stored in lex
  - ⋆ affixation is not always concatenative

6.3 Naive syllables

(0) **A simple problem first!** Call this language L_{CV}

a. A grammar for syllables σ:

$$\begin{array}{ll} \sigma \rightarrowtail Ons\ Rime & Ons \rightarrowtail C \\ \sigma \rightarrowtail Rime & Nuc \rightarrowtail V \\ Rime \rightarrowtail Nuc\ Coda & Coda \rightarrowtail C \\ Rime \rightarrowtail Nuc & \end{array}$$

Note that this defines a finite set of syllables.

b. The language L_{cv} is $(\sigma)^*$. This is an infinite language. We could say

$$\begin{array}{l} w \rightarrowtail \sigma\ w \\ w \rightarrowtail \sigma\ \epsilon \end{array}$$

L_{CV} is this language of words, an infinite language.

c. **Thm** $VVV \in L_{CV}$, but $CCC \notin L_{CV}$

d. Putting a period between syllables, everything we have said so far suggests that $VCVC$ could be analyzed as either $VC.VC$ or $V.CVC$, but that's not right. Only the latter analysis is possible.

e. To analyze a rule a word into syllables:

the phones: {C,V}

the possible onsets=the possible codas: {C}

the syllabification rule:

i. every V is a nucleus

ii. each syllable has as onset the the longest possible onset

iii. consonants not treated as onsets by ii are codas

(1) Can we write a grammar that, unlike (1a,b), actually defines just the syllable sequences that (1e) allows? Let's classify the syllables this way:

$$\begin{array}{rcl} \sigma_{o,c} & = & \text{syllables with onset and coda} \\ \sigma_{no,c} & = & \text{syllables with no onset and coda} \\ \sigma_{o,nc} & = & \text{syllables with onset and no coda} \\ \sigma_{no,nc} & = & \text{syllables with no onset and no coda} \end{array}$$

In a table of all the possible syllable-pairs, we see that 'maximize onsets' prohibits 4 of the 16 possible sequences, namely those sequences in which a syllable with a coda precedes one with no onset:

	$\sigma_{o,c}$	$\sigma_{no,c}$	$\sigma_{o,nc}$	$\sigma_{no,nc}$
$\sigma_{o,c}$		*		*
$\sigma_{no,c}$		*		*
$\sigma_{o,nc}$				
$\sigma_{no,nc}$				

So I think this grammar does the trick:

$$
\begin{array}{ll}
w \rightarrowtail \sigma_{o,c} w_o & \sigma_{o,c} \rightarrowtail Ons\ Rime_c \\
w \rightarrowtail \sigma_{no,c} w_o & \sigma_{no,c} \rightarrowtail Rime_c \\
w \rightarrowtail \sigma_{o,nc} w & \sigma_{o,nc} \rightarrowtail Ons\ Rime_{nc} \\
w \rightarrowtail \sigma_{no,nc} w & \sigma_{no,nc} \rightarrowtail Rime_{nc} \\
w \rightarrowtail \epsilon & Rime_c \rightarrowtail Nuc\ Coda \\
w_o \rightarrowtail \sigma_{o,c} w_o & Rime_{nc} \rightarrowtail Nuc \\
w_o \rightarrowtail \sigma_{o,nc} w & Ons \rightarrowtail C \\
w_o \rightarrowtail \epsilon & Nuc \rightarrowtail V \\
 & Coda \rightarrowtail C
\end{array}
$$

This grammar defines infinitely many words w, and now, every analysis allowed by the grammar will be one that conforms to the syllabification rules in 1e.

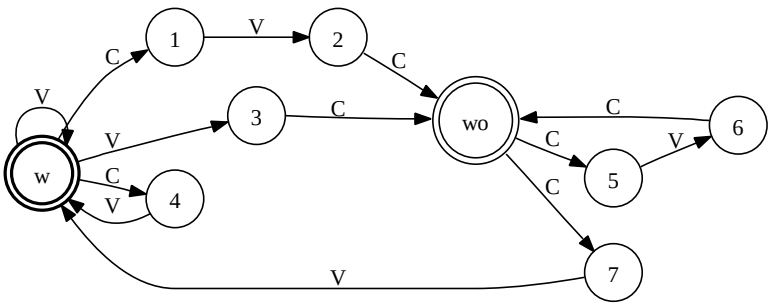
(2) **Thm.** The language L_{CV} is regular. That is, although the grammar just given is not right linear, the same language can be given by a right linear grammar.

Proof sketch: We can transform the grammar just given into a right linear grammar by noting that the first categories on the right sides of each rule rewrite to just finitely many strings, so these strings could be generated by appropriate right linear rules. Taking the first step, it is easy to see the the following grammar, obtained simply by putting in the V,C sequences for each syllable type, generates the same strings,

$$
\begin{array}{l}
w \rightarrowtail CVCw_o \\
w \rightarrowtail VCw_o \\
w \rightarrowtail CVw \\
w \rightarrowtail Vw \\
w \rightarrowtail \epsilon \\
w_o \rightarrowtail CVCw_o \\
w_o \rightarrowtail CVw \\
w_o \rightarrowtail \epsilon
\end{array}
$$

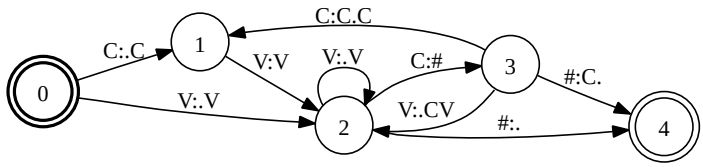
To make this grammar right linear, we need to introduce some new categories which we will simply number:

$$
\begin{array}{lll}
w \rightarrowtail C1 & 1 \rightarrowtail V2 & 2 \rightarrowtail Cw_o \\
w \rightarrowtail V3 & 3 \rightarrowtail Cw_o & \\
w \rightarrowtail C4 & 4 \rightarrowtail Vw & \\
w \rightarrowtail Vw & & \\
w \rightarrowtail \epsilon & & \\
w_o \rightarrowtail C5 & 5 \rightarrowtail V6 & 6 \rightarrowtail Cw_o \\
w_o \rightarrowtail C7 & 7 \rightarrowtail Vw & \\
w_o \rightarrowtail \epsilon & &
\end{array}
$$



This grammar is nondeterministic, and has more states than necessary, but suffices to establish the theorem.

(3) **Claim.** The mapping from L_{CV} that inserts periods between syllables is a finite transduction.



(This drawing is not quite clear: there is an arc from 2 to 4, and none from 4 to 2.)

```
(* a simple 1-subsequential finite state transducer SSFST *)
(* That is: no indeterminacy, but just a single empty transition to the final state *)

(* The 1-SSFST is a list of ((PrevState,NextIn),(NextOut,NextState)) pairs of pairs *)
(* We could put these into a hash table, but there are so few, it's not worth the trouble *)
let syllSSFST = [
  ((0,'C'),(['.';'C'],1));
  ((0,'V'),(['.';'V'],2));
  ((1,'V'),(['V'],2));
  ((2,'V'),(['.';'V'],2));
  ((2,'C'),([],3));
  ((2,'\255'),(['.'],4));
  ((3,'C'),(['C';'.';'C'],1));
  ((3,'V'),(['.';'C';'V'],2));
  ((3,'\255'),(['C';'.'],4));
];;

let rec transduceSSFST ssfst state input = match input with
    [] -> fst (List.assoc (state,'\255') ssfst)
  | c::cs ->
      let (output,nextState) = List.assoc (state,c) ssfst
      in
        begin
          Printf.fprintf stdout "(%i,%c)->(_,%i)\n" state c nextState;
          List.append output (transduceSSFST ssfst nextState cs);
        end;;

(* examples *)
transduceSSFST syllSSFST 0 ['V';];
transduceSSFST syllSSFST 0 ['V';'V';];
transduceSSFST syllSSFST 0 ['C';'V';'C';];
transduceSSFST syllSSFST 0 ['C';'V';'C';'C';'V';];
transduceSSFST syllSSFST 0 ['C';'V';'C';'V';];
```

6.4 Naive morphology

- **concatenation of transducers.** Given transducers $A_1 = \langle Q_1, \Sigma_1, \Sigma_2, \delta_1, I_1, F_1 \rangle$ and $A_2 = \langle Q_2, \Sigma_3, \Sigma_4, \delta_2, I_2, F_2 \rangle$, where $Q_1 \cap Q_2 = \emptyset$, intuitively, we merge all the elements of F_1 with all the elements of I_2. Define $T = \langle Q_1 \cup Q_2, \Sigma_1 \cup \Sigma_3, \Sigma_2 \cup \Sigma_4, \delta, I_1 F_2 \rangle$, where δ includes δ_1 and δ_2, and in addition, whenever $(q_1, a_1, a_2, q_2) \in \delta_1$ and $q_2 \in F_1$, $(q_1, a_1, a_2, q_3) \in \delta_1$ for each $q_3 \in I_2$.
- Regular expression notation extended to transducers

$$(a, b)^* = \{(a^n, b^n) |\ n \geq 0\}$$
$$(a : b) + (aa : bb)^* = \{(a^n, b^n) |\ n \in \{1, 2\}\}$$
$$(a : b)(c : d)^* = \{(ac^n, bd^n) |\ n \geq 0\}$$

- Regular plurals 1:

$$(hook, hUk)(s, s) = (hooks, hUks)$$

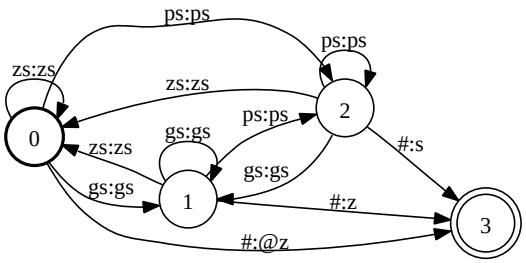
But this is only right for forms with pronunciation ending in [p,k,T,t,f,h]!

$$*(hug, hVg)(s, s) = (hugs, hVgs)$$

- **composition of transducers.** Given finite transducers $A = \langle Q_1, \Sigma_1, \Sigma_2, \delta_1, I_1, F_1 \rangle$ and $B = \langle Q_2, \Sigma_2, \Sigma_3, \delta_2, I_2, F_2 \rangle$,
  define $(A \circ B) = \langle Q_1 \times Q_2, \Sigma_1, \Sigma_2, \delta, I_1 \times I_2, F_1 \times F_2 \rangle$, where for all $a \in \Sigma_1, b \in \Sigma_2, c \in \Sigma_3, q_1, r_1 \in Q_1, q_2, r_2 \in Q_2$,

$$((q_1, q_2), a, c, (r_1, r_2)) \in \delta \text{ iff } (q_1, a, b, r_1) \in \delta_1 \text{ and } (q_2, b, c, r_2) \in \delta_2.$$

- Naive English regular plurals, spelling → pronunciation (cf. production).
  Recall that we partitioned the output (pronunciation) symbols of the lexicon into $\{ps, gs, zs\}$. Letting $ps : ps$ represent the arcs labeled $x : x$ for any $x \in ps$, and similarly for the other sets, consider the following transducer PL:



  Then, roughly:

$$\text{(hook,hUk)} \circ PL = \text{(hook,hUks)}$$
$$\text{(wish,wIS)} \circ PL = \text{(wish,wIS@z)}$$
$$\text{(hugs,hVg)} \circ PL = \text{(hug,hVgz)}$$

The proposal: morphological processes (concatenative and otherwise) can be implemented by composition. [134, 181, 16, 224]
Is that true??

6.5 Exercises

1. Get `plural1.ml` from the `code-snippets` page.
   (Or, if you're feeling on top of everything, you could use a hash table instead of a list, and get `plural2.ml`.)
   Then modify the previous transducer PL so that

$$\text{(hook,hUk)} \circ PL = \text{(hook,hUks)}$$
$$\text{(wish,wIS)} \circ PL = \text{(wish,wIS@z)}$$
$$\text{(hugs,hVg)} \circ PL = \text{(hug,hVgz)}$$
$$\text{(tooth,tuT)} \circ PL = \text{(tooth,tiT)}$$

   That is, modify PL so that it maps tuT to tiT. Implement and test your proposal.

2. Design a transducer, PERF, that takes any pronunciation, and deletes the final segment – a simplistic version of Papago:
   - ⋆ *neid*(see) → *nei*(perf), *hink*(bark) → *hin*(perf)
   - ⋆ *hiwa*(rub against) → *hiw*(perf), *moto*(carry) → *mot*(perf)

   Implement it (as done in `plural1.ml` or `plural2.ml`), and test your proposal.

3. Design a transducer, PAST, that takes any pronunciation, and inserts -um- before the first vowel, leaving everything else unchanged – a simplistic version of Tagalog:
   - ⋆ *takbuh*(run) → *tumakbuh*(ran), *lakad*(walk) → *lumakad*(walked)

   Implement it (as done in `plural1.ml` or `plural2.ml`), and test your proposal.

4. Some Tagalog loanwords begin with consonant clusters, in which case, -um- is sometimes also placed after one consonant away from the first vowel (but always one -um- per word), as in

   ⋆ *gradwet*(graduate) → *grumadwet* AND *gumradwet*(graduated)

   ⋆ *plahio*(plagiarize) → *pumlahi* AND *plumlahio*(plagiarized)

   Implement it (as done in `plural1.ml` or `plural2.ml`), but note that some modification will be needed in order to see more than one output. Make the modification as small and elegant as possible. Test your proposal.

§7 Morphophonology once more

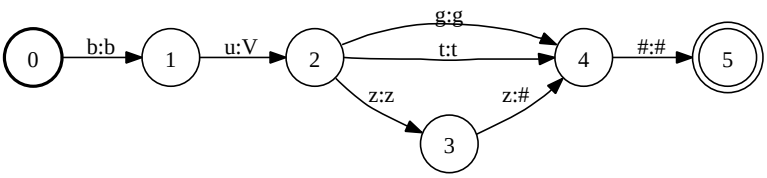
(0) Last time we saw that morphophonological rules *can* be separated from the lexical entries. In fact, an elegant representation seems adequate for much phonology: *many phonological processes are finite transductions.*

(1) We mentioned briefly some of the psychological evidence that it *is* separated in humans. But we did not yet talk about the question: *does that mean that derived forms are not (or not usually) stored?* This question turns out to be quite tricky: it could be that we notice regularities but still store the derived forms.

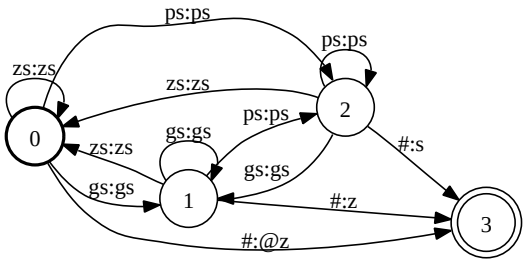
(Notice that compiler optimizations often multiply out regularities: recursion unfolding, loop unrolling, inlining,. . . Superficial representations can allow faster performance, depending on the architecture of the recognizer. Humans could well represent both superficial forms and their regularities.)

7.0 Morphophonological processes as transducer compositions

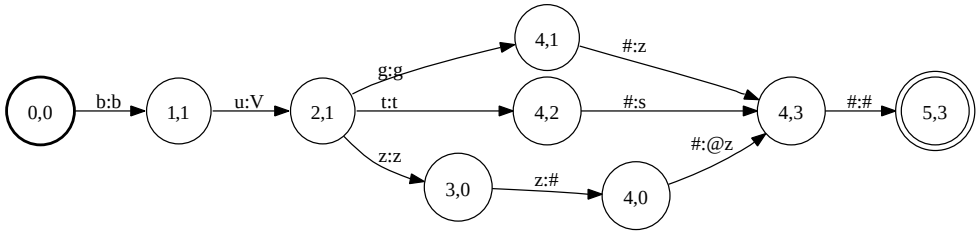
(2) **First example:** Consider for example the following transducer BUG:



And recall our transducer for English pluralization:



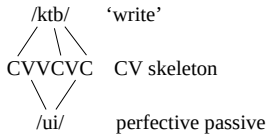
We can compose BUG∘PL. (Computing compositions in our Ocaml implementation is slightly tricky because we have required that there be exactly 1 symbol in the input. If output were restricted to at most 1 symbol, the implementation of this composition operation would be straightforward. But it is not too hard to extend the composition of transductions to allow multiple symbols are allowed in the output too.)



(3) **Another example:** This table from [173], abstracting away from effects of phonological rules and addition of affixes for agreement, mood, case:

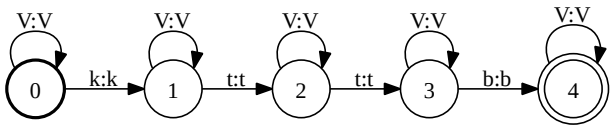
		/ktb/ 'write'	/drs/ 'study'	/sm/ 'poison'
perfect active		kattab	darras	sammam
perfect passive		kuttib	durris	summim
imperfect active	Cu+	kattib	darris	sammim
imperfect passive	Cu+	kattab	darras	sammam

> ...this small array of facts is sufficient to demonstrate the property of shape-invariance in Arabic templatic morphology. Moving across the columns...changes the consonantal root, the fundamental lexical unit of the language. Despite this change in the consonants, the canonical pattern remains the same. Similarly, moving down the rows...changes the vocalism: voice goes from active to passive or aspect from perfective to imperfective.



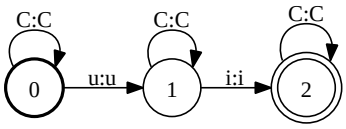
(4) **Transducer representation (first idea):** Represent the association lines in the previous figure by transducers. Cf. e.g. [224, §2.2.9], [33]

Let **ctemplate(kttb)** be the Id transducer that allows any number of Vs between the consonants:

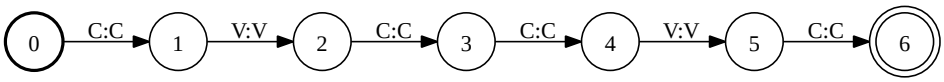


As in our construction of the plural transducer, we let V:V abbreviate the identity on the vowels, and similarly for the consonants C in the transducers below.

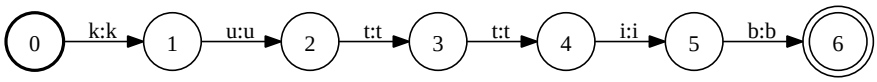
Let **vtemplate(ui)** be the Id transducer that allows any number of Cs between the vowels:



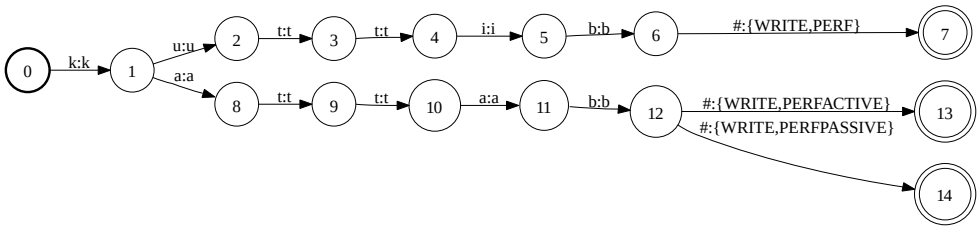
cvskeleton(cvccvc)



ctemplate(kttb)∘cvskeleton(cvccvc)∘vtemplate(ui) = Id(kutib):



Composition with restricted identity relations is really intersection. To define composition operation that intersects, while accumulating outputs, so that, for example,



(5) **2 important points:**

- Notice that *kuttib* and *kattab* must have separate paths after the first symbol, since we must enforce the constraints between the first and second vowel. This means that we end up having multiple copies of the /ui/ transducer. For some problems (e.g. handling a whole dictionary with all morphological changes), this can lead to enormous transducers.
- Parsing can be regarded as the composition of Id(input) with a transducer that accepts the language.

  (This idea mentioned in last Thursday's talk: 'parser states are grammars'.)

  When the (possibly nondeterministic) transducers have n and m states respectively, this step can be done in $\mathcal{O}(nm)^2$ steps [118, §4.3.3].

(6) Instead of calculating the large composed transducers, we can parse the input simultaneously with the two smaller, separate transducers.

7.1 Beyond finite state morphology

(7) **Thm** The language $XX = \{xx|\ x \in \Sigma^*\}$ is not regular.

Proof sketch: Assuming that Σ has more than one element (as standard), let $\Sigma = \{a, b, \ldots\}$. Then $XX \cap a^*ba^*b = \{a^nba^nb|\ n \geq 0\}$ By Nerode's theorem and the closure of regular languages under intersections, it is easy to show that this language is not regular, as we did for the similar language $\{a^na^n|\ n \geq 0\}$. □

(8) **Plurals in Pima.** [185]

C-copying	'lion'	mávit	mámvit
	'orange'	nálash	nánlash
CV-copying	'rock'	hódai	hóhodai
	'peach'	ñúlash	ñúñulash

Since there are finitely many CV pairs, this is in principle finite state, but the finite state representation *misses the simple generalization*!

(9) **Habitual-repetitive in Javanese.** [73]

'buy'	tuku	tuka-tuku
'puff'	bul	bal-bul
'walk'	melaku	meloka-melaku
'relapse'	kumat	kumat-kumet

(10) There are many many languages with this kind of copying in their morphophonologies. English has some [187], but less than most other languages, and still one finds reduplication creeping into speech/spelling errors [276] and into motherese [162].

We will look at grammars and transducers with copying later.

(11) **Controversies:**

a. ...the computational knights have presented themselves at the Royal Court of Linguistics, rushed up to the Princess of Phonology and Morphology in great excitement to deliver the same message "Dear Princess. I have wonderful news for you: You are not like some of your NP-complete sisters. You are regular. You are rational. You are finite state. Please marry me. Together we can do great things." And time after time, the put-down response from the Princess has been the same: "Not interested. You do not understand Theory. Go away you geek." [137]

- The "You are finite state" remark is odd. Of course, every finite creature – at least, every finite computing system of the sorts we can understand now – is a finite state system. That does not mean that all such systems are well described as finite state systems.

  > Viewing a computer itself as a finite state system is not satisfying mathematically or realistically... To properly capture the notion of computation we need a potentially infinite memory, even though each computer installation is finite... It is also tempting to view the human brain as a finite state system... [but] the number of states is so large that this approach is unlikely to result in useful observations about the human brain, any more than finite state assumptions help us understand large but finite computer systems. [119, p.16]

- Suppose linguists are interested in understanding what kinds of mechanisms people use in morphology. A reasonable hunch is that the phonological and morphological processes found in human languages are there because they are natural options, given the nature of our language and our memories. So then...

` the 'inelegance' of the treatments of compositions (as noted for example in the Arabic above) suggests (though not decisively) that these operations are not what people do. (I.e. that the range of finite state operations is too small.)

` the inability to represent the very natural operation of copying (noticed, grammatical repetition), except with a brute-force enumeration of all the finite options, suggests that this is not what people do. (I.e. the range of finite state operations is too small.)

` many finite state operations that are definable are not found in human morphologies. (I.e. the range of finite state operations is too big.)

b. ...if someone is interested in how humans process morphology, rather than the formal computational description of such processing, they might be inclined to wonder [upon being shown finite state implementations of alternative proposals]: "so what?" But this clearly appeals to psycholinguistic evidence, and there the data does not seem particularly friendly to either side: there is increasing evidence that human morphological processing involves a powerful set of analogical reasoning tools that are sensitive to various effects including frequency, semantic and phonetic similarity, register, and so on [115, 11]. [224, pp.66,86]

- the "...rather than ..." phrase in the first sentence is odd, since we can perfectly well aim for a formal, computational description of how humans process morphology

- The traditional strategy is to look for linguistically conditioned (structural) effects first, and the <u>obvious</u> fact that we are sensitive to 'various effects including frequency, semantic and phonetic similarity, register, and so on' does not show that there are no structural effects!

  (The motivation behind this traditional strategy is also obvious: it is only feasible to model effects of causes that we have some good theory about.)

  Ultimately, of course, we want models that capture frequency and semantic effects appropriately. In the present state of ignorance, that requires averaging across unknown but massive sources of influence with statistics.

§8 Segmentation and indeterminacy

8.0 Two segmentation problems

8.0.1 Segmenting the phones in the lexical entries

(0) We must now consider the diphthongs... Each of these sounds involves a change in quality within the one vowel. As a matter of convenience, they can be described as movements from one vowel to another. The first part of the diphthong is usually more prominent than the last. In fact, the last part is often so brief and transitory that it is difficult to determine its exact quality. Furthermore, contrary to the traditional transcriptions, the diphthongs often do not begin and end with any of the sounds that occur in simple vowels. [159, p.82]

(1) In the Mitton dictionary, we have the simple vowels:

Mitton:	i	I	e	&	A	0 (zero)	U	V	O	u	3	@
IPA:	i	ɪ	ɛ	æ	a	ɒ	ʊ	ʌ	ɛə	u	ɜ	ə
example:	bead	bid	bed	bad	bard	cod	good	but	cord	food	bird	about

and the diphthongs

Mitton:	eI	aI	oI	e@	@U	aU	I@	U@
IPA:	eɪ	aɪ	oɪ	ɛə	oʊ	aʊ	ɪə	ʊə
example:	day	eye	boy	bare	go	cow	beer	tour

As Ladefoged warns, in these transcriptions, diphthongs have symbols in common with the simple vowels.

(2) That means there are potential parsing ambiguities, ambiguities that could matter for syllabification, stress assignment, etc.

If there are never more than 2 vowels in a sequence, we could just say: always prefer diphthong analyses when possible. But this policy might not suffice for longer sequences, where there could there be "ties"! That is, there could two different analyses both use the same number of diphthongs.

Checking to see whether this happens, even in a dictionary where some of the syllable breaks are indicated by stress marks, it is easy to write a program that collects and counts maximal vowel sequences in the dictionary:

49512	I	97	@U@	3	i0
30671	@	94	eI@	3	A0
11535	&	92	iI	2	Ue
11296	e	68	UeI	2	@UaU
9710	eI	54	oI@	2	uA
7939	0	44	I&	2	oIeI
7124	V	32	OI	2	oI&
6463	i	24	aUI	2	O@
6404	aI	22	oII	2	IO
6250	@U	16	i@U	2	eIi
4690	O	16	IA	2	eIA
4487	u	15	UI	2	aIu
4179	A	15	aI&	2	aIi
3095	3	10	I0	2	AI
3056	I@	9	Ie	1	u@U
1948	aU	7	eI@U	1	Ui
1886	U	7	A@U	1	@Ue
961	e@	7	aI@U	1	UaI
840	aI@	4	@UII	1	u0
751	U@	4	@UI@	1	&@U
708	oI	4	U&	1	@U&
579	II	4	iO	1	I@UI
226	IeI	4	Ii	1	I@U&
205	aU@	4	Ie@	1	ieI
184	uI	4	eI0	1	@I
178	@UI	4	aIA	1	eIe
163	aII	3	uII		
130	i@	3	uI@		
114	eII	3	ueI		
108	u@	3	ie		
107	I@U	3	IaI		

Notice that @U@ appears in 97 entries! Should these be parsed ["@U";"@"] or ["@";"U@"]? E.g.,

```
("coalition",",k@U@'lISn");
("feather-boa",",feD@-'b@U@");
("slower","'sl@U@R");
("yellower","'jel@U@R");
```

Squib topic: Produce a version of mitton2.ml where the phones are correctly parsed, briefly reporting on your strategy and the code you write to assist in the task.

8.0.2 Segmenting the morphs in an utterance

(3) Segmenting the morphs from continuous speech presents many more ambiguities than we have in parsing the dictionary pronunciations.

In ordinary conversation, phones can be dropped or obscured by noise, but we find it even in the artificial situation where we are given perfect dictionary entry sequences, but without spaces between words. Consider, for example,

'aIskrim

Here are some relevant Mitton entries:

```
("I","aI");
("ay","aI");
("aye","aI");
("eye","aI");
("i","aI");
("ice","aIs");
("cream","krim");
("scree","skri");
("scream","skrim");
("ice-cream","aIs-'krim");
("ice-creams","aIs-'krimz");
("ice-lollies","aIs-'lOlIz");
("iceberg","'aIsb3g");
```

Oddly, some of these phonetic entries contain a hyphen, which is not pronounced; and there is no singular "ice-lolly"; but worse, these entries do not have the normal American compound stress, which we see only in `iceberg`. I think the ice-cream entries should be:

```
("ice-cream","'aIskrim");
("ice-creams","'aIskrimz");
("ice-lollies","'aIslOlIz");
```

The OED agrees with Mitton, so apparently British English differs from American English in this respect.

So there are 2 problems here:

The parsing problem. How to choose among the possible segmentations of an input sequence?

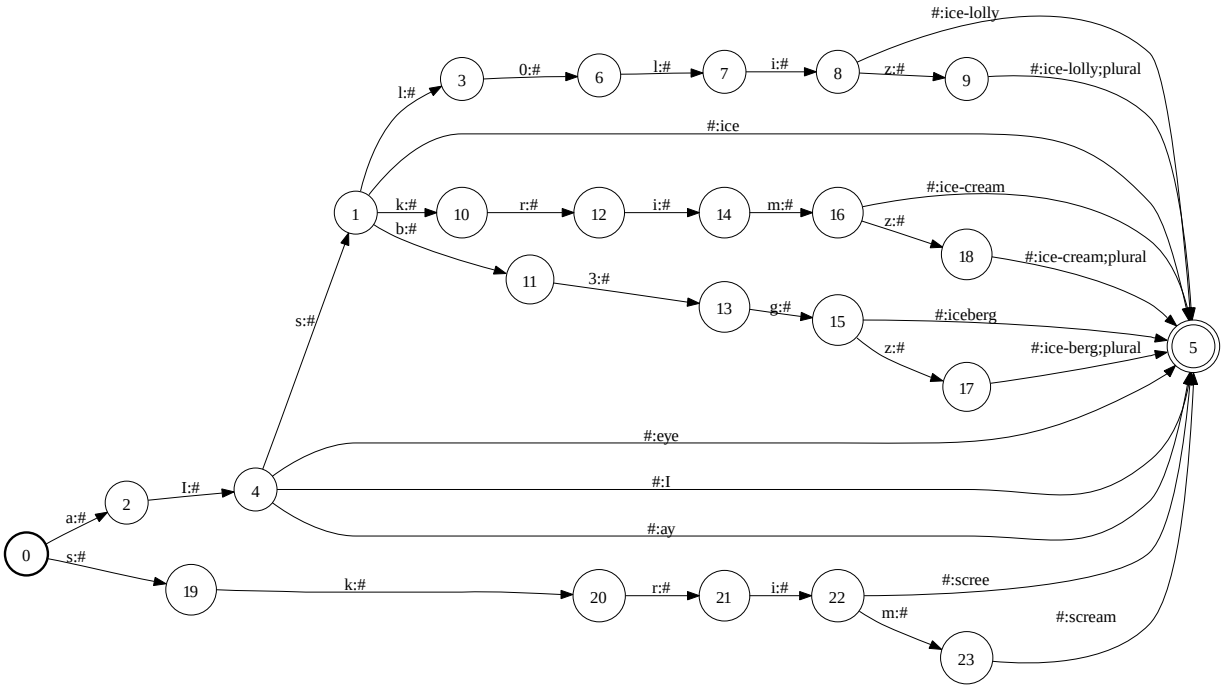
The 'noise' and 'variation' problem. How can you recognize something that is not pronounced exactly the way you would pronounce it? (e.g. with different stress)

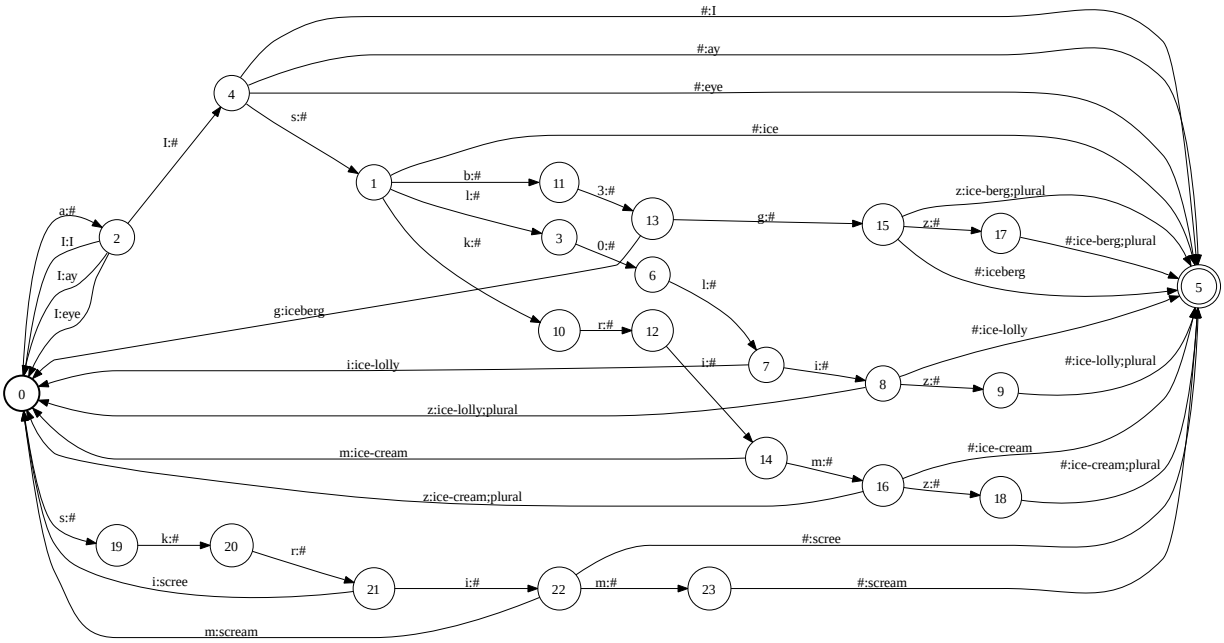
Let's ignore stress and other possibly variable aspects of pronunciation for the moment, and work on the parsing problem first. We return to 'noise' and 'variation' below.

(4) We will consider 3 very different responses to the parsing problem:

- **backtracking** – find one solution at a time, backtracking if necessary
- **all paths at once** – find all possible analyses
- **probabilistic methods** – find only the 'most probable' solution

8.1 Example





8.2 Backtracking

(5) A backtracking system takes the input, the finite state machine, and collects the output, collecting all next-step choices on a stack.

At each step, taking the first (state,remainingInput,OutputSoFar) off the stack, the next input symbol is processed, putting all choices onto the stack.

```
(* a simple k-subsequential finite state transducer SSFSA *)
(* That is: at most k empty transitions to the final states *)

(* This k-SSFSA is a list of ((PrevState,NextIn),(NextOut,NextState)) pairs of pairs *)
(* We could put these into a hash table, but there are so few, it's not worth the trouble *)
let dictSSFSA = [
  ((0,'a'),([],1));
  ((1,'I'),([],2));
  ((2,'\255'),(["I"],3));
  ((2,'\255'),(["ay"],3));
  ((2,'\255'),(["eye"],3));
  ((2,'s'),([],4));
  ((4,'\255'),(["ice"],3));
  ((4,'k'),([],5));
  ((5,'r'),([],6));
  ((6,'i'),([],7));
  ((7,'m'),([],8));
  ((8,'\255'),(["ice-cream"],3));
  ((8,'z'),([],9));
  ((9,'\255'),(["ice-cream";"plural"],3));
  ((4,'l'),([],10));
  ((10,'0'),([],11));
  ((11,'l'),([],12));
  ((12,'i'),([],13));
  ((13,'\255'),(["ice-lolly"],3));
  ((13,'z'),([],14));
  ((14,'\255'),(["ice-lolly";"plural"],3));
  ((4,'b'),([],15));
  ((15,'3'),([],16));
  ((16,'g'),([],17));
  ((17,'\255'),(["iceberg"],3));
  ((17,'z'),([],18));
  ((18,'\255'),(["ice-berg";"plural"],3));
  ((0,'s'),([],19));
  ((19,'k'),([],20));
  ((20,'r'),([],21));
  ((21,'i'),([],22));
  ((22,'\255'),(["scree"],3));
  ((22,'m'),([],23));
  ((23,'\255'),(["scream"],3));
];;

let rec transduceSSFSA ssfsa state input = match input with
    [] -> fsa (List.assoc (state,'\255') ssfsa)
  | c::cs ->
      let (output,nextState) = List.assoc (state,c) ssfsa
      in
        begin
          Printf.fprintf stdout "(%i,%c)->(_,%i)\n" state c nextState;
          List.append output (transduceSSFSA ssfsa nextState cs);
        end;;

(* examples *)
transduceSSFSA dictSSFSA 0 ['a';'I'];;
transduceSSFSA dictSSFSA 0 ['a';'I';'s'];;
transduceSSFSA dictSSFSA 0 ['a';'I';'s';'k';'r';'i';'m'];;
transduceSSFSA dictSSFSA 0 ['a';'I';'s';'k';'r';'i';'m';'z'];;
transduceSSFSA dictSSFSA 0 ['a';'I';'s';'b';'3';'g'];;
transduceSSFSA dictSSFSA 0 ['a';'I';'s';'b';'3';'g';'z'];;
transduceSSFSA dictSSFSA 0 ['a';'I';'s';'l';'0';'l';'i'];;
transduceSSFSA dictSSFSA 0 ['a';'I';'s';'l';'0';'l';'i';'z'];;
transduceSSFSA dictSSFSA 0 ['s';'k';'r';'i'];;
transduceSSFSA dictSSFSA 0 ['s';'k';'r';'i';'m'];;
```

```
(* a ** cyclic **  nondeterministic finite state transducer FSA *)
(* still: at most k empty transitions to the final states *)
(* This FSA is a list of ((PrevState,NextIn),(NextOut,NextState)) pairs of pairs *)
(* We could put these into a hash table, but there are so few, it's not worth the trouble *)
let dictFSA = [
  ((0,'a'),([],1));
  ((1,'I'),([],2));
  ((2,'\255'),(["I"],3));
  ((2,'\255'),(["ay"],3));
  ((2,'\255'),(["eye"],3));
  ((2,'s'),([],4));
  ((4,'\255'),(["ice"],3));
  ((4,'k'),([],5));
  ((5,'r'),([],6));
  ((6,'i'),([],7));
  ((7,'m'),([],8));
  ((8,'\255'),(["ice-cream"],3));
  ((8,'z'),([],9));
  ((9,'\255'),(["ice-cream";"plural"],3));
  ((4,'l'),([],10));
  ((10,'0'),([],11));
  ((11,'l'),([],12));
  ((12,'i'),([],13));
  ((13,'\255'),(["ice-lolly"],3));
  ((13,'z'),([],14));
  ((14,'\255'),(["ice-lolly";"plural"],3));
  ((4,'b'),([],15));
  ((15,'3'),([],16));
  ((16,'g'),([],17));
  ((17,'\255'),(["iceberg"],3));
  ((17,'z'),([],18));
  ((18,'\255'),(["ice-berg";"plural"],3));
  ((0,'s'),([],19));
  ((19,'k'),([],20));
  ((20,'r'),([],21));
  ((21,'i'),([],22));
  ((22,'\255'),(["scree"],3));
  ((22,'m'),([],23));
  ((23,'\255'),(["scream"],3));
(* for each final arc, we add an arc back to 0 ** nondeterminacy enters here! *)
  ((1,'I'),(["I"],0));
  ((1,'I'),(["ay"],0));
  ((1,'I'),(["eye"],0));
  ((7,'m'),(["ice-cream"],0));
  ((8,'z'),(["ice-cream";"plural"],0));
  ((12,'i'),(["ice-lolly"],0));
  ((13,'z'),(["ice-lolly";"plural"],0));
  ((16,'g'),(["iceberg"],0));
  ((17,'z'),(["ice-berg";"plural"],3));
  ((21,'i'),(["scree"],0));
  ((22,'m'),(["scream"],0));
];;

let rec next fsa fsastate c cs outputSoFar states = match fsa with
    [] -> states
  | ((s0,i0),(o0,s1))::arcs ->
      if s0=fsastate && i0=c
      then
        begin
          Printf.fprintf stdout "(%i,%c)->(_,%i)\n" fsastate c s1;
          (s1,cs,o0@outputSoFar)::next arcs fsastate c cs outputSoFar states;
        end
      else next arcs fsastate c cs outputSoFar states;;

let rec printStrings list = match list with
    [] -> Printf.fprintf stdout "\n"
  | s::ss -> begin Printf.fprintf stdout "%s; " s; printStrings ss; end;;

let rec output fsa fsastate outputSoFar states =  match fsa with
    [] -> states
  | ((s0,i0),(o0,s1))::arcs ->
      if s0=fsastate && i0='\255'
      then
        begin
          printStrings (List.rev (o0@outputSoFar));
          output arcs fsastate outputSoFar states;
        end
      else output arcs fsastate outputSoFar states;;

(* returns list of resume states *)
let rec transduceFSA fsa resume = match resume with
    [] -> []
  | (fsastate,[],outputSoFar)::states -> transduceFSA fsa (output fsa fsastate outputSoFar states)
  | (fsastate,c::cs,outputSoFar)::states -> transduceFSA fsa (next fsa fsastate c cs outputSoFar states);;

(* examples *)
transduceFSA dictFSA [(0, ['a';'I'], [])];;
transduceFSA dictFSA [(0, ['a';'I';'s'], [])];;
transduceFSA dictFSA [(0, ['a';'I';'s';'k';'r';'i';'m'], [])];;
transduceFSA dictFSA [(0, ['a';'I';'s';'k';'r';'i';'m';'z'], [])];;
transduceFSA dictFSA [(0, ['a';'I';'s';'b';'3';'g'], [])];;
transduceFSA dictFSA [(0, ['a';'I';'s';'b';'3';'g';'z'], [])];;
transduceFSA dictFSA [(0, ['a';'I';'s';'l';'0';'l';'i'], [])];;
transduceFSA dictFSA [(0, ['a';'I';'s';'l';'0';'l';'i';'z'], [])];;
transduceFSA dictFSA [(0, ['s';'k';'r';'i'], [])];;
```

```
(* Adding prompt for next solution to the ** cyclic **  nondeterministic FSA *)
(* still: at most k empty transitions to the final states *)
(* This FSA is a list of ((PrevState,NextIn),(NextOut,NextState)) pairs of pairs *)
let dictFSA = [
  ((0,'a'),([],1));                                  ((4,'l'),([],10));
  ((1,'I'),([],2));                                  ((10,'0'),([],11));
  ((2,'\255'),(["I"],3));                             ((11,'l'),([],12));
  ((2,'\255'),(["ay"],3));                            ((12,'i'),([],13));
  ((2,'\255'),(["eye"],3));                           ((13,'\255'),(["ice-lolly"],3));
  ((2,'s'),([],4));                                  ((13,'z'),([],14));
  ((4,'\255'),(["ice"],3));                           ((14,'\255'),(["ice-lolly";"plural"],3));
  ((4,'k'),([],5));                                  ((4,'b'),([],15));
  ((5,'r'),([],6));                                  ((15,'3'),([],16));
  ((6,'i'),([],7));                                  ((16,'g'),([],17));
  ((7,'m'),([],8));                                  ((17,'\255'),(["iceberg"],3));
  ((8,'\255'),(["ice-cream"],3));                     ((17,'z'),([],18));
  ((8,'z'),([],9));                                  ((18,'\255'),(["ice-berg";"plural"],3));
  ((9,'\255'),(["ice-cream";"plural"],3));            ((21,'i'),([],22));
  ((0,'s'),([],19));                                 ((22,'\255'),(["scree"],3));
  ((19,'k'),([],20));                                ((22,'m'),([],23));
  ((20,'r'),([],21));                                ((23,'\255'),(["scream"],3));
(* for each final arc, we add an arc back to 0 ** nondeterminacy enters here! *)
  ((1,'I'),(["I"],0));                               ((12,'i'),(["ice-lolly"],0));
  ((1,'I'),(["ay"],0));                              ((13,'z'),(["ice-lolly";"plural"],0));
  ((1,'I'),(["eye"],0));                             ((16,'g'),(["iceberg"],0));
  ((7,'m'),(["ice-cream"],0));                        ((17,'z'),(["ice-berg";"plural"],3));
  ((8,'z'),(["ice-cream";"plural"],0));               ((21,'i'),(["scree"],0));
                                                     ((22,'m'),(["scream"],0));
];;

let rec next fsa fsastate c cs outputSoFar states = match fsa with
    [] -> states
  | ((s0,i0),(o0,s1))::arcs ->
      if s0=fsastate && i0=c
      then
        begin
          Printf.fprintf stdout "(%i,%c)->(_,%i)\n" fsastate c s1;
          (s1,cs,o0@outputSoFar)::next arcs fsastate c cs outputSoFar states;
        end
      else next arcs fsastate c cs outputSoFar states;;

let rec printStrings list = match list with
    [] -> Printf.fprintf stdout "\n"
  | s::ss -> begin Printf.fprintf stdout "%s; " s; printStrings ss; end;;

exception HappyToBeOfServiceComeAgain;;

let promptForStop () =
  begin
    Printf.fprintf stdout "more? (y or n) ";
    if (read_line ()).[0] = 'n' then raise (HappyToBeOfServiceComeAgain) else ();
 end;;

let rec output fsa fsastate outputSoFar states =  match fsa with
    [] -> states
  | ((s0,i0),(o0,s1))::arcs ->
      if s0=fsastate && i0='\255'
      then
        begin
          printStrings (List.rev (o0@outputSoFar));
          promptForStop ();
          output arcs fsastate outputSoFar states;
        end
      else output arcs fsastate outputSoFar states;;

let apologize () = Printf.fprintf stdout "No (more) solutions.\n";;

(* returns list of resume states *)
let rec transduceFSA fsa resume = match resume with
    [] -> begin apologize ();  []; end
  | (fsastate,[],outputSoFar)::states -> transduceFSA fsa (output fsa fsastate outputSoFar states)
  | (fsastate,c::cs,outputSoFar)::states -> transduceFSA fsa (next fsa fsastate c cs outputSoFar states);;

(* examples *)
let eg0 () = transduceFSA dictFSA [(0, ['a';'I'], [])];;
let eg1 () = transduceFSA dictFSA [(0, ['a';'I';'s'], [])];;
let eg2 () = transduceFSA dictFSA [(0, ['a';'I';'s';'k';'r';'i';'m'], [])];;
let eg3 () = transduceFSA dictFSA [(0, ['a';'I';'s';'k';'r';'i';'m';'z'], [])];;
let eg4 () = transduceFSA dictFSA [(0, ['a';'I';'s';'b';'3';'g'], [])];;
let eg5 () = transduceFSA dictFSA [(0, ['a';'I';'s';'b';'3';'g';'z'], [])];;
let eg6 () = transduceFSA dictFSA [(0, ['a';'I';'s';'l';'0';'l';'i'], [])];;
let eg7 () = transduceFSA dictFSA [(0, ['a';'I';'s';'l';'0';'l';'i';'z'], [])];;
```

8.3 An exercise in 5 easy parts

Get a copy of the backtracking transducer above, which I called `icecream1.ml`. Recall that the 7 symbol input

```
['a';'I';'s';'k';'r';'i';'m']
```

produces 3 different analyses. Send me a session transcript with the following:

(1) Use `List.length` to calculate the number of arcs in the transducer.

(2) Modify `transduceFSA` in a minimal way so that at each step, the next input symbol and the number of elements on the backtrack stack is printed, like this:

```
# transduceFSA dictFSA [(0, ['a';'I';'s'], [])];;
Next input symbol:a  Stack depth:1
(0,a)->(_,1)
Next input symbol:I  Stack depth:1
(1,I)->(_,2)
(1,I)->(_,0)
(1,I)->(_,0)
(1,I)->(_,0)
Next input symbol:s  Stack depth:4
(2,s)->(_,4)
Stack depth:4
ice;
Next input symbol:s  Stack depth:3
(0,s)->(_,19)
Stack depth:3
Next input symbol:s  Stack depth:2
(0,s)->(_,19)
Stack depth:2
Next input symbol:s  Stack depth:1
(0,s)->(_,19)
Stack depth:1
- : 'a list = []
```

(3) Using this transducer, which 7 symbol input produces the most different outputs?

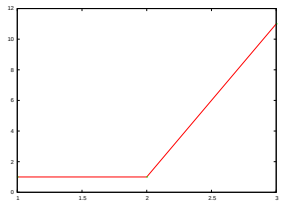
(4) For each symbol x in the input, let the *backtrack cost of* x be the sum of the backtrack stack lengths reported at that symbol. So in the example above the costs are a=1, I=1, s=11.

Summing the backtrack cost for each symbol and dividing by the length of the string gives an *average per-symbol backtrack cost*. So in the example above the average is $4\frac{1}{3}$.

Which 7 symbol input has the highest per-symbol backtrack cost? (* Show the session, and put your calculated per-symbol backtrack cost in a comment. *)

(5) (* In a brief comment, explain why the backtrack machine is not finite state, even though the transducer has only finitely many states. *)

(Optional) Various methods are used by psycholinguists to determine how much "effort" you are spending on reading or listening to each word in a presented sentence, and sometimes they display their results in word-by-word complexity graphs. Draw a character-by-character complexity graph showing the total stack depths at each character in the word "eiskrimz". That is, for each character, add up all the stack depths recorded at that position.

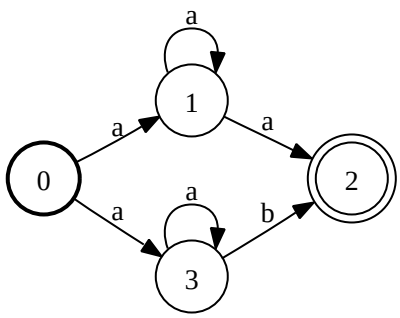


(Optional) Write a function that will take a transducer in the format used above, and write out all the state to state links in `dot` format. Or use the ATT FSM tools and the code in my file `attFST.ml` to draw the transducer.

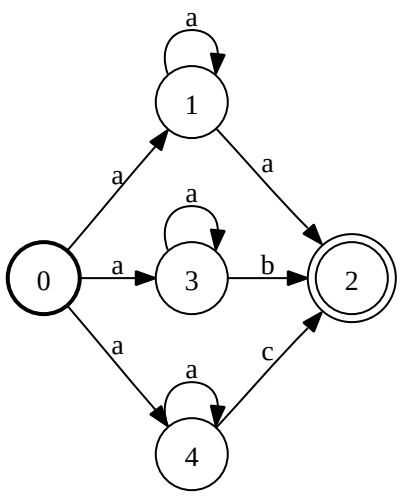
8.4 The complexity of backtracking

(6) **Thm.** Given an input of length n, finding an analysis using an nondeterministic acceptor or transducer using backtracking can take exponentially many steps, that is, c^n steps for some constant c.

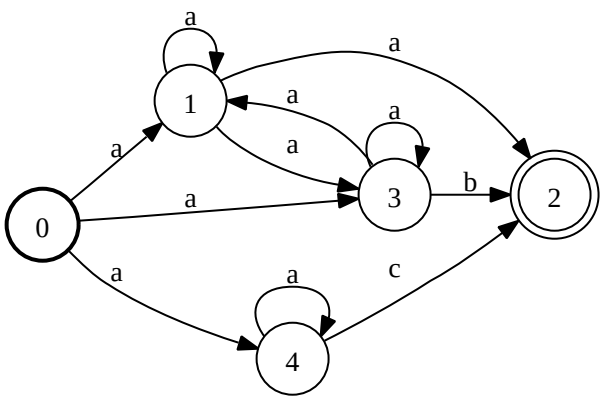
Proof: This can be established with example. Consider this transducer:



If the transducer takes the uppermost paths first in an attempt to parse an input $a^{n+1}b$, the analysis will visit the states 01^n03^n2, for a total of $2n+2$ steps. If we add another path like this:



With this machine, an attempt to parse $a^{n+1}b$, the analysis will visit the states $01^n03^n04^n2$, for a total of $3n+3$ steps. To make things worse, we just need to provide more false paths, as for example in this acceptor:



With this machine, an attempt to parse $a^{n+1}b$, the analysis will visit the states $0\{1,3\}^n042$, for a total of $2^n + 3$ steps.

(7) **Corollary.** Obviously, in an acceptor or transducer with empty input transitions, an attempt to analyze an input of length n can fail to terminate.

§9 indeterminacy: paths and probabilities

9.0 All paths at once

This is the strategy we have already studied!

(0) With any transducer, deterministic or not, you can get a representation of all possible analyses by intersecting Pt(input) with the word sequence transducer, or equivalently, composing the Id(Pt(input)) transducer with the word sequence transducer,

If you only want to know whether one of these analyses succeeds, you can throw away the states you've reached in previous steps: at each step through the input you need only compute all the next states you can get to.

WE WILL WORK OUT AN EXAMPLE IN CLASS

(1) As we mentioned before, this step requires at most $\mathcal{O}n^2$ steps, so it is, in the worst case, very much slower than using a deterministic transducer, but very much better than backtracking.

We will have more to say about this method later, because it generalizes to other language classes that are closed under intersection with regular languages.

9.1 PFSGs for most probable solutions

For a good survey of basic results on probabilistic finite state grammars/automata, see e.g. [264]. Mohri has some recent important results [182, 5].

(2) A **probabilistic regular (right linear) grammar** $G = \langle Cat, \Sigma, \rightarrowtail, S, P\rangle$ has these parts:

$$\begin{array}{rl} Cat & \text{finite set of Categories} \\ \Sigma & \text{finite nonempty set of vocabulary items disjoint from } C \\ \rightarrowtail \subseteq (Cat \times \Sigma Cat) \cup (Cat \times \{\epsilon\}) & \text{a finite set of 'rules'} \\ S \in Cat & \text{start category} \\ P : (\rightarrowtail) \rightarrow [0,1] & \text{such that for each } C \in Cat, \sum_{\alpha \in V^*} p(C \rightarrow \alpha) = 1 \end{array}$$

(3) For each rewrite grammar, letting $V = \Sigma \cup Cat$, we define the relation $\Rightarrow \subseteq V^* \times V^*$ as follows: $x \Rightarrow z$ iff there are $t, u, v, w \in V^*$ such that $x = tuv$, $u \rightarrowtail w$, and $z = twv$.

We will call $x \Rightarrow z$ a *step using rule* $u \rightarrowtail w$.

(4) As before, $\Rightarrow^*$ is the reflexive, transitive closure of $\Rightarrow$, and $L(G) = \{x \in \Sigma^* |\ S \Rightarrow^* x\}$.

(5) An n-step derivation is a sequence of elements related by $\Rightarrow$,

$$S \Rightarrow \alpha_0 \Rightarrow \ldots \Rightarrow \alpha_n$$

for $\alpha_n \in \Sigma^*$, using n rules $A_i \rightarrow \beta_i$ for $0 \leq i \leq n$, where $A_0 = S, \alpha_0 = \beta_0$.

Let $\Gamma(G)$ be the set of derivations.

(6) Each such derivation $S \Rightarrow \ldots \Rightarrow \alpha_n$, using rules $A_i \rightarrow \alpha_i$ for $0 \leq i \leq n$ has probability

$$\Pi_{0 \leq i \leq n} P(A_i \rightarrow \alpha_i).$$

(7) For $s \in \Sigma^*$, let $\Gamma(G, s)$ be the set of derivations terminating in s. Let's say

$$p(s) = \sum_{d \in \Gamma(G,s)} p(d).$$

(8) **Thm.** If G has no useless states, and no rules have probability 0, then

$$\sum_{s \in L(G)} p(s) = 1.$$

(9) Are there regular languages L with probability function p on Σ^* such that the set with "support," $\{x \in L|\ p(x) > 0\}$, is not regular? Yes, of course.

Can it happen that there is a probability function p on Σ^* such that $\{x \in L|\ p(x) > 0\}$ is regular, but there is no probabilistic regular grammar G such that $p = p_G$? Yes, this can happen too [79, 74, 157].

(10) **Example.** We could use some estimate of relative frequencies of the words to change our "ice cream" transducer into a weighted transducer.

Note that, in accord with our grammar-based definition above, there is only 1 start state, and that each final state needs to have a "stop" probability.

That transducer is not at all accurate though, in the sense that typical pronunciations of "ice cream" are different from typical pronunciations of "I scream". If these small variants in pronunciation were appropriately represented and assigned probabilities, we could get a much better parser for phone sequences.

(11) **Computing the most probable parse: the 'Viterbi method' [267].**

Viterbi was a Professor at UCLA from 1963 to 1973. In 1967 he published a famous paper that had the basic insight is that at any point in computing the probability of a derivation, we need only the probability of reaching the current state, and the probability of the transitions to the next state.

So in our machines which have a unique start state (corresponding to the 'start' category S), we start there with probability 1. Then for each state that can be reached, accepting the next symbol, we compute the probability of all the next states in the derivation. For the following symbols we compute maximum probabilities of reaching the next states, and so on.

The effect is often said to be a calculation with a trellis structure, since we must, for each possible current state, find the probability of reaching any of the next possible states, so that we can note the maximum resulting probabilities at all the next states.

At the end, the string is accepted only if at least one transition to has been taken, and we can find the most probable parse(s) by going to predecessors of that last step that have maximum probability.

9.1.1 Stochastic processes

(12) A **stochastic process** is a function X from times (or "indices") to random variables.

If the time is continuous, then $X : \mathbb{R} \to [\Omega \to \mathbb{R}]$, where $[\Omega \to \mathbb{R}]$ is the set of random variables.

If the time is discrete, then $X : \mathbb{N} \to [\Omega \to \mathbb{R}]$

(13) For stochastic processes X, instead of the usual argument notation $X(t)$, we use subscripts X_t, to avoid confusion with the arguments of the random variables.

So X_t is the value of the stochastic process X at time t, a random variable.

When time is discrete, for $t = 0, 1, 2, \ldots$ we have the sequence of random variables $X_0, X_1, X_2, \ldots$

(14) We will consider primarily discrete time stochastic processes, that is, sequences $X_0, X_1, X_2, \ldots$ of random variables.

So X_i is a random variable, namely the one that is the value of the stochastic process X at time i.

(15) $X_i = q$ is interpreted as before as the event (now understood as occurring at time i) which is the set of outcomes $\{e|\ X_i(e) = q\}$.

So, for example, $P(X_i = q)$ is just a notation for the probability, at time i, of an outcome that is named q by X_i, that is, $P(X_i = q)$ is short for $P(\{e|\ X_i(e) = q\})$.

(16) Notice that it would make perfect sense for all the variables in the sequence to be identical, $X_0 = X_1 = X_2 = \ldots$. In that case, we still think of the process as one that occurs in time, with the same classification of outcomes available at each time.

Let's call a stochastic process **time-invariant** (or stationary) iff all of its random variables are the same function. That is, for all $q, q' \in \mathbb{N}, X_q = X_{q'}$.

(17) A **finite stochastic process** X is one where sample space of all the stochastic variables, $\Omega_X = \bigcup_{i=0}^{\infty} \Omega_{X_i}$ is finite.

The elements of Ω_X name events, but in this context the elements of Ω_X are often called states.

(18) A stochastic process $X_0, X_1, \ldots$ is **first order** iff for each $0 \leq i$, $\sum_{x \in X_i} P(x) = 1$ and the events in Ω_{X_i} are all independent of one another.

(Some authors number from 0, calling this one 0-order).

(19) A stochastic process $X_0, X_1, \ldots$ has the **Markov property** (that is, it is **second order**) iff the probability of the next event may depend only on the current event, not on any other part of the history.

That is, for all $t \in \mathbb{R}$ and all $q_0, q_1, \ldots \in \Omega_X$,

$$P(X_{t+1} = q_{t+1} | X_0 = q_0, \ldots, X_t = q_t) = P(X_{t+1} = q_{t+1} | X_t = q_t)$$

The Russian mathematician Andrei Andreyevich Markov (1856-1922) was a student of Pafnuty Chebyshev. He used what we now call Markov chains in his study of consonant-vowel sequences in poetry.

(20) A **Markov chain** or Markov process is a stochastic process with the Markov property.

(21) A **finite Markov chain**, as expected, is a Markov chain where the sample space of the stochastic variables, Ω_X is finite.

(22) It is sometimes said that an n-state Markov chain can be specified with an $n \times n$ matrix that specifies, for each pair of events $s_i, s_j \in \Omega_X$ the probability of next event s_j given current event s_i.

Is this true? Do we know that for some X_j where $i \neq j$ that $P(X_{i+1} = q' | X_i = q) = P(X_{j+1} = q' | X_j = q)$? No.[1] For example, we can perfectly well allow that $P(\{e|\ X_{i+1}(e) = q'\}) \neq P(\{e|\ X_{j+1}(e) = q'\})$, simply by letting $\{e|\ X_{i+1}(e) = q'\}) \neq \{e|\ X_{j+1}(e) = q'\}$. This can happen quite naturally when the functions X_{i+1}, X_{j+1} are different, a quite natural assumption when these functions have in their domains outcomes that happen at different times.

The condition that disallows this is: time-invariance, defined in (16) above.

(23) Given a time-invariant, finite Markov process X, the probabilities of events in Ω_X can be specified by

i. an "initial probability vector" which defines a probability distribution over $\Omega_x = \{q_0, q_1, \ldots, q_{n-1}\}$. This can be given as a vector, a $1 \times n$ matrix, $[P_0(q_0) \quad \ldots \quad P_0(q_{n-1})]$, where $\sum_{i=0}^{n-1} P(q_i) = 1$

ii. a $|\Omega_X| \times |\Omega_X|$ matrix of conditional probabilities, here called **transition** or **digram probabilities**, specifying for each $s_i, s_j \in \Omega_X$ the probability of next event/state q_i given current event/state q_j.

We introduce the notation $P(q_i|q_j)$ for $P(X_{t+1} = q_i | X_t = q_j)$.

(24) Given an initial probability distribution I on Ω_X and the transition matrix M, the probability of state sequence $q_0 q_1 q_2 \ldots q_n$ is determined. Writing $P_0(q_i)$ for the initial probability of q_i,

$$P(q_0 q_1 q_2 \ldots q_n) = P_0(q_0) P(q_1|q_0) P(q_2|q_1) \ldots P(q_n|q_{n-1})$$

Writing $P(q_i|q_j)$ for the probability of the transition from state q_j to state q_i,

$$\begin{aligned} P(q_1 \ldots q_n) &= P_0(q_1) P(q_2|q_1) \ldots P(q_n|q_{n-1}) \\ &= P_0(q_1) \textstyle\prod_{1 \leq i \leq n-1} P(q_{i+1}|q_i) \end{aligned}$$

(25) Given an initial probability distribution I on Ω_X and the transition matrix M, the probability distribution for the events of a finite time-invariant Markov process at time t is given by the matrix product IM^t. That is, at time 0 $P = I$, at time 1 $P = IM$, at time 2 $P = IMM$, and so on.

[1] The better written texts are careful about this, as in [194, p637] and [72, p165], for example.

9.2 Linear algebra review 0

(26) An $m \times n$ matrix A is an array with m rows and n columns.
Let $A(i,j)$ be the element in row i and column j.

$$\begin{array}{cc} \text{rows/cols} & \begin{array}{cccc} 1 & 2 & \dots & n \end{array} \\ \begin{array}{c} 1 \\ \dots \\ m \end{array} & \begin{bmatrix} a & b & \dots & d \\ & & \dots & \\ a & b & \dots & d \end{bmatrix} \end{array}$$

(27) We can add the $m \times n$ matrices A, B to get the $m \times n$ matrix $A + B = C$ in which
$$C(i,j) = A(i,j) + B(i,j).$$

(28) Matrix addition is associative and commutative:
$$A + (B + C) = (A + B) + C$$
$$A + B = B + A$$

(29) For any $n \times m$ matrix M there is an $n \times m$ matrix M' such that $M + M' = M' + M = M$, namely the $n \times m$ matrix M' such that every $M(i,j) = 0$.

(30) We can multiply an $m \times n$ matrix A and a $n \times p$ matrix to get an $m \times p$ matrix C in which
$$C(i,j) = \sum_{k=1}^{n} A(i,k)B(k,j).$$
This definition is sometimes called the "row by column" rule. To find the value of $C(i,j)$, you add the products of all the elements in row i of A and column j of B. For example,

$$\begin{bmatrix} 1 & 4 \\ 2 & 5 \end{bmatrix} \begin{bmatrix} 8 & 9 & 0 \\ 2 & 6 & 0 \end{bmatrix} = \begin{bmatrix} (1 \cdot 8) + (4 \cdot 2) & (1 \cdot 9) + (4 \cdot 6) & (1 \cdot 0) + (4 \cdot 0) \\ (2 \cdot 8) + (5 \cdot 2) & (2 \cdot 9) + (5 \cdot 6) & (2 \cdot 2) + (5 \cdot 0) \end{bmatrix}$$

Here we see that to find $C(1,1)$ we sum the products of all the elements row 1 of A times the elements in column 1 of B. – The number of elements in the rows of A must match the number of elements in the columns of B or else AB is not defined.

(31) Matrix multiplication is associative, but not commutative:
$$A(BC) = (AB)C$$
$$\text{For example, } \begin{bmatrix} 3 & 5 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \end{bmatrix} \neq \begin{bmatrix} 2 \\ 4 \end{bmatrix} \begin{bmatrix} 3 & 5 \end{bmatrix}$$
It is interesting to notice that Lambek's (1958) composition operator is also associative but not commutative:
$(X \bullet Y) \bullet Z) \Rightarrow X \bullet (Y \bullet Z)$
$X/Y \bullet Y \Rightarrow X$
$Y \bullet X/Y \not\Rightarrow X$
The connection between the Lambek calculus and matrix algebra is actually a deep one [196].

(32) For any $m \times m$ matrix M there is an $m \times m$ matrix I_m such that $TI_m = I_mT = T$, namely the $m \times m$ matrix I_m such that every $I_m(i,i) = 1$ and for every $i \neq j, I_m(i,j) = 0$.

(33) *Exercise:*

a. Explain why the claims in 28 are obviously true.

b. Do the calculation to prove that my counterexample to commutativity in 31 is true.

c. Explain why 32 is true.

d. Make sure you can use `octave` or some other system to do the calculations once you know how to do them by hand:

```
1% octave
Octave, version 2.0.12 (i686-pc-linux-gnulibc1).
Copyright (C) 1996, 1997, 1998 John W. Eaton.
This is free software with ABSOLUTELY NO WARRANTY.
For details, type 'warranty'.

octave:1> x=[1,2]
x =
  1  2

octave:2> y=[3;4]
y =
  3
  4

octave:3> z=[5,6;7,8]
z =
```

```
  5  6
  7  8

octave:4> x+x
ans =
  2  4

octave:5> x+y
error: operator +: nonconformant arguments (op1 is 1x2, op2 is 2x1)
error: evaluating assignment expression near line 5, column 2
octave:5> 2*x
ans =
  2  4

octave:6> x*y
ans = 11
octave:7> x*z
ans =
  19  22

octave:8> y*x
ans =
  3  6
  4  8

octave:9> z*x
error: operator *: nonconformant arguments (op1 is 2x2, op2 is 1x2)
error: evaluating assignment expression near line 9, column 2
```

(34) To apply the idea in (25), we will always be multiplying a $1 \times n$ matrix times a square $n \times n$ matrix, to get the new $1 \times n$ probability distribution for the events of the n state Markov process.

(35) For example, suppose we have a coffee machine that (upon inserting money and pressing a button) will do one of 3 things:
(q_1) produce a cup of coffee,
(q_2) return the money with no coffee,
(q_3) keep the money and do nothing.
Furthermore, after an occurrence of (q_2), following occurrences of (q_2) or (q_3) are much more likely than they were before. We could capture something like this situation with the following initial distribution for q_1, q_2 and q_3 respectively,

$$I = \begin{bmatrix} 0.7 & 0.2 & 0.1 \end{bmatrix}$$

and if the transition matrix is:

$$\mathrm{T} = \begin{bmatrix} 0.7 & 0.2 & 0.1 \\ 0.1 & 0.7 & 0.2 \\ 0 & 0 & 1 \end{bmatrix}$$

a. What is the probability of state sequence $q_1 q_2 q_1$?

$$P(q_1 q_2 q_1) = P(q_1)P(q_2|q_1)p(q_1|q_2) = 0.7 \cdot 0.2 \cdot 0.1 = 0.014$$

b. What is the probability of the states Ω_X at a particular time t?
At time 0 (maybe, right after servicing) the probabilities of the events in Ω_X are given by I.
At time 1, the probabilities of the events in Ω_X are given by

$$\begin{aligned} \mathrm{IT} &= \begin{bmatrix} 0.7 \cdot 0.7 + 0.2 \cdot 0.1 + 0.1 \cdot 0 & 0.7 \cdot 0.2 + 0.2 \cdot 0.7 + 0.1 \cdot 0 & 0.7 \cdot 0.1 + 0.2 \cdot 0.2 + 0.1 \cdot 1 \end{bmatrix} \\ &= \begin{bmatrix} 0.49 + 0.02 & 0.14 + 0.14 & 0.07 + 0.04 + .1 \end{bmatrix} \\ &= \begin{bmatrix} 0.51 & 0.28 & .21 \end{bmatrix} \end{aligned}$$

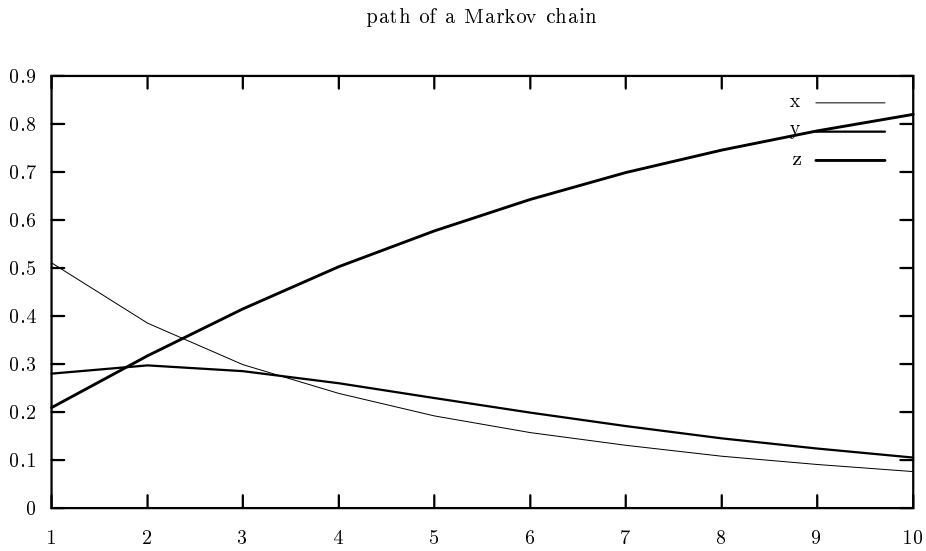
At time 2, the probabilities of the events in Ω_X are given by IT^2.
At time t, the probabilities of the events in Ω_X are given by IT^t.

```
octave:10> i=[0.7,0.2,0.1]
i =

  0.70000  0.20000  0.10000

octave:11> t=[0.7,0.2,0.1;0.1,0.7,0.2;0,0,1]
t =

  0.70000  0.20000  0.10000
  0.10000  0.70000  0.20000
  0.00000  0.00000  1.00000

octave:12> i*t
ans =

  0.51000  0.28000  0.21000

octave:13> i*t*t
ans =

  0.38500  0.29800  0.31700
```

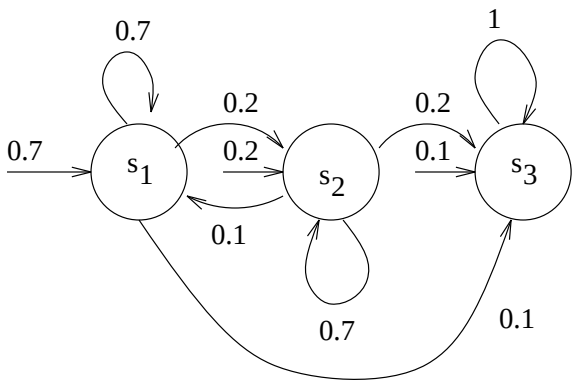
```
octave:14> i*t*t*t
ans =

  0.29930  0.28560  0.41510

octave:15> i*t*t*t*t
ans =

  0.23807  0.25978  0.50215

octave:16> i*(t**1)
ans =

  0.51000  0.28000  0.21000

octave:17> i*(t**2)
ans =

  0.38500  0.29800  0.31700

octave:18> result=zeros(10,4)
result =

  0  0  0  0
  0  0  0  0
  0  0  0  0
  0  0  0  0
  0  0  0  0
  0  0  0  0
  0  0  0  0
  0  0  0  0
  0  0  0  0
  0  0  0  0

octave:19>  for x=1:10
>   result(x,:)= [x,(i*(t**x))]
> endfor

octave:20>  result
result =

   1.000000   0.510000   0.280000   0.210000
   2.000000   0.385000   0.298000   0.317000
   3.000000   0.299300   0.285600   0.415100
   4.000000   0.238070   0.259780   0.502150
   5.000000   0.192627   0.229460   0.577913
   6.000000   0.157785   0.199147   0.643068
   7.000000   0.130364   0.170960   0.698676
   8.000000   0.108351   0.145745   0.745904
   9.000000   0.090420   0.123692   0.785888
  10.000000   0.075663   0.104668   0.819669

octave:21> gplot [1:10] result title "x",\
> result using 1:3 title "y", result using 1:4 title "z"
```

Gibert Strang's undergrad text *Introduction to Linear Algebra* is very readable, and videos of Strang's lectures are available on the web here `http://ocw.mit.edu/OcwWeb/Mathematics/` – you can get a good review of basic linear algebra, from the first few (or more!) of these lectures.

9.3 Markov chains and Markov models

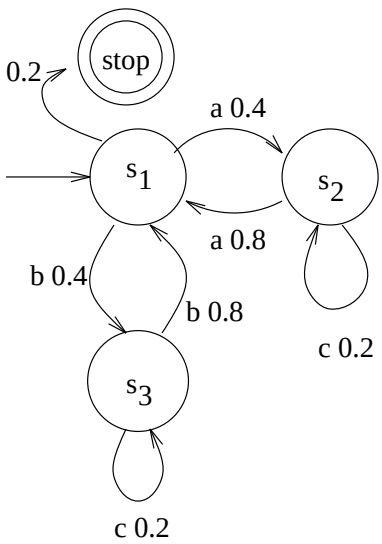


(36) Notice that the initial distribution and transition matrix can be represented by a finite state machine with no vocabulary and no final states:



(37) Notice that no Markov chain can be such that after a sequence of states *acccc* there is a probability of 0.8 that the next symbol will be an a, that is,
$P(a|acccc) = 0.8$
when it is also the case that
$P(b|bcccc) = 0.8$
This follows from the requirement mentioned in (23) that in each row i, the sum of the transition probabilities from that state $\sum_{q_j \in \Omega_X} P(q_j|q_i) = 1$, and so we cannot have both $P(b|c) = 0.8$ and $P(a|c) = 0.8$.

(38) Chomsky [44, p337] observes that the Markovian property that we see in state sequences does not always hold in regular languages. For example, the following finite state machine, to which we have added probabilities, is such that the probability of generating (or accepting) an a next, after generating *acccc* is $P(a|acccc) = 0.8$, and the probability of generating (or accepting) a b next, after generating *bcccc* is $P(b|bcccc) = 0.8$. That is, the strings show a kind of history dependence.



However the corresponding state sequences of this same machine are Markovian in some sense: they are not history dependent in the way the strings seem to be. That is, we can have both $P(q_1|q_1q_2q_2q_2q_2) = P(q_1|q_2) = 0.8$ and $P(q_1|q_1q_3q_3q_3q_3) = P(q_1|q_3) = 0.8$ since these involve transitions from different states. We will make this idea clearer in the next section.

(39) A Markov chain can be specified by an initial distribution and state-state transition probabilities can be augmented with stochastic outputs, so that we have in addition an initial output distribution and state-output emission probabilities.

One way to do this is to is to define a **Markov model** as a pair X, O where X is a Markov chain $X : \mathbb{N} \to [\Omega \to \mathbb{R}]$ and $O : \mathbb{N} \to [\Omega \to \Sigma]$ where the latter function provides a way to classify outcomes by the symbols $a \in \Sigma$ that they are associated with.

In a Markov chain, each number $n \in \mathbb{R}$ names an event under each X_i, namely $\{e|\ X_i(e) = n\}$.

In a Markov model, each output symbol $a \in \Sigma$ names an event in each O_i, namely $\{e|\ 0_i(e) = a\}$.

(40) In problems concerning Markov models where the state sequence is not given, the model is often said to be "hidden," a **hidden Markov model** (HMM).

See, e.g., Rabiner [214] for an introductory survey on HMMs. Some interesting recent ideas and applications appear in, e.g., [127], [126], [67], [69], [220], [219].

(41) Let's say that a Markov model (X, O) is a **Markov source** iff the functions X and O are "aligned" in the following sense:[2]

$$\forall e \in \not\leq,\ \forall i \in \mathbb{N},\ \forall n \in \mathbb{R},\ \exists a \in \Sigma, \qquad X_i(e) = n \text{ implies } O_i(e) = a$$

Then for every X_i, for all $n \in \Omega_{X_i}$, there is a particular output $a \in \Sigma$ such that $P(O_i = a | X_i = n) = 1$.

(42) Intuitively, in a Markov source, the symbol emitted at time i depends only on the state n of the process at that time. Let's formalize this idea as follows.

Observe that, given our definition of Markov source, when O_i extended pointwise to subsets of Ω, the set of outputs associated with outcomes named n has a single element, $O_i(\{e|\ X_i(e) = n\}) = \{a\}$.

So define $Out_i : \Omega_{X_i} \to \Sigma$ such that for any $n \in \Omega_{X_i}$, $Out_i(n) = a$ where $O_i(\{e|\ X_i(e) = n\}) = \{a\}$.

(43) Let a **pure Markov source** be a Markov model in which Out_i is the identity function on Ω_{X_i}.[3]

Then the outputs of the model are exactly the event sequences.

(44) Clearly, no Markov source can have outputs like those mentioned in (38) above, with $P(a|acccc) = 0.8$ and $P(b|bcccc) = 0.8$.

(45) Following Harris 1955 [113], a Markov source in which the functions Out_i are not 1-1 is a **grouped (or projected) Markov source**.

(46) The output sequences of a grouped Markov source may lack the Markov property. For example, it can easily happen that $P(a|acccc) = 0.8$ and $P(b|bcccc) = 0.8$.

This happens, for example, the 2-state Markov model given by the following initial state matrix I, transition matrix T and output matrix O:

$$I = [0.5 \quad 0.5]$$
$$\mathrm{T} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$
$$\mathrm{O} = \begin{bmatrix} 0.8 & 0 & 0.2 \\ 0 & 0.8 & 0.2 \end{bmatrix}$$

The entry in row i column j of the output matrix represents the probability of emitting the j'th element of $\langle a, b, c\rangle$ given that the system is in state i. Then we can see that we have described the desired situation, since the system can only emit a a if it is in state 1, and the transition table says that once the system is in state 1, it will stay in state 1. Furthermore, the output table shows that in state 1, the probability of emitting another a is 0.8. On the other hand, the system can only emit a b if it is in state 2, and the transition table says that once the system is in state 2, it will stay in state 2, with a probability of 0.8 of emitting another b.

(47) Miller and Chomsky 1964 [178, p427] say that any finite state automaton over which an appropriate probability measure is defined "can serve as" a Markov source, by letting the transitions of the finite state automaton correspond to states of a Markov source.

(Chomsky [44, p337] observes a related result by Schützenberger 1961 [232] which says that every regular language is the homomorphic image of a 1-limited finite state automaton.)

9.3.1 Computing the probabilities of output sequences: naive

(48) As noted in (24), given any Markov model and any sequence of states $q_1 \ldots q_n$,

$$P(q_1 \ldots q_n) = P_0(q_1) \prod_{1 \le i \le n-1} P(q_{i+1}|q_i) \tag{9.1}$$

Given $q_1 \ldots q_n$, the probability of output sequence $a_1 \ldots a_n$ is

[2] This is nonstandard. I think "Markov source" is usually just another name for a Markov model.

[3] With this definition, pure Markov sources are a special case of the general situation in which the functions Out_i are 1-1.

$$\prod_{t=1}^{n} P(a_t|q_t). \tag{9.2}$$

The probability of $q_1 \dots q_n$ occurring with outputs $a_1 \dots a_n$ is the product of the two probabilities (i) and (ii), that is,

$$P(q_1 \dots q_n, a_1 \dots a_n) = P_0(q_1) \prod_{1 \le i \le n-1} P(q_{i+1}|q_i) \prod_{t=1}^{n} P(a_t|q_t). \tag{9.3}$$

(49) Given any Markov model, the probability of output sequence $a_1 \dots a_n$ is the sum of the probabilities of this output for all the possible sequences of n states.

$$\sum_{q_i \in \Omega_X} P(q_1 \dots q_n, a_1 \dots a_n) \tag{9.4}$$

(50) Directly calculating this is infeasible, since there are $|\Omega_X|^n$ state sequences of length n.

9.3.2 Computing the probabilities of output sequences: forward

Here is a feasible way to compute the probability of an output sequence $a_1 \dots a_n$.

(51) a. Calculate, for each possible initial state $q_i \in \Omega_X$,

$$P(q_i, a_1) = P_0(q_i)P(a_1|q_i).$$

b. **Recursive step:** Given $P(q_i, a_1 \dots a_t)$ for all $q_i \in \Omega_X$, calculate $P(q_j, a_1 \dots a_{t+1})$ for all $q_j \in \Omega_X$ as follows

$$P(q_j, a_1 \dots a_{t+1}) = \big(\sum_{i \in \Omega_X} P(q_i, a_1 \dots a_t)P(q_j|q_i)\big)P(a_{t+1}|q_j)$$

c. Finally, given $P(q_i, a_1 \dots a_n)$ for all $q_i \in \Omega_X$,

$$P(a_1 \dots a_n) = \sum_{q_i \in \Omega_X} P(q_i, a_1 \dots a_n)$$

(52) Let's develop the coffee machine example from (35), adding outputs so that we have a Markov model instead of just a Markov chain. Suppose that there are 3 output messages:
(s_1) `thank you`
(s_2) `no change`
(s_3) `x@b*/!`
Assume that these outputs occur with the probabilities given in the following matrix where row i column j represents the probability of emitting symbol s_j when in state i:

$$\mathrm{O} = \begin{bmatrix} 0.8 & 0.1 & 0.1 \\ 0.1 & 0.8 & 0.1 \\ 0.2 & 0.2 & 0.6 \end{bmatrix}$$

Exercise: what is the probability of the output sequence

$$s_1 s_3 s_3$$

Solution sketch: (do it yourself first! note the trellis-like construction)

a. probability of the first symbol s_1 from one of the initial states

$$\begin{aligned} p(q_i|s_1) = p(q_i)p(s_1|q_i) &= \begin{bmatrix} 0.7 \cdot 0.8 & 0.2 \cdot 0.1 & 0.1 \cdot 0.2 \end{bmatrix} \\ &= \begin{bmatrix} 0.56 & 0.02 & 0.02 \end{bmatrix} \end{aligned}$$

b. probabilities of the following symbols from each state (transposed to column matrix)

$$
\begin{aligned}
p(q_i|s_1s_3)' &= \begin{bmatrix} ((p(q_1,s_1)\cdot p(q_1|q_1)) + (p(q_2,s_1)\cdot p(q_1|q_2)) + (p(q_3,s_1)\cdot p(q_1|q_3)))\cdot p(s_3|q_1) \\ ((p(q_1,s_1)\cdot p(q_2|q_1)) + (p(q_2,s_1)\cdot p(q_2|q_2)) + (p(q_3,s_1)\cdot p(q_2|q_3)))\cdot p(s_3|q_2) \\ ((p(q_1,s_1)\cdot p(q_3|q_1)) + (p(q_2,s_1)\cdot p(q_3|q_2)) + (p(q_3,s_1)\cdot p(q_3|q_3)))\cdot p(s_3|q_3) \end{bmatrix} \\
&= \begin{bmatrix} ((0.56\cdot 0.7) + (0.02\cdot 0.2) + (0.02\cdot 0))\cdot 0.1 \\ ((0.56\cdot 0.2) + (0.02\cdot 0.7) + (0.02\cdot 0))\cdot 0.1 \\ ((0.56\cdot 0.1) + (0.02\cdot 0.1) + (0.02\cdot 1))\cdot 0.6 \end{bmatrix} \\
&= \begin{bmatrix} (0.392 + 0.04)\cdot 0.1 \\ (0.112 + 0.014)\cdot 0.1 \\ (0.056 + 0.002 + 0.02)\cdot 0.6 \end{bmatrix} \\
&= \begin{bmatrix} 0.0432 \\ 0.0126 \\ 0.0456 \end{bmatrix}
\end{aligned}
$$

$$
\begin{aligned}
p(q_i|s_1s_3s_3)' &= \begin{bmatrix} ((p(q_1,s_1s_3)\cdot p(q_1|q_1)) + (p(q_2,s_1s_3)\cdot p(q_1|q_2)) + (p(q_3,s_1s_3)\cdot p(q_1|q_3)))\cdot p(s_3|q_1) \\ ((p(q_1,s_1s_3)\cdot p(q_2|q_1)) + (p(q_2,s_1s_3)\cdot p(q_2|q_2)) + (p(q_3,s_1s_3)\cdot p(q_2|q_3)))\cdot p(s_3|q_2) \\ ((p(q_1,s_1s_2)\cdot p(q_3|q_1)) + (p(q_2,s_1s_3)\cdot p(q_3|q_2)) + (p(q_3,s_1s_3)\cdot p(q_3|q_3)))\cdot p(s_3|q_3) \end{bmatrix} \\
&= \begin{bmatrix} ((0.0432\cdot 0.7) + (0.0126\cdot 0.2) + (0.0456\cdot 0))\cdot 0.1 \\ ((0.0432\cdot 0.2) + (0.0126\cdot 0.7) + (0.0456\cdot 0))\cdot 0.1 \\ ((0.0432\cdot 0.1) + (0.0126\cdot 0.1) + (0.0456\cdot 1))\cdot 0.6 \end{bmatrix} \\
&= \begin{bmatrix} (0.03024 + 0.00252)\cdot 0.1 \\ (0.00864 + 0.00882)\cdot 0.1 \\ (0.00432 + 0.00126 + 0.0456)\cdot 0.6 \end{bmatrix} \\
&= \begin{bmatrix} 0.003276 \\ 0.001746 \\ 0.030708 \end{bmatrix}
\end{aligned}
$$

c. Finally, we calculate $p(s_1s_3s_3)$ as the sum of the elements of the last matrix:

$$p(s_1s_3s_3) = 0.03285$$

9.3.3 Computing the probabilities of output sequences: backward

Another feasible way to compute the probability of an output sequence $a_1 \dots a_n$.

(53) a. Let $P(q_i \Rightarrow a_1 \dots a_n)$ be the probability of emitting $a_1 \dots a_n$ beginning from state q_i. And for each possible final state $q_i \in \Omega_X$, let

$$P(q_i \Rightarrow \epsilon) = 1$$

(With this base case, the first use of the recursive step calculates $P(q_j \Rightarrow a_n)$ for each $q_i \in \Omega_X$.)

b. **Recursive step:** Given $P(q_i \Rightarrow a_t \dots a_n)$ for all $q_i \in \Omega_X$, calculate $P(q_j \Rightarrow a_{t-1} \dots a_n)$ for all $q_j \in \Omega_X$ as follows:

$$P(q_j \Rightarrow a_{t-1} \dots a_n) = \big(\sum_{j \in \Omega_X} P(q_i \Rightarrow a_t \dots a_n) P(q_j|q_i)\big) P(a_{t-1}|q_j)$$

c. Finally, given $P(q_i \Rightarrow a_1 \dots a_n)$ for all $q_i \in \Omega_X$,

$$P(a_1 \dots a_n) = \sum_{q_i \in \Omega_X} P_0(q_i) P(q_i \Rightarrow a_1 \dots a_n)$$

(54) **Exercise:** Use the coffee machine as elaborated in (52) and the backward method to compute the probability of the output sequence

$$s_1s_3s_3.$$

9.3.4 Finding the most probable parse with Viterbi's algorithm

(55) Given a string $a_1 \dots a_n$ output by a Markov model, what is the most likely sequence of states that could have yielded this string? This is analogous to finding a most probable parse of a string.

Notice that we could solve this problem by calculating the probabilities of the output sequence for each of the $|\Omega_X|^n$ state sequences, but this is not feasible!

(56) The **Viterbi algorithm** allows efficient calculation of the most probable sequence of states producing a given output (Viterbi 1967; Forney 1973), using an idea that is similar to the forward calculation of output sequence probabilities in §9.3.2 above.

Intuitively, once we know the best way to get to any state in Ω_X at a time t, the best path to the next state is an extension of one of those.

(57) a. Calculate, for each possible initial state $q_i \in \Omega_X$,

$$P(q_i, a_1) = P_0(q_i)P(a_1|q_i).$$

and record: $q_i : P(q_i, a_1)@\epsilon$.

That is, for each state q_i, we record the probability of the state sequence ending in q_i.

b. **Recursive step:** Given $q_i : P(\vec{q}q_i, a_1 \ldots a_t)@\vec{q}$ for each $q_i \in \Omega_X$, for each $q_j \in \Omega_X$ find a q_i that maximizes

$$P(\vec{q}q_iq_j, a_1 \ldots a_ta_{t+1}) = P(\vec{q}q_i, a_1 \ldots a_t)P(q_j|q_i)P(a_{t+1}|q_j)$$

and record: $q_j : P(\vec{q}q_iq_j, a_1 \ldots a_ta_{t+1})@\vec{q}q_i$.[4]

c. After these values have been computed up to the final state t_n, we choose a $q_i : P(\vec{q}q_i, a_1 \ldots a_n)@\vec{q}$ with a maximum probability $P(\vec{q}q_i, a_1 \ldots a_n)$.

9.4 More examples.

(58) **Word associations, priming, etc: naive models.** There are word association thesauri that could be used to dynamically adjust weights for lexical access. See, e.g. the Edinburgh Associative Thesaurus (EAT) at `http://www.eat.rl.ac.uk/` [149, 65, 66]

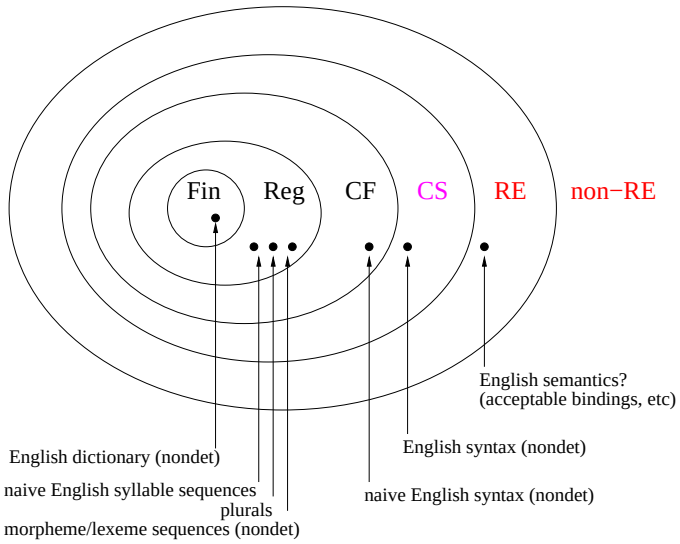
(59) **Regularities in fluent speech.**

(60) **Noise: confusion matrices etc.**

[4] In case more than one q_i ties for the maximum $P(\vec{q}q_iq_j, a_1 \ldots a_ta_{t+1})$, we can either make a choice, or else carry all the winning options forward.

§10 Phrase structure syntax

(0) The Chomsky hierarchy can be defined by the kinds of rules allowed in grammmars.



(1) A finite dictionary can be listed.

a. faster, more compact to use a compressed, subsequential acyclic finite state transducer

b. Loose end: how to handle indeterminacy of outputs?

(2) Morphology can be separated:

a. Simple approaches to syllabification: cyclic deterministic transducer

b. Simple approaches to plurals: cyclic deterministic transducer

c. Can all of morphology be represented this way? No – reduplication impossible, other things awkward

d. Loose end: how to handle reduplication etc?

(3) Segmenting morphs in phone sequences highly nondeterministic

a. human determination of 'intended parse' of fluent speech usually rapid, effortless

b. but this selection of 'intended' parse is influenced by nonlinguistic factors! (priming, etc!)

(4) 3 ways to handle nondeterminism (for loose end 1b, problem 3, and many problems later. . .)

- backtracking
  - \+ very simple
  - \- can take $> 2^n$ or even ∞ steps to process input of length n
- all paths at once by automaton intersection ('chart parsing', 'dynamic progamming')
  - \+ very simple
  - \+ works even when there is ∞ ambiguity, because we do not list each derivation
  - \- takes no more than n^2 steps to process input of length n (too much for fluent speech)
  - \- loose end: how to choose the analysis you want from the result

- most probable parse
  - \+ very simple
  - \+ works even when there is ∞ ambiguity
  - \- Viterbi algorithm: no more than n^2 steps needed for input of length n (too much for fluent speech)
  - \- loose end: what are the approp probabilities? (unsolved for many problems, like speech recognition)
  - \- loose end: can appropriate probabilities be defined by a regular grammar? (ES claims: no!)
- With standard assumptions about English grammar, English is not regular
  (because the Nerode equivalence relation in English has infinitely many blocks)

10.0 CF recognition: top-down

(5) A context free grammar G has 4 parts, $G = \langle \Sigma, N, (\rightarrow), \mathtt{S} \rangle$, where

Σ is a finite nonempty vocabulary

N is a set of categories (disjoint from Σ)

$\rightarrow$ is a rewrite relation that is a subset of $N \times (\Sigma \cup N)^*$. So each pair (or "rule" or "production") in this relation has the form

$$(C, \alpha), \text{ often written } C \rightarrow \alpha,$$

where $C \in N$ is a category and α is a (possibly empty) sequence of vocabulary elements and categories.

S is some category chosen as the "start" symbol.

(6) A production is **right branching** iff it has the form $A \rightarrow \Delta B$ where $\Delta \in \Sigma^*$.
A production is **left branching** iff it has the form $A \rightarrow B\Delta$ where $\Delta \in \Sigma^*$.
A production is **non-branching** iff it has the form $A \rightarrow \Delta$ for some $\Delta \in \Sigma^*$.
A **chain production** has the form $A \rightarrow B$ (and so it is both left and right branching).

(7) Define $\Rightarrow_G$ as follows: for any $x, y, z \in (\Sigma \cup N)^*$ and any $A \in N$

$$xAy \Rightarrow xzy \text{ iff } A \rightarrow z.$$

Let $\Rightarrow_G^*$ be the reflexive, transitive closure of $\Rightarrow$.

(8) A derivation of $s \in \Sigma^*$ from $A \in N$ is a finite sequence of steps:

$$A \Rightarrow \ldots \Rightarrow s.$$

If there is such a derivation, then of course $A \Rightarrow_G^* s$. Such a derivation is leftmost(rightmost) if every step rewrites the leftmost(rightmost) category. We sometimes write $x \Rightarrow_{lm} y$ to indicate a leftmost rewrite step, and similarly $x \Rightarrow_{rm} y$ for rightmost.

(9) The language defined by the grammar $L_G = \{s \in \Sigma^* | \ S \Rightarrow_G^* s\}$.

(10) In general, $yield_G(A) = \{s \in \Sigma^* | \ A \Rightarrow_G^* s\}$, so $L_G = yield_G(S)$.

(11) $s \in yield(G, A)$ is **right (left) branching** iff every derivation of s from any category uses only right (left) branching and non-branching productions.

(12) $S \subseteq yield(G, A)$ is **right (left) branching** iff every $s \in S$ is right (left) branching.

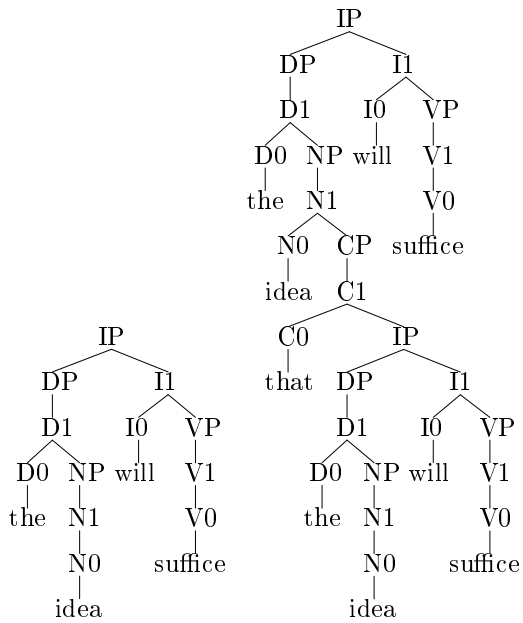
(13) Production $A \rightarrow \gamma$ is **recursive** iff $\gamma \Rightarrow_G^* \alpha A \beta$, for some $\alpha, \beta \in (N \cup \Sigma)^*$.
Production $A \rightarrow \gamma$ is **left recursive** iff $\gamma \Rightarrow_G^* A\beta$, for some $\beta \in (N \cup \Sigma)^*$.
Production $A \rightarrow \gamma$ is **right recursive** iff $\gamma \Rightarrow_G^* \beta A$, for some $\beta \in (N \cup \Sigma)^*$.

(notation) We usually leave off the G subscripts when no confusion can result.

(14) Category $A \in N$ is (left,right)**recursive** iff it the left side of a (left,right) recursive production.

(15) Grammar G is (left,right) **recursive** iff it has a (left,right) recursive production.

IP → DP I1	I1 → I0 VP	I0 → will
DP → D1	D1 → D0 NP	D0 → the
NP → N1	N1 → N0	N0 → idea
	N1 → N0 CP	
VP → V1	V1 → V0	V0 → suffice
CP → C1	C1 → C0 IP	C0 → that



(16) **Top-down, "LL" parsing.** Where IP is the start category, and we want to part *input*, we begin by putting the predicted IP on the stack. The state of the computation can be regarded as the pair $(input, \overline{IP})$, but remember that we can imagine that the input is coming into the machine from a tape, while the "stacks" represent states of the device.

We modify the stack and input with these rules, trying to get to ([],[]),

$$\frac{(input, \overline{C} :: stack)}{(input, \overline{\alpha}@stack)}\,[\text{expandComplete}] \qquad \text{if } C \rightarrow \alpha$$

$$\frac{(w :: input, \overline{w} :: stack)}{(input, stack)}\,[\text{shiftComplete}]$$

(17) A successful computation can be displayed like a proof, where $(input, \overline{IP})$ is the starting axiom, and each intermediate result is derived using the rules given above:

(the idea will suffice, ¯IP)

(18) There can be considerable indeterminacy in the search for such a proof though, since [expandComplete] is non-deterministic. To make sure we explore all available alternatives until we find a solution if there is one, we use a backtrack stack. The state of the computation is given by (remainingInput,stack), so these are what we need in the backtrack stack.

```
(*  file: cf-td1.ml
 creator: E Stabler
    date: Wed Feb 21 10:59:41 PST 2007
 purpose: first top-down "ll" backtracking recognizer
*)

let cfg1 = [
  ("IP",["DP";"I1"]);   ("I1",["I0";"VP"]);   ("I0",["will"]);
  ("DP",["D1"]);       ("D1",["D0";"NP"]);   ("D0",["the"]);
  ("NP",["N1"]);       ("N1",["N0"]);       ("N0",["idea"]);
```

```
                    ("N1",["N0";"CP"]);
  ("VP",["V1"]);        ("V1",["V0"]);        ("V0",["suffice"]);
  ("CP",["C1"]);        ("C1",["C0";"IP"]);   ("C0",["that"]);
];;

let rec predict cats revCats stack = match cats with
    [] -> List.rev_append revCats stack
  | c::cs -> predict cs ((c,false)::revCats) stack;;

let rec expandCompleteAll grammar input cat stack backtrack = match grammar with
    [] -> backtrack
  | (c,rhs)::moreRules -> if c=cat
    then expandCompleteAll moreRules input cat stack ((input,predict rhs [] stack)::backtrack)
    else expandCompleteAll moreRules input cat stack backtrack;;

let rec shiftComplete grammar input cat stack backtrack = match input with
    [] -> backtrack
  | word::words -> if word=cat then ((words,stack)::backtrack) else backtrack;;

let rec ll grammar backtrackStack = match backtrackStack with
    [] -> (false,[])
  | ([],[])::backtrack -> (true,backtrack)
  | (input,(cat,false)::stack)::backtrack ->
      let expanded = expandCompleteAll grammar input cat stack backtrack in
      let scannedAndExpanded = shiftComplete grammar input cat stack expanded in
        begin
          printBacktrackTop backtrackStack;
          ll grammar scannedAndExpanded;
        end
  | _::backtrack -> ll grammar backtrack;;

(* test *)
ll cfg1 [(["the";"idea";"will";"suffice"],[("IP",false)])];;
ll cfg1 [(["the";"idea";"that";"the";"idea";"will";"suffice";"will";"suffice"],[("IP",false)])];;

# ll cfg1 [(["the";"idea";"will";"suffice"],[("IP",false)])];;
([the;idea;will;suffice;],[(IP, false);])
([the;idea;will;suffice;],[(DP, false);(I1, false);])
([the;idea;will;suffice;],[(D1, false);(I1, false);])
([the;idea;will;suffice;],[(D0, false);(NP, false);(I1, false);])
([the;idea;will;suffice;],[(the, false);(NP, false);(I1, false);])
([idea;will;suffice;],[(NP, false);(I1, false);])
([idea;will;suffice;],[(N1, false);(I1, false);])
([idea;will;suffice;],[(N0, false);(CP, false);(I1, false);])
([idea;will;suffice;],[(idea, false);(CP, false);(I1, false);])
([will;suffice;],[(CP, false);(I1, false);])
([will;suffice;],[(C1, false);(I1, false);])
([will;suffice;],[(C0, false);(IP, false);(I1, false);])
([will;suffice;],[(that, false);(IP, false);(I1, false);])
([idea;will;suffice;],[(N0, false);(I1, false);])
([idea;will;suffice;],[(idea, false);(I1, false);])
([will;suffice;],[(I1, false);])
([will;suffice;],[(I0, false);(VP, false);])
([will;suffice;],[(will, false);(VP, false);])
([suffice;],[(VP, false);])
([suffice;],[(V1, false);])
([suffice;],[(V0, false);])
([suffice;],[(suffice, false);])
- : bool * (string list * (string * bool) list) list = (true, [])
```

10.1 CF parsing: top-down

```
(*  file: cf-tdp1.ml
 creator: E Stabler
    date: Wed Feb 21 10:59:41 PST 2007
 purpose: first top-down "ll" backtracking parser, with states (input, output, stack, backtrack),
   where the output is a list of the (input,stack) states in the proof
*)

let cfg1 = [
  ("IP",["DP";"I1"]);   ("I1",["I0";"VP"]);   ("I0",["will"]);
  ("DP",["D1"]);        ("D1",["D0";"NP"]);   ("D0",["the"]);
  ("NP",["N1"]);        ("N1",["N0"]);        ("N0",["idea"]);
                    ("N1",["N0";"CP"]);
  ("VP",["V1"]);        ("V1",["V0"]);        ("V0",["suffice"]);
  ("CP",["C1"]);        ("C1",["C0";"IP"]);   ("C0",["that"]);
];;

let rec predict cats revCats stack = match cats with
    [] -> List.rev_append revCats stack
  | c::cs -> predict cs ((c,false)::revCats) stack;;

let rec expandCompleteAll grammar input output cat stack backtrack = match grammar with
    [] -> backtrack
  | (c,rhs)::moreRules -> if c=cat
    then let newStack = predict rhs [] stack in
      expandCompleteAll moreRules input ((input,newStack)::output) cat stack ((input,(input,newStack)::output,newStack)::backtrack)
    else expandCompleteAll moreRules input output cat stack backtrack;;

let rec shiftComplete grammar input output cat stack backtrack = match input with
    [] -> backtrack
  | word::words -> if word=cat then ((words,(words,stack)::output,stack)::backtrack) else backtrack;;

let rec ll grammar backtrackStack = match backtrackStack with
    [] -> ([],[])
  | ([],output,[])::backtrack -> (List.rev output,backtrack)
  | (input,output,(cat,false)::stack)::backtrack ->
      let expanded = expandCompleteAll grammar input output cat stack backtrack in
      let scannedAndExpanded = shiftComplete grammar input output cat stack expanded in
        begin
          (* printBacktrackTop backtrackStack; *)
          ll grammar scannedAndExpanded;
        end
  | _::backtrack -> ll grammar backtrack;;

(* test *)
# let (output,backtrack) =
    ll cfg1 [(["the";"idea";"will";"suffice"],[(["the";"idea";"will";"suffice"],[("IP",false)])],[("IP",false)])]
    in printOutput 0 output;;
0. ([the;idea;will;suffice;],[(IP, false);])
1. ([the;idea;will;suffice;],[(DP, false);(I1, false);])
2. ([the;idea;will;suffice;],[(D1, false);(I1, false);])
3. ([the;idea;will;suffice;],[(D0, false);(NP, false);(I1, false);])
4. ([the;idea;will;suffice;],[(the, false);(NP, false);(I1, false);])
5. ([idea;will;suffice;],[(NP, false);(I1, false);])
6. ([idea;will;suffice;],[(N1, false);(I1, false);])
7. ([idea;will;suffice;],[(N0, false);(I1, false);])
8. ([idea;will;suffice;],[(idea, false);(I1, false);])
9. ([will;suffice;],[(I1, false);])
10. ([will;suffice;],[(I0, false);(VP, false);])
11. ([will;suffice;],[(will, false);(VP, false);])
12. ([suffice;],[(VP, false);])
13. ([suffice;],[(V1, false);])
14. ([suffice;],[(V0, false);])
15. ([suffice;],[(suffice, false);])
16. ([],[])
- : 'a list = []

# let (output,backtrack) =
    ll cfg1 [(["the";"idea";"that";"the";"idea";"will";"suffice";"will";"suffice"],[],[("IP",false)])]
```

```
    in printOutput 0 output;;
0. ([the;idea;that;the;idea;will;suffice;will;suffice;],[(DP, false);(I1, false);])
1. ([the;idea;that;the;idea;will;suffice;will;suffice;],[(D1, false);(I1, false);])
2. ([the;idea;that;the;idea;will;suffice;will;suffice;],[(D0, false);(NP, false);(I1, false);])
3. ([the;idea;that;the;idea;will;suffice;will;suffice;],[(the, false);(NP, false);(I1, false);])
4. ([idea;that;the;idea;will;suffice;will;suffice;],[(NP, false);(I1, false);])
5. ([idea;that;the;idea;will;suffice;will;suffice;],[(N1, false);(I1, false);])
6. ([idea;that;the;idea;will;suffice;will;suffice;],[(N0, false);(I1, false);])
7. ([idea;that;the;idea;will;suffice;will;suffice;],[(N0, false);(CP, false);(I1, false);])
8. ([idea;that;the;idea;will;suffice;will;suffice;],[(idea, false);(CP, false);(I1, false);])
9. ([that;the;idea;will;suffice;will;suffice;],[(CP, false);(I1, false);])
10. ([that;the;idea;will;suffice;will;suffice;],[(C1, false);(I1, false);])
11. ([that;the;idea;will;suffice;will;suffice;],[(C0, false);(IP, false);(I1, false);])
12. ([that;the;idea;will;suffice;will;suffice;],[(that, false);(IP, false);(I1, false);])
13. ([the;idea;will;suffice;will;suffice;],[(IP, false);(I1, false);])
14. ([the;idea;will;suffice;will;suffice;],[(DP, false);(I1, false);(I1, false);])
15. ([the;idea;will;suffice;will;suffice;],[(D1, false);(I1, false);(I1, false);])
16. ([the;idea;will;suffice;will;suffice;],[(D0, false);(NP, false);(I1, false);(I1, false);])
17. ([the;idea;will;suffice;will;suffice;],[(the, false);(NP, false);(I1, false);(I1, false);])
18. ([idea;will;suffice;will;suffice;],[(NP, false);(I1, false);(I1, false);])
19. ([idea;will;suffice;will;suffice;],[(N1, false);(I1, false);(I1, false);])
20. ([idea;will;suffice;will;suffice;],[(N0, false);(I1, false);(I1, false);])
21. ([idea;will;suffice;will;suffice;],[(idea, false);(I1, false);(I1, false);])
22. ([will;suffice;will;suffice;],[(I1, false);(I1, false);])
23. ([will;suffice;will;suffice;],[(I0, false);(VP, false);(I1, false);])
24. ([will;suffice;will;suffice;],[(will, false);(VP, false);(I1, false);])
25. ([suffice;will;suffice;],[(VP, false);(I1, false);])
26. ([suffice;will;suffice;],[(V1, false);(I1, false);])
27. ([suffice;will;suffice;],[(V0, false);(I1, false);])
28. ([suffice;will;suffice;],[(suffice, false);(I1, false);])
29. ([will;suffice;],[(I1, false);])
30. ([will;suffice;],[(I0, false);(VP, false);])
31. ([will;suffice;],[(will, false);(VP, false);])
32. ([suffice;],[(VP, false);])
33. ([suffice;],[(V1, false);])
34. ([suffice;],[(V0, false);])
35. ([suffice;],[(suffice, false);])
36. ([],[])
- : 'a list = []
```

10.1.1 Some basic properties of the top-down recognizer

(19) Throughout our study, we will keep an eye on these basic properties of syntactic analysis algorithms which are mentioned in these facts:

1. A recognition method is **sound** when: if s is recognized as having category A then $s \in yield_G(A)$.
   It is **complete** when: if $s \in yield_G(A)$ then s is recognized as having category A.
   It avoids **spurious ambiguity** when there are n successful recognition computations of s just in case there are n corresponding derivations of s from the grammar.
2. For recognition methods that involve searching through derivable configurations, the set of all proofs from any input and category is the **search space** of that problem.
   A recognition method has a **finitely bounded search space** iff the search space for every grammar, input and category is finite. Otherwise, the search space of the grammar is said to be infinite.

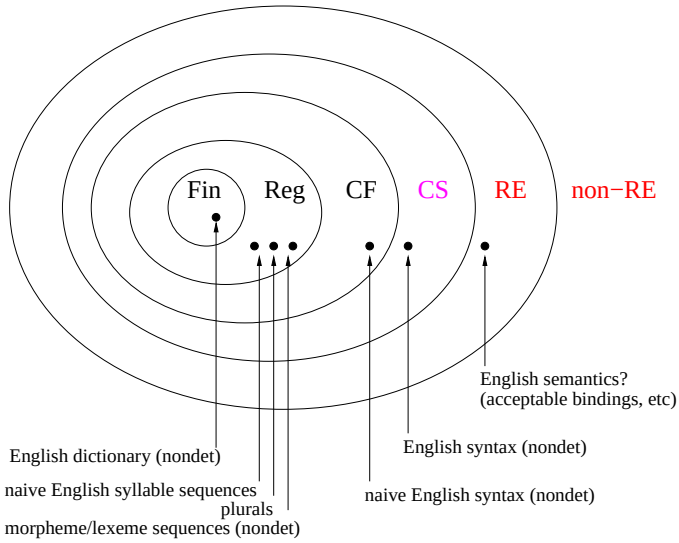
(20) **LL is sound, complete, and has no spurious ambiguity.**

(21) **With LL, every right branching $RS \subseteq yield(G, A)$ has finitely bounded search space.**

(22) **With LL, grammars with left recursion can have infinite search spaces.** For example, every infinite left branching language has an infinite search space.

§11 Context free parsing: 1 path at a time

(0) The Chomsky hierarchy can be defined by the kinds of rules allowed in grammmars.



(1) A finite dictionary can be listed, but compressed transducer is faster, more compact

(2) Morphology can be separated, and re-introduced by compositions
(We haven't yet seen any way to handle morphological operations like reduplication)

(3) Segmenting morphs in phone sequences highly <u>nondeterministic</u>

- backtracking
- all paths at once ('chart parsing', 'dynamic progamming')
- most probable parse

(4) With standard assumptions about English grammar, English is not regular
(because the Nerode equivalence relation in English has infinitely many blocks)

(5) Context free phrase structure grammars can be parsed

- with a top-down backtracking parser (sometimes)
- with a bottom-up backtracking parser (sometimes)
- with a left-corner backtracking parser (sometimes)

11.0 CFGs

(6) A context free grammar G has 4 parts, $G = \langle \Sigma, N, (\rightarrow), \mathtt{S} \rangle$, where

Σ is a finite nonempty vocabulary

N is a set of categories (disjoint from Σ)

$\rightarrow$ is a rewrite relation that is a subset of $N \times (\Sigma \cup N)^*$. So each pair (or "rule" or "production") in this relation has the form

$$(C, \alpha), \text{ often written } C \rightarrow \alpha,$$

where $C \in N$ is a category and α is a (possibly empty) sequence of vocabulary elements and categories.

S is some category chosen as the "start" symbol.

(7) A production is **right branching** iff it has the form $A \rightarrow \Delta B$ where $\Delta \in \Sigma^*$.
A production is **left branching** iff it has the form $A \rightarrow B\Delta$ where $\Delta \in \Sigma^*$.
A production is **non-branching** iff it has the form $A \rightarrow \Delta$ for some $\Delta \in \Sigma^*$.
A **chain production** has the form $A \rightarrow B$ (and so it is both left and right branching).

(8) Define $\Rightarrow_G$ as follows: for any $x, y, z \in (\Sigma \cup N)^*$ and any $A \in N$

$$xAy \Rightarrow xzy \text{ iff } A \rightarrow z.$$

Let $\Rightarrow_G^*$ be the reflexive, transitive closure of $\Rightarrow$.

(9) A derivation of $s \in \Sigma^*$ from $A \in N$ is a finite sequence of steps:

$$A \Rightarrow \ldots \Rightarrow s.$$

If there is such a derivation, then of course $A \Rightarrow_G^* s$. Such a derivation is leftmost(rightmost) if every step rewrites the leftmost(rightmost) category. We sometimes write $x \Rightarrow_{lm} y$ to indicate a leftmost rewrite step, and similarly $x \Rightarrow_{rm} y$ for rightmost.

(10) The language defined by the grammar $L_G = \{s \in \Sigma^* | \ S \Rightarrow_G^* s\}$.

(11) In general, $yield_G(A) = \{s \in \Sigma^* | \ A \Rightarrow_G^* s\}$, so $L_G = yield_G(S)$.

(12) $s \in yield(G, A)$ is **right (left) branching** iff every derivation of s from any category uses only right (left) branching and non-branching productions.

(13) $S \subseteq yield(G, A)$ is **right (left) branching** iff every $s \in S$ is right (left) branching.

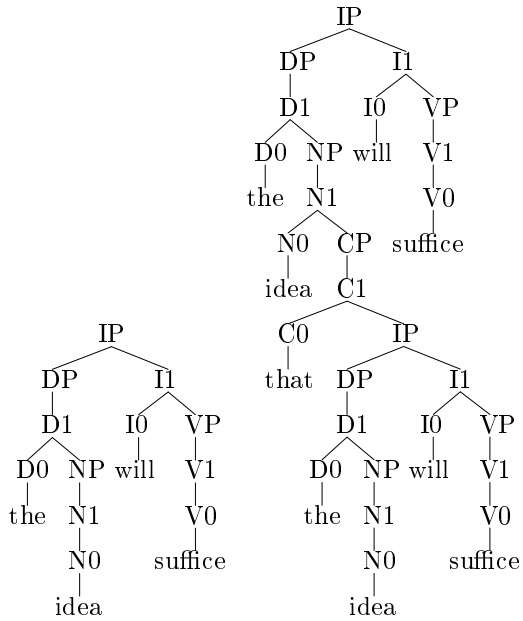
(14) Production $A \rightarrow \gamma$ is **recursive** iff $\gamma \Rightarrow_G^* \alpha A \beta$, for some $\alpha, \beta \in (N \cup \Sigma)^*$.
Production $A \rightarrow \gamma$ is **left recursive** iff $\gamma \Rightarrow_G^* A\beta$, for some $\beta \in (N \cup \Sigma)^*$.
Production $A \rightarrow \gamma$ is **right recursive** iff $\gamma \Rightarrow_G^* \beta A$, for some $\beta \in (N \cup \Sigma)^*$.

(notation) We usually leave off the G subscripts when no confusion can result.

(15) Category $A \in N$ is (left,right)**recursive** iff it the left side of a (left,right) recursive production.

(16) Grammar G is (left,right) **recursive** iff it has a (left,right) recursive production.

IP $\rightarrow$ DP I1	I1 $\rightarrow$ I0 VP	I0 $\rightarrow$ will
DP $\rightarrow$ D1	D1 $\rightarrow$ D0 NP	D0 $\rightarrow$ the
NP $\rightarrow$ N1	N1 $\rightarrow$ N0	N0 $\rightarrow$ idea
	N1 $\rightarrow$ N0 CP	
VP $\rightarrow$ V1	V1 $\rightarrow$ V0	V0 $\rightarrow$ suffice
CP $\rightarrow$ C1	C1 $\rightarrow$ C0 IP	C0 $\rightarrow$ that


IP
DP
I1
D1
I0
VP
D0
NP
will
V1
the
N1
V0
N0
CP
suffice
idea
C1
C0
that
IP
DP
I1
D1
I0
VP
D0
NP
will
V1
the
N1
V0
N0
suffice
idea
IP
DP
I1
D1
I0
VP
D0
NP
will
V1
the
N1
V0
N0
suffice
idea

11.1 Top-down recognition

This recognition method is sometimes called 'recursive descent', or 'LL' since it goes through the input string left-to-right constructing a leftmost derivation.

We can give the top-down recognition method as a pair of functions which, given certain (input,stack) pairs and corresponding rules C$\rightarrow$ $A1...An$, will define a new (input,stack) pair. To recognize an input, we begin with the pair (input,$\overline{S}$) where S is the start category, and use these functions to get to the pair (ϵ, ϵ). From any given (input,stack) pair, if shiftComplete applies, then expandComplete cannot apply, and vice versa. But when using expandComplete, there may often be choices in which grammar rule C$\rightarrow$ $A_1...A_n$ to use. The standard top-down backtracking method handles these choices by putting them on a backtrack stack, and pursuing just one of the choices. We use x for symbols that can be either categories in N or vocabulary items in Σ.

for each $C \rightarrow x_1...x_n$, a rule [expandComplete] $\quad \dfrac{(\text{input}, \overline{C}\ \text{stack})}{(\text{input}, \overline{x_1}...\overline{x_n}\ \text{stack})}$

[shiftComplete] $\quad \dfrac{(w\ \text{input}, \overline{w}\ \text{stack})}{(\text{input, stack})}$

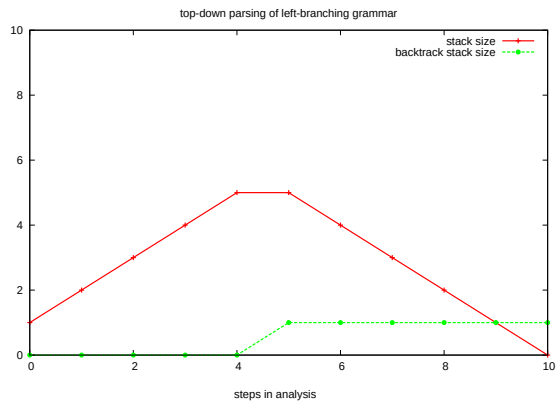
It is easy to establish these important properties of this recognition method:

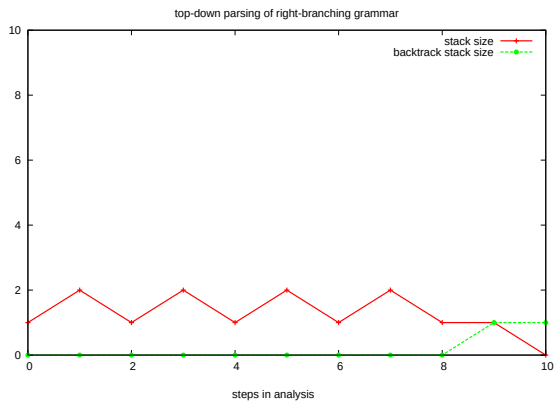
(17) Left recursion can lead to non-termination – so top-down recognition does not deserve to be called an 'algorithm' unless it is restricted to grammars with no left recursion.

(18) Even when it terminates, in the presence of local (and global) ambiguities it can be intractable [2, Thm4.3].

(19) A grammar with purely left branching constructions can require unbounded stack depths. Consider for example recognizing the string *abcde* using the following simple grammars:

A $\rightarrow$ B a	A $\rightarrow$ a B
B $\rightarrow$ C b	B $\rightarrow$ b C
D $\rightarrow$ D c	D $\rightarrow$ c D
D $\rightarrow$ E d	D $\rightarrow$ d E
E $\rightarrow$ F e	E $\rightarrow$ e F
E $\rightarrow$ e	E $\rightarrow$ e





(20) A 'more intelligent' exploration of the choices can be guided by an 'oracle', as discussed in §11.5 below. A language is said to be LL(k) iff, with an oracle that uses k symbols of 'lookahead', LL is deterministic. As we will see, English is not LL(k) for any k.

11.2 Bottom-up recognition

This recognition method is sometimes called 'recursive ascent', or 'shift-reduce backtracking', or 'LR' since it goes through the input string left-to-right constructing a rightmost derivation in reverse.

$$\text{for each } C \rightarrow x_1 ... x_n, \text{ [reduce]} \qquad \frac{(\text{input}, x_n x_{n-1} \ldots x_1 \text{ stack})}{(\text{input}, C\text{stack})}$$

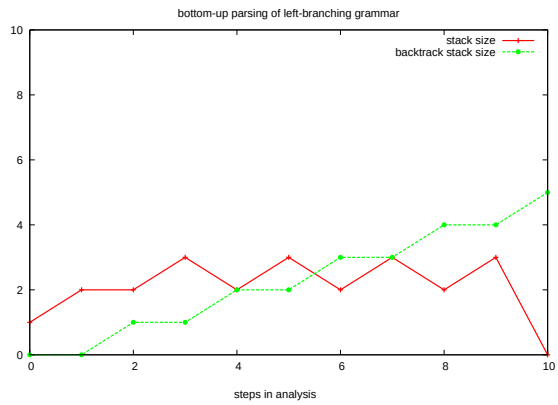
$$\text{for each } C \rightarrow x_1 ... x_n, \text{ [reduceComplete]} \qquad \frac{(\text{input}, x_n x_{n-1} \ldots x_1 \overline{C} \text{ stack})}{(\text{input}, \text{stack})}$$

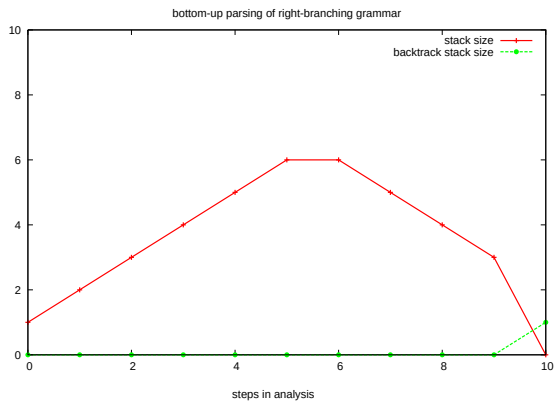
$$\text{[shift]} \qquad \frac{(w \text{ input}, \text{stack})}{(\text{input}, w \text{ stack})}$$

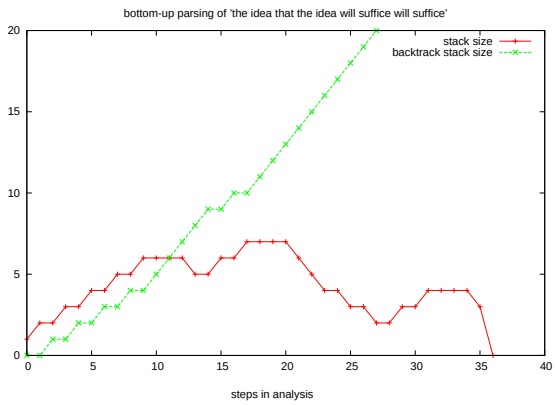
Consider for example the statistics recognizing "edcba" with a left-branching grammar versus recognizing "abcde" with a right branching grammar:

(21) Empty categories in the grammar can produce infinite search spaces, leading to nontermination on some problems – so this recognition method does not deserve to be an 'algorithm' unless it is restricted.

(22) Unbounded memory requirements on simple right branching. Consider for example, the simple grammars for *abcde* from the previous section:







(23) Attachment preferences like 'right association' and 'minimal attachment' are naturally realized in a bu parser [199, 90, 148]

(24) A 'more intelligent' exploration of the choices can be guided by an 'oracle', as discussed in §11.5 below. A language is said to be LR(k) iff, with an oracle using k symbols of 'lookahead', LR is deterministic. English is not LR(k) for any k. [168, 18, 191]

11.3 Left corner (LC) recognition

for each C→ $x_1 ... x_n$, [lc], $\dfrac{(\text{input}, x_1\text{stack})}{(\text{input}, \overline{x_2} \ldots \overline{x_n} C\text{stack})}$

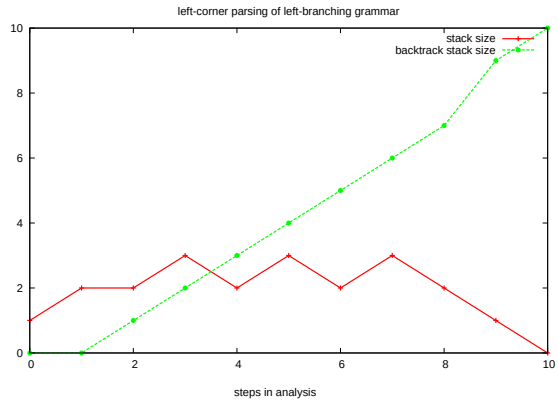
for each C→ $x_1 ... x_n$, [lcComplete] $\dfrac{(\text{input}, x_1 \overline{C}\text{stack})}{(\text{input}, \overline{x_2} \ldots \overline{x_n}\text{stack})}$

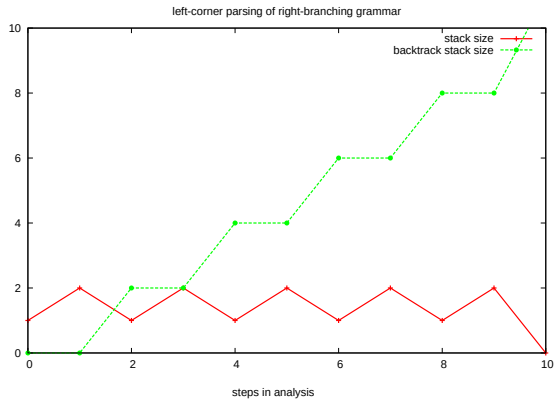
[shift] $\dfrac{(w \text{ input}, \text{stack})}{(\text{input}, w \text{ stack})}$

[shiftComplete] $\dfrac{(w \text{ input}, \overline{w} \text{ stack})}{(\text{input}, \text{stack})}$

(25) Empty categories in the grammar can produce infinite search spaces, since their 'left corners' will also be empty, leading to nontermination on some problems – so this recognition method does not deserve to be an 'algorithm' unless it is restricted.

(26) But unlike LL and LR, this method has bounded memory requirements on simple right and left branching. It has unbounded memory requirements for recursive center embedding (of course).





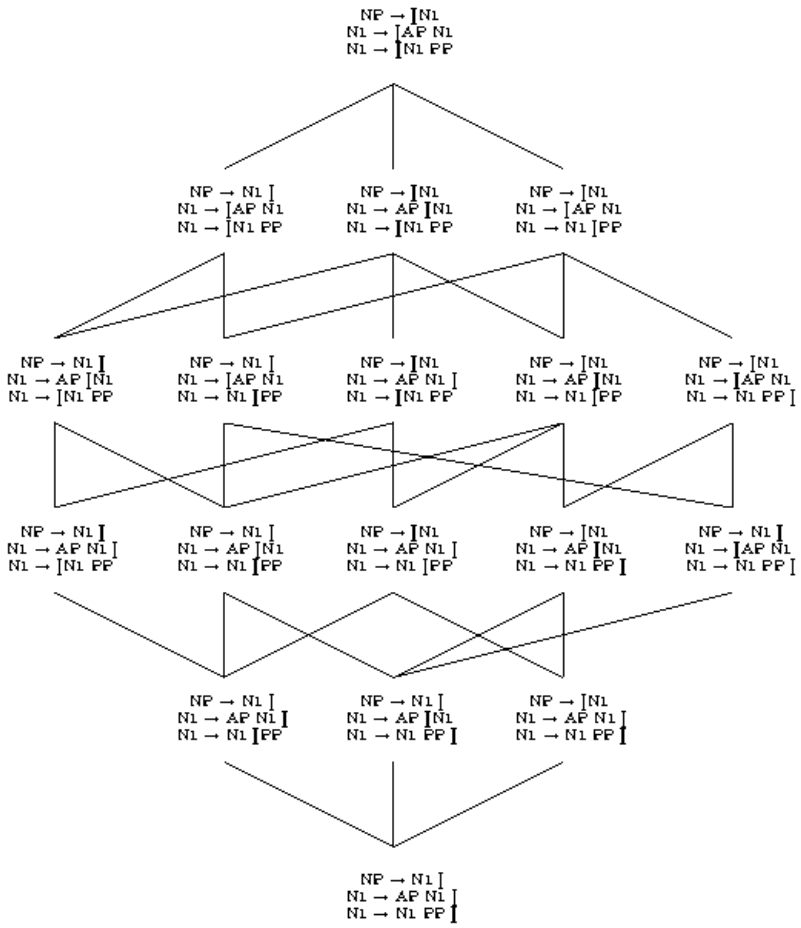
(27) Psychologists like this method because it is almost LL (top-down), but avoids the problem with left recursion [100, 101]

(28) When used with a treebank grammar, the LC stack never gets very deep [167]

(29) Probabilistic versions of this method sometimes used for NLP [183].
the left corner rules provide more information for better probabilistic recognition [129],
though now head-labeled 'bilexical' grammars are more common [58, 21, 78]

11.4 Generalized left corner recognition

Adapting ideas from [31, 68, 188], we can define different 'generalized left corner' (GLC) strategies depending on how much of the right side of a rule needs to be found bottom-up before the rule is used, by predicting the rest of the right side. We mark the 'trigger point', the division between what must be found bottom-up and what must be predicted, with the symbol][. The whole collection of recognition strategies, with LL on top and LR on the bottom, forms a lattice:



$$\text{for each } C \to x_1 ... x_i][x_{i+1} ... x_n, \text{ [glc]}, \qquad \frac{(\text{input}, x_i \ldots x_1 \text{ stack})}{(\text{input}, \overline{x_{i+1}} \ldots \overline{x_n} C \text{ stack})}$$

$$\text{for each } C \to x_1 ... x_i][x_{i+1} ... x_n, \text{ [glcComplete]} \qquad \frac{(\text{input}, x_i \ldots x_1 \overline{C} \text{ stack})}{(\text{input}, \overline{x_{i+1}} \ldots \overline{x_n} \text{ stack})}$$

$$\text{[shift]} \qquad \frac{(w \text{ input}, \text{stack})}{(\text{input}, w \text{ stack})}$$

$$\text{[shiftComplete]} \qquad \frac{(w \text{ input}, \overline{w} \text{ stack})}{(\text{input}, \text{stack})}$$

(30) All these methods are sound and complete, finding an n step proof that a string is in the language for each n-node derivation tree that the string has.

(31) As we have already seen by comparing LL, LR, LC, these strategies have different memory requirements and different degrees of indeterminacy. Mixed strategies other than LC are rarely explored – cf. [221]

(32) A dilemma:

- 'Incremental' comprehension suggests that human syntactic recognition is quite top-down [212, 235]
- Easy recognition of relative clause in examples like this suggests that structure building cannot be top-down:

The daughter of the colonel who was standing on the balcony waved to us.

(33) Search problems make these methods mainly unusable with linguists' grammars.

11.5 Oracles for the GLC parsers

There are two kinds of information an oracle can use. The oracle can use the grammar to make sure that the current stack of the parser is 'consistent' in the sense that it could possibly be reduced to ϵ, and the oracle can 'look ahead' a finite amount into the input to see what k words come next.

11.5.1 Stack consistency

(34) 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.

(35) 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.

- In top-down recognition, the triggers are always empty.
- In left-corner recognition, the possible trigger sequences are always exactly one completed category.
- In bottom-up and some other methods, trigger sequences can sometimes be arbitrarily long, but in some cases they are finitely bounded and can easily be calculated in advance.

A test that can check these precalculated possibilities is called an **oracle**.

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

$$\text{for each } C \rightarrow x_1 \ldots x_i \,[\![\, x_{i+1} \ldots x_n, \text{ [axiom]} \qquad \frac{}{(x_1 \ldots x_i, C)}$$

$$\text{if } x_i \in \Sigma \text{ [unscan]} \qquad \frac{(x_1 \ldots x_i, C)}{(x_1 \ldots x_{i-1}, C)}$$

$$\text{if } x_i \rightarrow y_1 \ldots y_j \,[\![y_{j+1} \ldots y_n, \text{ [unreduce]} \qquad \frac{(x_1 \ldots x_i, C)}{(x_1 \ldots x_{i-1} y_1 \ldots y_j, C)}$$

(37) **Example:** Consider the following grammar g5-mix with the triggers indicated:

IP → DP I1][	I1 →][I0 VP	I0 →][will.
DP → D1][	D1 → D0][NP	D0 → the][
NP → N1][	N1 → N0][	N0 → idea][
	N1 → N0][PP	
VP → V1][	V1 → V0][	V0 → suffice][
PP → P1][	P1 → P0][DP	P0 → that][

For this grammar, the following proof shows that the beginnings of `IP` include DP I1, DP, D1, D0, `the`, and ϵ:

$$\frac{\frac{\frac{\frac{\frac{(\text{DP I1}, \text{IP})}{(\text{DP}, \text{IP})}\text{[unreduce]}}{(\text{D1}, \text{IP})}\text{[unreduce]}}{(\text{D0}, \text{IP})}\text{[unreduce]}}{(\mathtt{the}, \text{IP})}\text{[unreduce]}}{(\epsilon, \text{IP})}\text{[unshift]}$$

Notice that the beginnings of IP do <u>not</u> include `the idea`, `the` I1, D0 I0, I0, or I1.

(38) 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.

Let's say that a sequence C is **reducible** iff the sequence C is the concatenation of two sequences $C = C_1C_2$ where

a. C_1 is a sequence $A_i, \ldots, A_1$ of $0 \leq i$ completed elements

b. C_2 begins with a predicted element $\overline{C}$, and

c. $A_1, \ldots, A_i$ is a beginning of C

(39) GLC with an oracle:

$$\frac{(input,\ x_i \ldots x_1\alpha)}{(input,\ \overline{x_{i+1}} \ldots \overline{x_n}C\alpha)} \quad \text{if } C \rightarrow x_1, \ldots, x_i \overset{\text{[glc]}}{][} x_{i+1}, \ldots x_n \quad \text{and } C\alpha \text{ is reducible}$$

$$\frac{(input,\ x_i \ldots x_1\overline{C}\alpha)}{(input,\ \overline{x_{i+1}} \ldots \overline{x_n}\alpha)}\ \text{[glc-complete]} \quad \text{if } C \rightarrow x_1, \ldots, x_i][x_{i+1}, \ldots x_n$$

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

$$\frac{(\mathtt{w}\ input, \overline{\mathtt{w}}\ \alpha)}{(input,\ \alpha)}\ \text{[shift-complete]}$$

(40) **Example.** To implement GLC recognition with this oracle, we precalculate the beginnings of every category. In effect, we want to find every theorem of the logic given in (36) above, for every category C in the grammar, if possible, so that it can be quickly checked by the glc rules.

Consider again `g5-mix.pl` given in (37) above. There are infinitely many derivations of this trivial result, as we see in this infinite sequence of steps:

$$\cfrac{\cfrac{\cfrac{(\text{N1}, \text{N1})}{(\text{N1}, \text{N1})}\ \text{[unreduce]}}{(\text{N1}, \text{N1})}\ \text{[unreduce]}}{\cdots}\ \text{[unreduce]}$$

Nevertheless, the set of theorems, the set of pairs in the "is a beginning of" relation for the grammar G5-mix above, with the trigger relations indicated there, is finite. So we can compute the whole set by taking the closure of the axioms under the inference relation, using the same kind of strategy that we used in the "all paths at once" recognition methods. See exercise 4 on page 87 below.

The calculation of the oracle will of course fail to terminate when the set of beginnings of any category is infinite. To ensure termination, a standard strategy is to restrict the length of the trigger sequences. With such a restriction, the oracle may fail to block some inconsistent stack sequences.

11.5.2 Lookahead

(41) In top-down recognition, there is no point in using an expansion $p \rightarrow q, r$ if the next symbol to be parsed could not possibly be the beginning of a q.

To guide top-down steps, it would be useful to know what symbol (or what k symbols) are next, waiting to be parsed. 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, structure building waits for a word (or in general, k words) to be heard.

Again, we will 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.

(42) In calculating lookahead, we ignore the triggers.

One kind of situation that we must allow for is this. If $p \to q_1, \ldots, q_n$ and $q_1, \ldots, q_i \Rightarrow^* \epsilon$, then every next symbol for q_{i+1} is a next symbol for p.

(43) For any $S \in \Sigma^*$, let $\text{first}_k(S)$ be the first k symbols of S if $|S| \geq k$, and otherwise $\text{first}_k(S) = S$.

We can use the following reasoning to calculate all of the next k words that can be waiting to be parsed as each category symbol is expanded. For some $k > 0$:

$wLAw$ [axiom] if $w \in \Sigma$

$xLA\ C$ [axiom] if $C \to x_1, \ldots, x_n$ and either $x = x_1, \ldots, x_k \in \Sigma^k$ for $k \leq n$, or $x = x_1,$

$$\frac{y_1LAx_1 \quad \ldots \quad y_nLAx_n}{xLA}\ [\text{la}] \qquad \text{if } C \to x_1, \ldots, x_n, \text{ and } x \in \text{first}_k(y_1 \ldots y_n)$$

And we let $first_k(y_1 \ldots y_n)LA(x_1, \ldots, x_n)$ if y_iLAx_i for $1 \leq i \leq n$.

(44) GLC with stack consistency *and* lookahead oracles:

$$\frac{(input,\ x_i \ldots x_1\alpha)}{(input,\ \overline{x_{i+1}} \ldots \overline{x_n}C\alpha)} \qquad \text{if } C \to x_1, \ldots, x_i][x_{i+1}, \ldots x_n \text{ and } C\alpha \text{ is reducible, and } \text{first}_k(S)LA(x_{i+1}, \ldots, x$$

[glc]

$$\frac{(input,\ x_i \ldots x_1\overline{C}\alpha)}{(input,\ \overline{x_{i+1}} \ldots \overline{x_n}\alpha)} \qquad \text{if } C \to x_1, \ldots, x_i][x_{i+1}, \ldots x_n \text{ and } \text{first}_k(S)LA(x_{i+1}, \ldots, x_n)$$

[glc-complete]

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

$$\frac{(\texttt{w}\ input, \overline{\texttt{w}}\ \alpha)}{(input,\ \alpha)}\ [\text{shift-complete}]$$

(45) Adding bottom-up k-symbol lookahead to the `glco` recognizers with a bottom-up oracle, we have `glcola(`k`)` recognizers.

For any of the GLC parsers, a language is said to be `glcola(`k`)` iff there is at most one step that can be taken at every point in every proof.

Obviously, when a language has genuine structural ambiguity – more than one successful parse for some strings in the language – the language cannot be `glcola(`k`)` for any k (e.g. LL(k), LC(k), LR(k),...) In the case of an ambiguous language, though, we can consider whether the recognition of unambiguous strings is deterministic, or whether the indeterminism that is encountered is all due to global ambiguities.

11.6 Exercises

1. Consider the example grammar from class and the web, with the trigger points given by the][symbol:

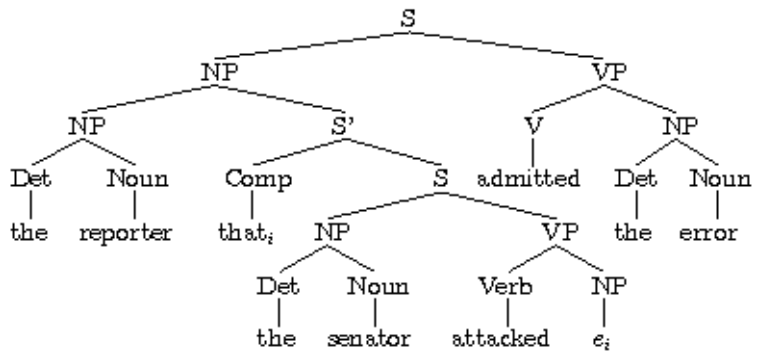
IP → DP][I1	I1 →][I0 VP	I0 → will][
DP →][D1	D1 → D0][NP	D0 → the][
NP →][N1	N1 → N0][	N0 →][idea
	N1 → N0 CP][	
VP →][V1	V1 → V0][	V0 →][suffice
CP →][C1	C1 → C0][IP	C0 → that][

Write out the complete, 16-step GLC parse for *the idea will suffice* using these trigger positions. (You can type this up by hand or write code to calculate and display it.)

2. Suppose you start at the top of the GLC lattice of parsers for this grammar, and take one of the shortest paths from the top to the trigger positions shown above. How long is this path – that is, how many steps from the top are needed to get to the positions given above?
3. Gibson [101] proposes

   For initial purposes, a syntactic theory with a minimal number of syntactic categories, such as Head-driven Phrase Structure Grammar [207] or Lexical Functional Grammar [26], will be assumed. [Note: The SPLT is also compatible with grammars assuming a range of functional categories such as Infl, Agr, Tense, etc. (e.g. Chomsky 1995) under the assumption that memory cost indexes predicted *chains* rather than predicted categories, where a chain is a set of categories coindexed through movement (Chomsky 1981).] Under these theories, the minimal number of syntactic head categories in a sentence is two: a head noun for the subject and a head verb for the predicate. If words are encountered that necessitate other syntactic heads to form a grammatical sentence, then these categories are also predicted, and an additional memory load is incurred. For example, at the point of processing the second occurrence of the word "the" in the object-extracted RC example,

   1a. The reporter who the senator attacked admitted his error,



   there are four obligatory syntactic predictions: 1) a verb for the matrix clause, 2) a verb for the embedded clause, 3) a subject noun for the embedded clause, and an empty category NP for the wh-pronoun "who."

   Is the proposed model a glc recognizer? If not, is the proposal a cogent one, one that conforms to the behavior of a parsing model that could possibly work?
4. If the beginnings of each category are finite, it is not difficult to calculate the 'stack consistency' results needed for the oracle in GLC recognition. Here is one standard way to do it, based on the strategy we have used in 'all-paths' recognition. Recall the rules given earlier, in (36), for each category C:

$$\text{for each } C \rightarrow x_1 \ldots x_i \,[\![\, x_{i+1} \ldots x_n, \text{ [axiom]} \qquad \frac{}{(x_1 \ldots x_i, C)}$$

$$\text{if } x_i \in \Sigma \text{ [unscan]} \qquad \frac{(x_1 \ldots x_i, C)}{(x_1 \ldots x_{i-1}, C)}$$

$$\text{if } x_i \rightarrow y_1 \ldots y_j \,[\![\, y_{j+1} \ldots y_n, \text{ [unreduce]} \qquad \frac{(x_1 \ldots x_i, C)}{(x_1 \ldots x_{i-1} y_1 \ldots y_j, C)}$$

   So define an OCaml function that takes a category C and two lists (or arrays or hash tables) $(agenda, chart)$, and initially puts all instances of [axiom] into both the agenda and the chart. Then define a recursive function that pops the top element off the agenda, computes the result of applying unscan and unreduce to it in all possible ways, using ensureMember to put all results into the chart, and whenever the result is new, also putting the result into the agenda.

   Apply this function with the grammar g5mix given in (37) above, to calculate the beginnings of every category C in that grammar.
5. Read the description of non-canonical LR(k,t) recognizers in [2, §6.2] or in [255] and modify the Ocaml bu parser to implement it. (These ideas were the basis of the strategies in the Marcus parser [168].)

6. A first response to the fact that human speakers seem to understand fluent speech on a word-by-word basis, without the need to wait for sentence or phrase endings, has been to assume that humans are using a one-path-at-a-time model, perhaps one of the sorts surveyed above.

   Since humans seem to have no trouble with left recursion, it is unlikely that we use a simple top-down model like the one described above. And since all these models face problems with local ambiguities, there has been great interest in how humans tend to resolve local ambiguities, and how they recover when they have resolved them incorrectly.

   One idea is that people prefer to attach new elements low, in the current phrase rather than in a higher position. There is evidence that people initially misanalyze *in the library* in sentences like,

   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.

   (These examples are from Frazier's classic survey article [89], and compare the recent Pickering and van Gompel survey [204].) Frazier calls this a "late closure" preference: listeners prefer to "close" each phrase as late as possible, so if they can attach a consituent low, they will.

   How could you adapt the bottom-up parser above to exhibit this preference? (Hint: Pereira offers an answer here: [199]). Can you implement this by modifying our Ocaml code `cf-bu1.ml`? How would you modify the top-down parser to exhibit the same preference?

7. 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 [7] 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.

   (Altmann and others [6, 8, 132] later showed certain limitations in the contexts in which an effect of referential support can be seen. It is interesting to consider whether these proposals make sense in a computational model – good squib topic.)

8. The memory used by the models above is two stacks: the stack of syntactic elements and the backtrack stack of unexplored alternatives. Human memory is not like a stack: for example in people there can be interference effects among the things remembered in a way that does not happen in a stack. And human memory also seems to be structured in various ways.

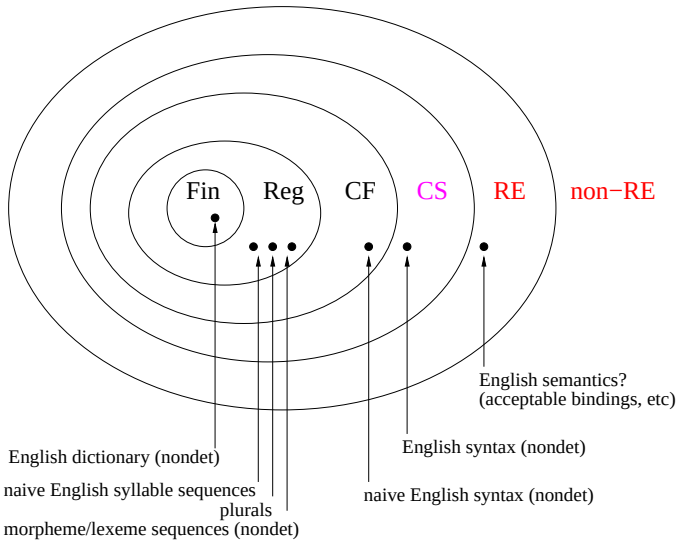
   Read these surveys of recent proposals [82], [163].

   Explain: How would you design a model that (i) can actually succeed in recognizing a reasonable range of sentences (of English or some other human language) and yet (ii) uses a memory that is more human-like?

   Or implement: a small version of one of the ideas in the surveys.

§12 Context free parsing: all paths at once

(0) The Chomsky hierarchy can be defined by the kinds of rules allowed in grammmars.



(1) A finite dictionary can be listed, but compressed transducer is faster, more compact

(2) Morphology can be separated, and re-introduced by compositions
(We haven't yet seen any way to handle morphological operations like reduplication)

(3) Segmenting morphs in phone sequences highly <u>nondeterministic</u>

- backtracking (possible nontermination when ambiguity infinite)
- all paths at once ('chart parsing', 'dynamic progamming')
- most probable parse

(4) With standard assumptions about English grammar, English is not regular
(because the Nerode equivalence relation in English has infinitely many blocks)

(5) CFGs for human languages are ambiguous (nondeterministic). They can be parsed by

- backtracking
  - with a top-down backtracking parser (possible nontermination when ambiguity infinite or when left recursive)
    oracle cannot resolve even local ambiguities with k symbols of lookahead, for any k
  - with a bottom-up backtracking parser (possible nontermination when ambiguity infinite or when categories can be empty)
    oracle cannot resolve even local ambiguities with k symbols of lookahead, for any k
  - with a left-corner backtracking parser (possible nontermination when ambiguity infinite or when left corners can be empty)
    oracle cannot resolve even local ambiguities with k symbols of lookahead, for any k
  - none of the GLC parsers seem to work – have we framed the problem incorrectly? (yes says Marcus, Berwick&Weinberg, ES,..., psychologists)

___ (we have gotten this far)

- all paths at once – 'works' (i.e. terminates, efficiently) even if we have framed the problem incorrectly
- most probable parse – 'works' (i.e. terminates, efficiently) even if we have framed the problem incorrectly

(6) If English includes "X or no X", for X any NP [27, 236, 209], or if English has any remnant movement [117, 140, 141] CFGs cannot provide appropriate descriptions (though they could 'approximate' in various senses)

(7) Beyond context-free:
- a simple, mildly context sensitive, formal minimalist grammar (MG)
- MGs with head movement, affix hopping, adjunction
- Beyond mildly context sensitive: PMCFGs

12.0 Ambiguity again, more carefully

- PP attachment in $[_{VP}$ V D N PP1 PP2 ...]

  Consider the grammar:

$$\text{VP} \rightarrow \text{V NP PP}^*$$
$$\text{NP} \rightarrow \text{D N PP}^*$$

  A grammar like this cannot be directly expressed in standard context free form. It defines a context free language, but it is equivalent to the following infinite grammar:

NP → D N	VP → V NP
NP → D N PP	VP → V NP PP
NP → D N PP PP	VP → V NP PP PP
NP → D N PP PP PP	VP → V NP PP PP PP
NP → D N PP PP PP PP	VP → V NP PP PP PP PP
NP → D N PP PP PP PP PP	VP → V NP PP PP PP PP PP
...	...

  The number of structures defined by this grammar is an exponential function of the number of words.

N =	1	2	3	4	5	6	7	8	9	10
#trees =	2	5	14	132	469	1430	4862	16796	1053686	...

- N compounds $[_N$ N N]

  N → N N

  This series same as for PPs, except PP^1 is like a N^3 compound:

n	1	2	3	4	5	6	7	8	9	10
#trees	1	1	2	5	14	42	132	429	1420	4862

  We see the Catalan sequence here, which as we mentioned before is the number of binary trees with n leaves. At larger values the n'th Catalan number approximates the exponential function 4**n. So both PP attachment and noun compounding give rise to exponential ambiguity.

- Coordination $[_X$ X and X]

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

  This is equivalent to the grammar:

  NP → NP and NP
  NP → NP and NP and NP
  NP → NP and NP and NP and NP
  NP → NP and NP and NP and NP and NP
  NP → NP and NP and NP and NP and NP and NP
  NP → NP and NP and NP and NP and NP and NP and NP
  NP → NP and NP and NP and NP and NP and NP and NP and NP
  NP → NP and NP and NP and NP and NP and NP and NP and NP and NP
  ...

n	1	2	3	4	5	6	7	8	9	10
#trees	1	1	3	11	45	197	903	4279	20793	103049

Again we have exponential ambiguity.

12.1 Structurally resolved (local) ambiguity

- Agreement: In simple English clauses, the subject and verb agree, even though the subject and verb can be arbitrarily far apart:
  - The deer {are, is} in the field
  - The deer, the guide claims, {are, is} in the field
- Binding: The number of the embedded subject is unspecified in the following sentence:
  - I expect the deer to admire the pond.

  But in similar sentences it can be specified by binding relations:
  - I expect the deer to admire {itself,themselves} in the reflecions of the pond.
- Head movement:
  - * Have the students take the exam!
    * Have the students taken the exam?
  - * Is the block sitting in the box?
    * Is the block sitting in the box red?
- A movement:
  - * The chair$_i$ is t_i too wobbly for the old lady to sit on it
    * The chair$_i$ is t_i too wobbly for the old lady to sit on t_i
- A' movement:
  - * Who$_i$ did you help t_i to prove that John was unaware that he had been leaking secrets
    * Who$_i$ did you help t_i to prove that John was unaware t_i that he had been leaking secrets to t_i

12.2 Even without global ambiguity: English is not LR(k) for any k

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

- a. The chair$_i$ is t_i too wobbly for the old lady to sit on it
  b. The chair$_i$ is t_i too wobbly for the old lady to sit on t_i
- a. Who$_i$ did you help t_i to prove that John was unaware that he had been leaking secrets
  b. Who$_i$ did you help t_i to prove that John was unaware t_i that he had been leaking secrets to t_i

Mitch Marcus [168] proposes: (1) **(At least some) garden paths indicate failed local ambiguity resolution.** (2) **To reduce backtracking to human level, delay decisions until next constituent is built.**

It follows that English is not GLC(k) for any GLC method. Instead, he explores "non-canonical" methods, were noticed by Knuth in [151], and studied further by Szymanski and Williams [255] and by Aho and Ullman [2, §6.2]. These are relevant in attempts to model the "incremental" nature of human language understanding!

12.3 All paths at once

12.3.1 CKY recognition for CFGs

(8) For simplicity, we first consider context free grammars $G = \langle \Sigma, N, \rightarrow, \mathtt{S} \rangle$ in Chomsky normal form: Chomsky normal form grammars have rules of only the following forms, for some $A, B, C \in N$, $w \in \Sigma$,

$$\mathrm{A} \rightarrow \mathrm{BC} \qquad \mathrm{A} \rightarrow \mathrm{w}$$

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:

$$\mathrm{S} \rightarrow \epsilon$$

(9) A Chomsky normal form grammar has no unit or empty productions, and hence no "cycles" $A \Rightarrow^+ A$, and no infinitely ambiguous strings.

(10) These grammars allow an especially simple CKY-like tabular parsing method (named after Cock, Kasami, and Younger). To parse a string $w_1, \ldots, w_n$, for $n > 0$ we use the following logic:

$$(i-1, i) : w_i \qquad \text{[axioms]}$$

$$\frac{(i,j) : w}{(i,j) : A} \qquad \text{[reduce1]} \qquad \text{if } A \rightarrow w$$

$$\frac{(i,j) : B \quad (j,k) : C}{(i,k) : A} \qquad \text{[reduce2]} \qquad \text{if } A \rightarrow BC$$

Recognition is successful iff $(0, n) : S$ is in the closure of the axioms under these inference rules.

This recognition method is based on proposals of Cocke, Kasami and Younger [2, 237, 238].

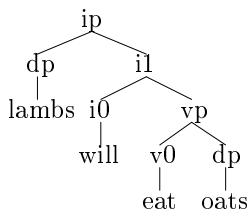
As will become clear when we collect trees from the closure, the closure, in effect, represents all derivations, but the representation is reasonably sized even when the number of derivations is infinite, because the number of possible items is finite.

(11) The soundness and completeness of this method are shown in [237]. Aho & Ullman [2, §4.2.1] show in addition that for a sentence of length n, the maximum number of steps needed to compute the closure is proportional to n^3. They say of this recognition method [2, p.314]:

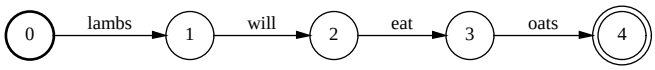
> It is essentially a "dynamic programming" method and it is included here because of its simplicity. It is doubtful, however, that it will find practical use, for three reasons:
>
> 1. n^3 time is too much to allow for parsing.
> 2. The method uses an amount of space proportional to the square of the 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.

CKY example 1

```
ip → dp i1
dp → lambs
i1 → i0 vp
i0 → will
vp → v0 dp
v0 → eat
dp → oats
```



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:[1]

	0	1	2	3	4
0		(0,1):dp (0,1):lambs			(0,4):ip
1			(1,2):i0 (1,2):will		(1,4):i1
2				(2,3):v0 (2,3):eat	(2,4):vp
3					(3,4):dp (3,4):oats
4					

CKY extended

(12) We can relax the requirement that the grammar be in Chomsky normal form. For example, to allow arbitrary empty productions, and rules with right sides of length 3,4,5,6, we could add the following rules:

$$\frac{}{(i,i):A} \quad \text{[reduce0]} \quad \text{if } A \rightarrow \epsilon$$

$$\frac{(i,j):B \quad (j,k):C \quad (k,l):D}{(i,l):A} \quad \text{[reduce3]} \quad \text{if } A \rightarrow BCD$$

$$\frac{(i,j):B \quad (j,k):C \quad (k,l):D \quad (l,m):E}{(i,m):A} \quad \text{[reduce4]} \quad \text{if } A \rightarrow BCDE$$

$$\frac{(i,j):B \quad (j,k):C \quad (k,l):D \quad (l,m):E \quad (m,n):F}{(i,n):A} \quad \text{[reduce5]} \quad \text{if } A \rightarrow BCDEF$$

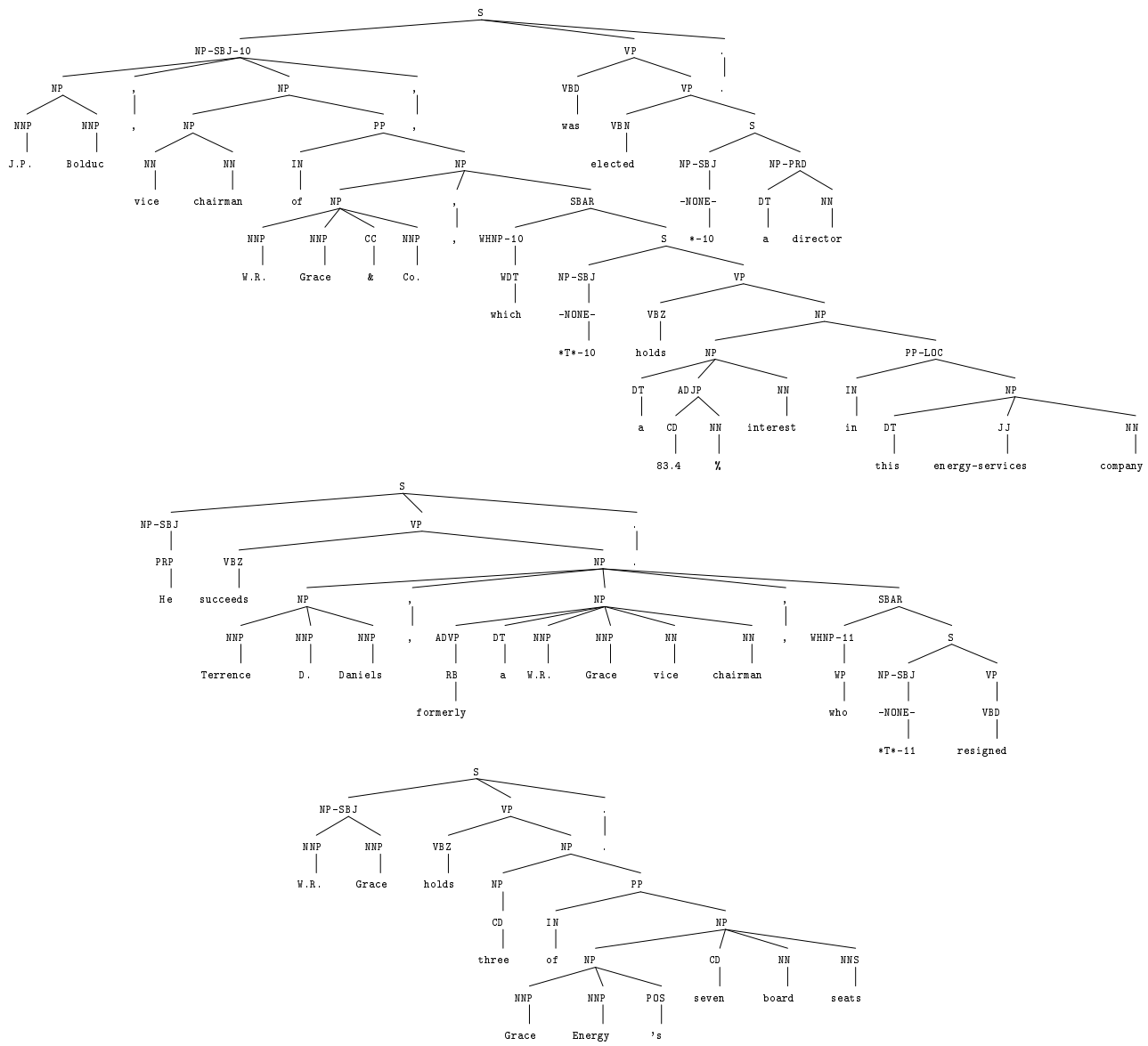
$$\frac{(i,j):B \quad (j,k):C \quad (k,l):D \quad (l,m):E \quad (m,n):F \quad (n,o):G}{(i,o):A} \quad \text{[reduce6]} \quad \text{if } A \rightarrow BCDEFG$$

(13) While this augmented parsing method is correct, it pays a price in efficiency. The Earley method of the next section can do better.

CKY example 2

Since we can now recognize the language of any context free grammar, we can take grammars written by anyone else and try them out. For example, we can take the grammar defined by the Penn Treebank and try to parse with it. For example, in the file `wsj_0005.mrg` we find the following 3 trees:

[1] CKY tables and other similar structures of intermediate results are frequently constructed by matrix operations. This idea has been important in complexity analysis and in attempts to find the fastest possible parsing methods [261, 161]. Extensions of matrix methods to more expressive grammars are considered by [230] and others.



Notice that these trees indicate movement relations, with co-indexed traces. If we ignore the movement relations and just treat the traces as empty, though, we have a CFG – one that will accept all the strings that are parsed in the treebank plus some others as well.

We will study how to parse movements later, but for the moment, let's collect the (overgenerating) context free rules from these trees. Dividing the lexical rules from the others, and showing how many times each rule is used, we have first:

```
1 ('SBAR' -> ['WHNP-11','S']).
1 ('SBAR' -> ['WHNP-10','S']).
2 ('S' -> ['NP-SBJ','VP']).
2 ('S' -> ['NP-SBJ','VP','.']).
1 ('S' -> ['NP-SBJ-10','VP','.']).
1 ('S' -> ['NP-SBJ','NP-PRD']).
3 ('VP' -> ['VBZ','NP']).
1 ('VP' -> ['VBN','S']).
1 ('VP' -> ['VBD']).
1 ('VP' -> ['VBD','VP']).
3 ('NP-SBJ' -> ['-NONE-']).
2 ('PP' -> ['IN','NP']).
2 ('NP' -> ['NP','PP']).
1 ('PP-LOC' -> ['IN','NP']).
1 ('NP-SBJ-10' -> ['NP',',','NP',',']).
1 ('NP-SBJ' -> ['PRP']).
1 ('NP-SBJ' -> ['NNP','NNP']).
1 ('NP-PRD' -> ['DT','NN']).
1 ('NP' -> ['NP','PP-LOC']).
1 ('NP' -> ['NP','CD','NN','NNS']).
1 ('NP' -> ['NP',',','SBAR']).
1 ('NP' -> ['NP',',','NP',',','SBAR']).
1 ('NP' -> ['NNP','NNP']).
1 ('NP' -> ['NNP','NNP','POS']).
1 ('NP' -> ['NNP','NNP','NNP']).
```

```
1 ('NP' -> ['NNP','NNP','CC','NNP']).
1 ('NP' -> ['NN','NN']).
1 ('NP' -> ['DT','JJ','NN']).
1 ('NP' -> ['DT','ADJP','NN']).
1 ('NP' -> ['CD']).
1 ('NP' -> ['ADVP','DT','NNP','NNP','NN','NN']).
1 ('ADVP' -> ['RB']).
1 ('ADJP' -> ['CD','NN']).
1 ('WHNP-11' -> ['WP']).
1 ('WHNP-10' -> ['WDT']).
```

And then the lexical rules:

```
5 (',' -> [',']).
4 ('NNP' -> ['Grace']).
3 ('NNP' -> ['W.R.']).
3 ('DT' -> [a]).
3 ('.' -> ['.']).
2 ('VBZ' -> [holds]).
2 ('NN' -> [vice]).
2 ('NN' -> [chairman]).
2 ('IN' -> [of]).
1 ('WP' -> [who]).
1 ('WDT' -> [which]).
1 ('VBZ' -> [succeeds]).
1 ('VBN' -> [elected]).
1 ('VBD' -> [was]).
1 ('VBD' -> [resigned]).
1 ('RB' -> [formerly]).
1 ('PRP' -> ['He']).
1 ('POS' -> ['\'s']).
1 ('NNS' -> [seats]).
1 ('NNP' -> ['Terrence']).
1 ('NNP' -> ['J.P.']).
1 ('NNP' -> ['Energy']).
1 ('NNP' -> ['Daniels']).
1 ('NNP' -> ['D.']).
1 ('NNP' -> ['Co.']).
1 ('NNP' -> ['Bolduc']).
1 ('NN' -> [interest]).
1 ('NN' -> [director]).
1 ('NN' -> [company]).
1 ('NN' -> [board]).
1 ('NN' -> ['%']).
1 ('JJ' -> ['energy-services']).
1 ('IN' -> [in]).
1 ('DT' -> [this]).
1 ('CD' -> [three]).
1 ('CD' -> [seven]).
1 ('CD' -> [83.4]).
1 ('CC' -> [&]).
1 ('-NONE-' -> ['*T*-11']).
1 ('-NONE-' -> ['*T*-10']).
1 ('-NONE-' -> ['*-10']).
```

12.3.2 Earley recognition for CFGs

(14) Earley [75] showed, in effect, how to build an oracle into a chart construction algorithm for any grammar $G = \langle \Sigma, N, \rightarrow, \mathtt{s} \rangle$. With this strategy, the 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 $S, T, U, V \in (N \cup \Sigma)^*$, and for input $w_1 \ldots w_n \in \Sigma^n$,

$$(0,0) : S' \rightarrow [] \bullet S \qquad \text{[axiom]}$$

$$\frac{(i,j) : A \rightarrow S \bullet w_{j+1} T}{(i,j+1) : A \rightarrow S w_{j+1} \bullet T} \qquad \text{[scan]}$$

$$\frac{(i,j) : A \rightarrow S \bullet BT}{(j,j) : B \rightarrow \bullet U} \qquad \text{[predict]} \qquad \text{if } B \rightarrow U \text{ and } (U = \epsilon \vee U = CV \vee (U = w_{j+1} V)$$

$$\frac{(i,k) : A \rightarrow S \bullet BT \quad (k,j) : B \rightarrow U \bullet}{(i,j) : A \rightarrow SB \bullet T} \qquad \text{[complete]}$$

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.

Also note that in order to apply the scan rule, we need to be able to tell which word is in the $j+1$'th position.

12.4 PCFGs

(15) A probabilistic context free grammar (PCFG)

$$G = \langle \Sigma, N, (\rightarrow), \mathtt{S}, P \rangle,$$

where

1. Σ, N are finite, nonempty sets,
2. S is some symbol in N,
3. the binary relation $(\rightarrow) \subseteq N \times (\Sigma \cup N)^*$ is also finite (i.e. it has finitely many pairs),
4. the function $P : (\rightarrow) \rightarrow [0, 1]$ maps productions to real numbers in the closed interval between 0 and 1 in such a way that

$$\sum_{\langle \mathsf{c}, \beta \rangle \in (\rightarrow)} P(\mathsf{c} \rightarrow \beta) = 1$$

We will often write the probabilities assigned to each production on the arrow: $c \xrightarrow{p} \beta$

(16) The probability of a parse is the product of the probabilities of the productions in the parse

(17) The probability of a string is the sum of the probabilities of its parses

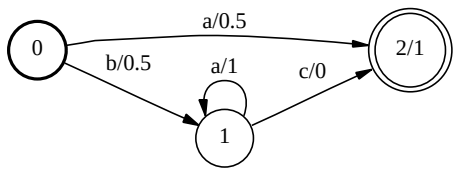
(18) A PCFG is said to be **consistent** iff $p(L(G)) = 1$.

That is, where G defines the whole space of linguistic events, given that a linguistic event is going to occur, the probability that some linguistic event occurs should be 1. Not more, not less!

(19) **Thm.** A PCFG is consistent if the probabilities of its rules are the relative frequencies of those rules in a finite sample of derivations. [41]

(20) **Thm: Linear grammar consistency.** A linear grammar is consistent iff from every state there is a non-zero probability of reaching some terminal string.

(21) Example inconsistent PFSA:



(22) Example inconsistent PCFG [40, p.132]:

$$S \xrightarrow{0.4} \epsilon$$
$$S \xrightarrow{0.6} SS$$

(23) **Thm: PCFG consistency.** (Grenander, Booth&Thompson) Consider the matrix $T(G)$ with rows indexed by N and columns indexed by $(\rightarrow)$, where for any $A \in N, r \in (\rightarrow)$,

$$T(A, r) = \begin{cases} 0 & \text{if } R \neq A \rightarrow \gamma \qquad \text{(for some } \gamma) \\ p(r) & \text{otherwise.} \end{cases}$$

This is sometimes called the first-moment matrix for G. Let ρ be the largest eigenvalue of the first-moment matrix for PCFG G. Then G is consistent if $|\rho| < 1$ and inconsistent if $|\rho| > 1$. When $|\rho| = 1$, G is consistent iff there is no 'final class' of nonterminals, where a final class is a set of nonterminals which, when rewritten, always produce at least one other member of the class.[2] See [271, 114], for algorithms to check for $|\rho| < 1$.

(24) Does any PCFG (or similar device) define the actual frequencies of human language sentences? (no!)

Do they 'approximate' those of a probabilistic model? (what does that mean?)

[2] This result is sometimes attributed to an unpublished technical report by Ulf Grenander, but it is independently presented in [23]. It is also covered in the text [105].

(25) We have extended the CKY parsing strategy can handle any CFG, and augmented the chart entries so that they indicate the rule used to generate each item and the positions of internal boundaries.

We still have a problem getting the parses out of the chart, since there can be too many of them: we do not want to take out one at a time!

One thing we can do is to extract just the most probable parse. An equivalent idea is to make all the relevant comparisons before adding an item to the chart.

(26) For any input string, the CKY parser chart represents a grammar that generates only the input string. We can find the most probable parse using the Trellis-like construction familiar from Viterbi's algorithm.

(27) For any positions $0 \leq i, j \leq |input|$, we can find the rule $(i, j) : A : \ X$ with maximal probability.

$$(i-1, a_i, i) \qquad [axiom]$$

$$\frac{(i, a, j)}{(i, A, j, p)} \qquad [reduce1] \quad \text{if } A \xrightarrow{p} a \text{ and } \neg\exists A \xrightarrow{p'} a \text{ such that } p' > p$$

$$\frac{(i, B, j, p_1) \quad (j, C, k, p_2)}{(i, A, k, p_1 * p_2 * p_3)} \qquad [reduce2] \quad \text{if } A \xrightarrow{p_3} BC \text{ and } \neg\exists(i, B', j, p_1'), (j, C', k, p_2'), A \xrightarrow{p_3'} B'C' \text{ such that } p_1' * p_2' * p_3' > p_1 * p_2 * p_3$$

(28) This algorithm does (approximately) as many comparisons of items as the non-probabilistic version, since the reduce rules require identifying the most probable items of each category over each span of the input.

To reduce the chart size, we need to restrict the rules so that we do not get all the items in there – and then there is a risk of missing some analyses.

12.4.1 Exercises

a. Important properties of PCFGs and of CFGs with other distributions are established in [40].

b. Train a stochastic context free grammar with a "treebank," etc, and then "smooth" to handle the sparse data problem: [39, 58, 21].

c. Transform the grammar to carry lexical particulars up into categories: [130, 78].

d. Probabilistic Earley parsing and other strategies: [253, 165]

e. Hack Viterbi parse selection to make it as fast as possible [150]

f. Instead of finding the very best parse, use an "n-best" strategy: [38] and many others.

g. Use multiple information sources: [216] and many others.

Instead of pursuing these topics, we will first look at grammars that do better than CFGs at defining the structure of human language syntax.

12.5 Appendix: implementing cf-cky

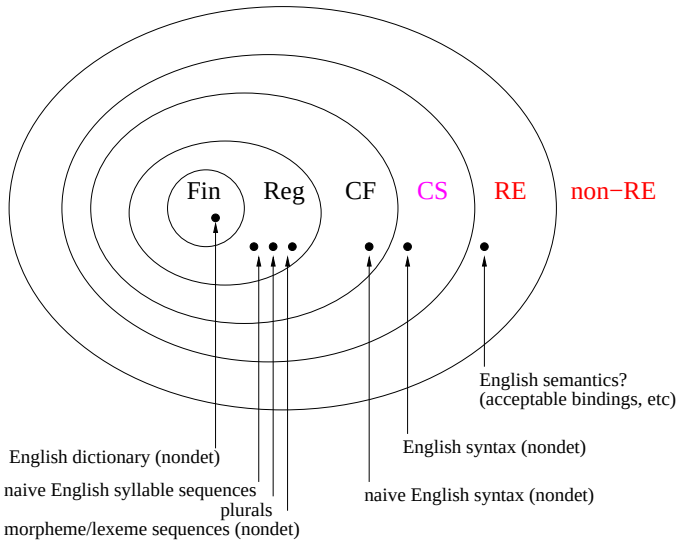
```
(*  file: cf-cky.ml
 creator: E Stabler
 updated: 2008-02-26 09:11:10 PST
 purpose: CKY-like parser

 the CKY algorithm for Chomsky normal form (CNF) grammars has an efficient matrix-based implementation
 (using e.g the Floyd-Warshall algorithm to compute the transitive closure)
 but here we implement the more intuitive slightly less efficient chart-based method

 For a string of length n, we implement the (n+1 * n+1) chart with an array.
 We implement the agenda as a list (i.e. a stack)
 Each time a *new* item is added to the chart, it goes onto the agenda too.
*)

(** print functions for tracing – for this i/o, we use an imperative style, with loops **)

let rec printChart chart =
  for i=0 to (Array.length chart)-1 do
    for j=0 to (Array.length chart)-1 do
      (List.iter (function x -> Printf.fprintf stdout "(%i,%s,%i)\n" i x j)) chart.(i).(j)
    done
  done;;

let rec printAgenda agenda =
  begin
    Printf.fprintf stdout "---agenda:\n";
    (List.iter (function (i,x,j) -> Printf.fprintf stdout "(%i,%s,%i)\n" i x j)) agenda;
    Printf.fprintf stdout "---\n";
  end;;

(* test *)

(** END print functions for tracing **)

let cfg1 = [
  ("IP",["DP";"I1"]);   ("I1",["I0";"VP"]);   ("I0",["will"]);
  ("DP",["D1"]);        ("D1",["D0";"NP"]);   ("D0",["the"]);
  ("NP",["N1"]);        ("N1",["N0"]);        ("N0",["idea"]);
                        ("N1",["N0";"CP"]);
  ("VP",["V1"]);        ("V1",["V0"]);        ("V0",["suffice"]);
  ("CP",["C1"]);        ("C1",["C0";"IP"]);   ("C0",["that"]);
];;

(* for e and list, if e not in list then return (newlist,true), else return (list,false) *)
let rec ensureMemberVal e list = match list with
    [] -> ([e],true)
  | x::xs -> if e=x
    then (x::xs,false)
    else let (list,added) = (ensureMemberVal e xs) in (x::list, added);;

(* ensure that element e is in chart, with added=false if e was already there *)
let rec ensureElementVal (i,e,j) chart =
  let (list,added) = ensureMemberVal e chart.(i).(j) in
    if added then
      begin
        chart.(i).(j) <- list;
        (chart, added);
      end
    else (chart, added);;

(* add words to the chart *)
let rec scanAll agenda chart input i = match input with
    [] -> (agenda,chart)
  | word::words ->
      let (agenda1,_) = ensureMemberVal (i,word,i+1) agenda in
      let (chart1,_) = ensureElementVal (i,word,i+1) chart in
        scanAll agenda1 chart1 words (i+1);;
```

```
(* test*)
let agenda0a0=[] in
let chart0=Array.make_matrix 5 5 [] in
let input = ["the";"idea";"will";"suffice"] in
let (agenda,chart) = scanAll agenda0a0 chart0 input 0 in
  begin
    printAgenda agenda;
    printChart chart;
  end;;

(* look for preceding category of right side of c expansion, if any, *)
(* adding (k,c,j) for each (k,d,i) that is found *)
let rec reduce2a k d i c j cell agenda chart = match cell with
    [] ->
      if k < i
      then reduce2a (k+1) d i c j chart.(k+1).(i) agenda chart
      else (agenda,chart)
  | cat::cats ->
      if cat=d then
        let (chart1,added) = ensureElementVal (k,c,j) chart in
            if added
            then reduce2a k d i c j cats ((k,c,j)::agenda) chart1
            else reduce2a k d i c j cats agenda chart
      else reduce2a k d i c j cats agenda chart;;

(* find all reductions using elements on agenda,
   each new value added to chart is put on the agenda too
*)
let rec reduce grammar (i,x,j) (agenda,chart) = match grammar with
    [] -> (agenda,chart)
  | (c,[d])::rules -> (* reduce 1 *)
      if d=x then
        let (chart1,added) = ensureElementVal (i,c,j) chart in
            if added
            then reduce rules  (i,x,j) (((i,c,j)::agenda),chart1)
            else reduce rules (i,x,j) (agenda,chart)
      else reduce rules (i,x,j) (agenda,chart)
  | (c,[d;e])::rules ->
      if e=x
      then reduce2a 0 d i c j chart.(0).(i) agenda chart
      else reduce rules (i,x,j) (agenda,chart);
  | _ -> raise (failwith "reduce: unusable rule");;

(* call reduce repeatedly until agenda is empty *)
let rec reduceAll grammar (agenda,chart) = match agenda with
    [] -> (agenda,chart)
  | (i,x,j)::moreAgendaa -> reduceAll grammar (reduce grammar (i,x,j) (moreAgendaa,chart));;

(* make a square array of lists, then fill it in all possible ways using CKY rules, and print it out *)
let rec cky grammar input =
  let n = (List.length input)+1 in
  let chart0 = Array.make_matrix n n [] in
  let (agenda1,chart1) = reduceAll grammar (scanAll [] chart0 input 0) in
    printChart chart1;;

(* tests *)
cky cfg1 ["the";"idea"];;
cky cfg1 ["the";"idea";"that";"the";"idea";"will";"suffice";"will";"suffice"];;
cky cfg1 ["the";"idea";"will";"suffice"];;
```

12.6 Exercise

(29) The Floyd-Warshall algorithm is one of the basic tricks every programmer should know. See, e.g. [59, §26.2]. Implement CKY in Ocaml in a way that allows you to use it.

§13 Beyond phrase structure grammars



(0) Chomsky suggested in the 1950's that CFGs are too restrictive: they can describe some things in human language only awkwardly, and other things seem impossible.

Marie will have -∅ be -en prais -ing Pierre

> When we turn to the complexity of description...we find that there are simple grounds for the conclusion that [rewrite grammars for such constructions are] fundamentally inadequate...We can, in fact, give a very simple analysis ...but only by selecting as elements certain discontinuous strings. (Chomsky 1956, pp119-20)

will Marie have -∅ be -en prais -ing Pierre

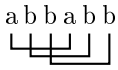
(1) Dutch also has crossing dependencies, but here they seem to lack any principled, finite bound [125, 27]:

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

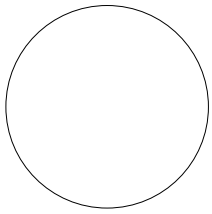
(2) Swiss-German also has crossing dependencies, and here they are syntactically coded in the case agreement requirements of the verbs [236]

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

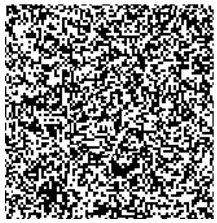
(3) The language $\{xx | x \in \{a,b\}^*\}$ is simpler but similar. It is usually shown to be beyond the expressive power of context free grammars with the pumping lemma. See, e.g., [239, p119] or [118, p279].



(4) It is natural to regard perception as the identification of invariants, regularities, that allow a simple description.



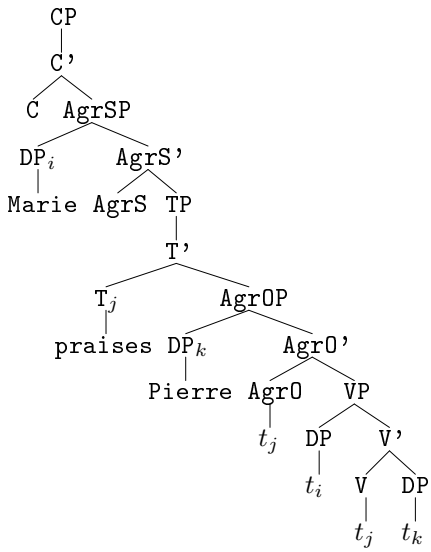
Marie praises Pierre



rpes eaiiPrM rkireea

In perception, we recognize certain regularities in perceptual data. Can we adopt a similar perspective on language?

'Symmetric figures' have relatively simple descriptions, since they cannot vary along parameters that can be affected by the designated operations on the plane, the symmetries. It is natural to think that language recognition is similar, except the ability to identify the simple description is acquired, with considerably more plasticity.



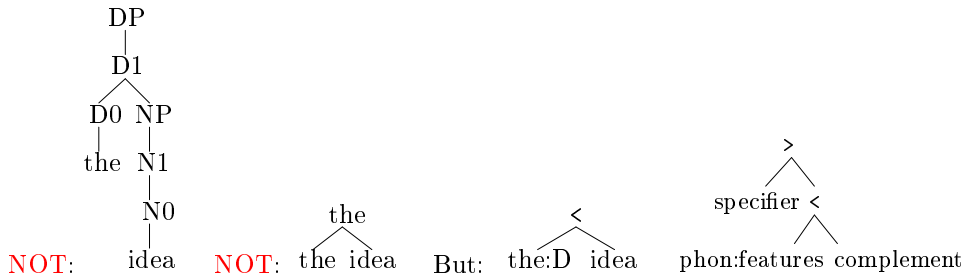
Marie praises Pierre ⇒

(P1a) Can the structure of a sentence be smaller than the sentence itself? Yes

Traditional representations of linguistic structure are very redundant!

13.0 Minimalist grammars on trees

(5) **Goal:** We would like grammars that allow elegant description of patterns like these but keeping the simplicity of CFGs insofar as possible. A first step is taken with formal **minimalist grammars** (MGs) that we will now define [243, 112, 176], inspired by Chomsky's 'minimalist program' [50, 51].



- The arrows ('order' symbols) <, > "point toward" the **head** of the phrase.
- The largest subtree with a given head is a **maximal** projection, or **phrase** XP.
- Initially, each phrase will have at most one complement, but any number of specifiers.

(6) Tentatively, following [243, 245, 245], let a minimalist grammar = (Lexicon,{merge,move}), where

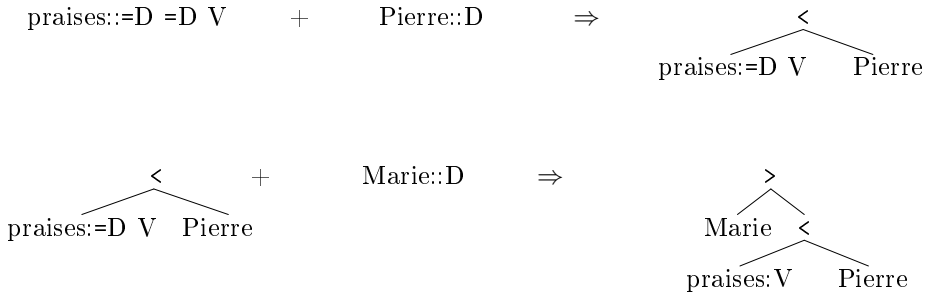
Lexicon: associates vocabulary with feature sequences:

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

(5a) **Merge** triggered by =X, attaches X on right if simple, left otherwise

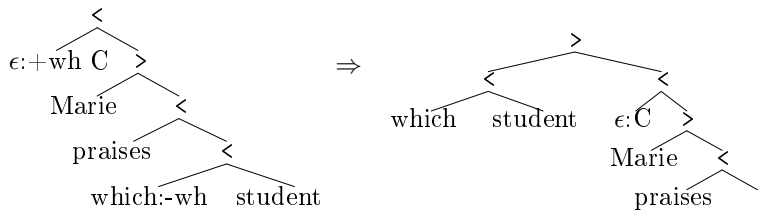


Each structure building operation applies to expressions, deleting a pair of features (shown in red), and building headed binary tree structures like those shown here, with the order symbols "pointing" towards the head.

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*.

(5b) **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.

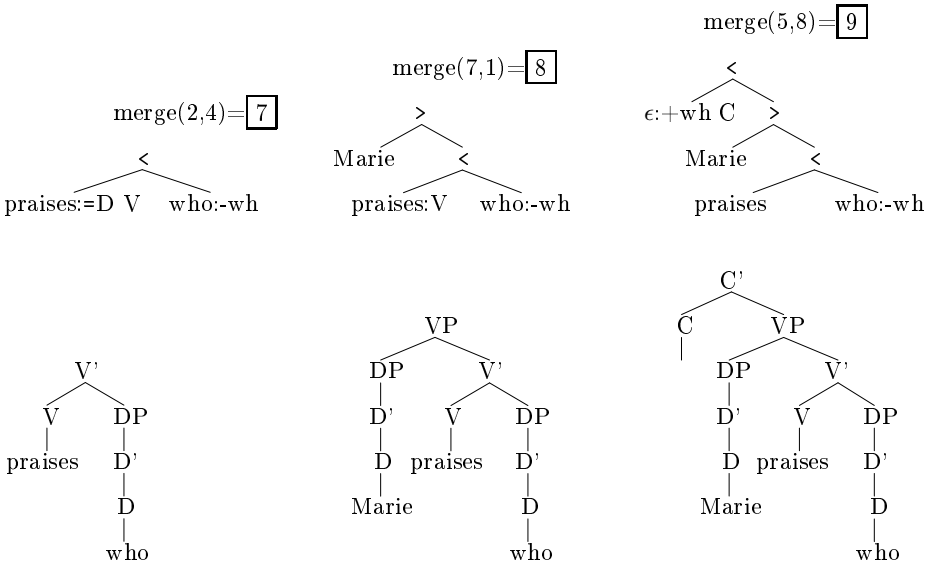
This is a unary, simplification step, but like merge, it deletes a pair of features and adds an "arrow".

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

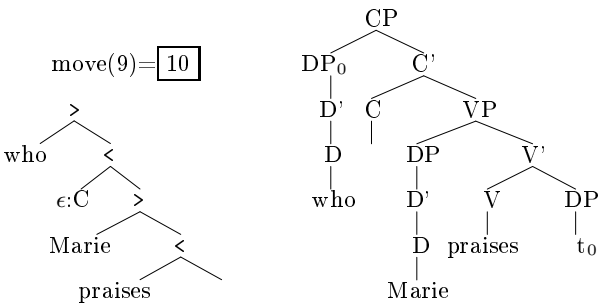
(7) **example MG1,** with 7 lexical items.

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`		

steps 1,2,3



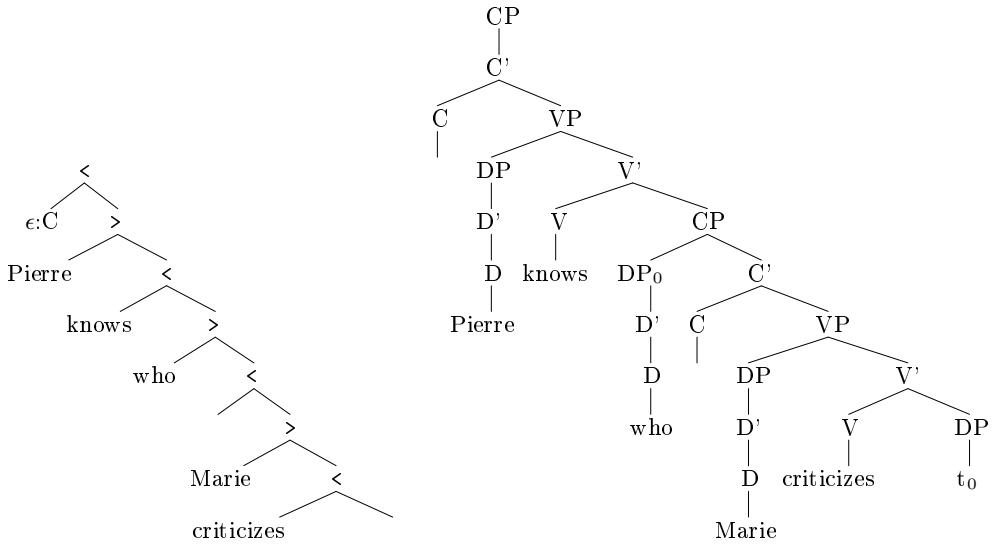
step 4



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

13.1 Minimalist grammars: basic properties

example 1 extended: knows::=C =D V



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`	`knows::=C =D V`	7

Pierre knows who Marie criticizes ___
Pierre knows who ___ criticizes Marie
who ___ knows Marie criticizes Pierre
who Pierre knows Marie criticizes___ (needs do-support)
who Pierre knows ___ criticizes Marie
*who Pierre knows who ___ criticizes ___ (by SMC!)

SMC immediately gives us a kind of 'relativized minimality' effect where the domains are determined by the identities of the licensee/licensor pairs [222, 223]. Challenges: multiple wh-movements, etc.

(8) Easy constructions easily defined: subjects before objects, question words tend to be initial.

[Subject [VP]] [Wh [Sentence]]

This claim about SVO is classically discussed in [12, 106]. Getting something like [[S V] O] is not impossible but *harder* than [S [V O]], in the sense of requiring a larger grammar and more derivational steps. (We get a direct relation between grammar size and derivation complexity in these systems, since each derivation step checks and deletes exactly 2 features.)

In Hungarian verbal complexes,

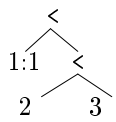
- Nem fogok akarni kezdeni haza-menni
  not will-1s want-inf begin-inf home-go-inf

  V_1 V_2 M-V_3 123 132 321 *213 Koopman & Szabolcsi [156]

And, across languages,

D Num A N 1234 4123 4321 *4213 Greenberg [106]

1::=2 1 2::=3 2 3::3



Claim. What re-orderings can we get with movement? **5 out of the 6. We cannot get *213**
(That is, we get these by movement alone, with no empty heads involved.)

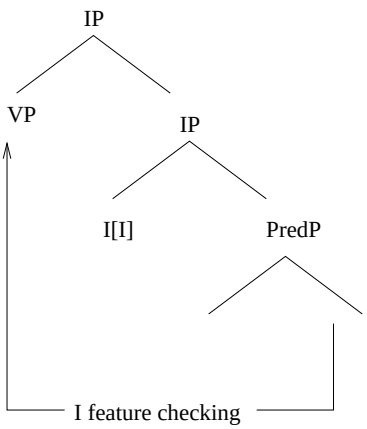
- Greenberg says: ...when any of the items (demonstrative, numeral, and descriptive adjective) precede the noun, they are always found in that order. If they follow, the order is either the same or the exact opposite

1::=2 1 2::=3 2 3::=4 3 4

Claim. What re-orderings can we get with movement? **15 out of the 24. We cannot get *4213**

Exercise 1. Produce the grammars and the derivations for the possible orderings of 123,

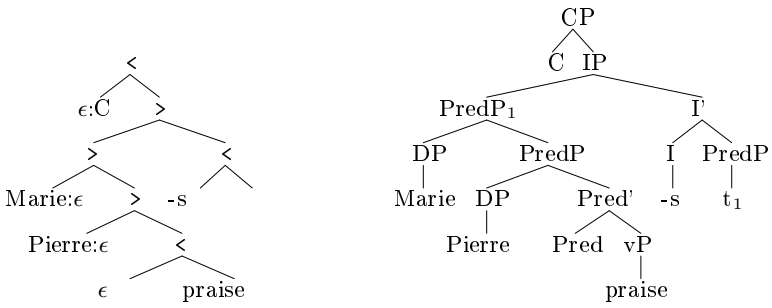
13.2 Example: SOVI "Naive Tamil"



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 [166] 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                    Marie::D
praise::v                    criticize::v
ϵ::=I C                      -s::=Pred +v I
ϵ::=v =D =D Pred -v
```

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

Exercise 2. Display the step-by-step derivation of this structure.

13.3 Example: Cable on Tlingit

mainly SOV, head-final: post-positions; possessors precede possessed; adjectives precede nouns; aux follows main V

(9) Obligatory *sá*

a. Daa *(sá) aawa<u>x</u>aa i éesh?
what Q he.ate.it your father
'What did your father eat?'

b. Goodéi *(sá) k<u>k</u>wagút?
where.to Q I.will.go
'Where will I go?'

(10) *sá* to right of wh-word

a. Aadóo yaagu sá ysiteen?
who boat Q you.saw.it
'Whose boat did you see?'

(11) but not anywhere to right

a. [aadóo jeet] sá [wé sakwnéin] aawatee
who hand.to Q that bread he.bought.it
'Who did he give the bread to?'

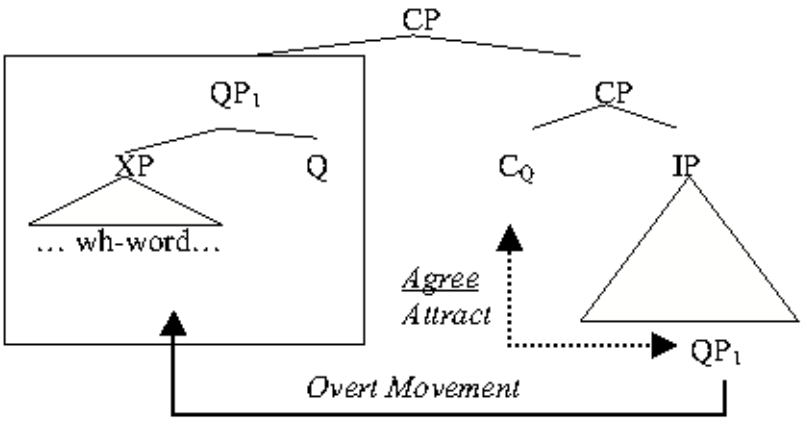
b. * [aadóo jeet] [wé sakwnéin] sá aawatee
who hand.to that bread Q he.bought.it

(12) All questioned elements are fronted

a. [aadóo sá]$_1$ [daa sá]$_2$ [t_1 yéiuwajée [t_2 du jee yéiteeyí]]
who Q what Q they.think their hand it.is.there
'Who thinks they have what?'

(13) *Sá projects a QP over the phrase containing the wh-word*

(14) *Wh-fronting is not directly 'triggered' by any properties of the wh-word. Rather, it is due to the properties of a distinct, formal element: a 'Q-particle'*

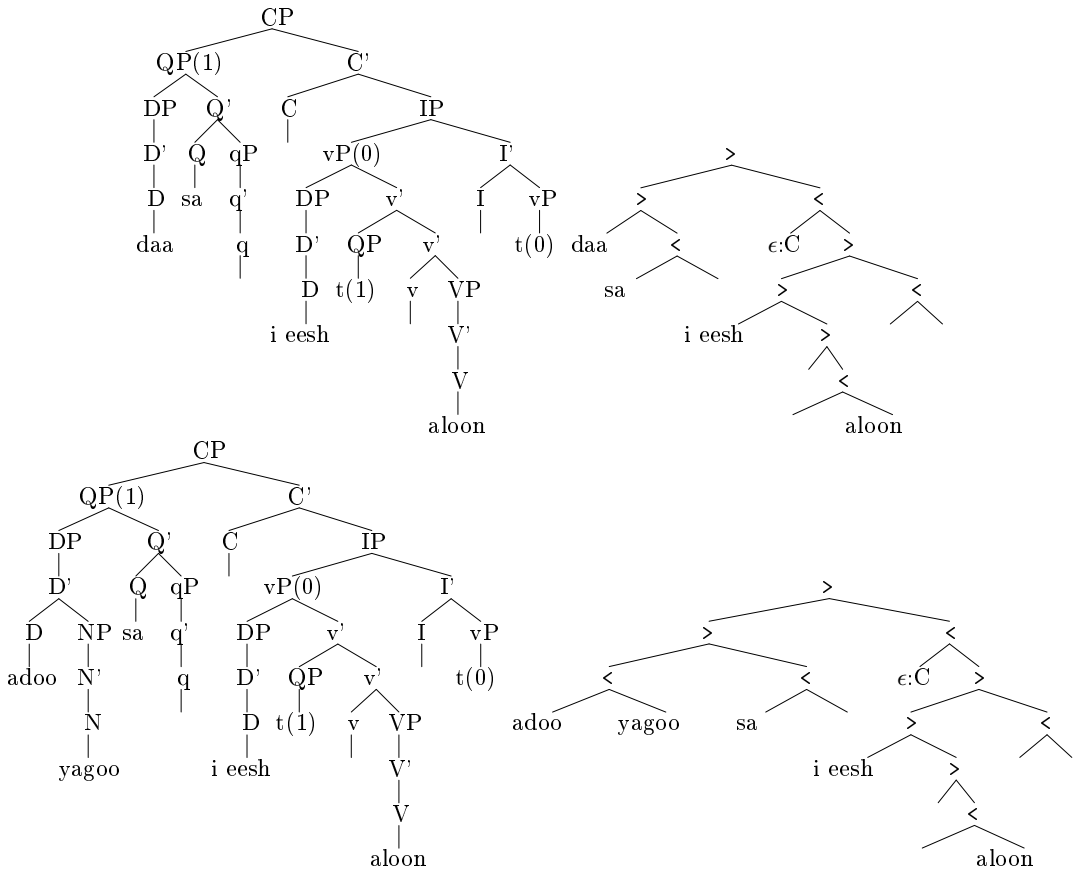


(15) a. how to get [DP Q] but [C IP], [D NP]? (we could assume that Q has a q 'layer')

b. how to properly select [DP Q], [PP Q], [IP Q]? (those will all need to be diff categories)

c. how to get multiple questions?

```
I eesh::D          daa::D             adoo::=N D         yaagu::N
aloon::V           ysiteen::V         sa::=q Q -q        ϵ::q
ϵ::=I C            ϵ::=v +v I
ϵ::=V =D =D v -v   ϵ::=V =Q =D v -v   ϵ::=V =D =Q v -v
```



Exercise 3. Display the step-by-step derivation of one of these structures.

13.4 Example: Adjective orders

(16) Dimitrova, Vulchanova & Giusti [71] observe some near symmetries in the nominal systems of English and Romance, symmetries that have interesting variants in the Balkan languages. [1]

It is interesting to consider whether the restricted grammars we have introduced – grammars with only overt, phrasal movement – can at least get the basic word orders.

In fact, this turns out to be very easy to do. The adjectives in English order appear in the preferred order,

`poss>card>ord>qual>size>shape>color>nationality>n`

with (partial) mirror constructions:

a. an expensive English fabric

b. un tissu anglais cher

(17) sometimes, though, other orders, as in Albanian:

a. një fustan fantastik blu
   a dress fantastic blue

b. fustan-i fantastik blu
   dress-the fantastic blue

(18) AP can sometimes appear on either side of the N, but with a (sometimes subtle) meaning change:

a. un bon chef (good at cooking)

b. un chef bon (good, more generally)

(19) scopal properties, e.g. `obstinate>young` in both

[1] Cf. also [240], [262], [55].

a. an obstinate young man

b. un jeune homme obstiné

To represent these possible selection possibilities elegantly, we use the notation `>poss` to indicate a feature that is either `poss` or a feature that follows `poss` in this hierarchy. And similarly for the other features, so `>color` indicates one of the features in the set `>color={color,nat,n}`. (We may return to a more general and principled account of this kind of ordering of categories later.)

We also put a feature `(-f)` in parentheses to indicate that it is optional.

```
% English
a(n)::=>poss d -case
expensive::=>qual qual
English::=>nat nat
fabric::n
```

```
% French
un::  =>poss d -case
cher::=>qual +f qual (-f)
anglais::  =>nat +f nat (-f)
tissu::n (-f)

bon::  =>qual (+f) qual (-f)
chef::  n (-f)
```

```
% Albanian
i::  =>poss +f d -case
nje::  =>poss d -case
fantastik::  =>qual (+f) qual
blu::  =>color color
fustan::  n -f
```

This grammar gets the word orders shown in (16-18).

The 4 English lexical items allow us to derive [a fabric], [an expensive fabric] and [an expensive English fabric] as determiner phrases (i.e. as trees with no unchecked syntactic features except the feature `d`), but NOT: [an English expensive fabric].

The first 4 French items are almost the same as the corresponding English ones, except for `+f,-f` features that trigger inversions of exactly the same sort that we saw in the approach to Hungarian verbal complexes in [245]. To derive [un tissu], we must use the lexical item `n tissu` – that is, we cannot include the optional feature `-f`, because that feature can only be checked by inversion with an adjective. The derivation of [un [tissu anglais] cher] has the following schematic form:

i. [anglais tissu] $\longrightarrow$ (`nat` selects `n`)

ii. [tissu$_i$ anglais t_i] $\longrightarrow$ (`nat` triggers inversion of `n`)

iii. cher [tissu$_i$ anglais t_i] $\longrightarrow$ (`qual` selects `nat`)

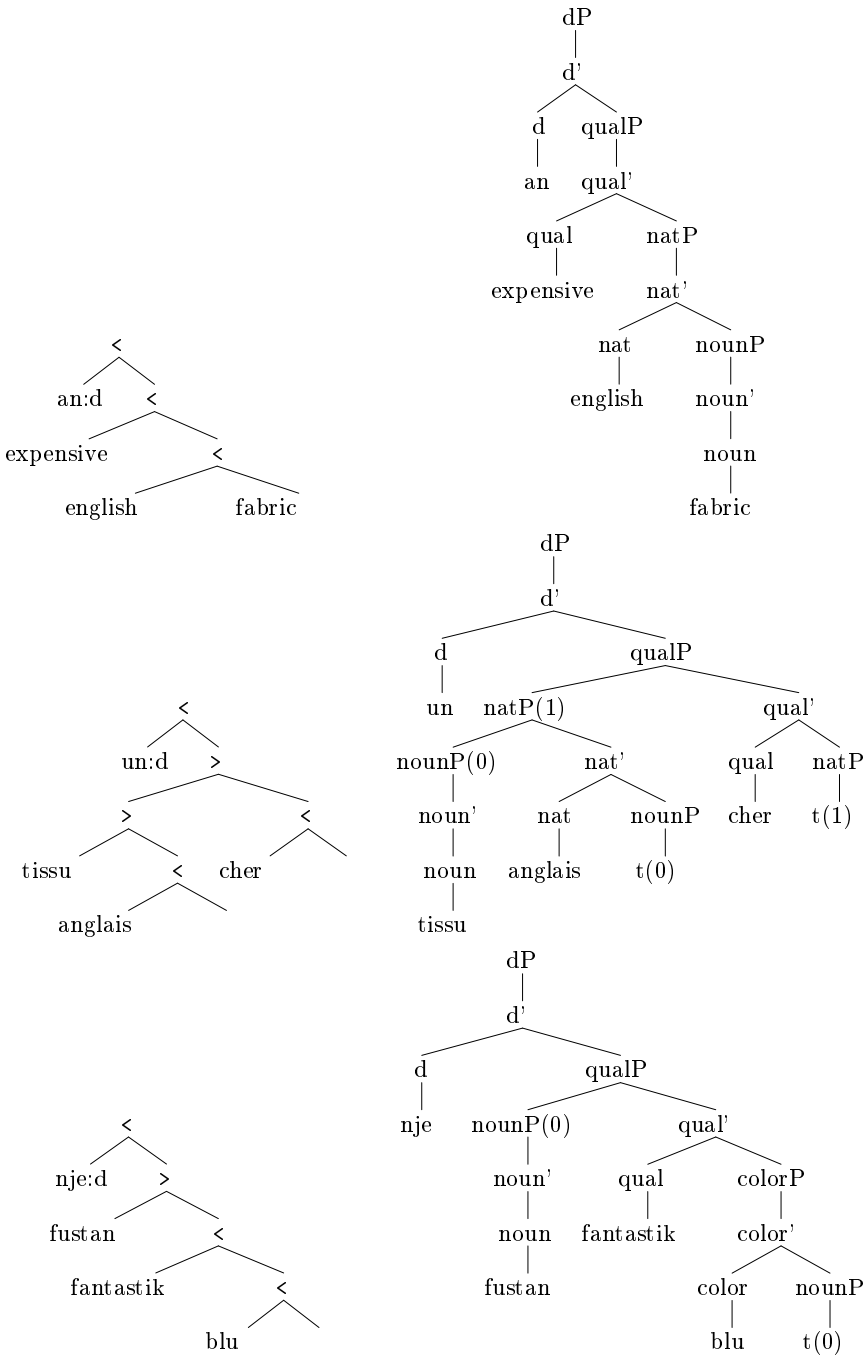
iv. [[tissu$_i$ anglais t_i]$_j$ cher t_j] $\longrightarrow$ (`qual` triggers inversion of `nat`)

v. un [[tissu$_i$ anglais t_i]$_j$ cher t_j] $\longrightarrow$ (`d` selects `qual`)

(The entries for `bon`, on the other hand, derive both orders.) So we see that with this grammar, the APs are in different structural configurations when they appear on different sides of the NP, which fits with the (sometimes subtle) semantic differences.

The lexical items for Albanian show how to get English order but with N raised to one side or the other of the article. We can derive [nje fustan fantastik blu] and [fustan-i fantastik blu] but not the other, impossible orders.

NB: we will use the symbol `=>` for a different purpose later.



Compare the work on adverbs by Cinque et al [55, 56]

13.5 Example Relative clauses according to Kayne.

As noted just above, we can capture order preferences among adjectives by assuming that they are heads selecting nominal phrases rather than left adjuncts of nominal phrases. The idea that some such adjustment is needed proceeds from a long line of interesting work including [15, 262, 240, 140, 200, 55, 71].

Since right adjuncts are not generated by our grammar, [140, §8] proposes that the raising analyses of relative clauses look most promising in this framework, in which the "head" of the relative is raised out of the clause. This kind of analysis was independently proposed much earlier because of an apparent similarity between relative clauses and certain kinds of focus constructions [231, 263, 1, 19]:

1. a. This is the cat that chased the rat

b. It's the cat that chased the rat

2. a. * That's the rat that this is the cat that chased

b. * It's that rat that this is the cat that chased

3. 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'

4. 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.

(20) Kayne assumes that the relative pronoun also originates in the same projection as the promoted head, so we get analyses with the structure:

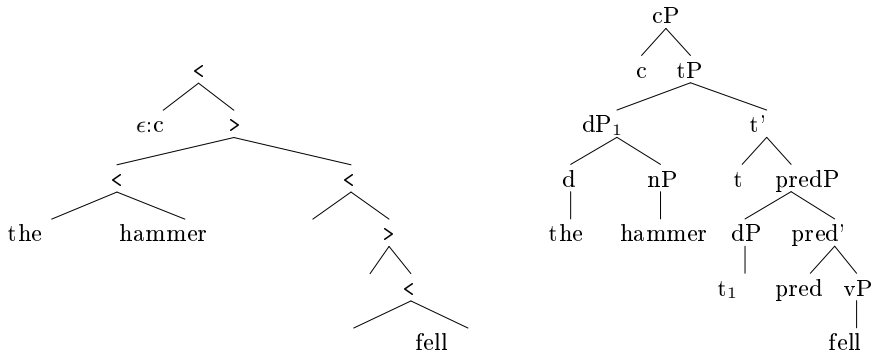
a. The hammer$_i$ [which t_i]$_j$ [$t_j$broke t_h]$_k$ [the window]$_h$ t_k

b. The window$_i$ [which t_i]$_j$ [the hammer]$_h$ [$t_h$broke t_j]$_k$ t_k

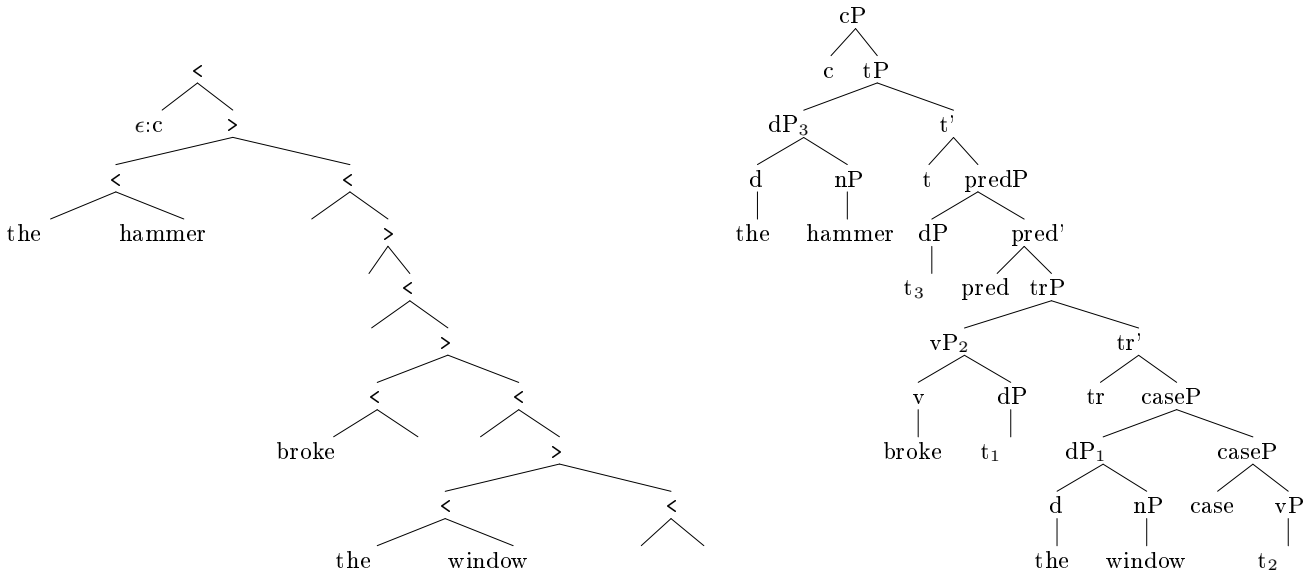
We can obtain this kind of analysis by allowing noun heads of relative clauses to be focused, entering the derivation with some kind of focus feature `-f`.

ϵ::=t c	ϵ::=t +wh$_{rel}$ c$_{rel}$
ϵ::=pred +case t	
the::=n d -case the	which::=n +f d -case -wh$_{rel}$
	the::=c$_{rel}$ d -case
hammer::n	hammer::n -f
window::n	window::n -f
ϵ::=v +case case	
ϵ::=tr =d pred	
ϵ::=case +tr tr	
broke::=d v -tr	

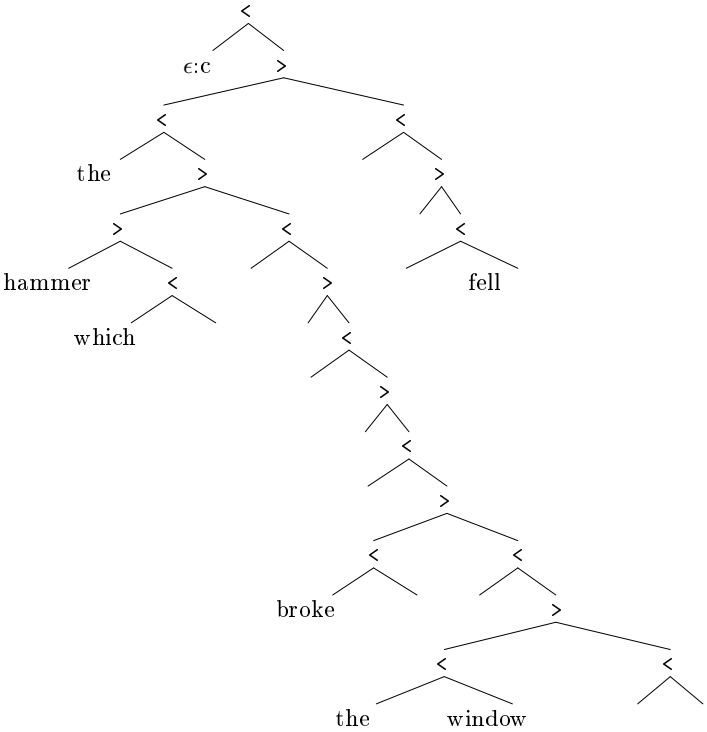
NB: focused lexical items in the second column.

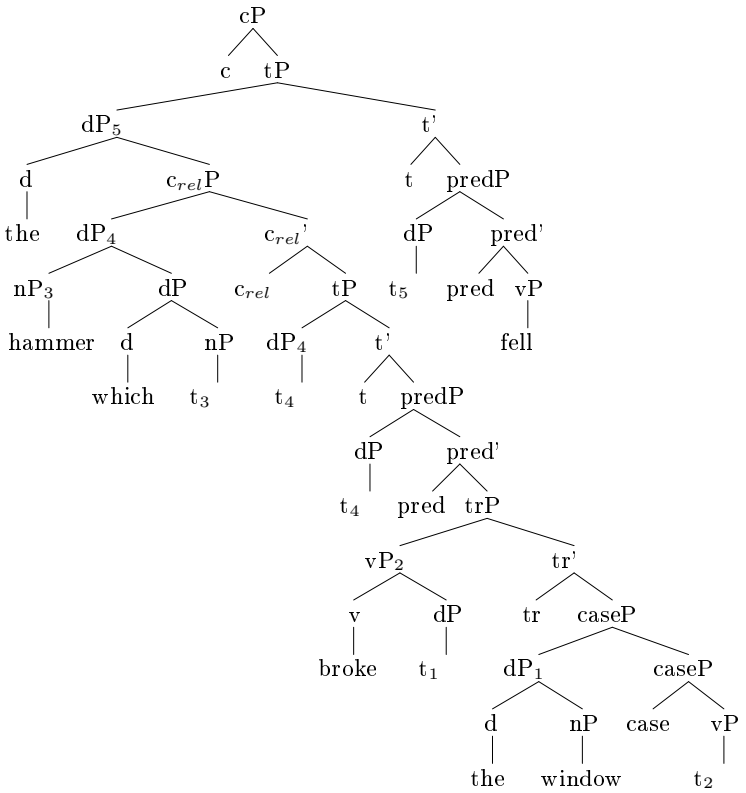


Exercise 4. Display the step-by-step derivation of the structure above.



Exercise 5. Display the step-by-step derivation of the structure above.





(21) Buell [35] 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`:

a. 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'

b. 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 [24] and some of them are answered in [19] and elsewhere...

13.6 Summary

- **vocabulary** $\Sigma = \{$`every,some,student,...`$\}$
- **types** $T = \{$`::, :`$\}$ "lexical" and "derived," respectively
- **syntactic features** F**:**

  `C, T, D, N, V, P,...` (selected categories)

  `=C, =T, =D, =N, =V, =P,...` (selector features)

  `+wh, +case, +focus,...` (licensors)

  `-wh, -case, -focus,...` (licensees)

- **expressions** E**:** trees with non-root nodes $\{$`<,>`$\}$, leaves $\Sigma^* \times T \times F^*$
- **Notation:**

  $t[f]$ = tree with 1st feature f at its head,
  so then t is the result of removing f from $t[f]$ and changing the type :: to :

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

  $t_1^{>}$ = the maximal projection of the head of t_1

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

- a **minimalist grammar** on trees G=$\langle Lex, \{$merge,move$\}\rangle$, where
- **lexicon** $Lex \subset \Sigma^* \times \{::\} \times F^*$, a finite set of 1-node trees
- $$\textbf{merge}(t_1[\texttt{=c}], t_2[\texttt{c}]) = \begin{cases} \overset{<}{\wedge}(t_1, t_2) & \text{if } t_1 \text{ has exactly 1 node} \\ \overset{>}{\wedge}(t_2, t_1) & \text{otherwise} \end{cases}$$

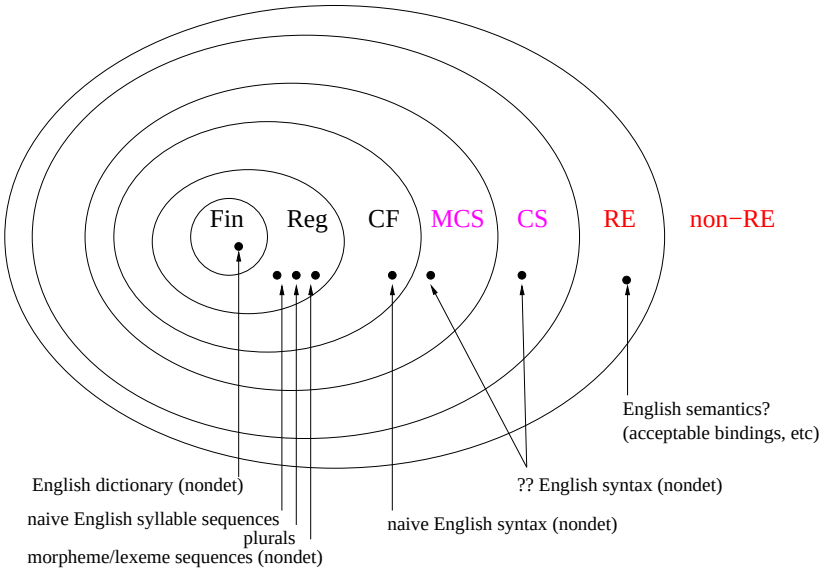
$$\textbf{move}(t_1[\texttt{+f}]) = \overset{>}{\wedge}(t_2^{>}, t_1\{t_2[\texttt{-f}]^{>}/\epsilon\}) \qquad \text{if (SMC) only } \underline{\text{one}} \text{ head has } \texttt{-f} \text{ as its first feature}$$

This was how we defined the rules. The idea is that these rules should be invariant across languages.

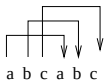
- All variation is lexical. So parameters of variation will have to be captured in the requirements of lexical items. One line of work coming from Pollock, Borer, Rizzi suggests that the parameters are the requirements of the functional categories
- Note that MGs are closed under unions, so this allows the functional categories of different grammars to be present at once, used by the same mechanism (and so this also predicts that code-switching should be possible). It is not so clear how the idea that language involves different UG settings – switches on some control mechanism – can accomodate diglossia.
- All movement is phrasal, overt, upward and to the left, with locality given by this simple SMC.
- All structure-building is feature-driven. This is something we could try to adjust later, when we can replace the features by a characterization of the properties of expressions in which the structure-building operations can apply. That is, we can think of the features as now explicitly labelling something that could actually follow from something else.

These grammars are very 'simple' by linguistic standards, but there is a lot going on here! We can do very much better.

§14 Minimalist grammars (MGs)



(0) The 'crossing correspondences' in $L_{xx} = \{xx|\ x \in \{a,b\}^+\}$ are beyond the power of CFGs:



(1) Various kinds of reduplication and similar crossing dependencies in natural languages:

English [42]:
John will have -∅ be -en eat -ing pie

Dutch [125, 27]:
... because I Cecilia Henk the hippo saw help feed
... omdat ik Cecilia Henk de nijlpaarden zag helpen voeren

Swiss-German [236]:
... that we the children Hans the house let help paint
... das mer d'chind em Hans es huus lönd hälfe aastriiche

English [99]:
we're not LIVING TOGETHER living together

Bambara [61]: 'whatever dog'
w u l u -o- w u l u

Pima [218]: mountain lion -s g egevho

(2) If reduplication is easy and natural, we might expect it to apply recursively, which yields an 'exponential' kind of pattern like this:

$$L = \{a, aa, aaaa, aaaaaaaa, \ldots\} = \{a^{2^n} | n \geq 0\}.$$

This kind of pattern is hard to find definitive evidence for, but there is suggestive evidence of this kind of reduplication in both syntax [152, 20, 177] and phonology [32, 260]. If we need this, MGs are not powerful enough. (We will consider several extensions of MGs. . .)

14.0 Context sensitive languages

(3) A context sensitive rewrite grammar is a rewrite grammar where the rules have the form $s \rightarrow t$ where $|s| \leq |t|$, or else $\mathrm{S} \rightarrow \epsilon$ where S is the 'start category'.

(4) Consider the following context sensitive grammar, for example:

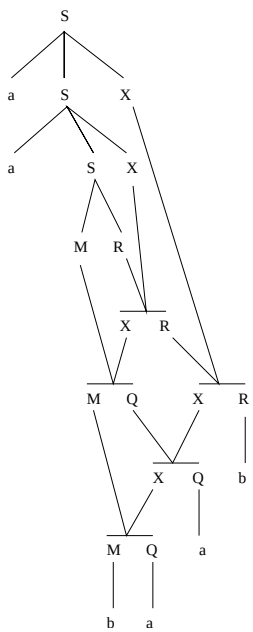
$$\mathrm{V} = \{\mathrm{a}, \mathrm{b}\} \qquad \mathrm{Cat} = \{\mathrm{S}, \mathrm{L}, \mathrm{M}, \mathrm{X}, \mathrm{Y}, \mathrm{Q}, \mathrm{R}\}$$

S → aSX	S → bSY	S → LQ	S → MR
QX → XQ	RX → XR	QY → YQ	RY → YR
LX → LQ	MX → MQ	LY → LR	MY → MR
L → a	M → b	Q → a	R → b

Mattescu and Salomaa [171] show that this grammar generates the simple non-context-free language $L_{xx} = \{xx|\ x \in \{a, b\}^+\}$, the language consisting of strings comprising two copies of any non-empty string of a's and b's. This grammar has derivations like the following:

S	⇒ aSX	⇒ aaSXX	⇒ aaMRXX	⇒ aaMXRX	⇒ aaMXXR
	⇒ aaMXXb	⇒ aaMQXb	⇒ aaMXQb	⇒ aaMXab	⇒ aaMQab
	⇒ aabQab	⇒ aabaab			

(5) It would be nice to present this derivation in some kind of phrase structure tree, but it is not clear how to portray the action of the non-CF rules. We could try something like this:



It is not obvious how to identify the "constituents" in a derivation like this.

(6) Context sensitive rewrite grammars can define languages that are intractable in the technical sense that they cannot be recognized in polynomial time [158, 135], and apparently non-human-like languages like $\{a^n|\ n \text{ is prime}\}$ and

$$L = \{a, aa, aaaa, aaaaaaaa, \ldots\} = \{a^{2^n} | n \geq 0\}.$$

(7) Simple 'reduplicating' languages like L_{xx} can also be defined with rules that apply to constituents regardless of context, if the constituents have more structure than strings (like trees, or tuples of strings) or if the rules can analyze and test the string components of their arguments. We will see this in the MCS grammars.

14.1 'Mildly context sensitive' MGs

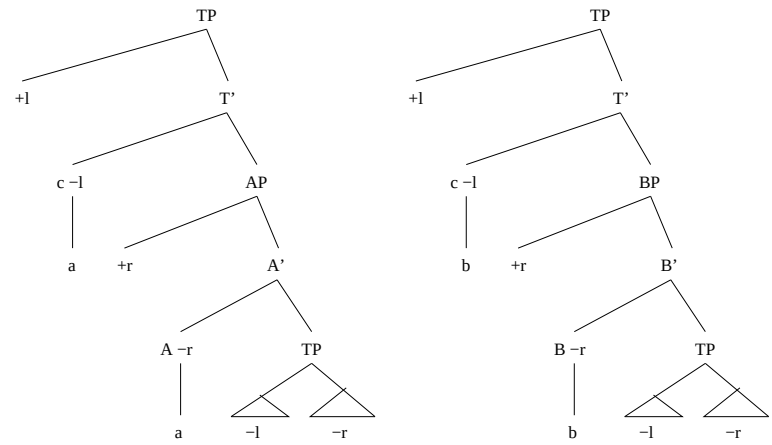
(8) To construct an MG for $L_{xx} = \{xx|\ x \in \{a,b\}^+\}$ is slightly complicated. We use movement to keep the two halves of each sentence 'synchronized' during the derivation.

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:

```
a::=A +l T -l          b::=B +l T -l b
a::=T +r A -r          b::=T +r B -r
```

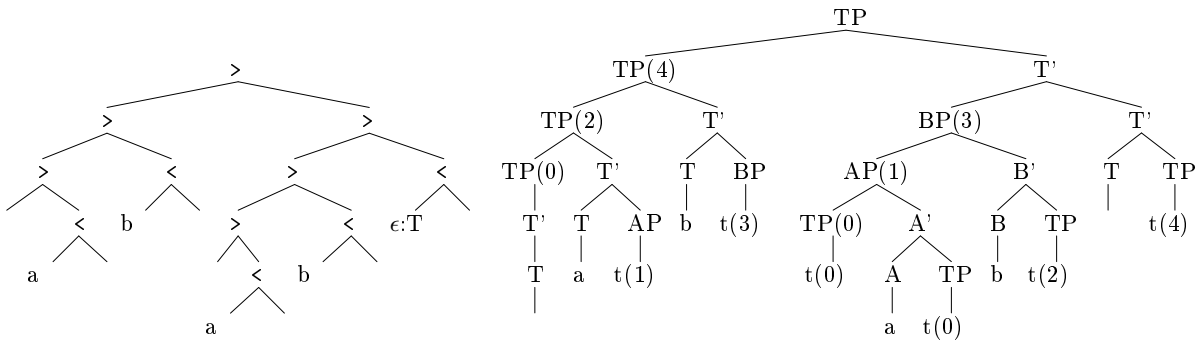
With this recursion, we only need to add the base case. Since we already have recursive structures that expect CPs to have a +r and -r, so we just need

ϵ::=T +r +l T

to finish the derivation, and to begin a derivation, we use one of these:

ϵ::T -r -l ϵ::T

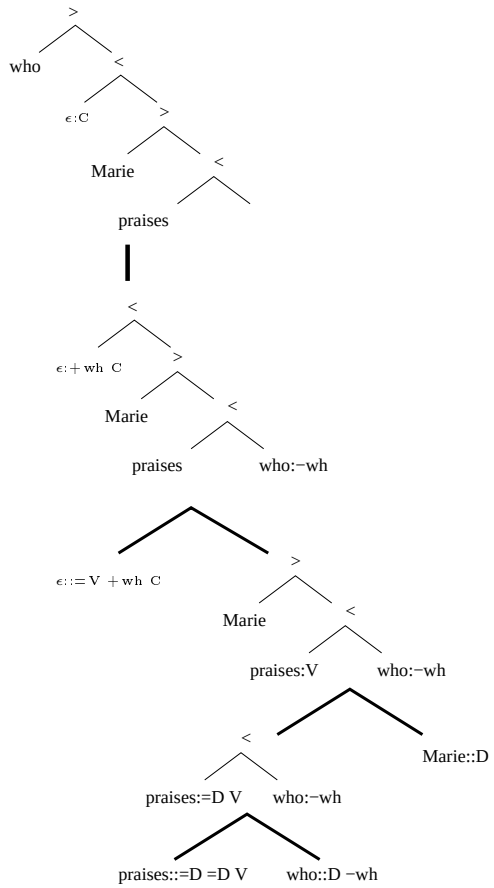
This grammar, has just 7 lexical items. (See if you can find a simpler formulation!)



14.2 MG derivations

MG = ⟨Lexicon, {merge, move}⟩	where Lexicon varies but the generating functions are fixed
structures S(G)	= closure(Lexicon, {merge,move})
completed structures	= trees $t \in S(G)$ with exactly 1 feature, the 'start' category at the head
sentences L(G)	= phonetic yields of completed structures

A standard **derivation tree** shows how any expression is derived, with lexical items at the leaves and derived elements at the internal nodes. For example, the first derivation we considered was this one:



(9) Note these properties of MG derivation trees:

a. MG derivation trees do not generally have the same shape or the same number of nodes as the derived trees that label their root nodes.

b. MG derivation trees have a binary branch for every merge, and a unary branch for every move.

(10) MGs have these properties:

a. All variation is lexical. So parameters of variation will have to be captured in the requirements of lexical items. One line of work coming from Pollock, Borer, Rizzi suggests that the parameters are the requirements of the functional categories

b. All languages underlyingly have the order spec-head-comp.

c. All movement is overt, phrasal, to a leftward and c-commanding position, with a locality condition (SMC) relativized to the nature of the movement.

d. MGs regarded as sets of lexical items are obviously closed under unions, so this allows the functional categories of different grammars to be present at once, used by the same mechanism (and so this also predicts that code-switching should be possible). It is not so clear how the idea that language involves different UG settings – switches on some control mechanism – can accomodate diglossia.

e. All structure-building is feature-driven. Does this have any consequences? (We will return to this question later)

These grammars may be 'simple' by linguistic standards, but there is a lot going on here! We can simplify these much more without changing the language defined or the structure of the derivation.

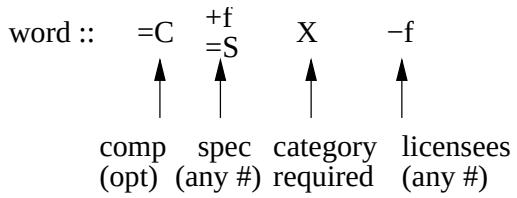
(11) Notice that MG derivations are precisely defined, so that:

a. An expression is lexical iff it has 1 node
b. An n-step completed derivation uses lexical items that have $2n + 1$ syntactic features
c. If merge(X,Y) is defined then neither move(X) nor move(Y) is
d. If X has i nodes, Y has j nodes, and merge(X,Y) is defined, then merge(X,Y) has $i + j + 1$ nodes
e. If X has i nodes, and move(X) is defined, then move(X) has $i + 2$ nodes
f. If X has i features, Y has j features, and merge(X,Y) is defined, then merge(X,Y) has $i + j - 2$ features
g. If X has i features and move(X) is defined, move(X) has $i - 2$ features

(12) Lexical items that can be used in completed derivations have certain properties:

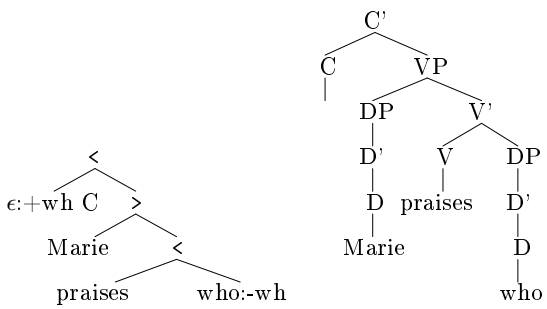
a. No completed derivation uses any lexical item whose first feature is a licensor
b. No completed derivation uses any lexical item with 0 categories
c. No completed derivation uses any lexical item with 2 categories
d. No completed derivation uses any lexical item with a selection or licensor feature following a category
e. No completed derivation uses any lexical item with a licensee feature preceding a category

Lexical items that occur in completed derivations have this structure:



left of category (selectors, licensors) defines phrase-internal requirements;
right of the category (licensees) defines phrase-external requirements. Only the category feature is always present.

14.3 What do derived structures represent?

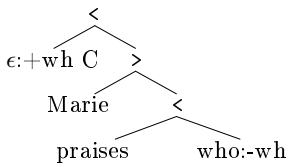


In the course of a derivation from an MG, we get to structures like the one on the left, which is much simpler than the one on the right. But still, there are many empty nodes here, and lots of branching. We want to ask: *what does the syntax need to be able to see in this structure*?

The syntax needs know (i) the features of the head, and (ii) needs to be able to find the subconstituent that is going to move.

Roughly, the syntax needs to be able to see (i) the phrases with non-empty syntactic feature sequences at their heads, and (ii) what those feature sequences are. This is rough, because the moving phrases in effect need to be movable. Let's explain this...

14.4 Trees and labeled categorizations



[$_<$ (ϵ:+wh C) Marie praises (who:-wh)]
[$_<$ [ϵ:+wh C] [$_>$Marie praises (who:-wh)]]
[$_<$ [ϵ:+wh C] [$_>$Marie [$_<$praises (who:-wh)]]]
[$_<$ [ϵ:+wh C] [$_>$[Marie] [$_<$[praises] [who:-wh]]]]

roughly: we do not need bracketing of elements with no syntactic features left

A tree can be represented by a labeled bracketing, at various levels of detail. *A tree is a structure that stores a set of categorized strings.*

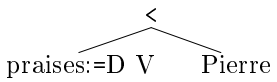
The question now is: *What does the syntax need to be able to see in this structure?*

Intuitively, 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.

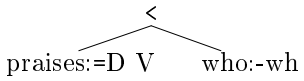
So roughly and more generally, the syntax needs to be able to see (i) the phrases with non-empty syntactic feature sequences at their heads, and (ii) what those feature sequences are. This is rough, because the moving phrases in effect need to be movable. Let's explain this...

14.5 Merge: two different cases

praises::=D =D V + Pierre::D ⇒



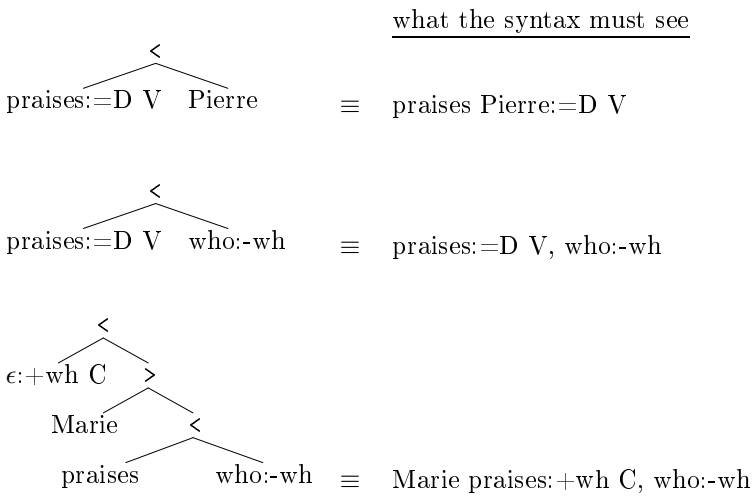
praises::=D =D V + who::D -wh ⇒



unlike *Pierre*, the DP *who* is a **mover**
second result has 2 active 'chains'

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, and its features need to be visible to the grammar.

14.6 The essentials of derived structure

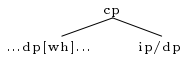


The syntax does not need the tree structure at all: each constituent can be represented by the phrase built so far, plus the moving phrases that have not yet been placed in their final positions.

In short: each **tree** can be replaced by a **tuple of categorized strings**. (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 can be 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.

14.7 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:

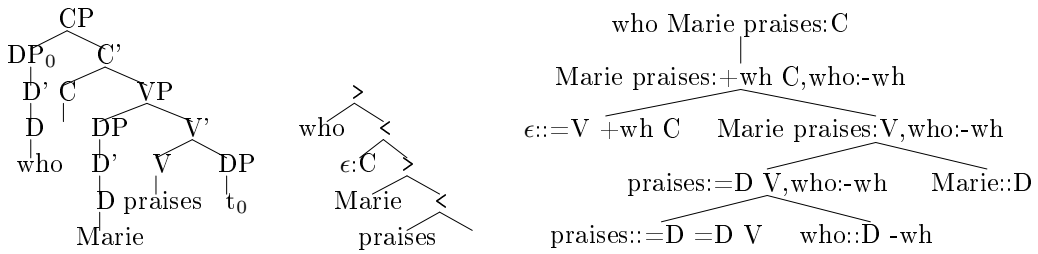
`merge(`2`,`4`)` = `praises:=D V, who:-wh` A

`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

We now have three ways of looking at what happened here:



In elementary syntax 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'. 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. Notice that this derivation tree has exactly the same shape as the derivation tree for bare trees shown on the first page. 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.

14.8 minimalist grammar on tuples G=⟨Lex,{merge,move}⟩

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

  `C, T, D, N, V, P,...` (selected categories)

  `=C, =T, =D, =N, =V, =P,...` (selector features)

  `+wh, +case, +focus,...` (licensors)

  `-wh, -case, -focus,...` (licensees)

- **chains** $C = \Sigma^* \times T \times F^*$
- **expressions** $E = C^*$
- **lexicon** $Lex \subset \Sigma^* \times \{::\} \times F^*$, a finite set

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

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

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

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

Here, $\alpha_1, \ldots, \alpha_k, \iota_1, \ldots, \iota_l$ $(0 \leq k, l)$ 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 $\alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k$ has $-f$ as its first feature,

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

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

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

structures S(G)=closure(Lex,{merge,move})
completed structures = expressions $w \cdot C$, for C the "start" category and any type $\cdot \in \{:, ::\}$
sentences L(G) $= \{w |\ w \cdot C \in S(G)$ for some $\cdot \in \{:, ::\}\}$, the strings of category C

This is the grammar presented in [249].

14.9 Example 4: a logical language

Standard propositional logics avoid structural ambiguity, rejecting

$$\neg p \wedge q$$

unless with conventions about "operator precedence." More often:

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

Or else, a Polish (prefix) notation:

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

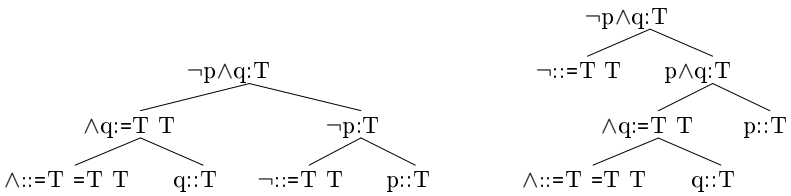
(Most good logic texts prove the unambiguity of their notation.)

Theorem [84]: standard logics (prefix form or parenthesized) are not only unambiguous, but
transparent: if any subsequence of an expression forms a complete constituent, that constituent occurs in the unique derivation of the expression.

Now let's look at an ambiguous MG grammar for propositional logic to make a different point.

Lexicon:

1	p::T	2	q::T	3	r::T
4	¬::=T T	5	∨::=T =T T	6	∧::=T =T T



Specifying the lexical items by their number, the two parses are 6241 and 4621 respectively.

'minimalist grammar' derivations

For <u>any</u> minimalist grammar G, let $\Gamma(G)$ be its completed derivation trees.

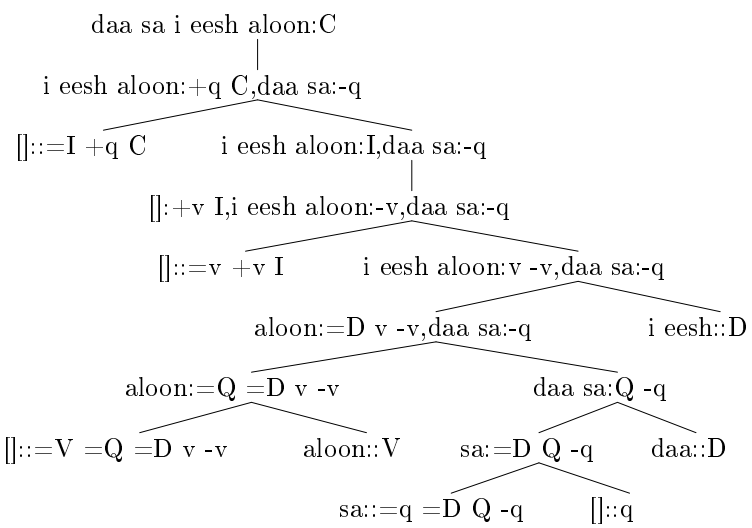
Let strings$(\Gamma(G)) \subseteq Lex^*$ be the strings of lexical items at the leaves of those trees (in order).

These languages of lexical strings are context free.

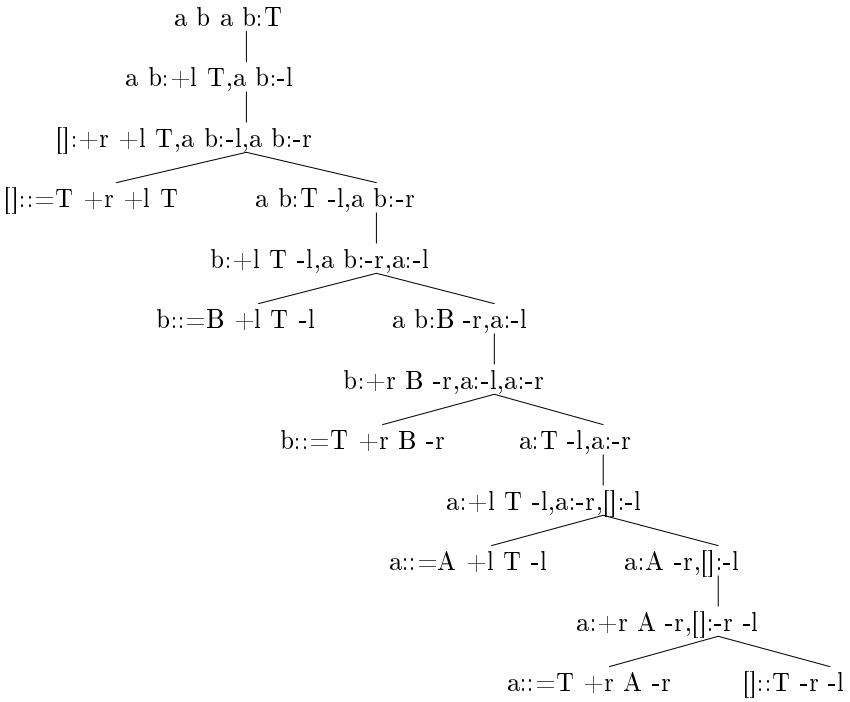
14.10 Earlier examples, reformulated

With our new, compact notation, we can provide succinct and readable presentations of some of the tricky derivations we looked at earlier.

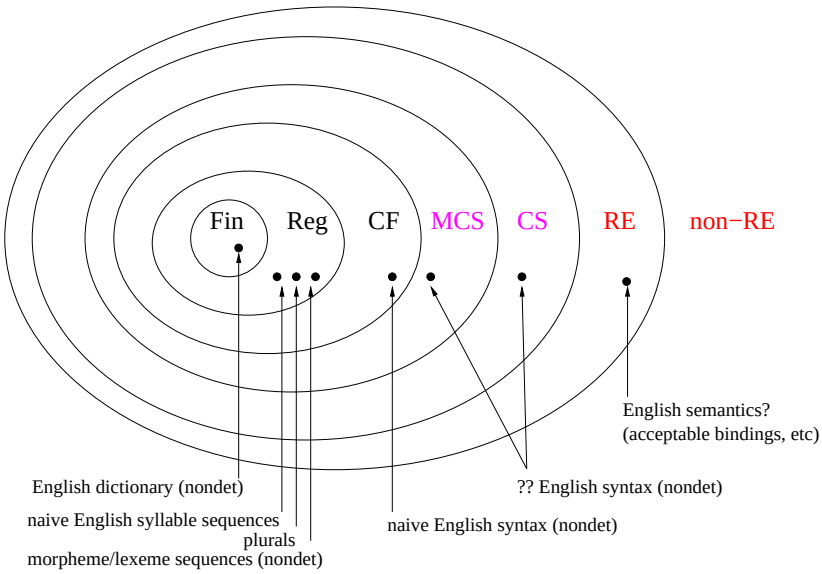
Here is the Tlingit:



Here is abab in the copy language:


a b a b:T
a b:+l T,a b:-l
[]:+r +l T,a b:-l,a b:-r
[]::=T +r +l T
a b:T -l,a b:-r
b:+l T -l,a b:-r,a:-l
b::=B +l T -l
a b:B -r,a:-l
b:+r B -r,a:-l,a:-r
b::=T +r B -r
a:T -l,a:-r
a:+l T -l,a:-r,[]:-l
a::=A +l T -l
a:A -r,[]:-l
a:+r A -r,[]:-r -l
a::=T +r A -r
[]::T -r -l

§15 Basic English syntax



15.0 Head movement

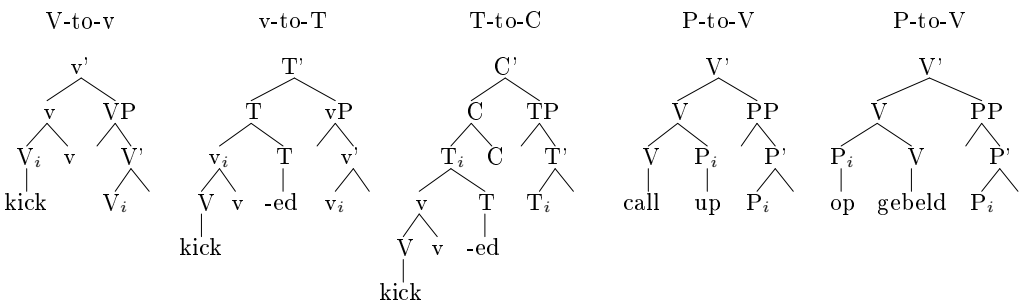
(0) **Phrasal movement:**

[which logic book] have [you been telling Mary that [you would read _]]?

(1) **Head movement:**

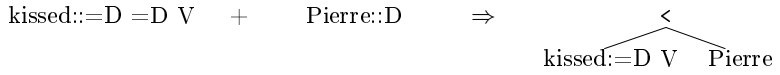
Have you _ been telling Mary you are reading it?
* Been you have _ telling Mary you are reading it?
* Are you have been telling Mary you _ reading it?

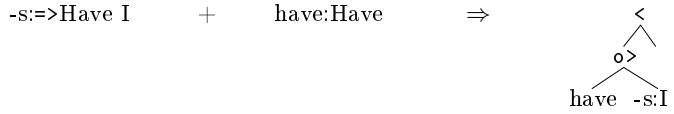
(2) **Canonical head movement:**



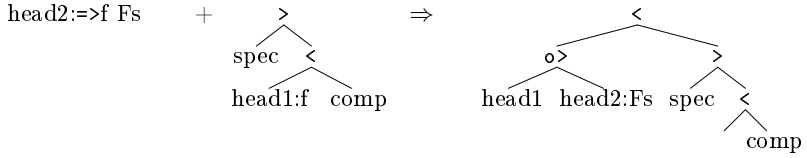
(3) **Head movement constraint:** (a strong version!)
head movement in the configuration of selection. Compare Travis [258]

(4) **Merge**:





In general:



(5) **extending the tuple style definition of merge**

replace this rule: $$\frac{s :: {=}f\gamma \qquad t \cdot f, \alpha_1, \ldots, \alpha_k}{st : \gamma, \alpha_1, \ldots, \alpha_k}\ \text{r1}$$

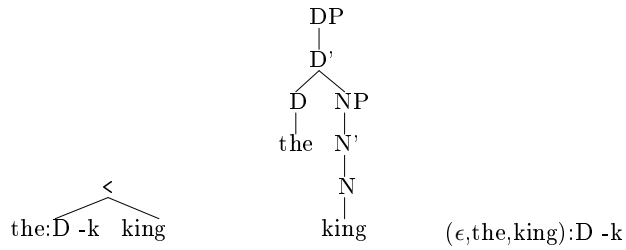
by these rules: $$\frac{(\epsilon, s, \epsilon) :: {=}f\gamma \qquad (t_s, t_h, t_c) \cdot f, \alpha_1, \ldots, \alpha_k}{(\epsilon, s, t_s t_h t_c) : \gamma, \alpha_1, \ldots, \alpha_k}\ \text{r1'}$$

$$\frac{(\epsilon, s, \epsilon) :: {=>}f\gamma \qquad (t_s, t_h, t_c) \cdot f, \alpha_1, \ldots, \alpha_k}{(\epsilon, t_h s, t_s t_c) : \gamma, \alpha_1, \ldots, \alpha_k}\ \text{r1left}$$

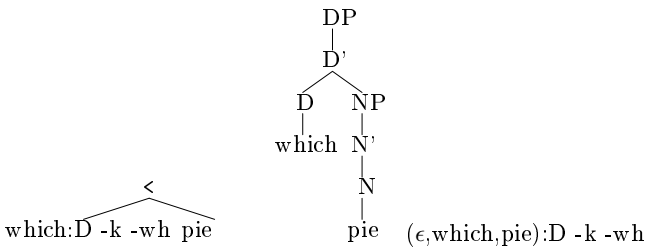
(6) **Example:**

ϵ::=T C	ϵ::=>T C	ϵ::=>T +wh C
-s::=>Have +k T	have::=Been Have	been::=ving Been
-ing::=>V =D ving	eat::=D +k V	
the::=N D -k	which::=N D -k -wh	
king::N	pie::N	

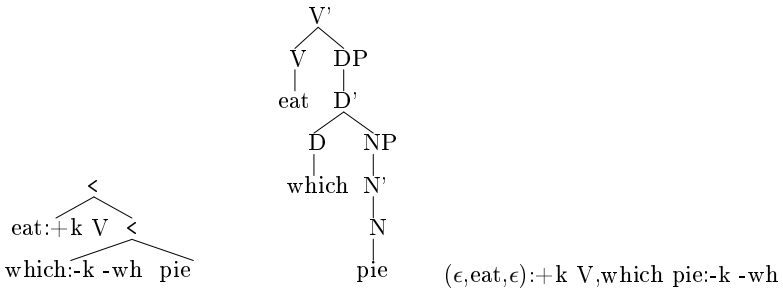
(step 1) merge



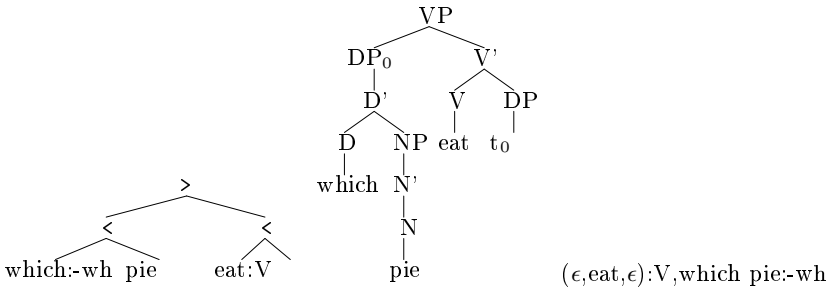
(step 2) merge



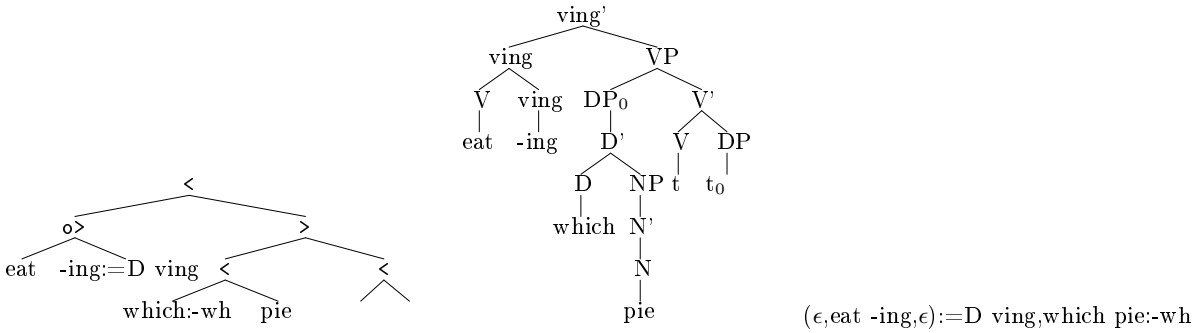
(step 3) merge



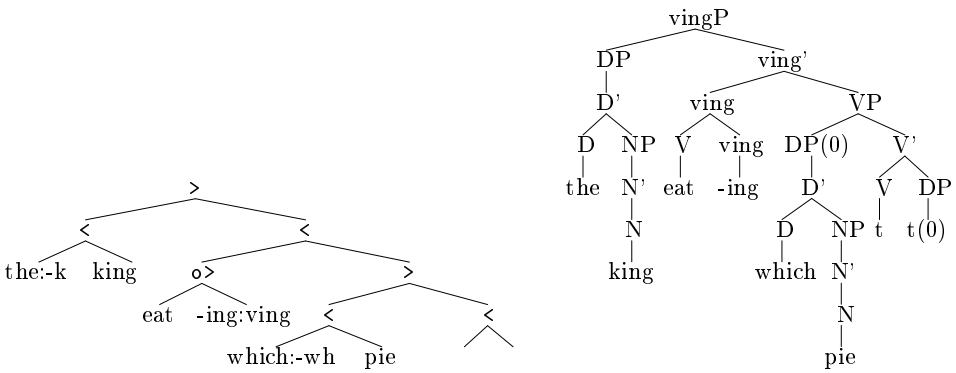
(step 4) move



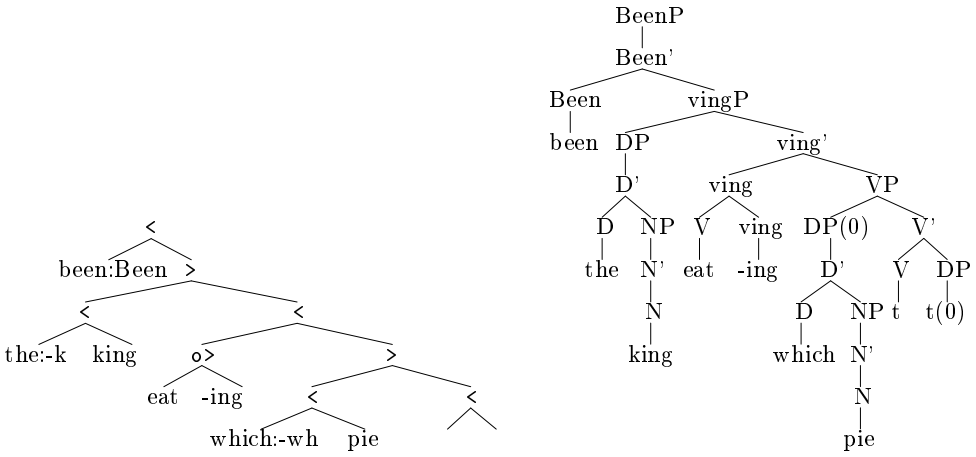
(step 5) merge



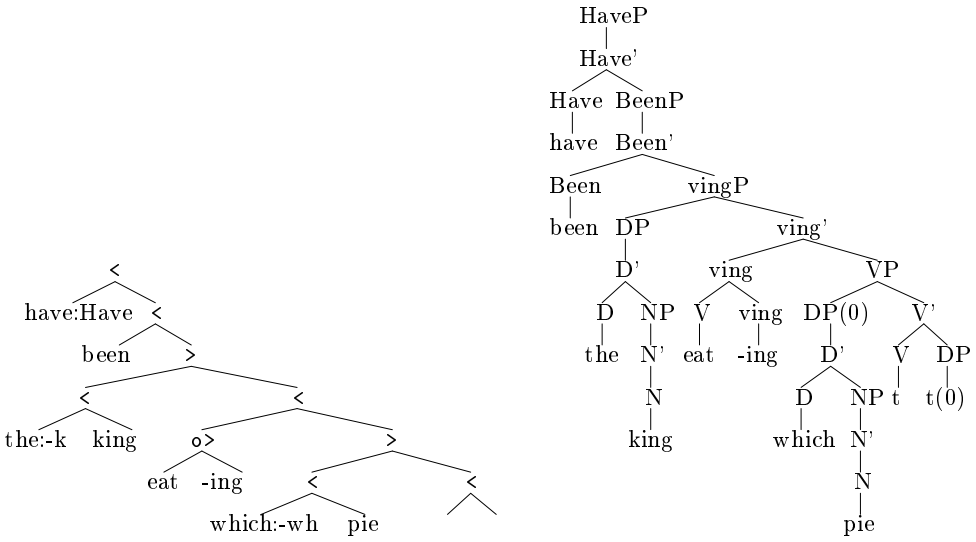
(step 6) merge



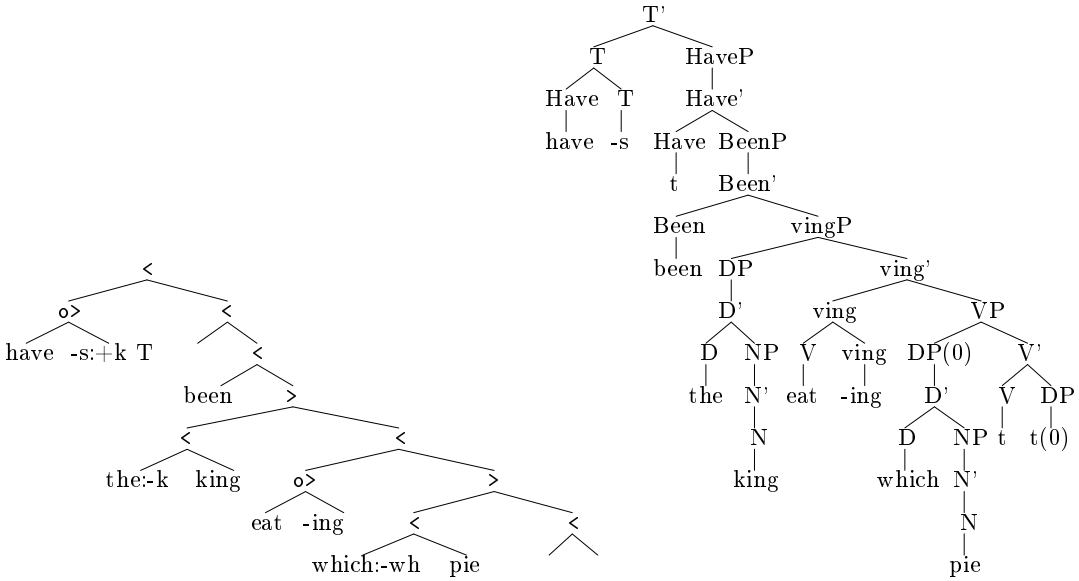
(step 7) merge



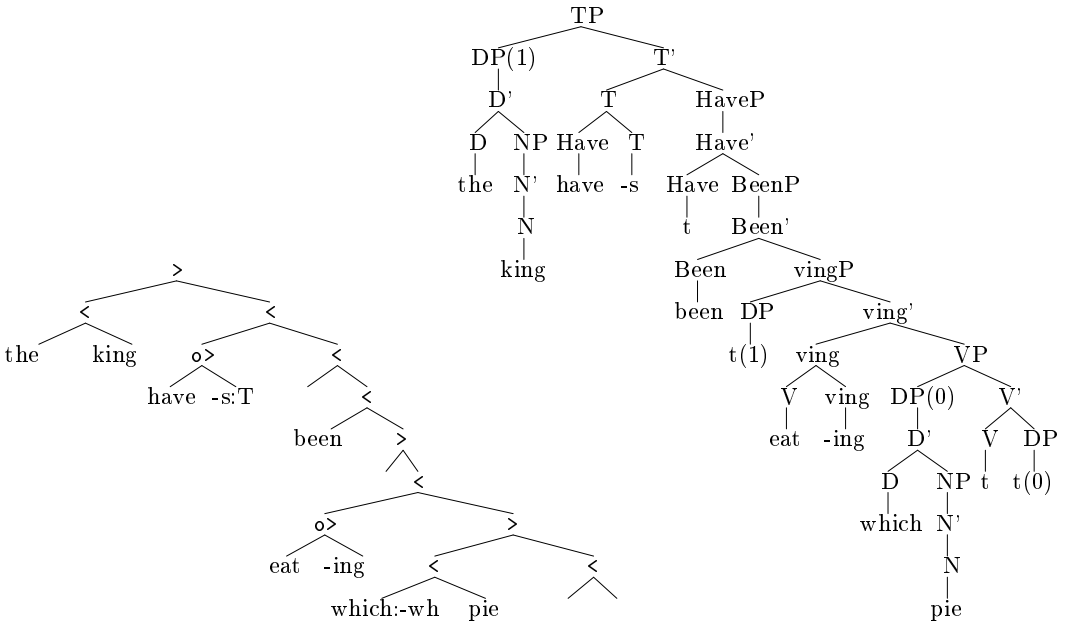
(step 8) merge



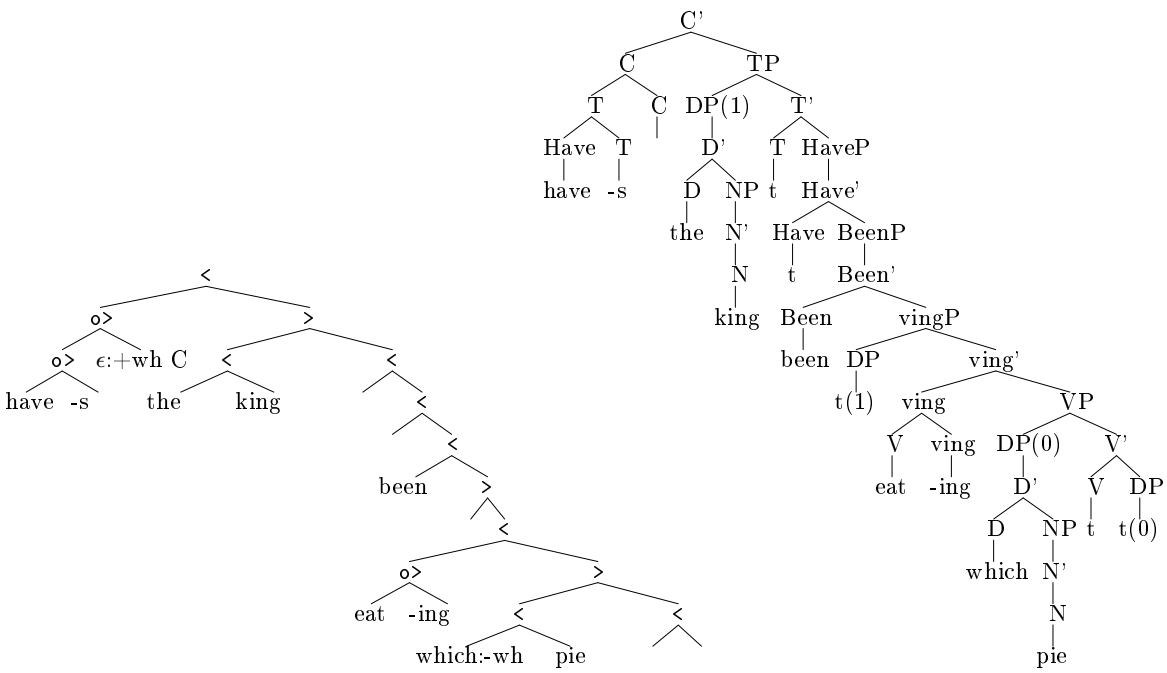
(step 9) merge



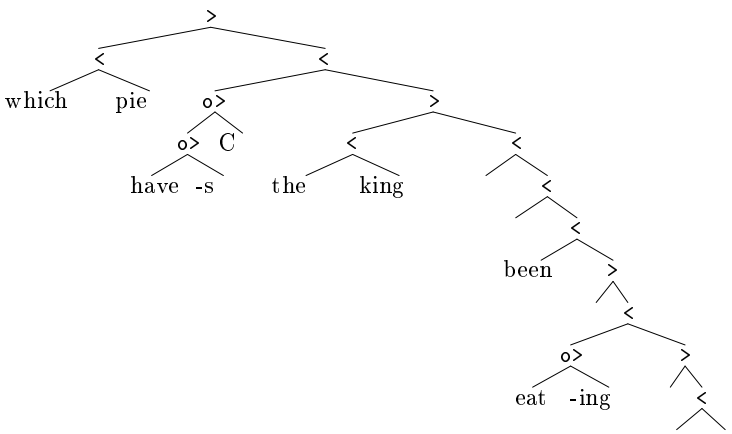
(step 10) move

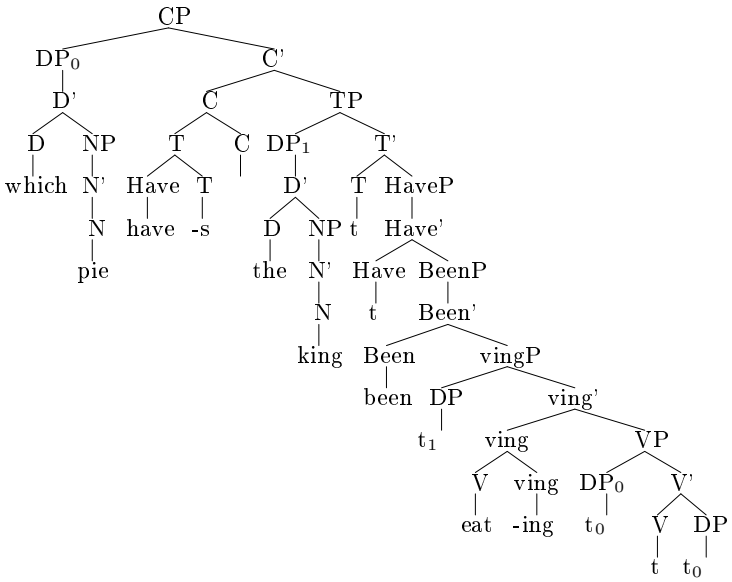


(step 11) merge



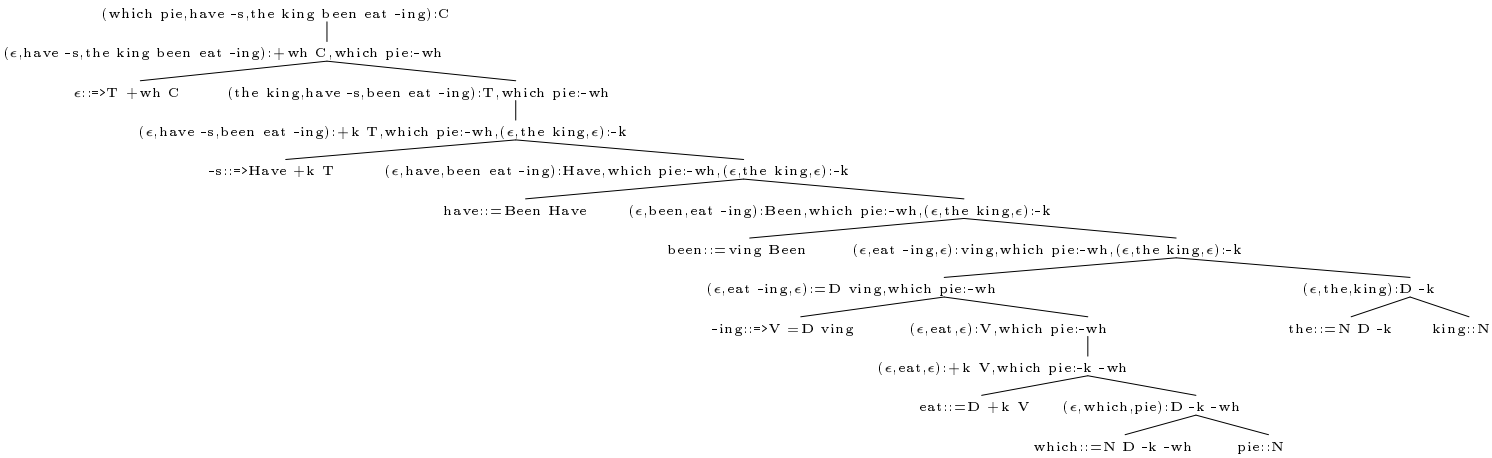
(step 12) move





(which pie,have -s,the king been eat -ing):C

the whole derivation: 12 internal nodes



(7) (Michaelis 2001, Harkema 2001):

i. MGs extended with canonical head movement define exactly the same languages as MGs without head movement, namely, exactly the MCFG languages.

ii. MGs with canonical head movement can be parsed efficiently.

(8) L(k-MG) $\subset$ L(k-MG[hm])

Proof: Easily shown that the following 0-MG[hm] grammar generates $\{xx|\ x \in \{a,b\}^*\} \notin$ 0-MG

ϵ::C	a::=>C =A C	b::=>C =B C
a::A	b::B	

(9) **Conclusions, so far:**

- MGs easily defined directly with MCFG rules
- MGs easily extended to (certain forms of) head movement, affix hop (and still MCFG equivalent)
- **Open extensions:**
  - ⋆ other versions of SMC, island conditions, economy, deletion, insertion rules
  - ⋆ performance models of human parsing, production, acquisition

15.1 Head movement and affix hopping: the details

(10) On the assumption that head movement takes place only in the configuration of selection, we regard it as part of the *merge* operation [243]. To implement this idea, the key thing is to have the phonetic contents of any movable head available in a separate component. A head X is not movable after its phrase XP has been merged, so we only need to distinguish the head components of phrases until they have been merged. So rather than expressions of the form:

$$\texttt{s}_1\cdot\texttt{Features}_1,\ \texttt{s}_2\texttt{:Features}_2,\ldots,\texttt{s}_k\texttt{:Features}_k,$$

we will use expressions in which the string part $\texttt{s}_1$ of the first chain is split into three (possibly empty) pieces `s(pecifier), h(head), c(omplement)`:

$$\texttt{(s,h,c)}\cdot\texttt{Features}_1,\ \texttt{s}_2\texttt{:Features}_2,\ldots,\texttt{s}_k\texttt{:Features}_k.$$

(11) So **lexical chains** now have a triple of strings, but only the head can be non-empty: $LC = (\epsilon, \Sigma^*, \epsilon) :: F^*$. As before, a **lexicon** is a finite set of lexical chains.

(12) As discussed above, head movement is triggered by a specialization of the selecting feature.

The feature =>V will indicate that the head of the selected VP is to be adjoined on the left.

The feature V<= will indicate that the head of the selected VP is to be adjoined on the right.

The former set of features is thus extended by adding these two new functions on the base categories B: **right-incorporators** $R = \{f\texttt{<=}|\ f \in B\}$, and **left-incorporators** $L = \{\texttt{=>}f|\ f \in B\}$.

So now the set of syntactic features $F = B \cup S \cup M \cup N \cup R \cup L$. The new work of placing heads properly is done by the *merge* function, so the earlier functions r1 and r3 each break into 3 cases.

(13) Define *merge* as the union of the following 7 functions:

$$\frac{(\epsilon, s, \epsilon) :: \texttt{=}f\gamma \qquad (t_s, t_h, t_c)\cdot f, \alpha_1, \ldots, \alpha_k}{(\epsilon, s, t_s t_h t_c) : \gamma, \alpha_1, \ldots, \alpha_k}\ \text{r1'}$$

$$\frac{(\epsilon, s, \epsilon) :: f\texttt{<=}\gamma \qquad (t_s, t_h, t_c)\cdot f, \alpha_1, \ldots, \alpha_k}{(\epsilon, s t_h, t_s t_c) : \gamma, \alpha_1, \ldots, \alpha_k}\ \text{r1right}$$

$$\frac{(\epsilon, s, \epsilon) :: \texttt{=>}f\gamma \qquad (t_s, t_h, t_c)\cdot f, \alpha_1, \ldots, \alpha_k}{(\epsilon, t_h s, t_s t_c) : \gamma, \alpha_1, \ldots, \alpha_k}\ \text{r1left}$$

$$\frac{(s_s, s_h, s_c) : \texttt{=}f\gamma, \alpha_1, \ldots, \alpha_k \qquad (t_s, t_h, t_c)\cdot f, \iota_1, \ldots, \iota_l}{(t_s t_h t_c s_s, s_h, s_c) : \gamma, \alpha_1, \ldots, \alpha_k, \iota_1, \ldots, \iota_l}\ \text{r2'}$$

$$\frac{(s_s, s_h, s_c) \cdot \texttt{=}f\gamma, \alpha_1, \ldots, \alpha_k \qquad (t_s, t_h, t_c) \cdot f\delta, \iota_1, \ldots, \iota_l}{(s_s, s_h, s_c) : \gamma, t_s t_h t_c : \delta, \alpha_1, \ldots, \alpha_k, \iota_1, \ldots, \iota_l} \text{r3'}$$

$$\frac{(s_s, s_h, s_c) :: f\texttt{<=}\gamma, \alpha_1, \ldots, \alpha_k \qquad (t_s, t_h, t_c) \cdot f\delta, \iota_1, \ldots, \iota_l}{(s_s, s_h t_h, s_c) : \gamma, t_s t_c : \delta, \alpha_1, \ldots, \alpha_k, \iota_1, \ldots, \iota_l} \text{r3right}$$

$$\frac{(s_s, s_h, s_c) :: \texttt{=>}f\gamma, \alpha_1, \ldots, \alpha_k \qquad (t_s, t_h, t_c) \cdot f\delta, \iota_1, \ldots, \iota_l}{(s_s, t_h s_h, s_c) : \gamma, t_s t_c : \delta, \alpha_1, \ldots, \alpha_k, \iota_1, \ldots, \iota_l} \text{r3left}$$

(14) And *move* changes only trivially. It is the union of the following functions:

$$\frac{(s_s, s_h, s_c) : +f\gamma, \alpha_1, \ldots, \alpha_{i-1}, t : -f, \alpha_{i+1}, \ldots, \alpha_k}{(ts_s, s_h, s_c) : \gamma, \alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k} \text{m1'}$$

$$\frac{(s_s, s_h, s_c) : +f\gamma, \alpha_1, \ldots, \alpha_{i-1}, t : -f\delta, \alpha_{i+1}, \ldots, \alpha_k}{(s_s, s_h, s_c) : \gamma, \alpha_1, \ldots, \alpha_{i-1}, t : \delta, \alpha_{i+1}, \ldots, \alpha_k} \text{m2'}$$

(15) The grammar above does not derive the simple tensed clause: *the king eat -s the pie*. The problem is that if we simply allow the verb *eat* to pick up this inflection by head movement to T, as the auxiliary verbs do, then we will mistakenly also derive **eat -s the king the pie*. Also, assuming that *will* fills T, there are VP modifiers that can follow T

He will completely solve the problem.

So if the verb moves to the T affix -*s*, we would expect to find it before such a modifier, which is not what we find:

He completely solve -s the problem.
* He solve -s completely the problem.

Since Chomsky 1957 [43], one common proposal about this is that when there is no auxiliary verb, the inflection can lower to the main verb. This lowering is sometimes called "affix hopping." In the present context, it is interesting to notice that once the head of unmerged phrases is distinguished for head movement, no further components are required for affix hopping.

(16) We can formalize this idea in our grammars as follows. We introduce two new kinds of features $\texttt{<=}f$ and $f\texttt{=>}$ (for any $f \in B$), and we add the following additional cases to definition of *merge*:

$$\frac{(\epsilon, s, \epsilon) :: f\texttt{=>}\gamma \qquad (t_s, t_h, t_c) \cdot f, \alpha_1, \ldots, \alpha_k}{(\epsilon, \epsilon, t_s t_h s t_c) : \gamma, \alpha_1, \ldots, \alpha_k} \text{r1hopright}$$

$$\frac{(\epsilon, s, \epsilon) :: \texttt{<=}f\gamma \qquad (t_s, t_h, t_c) \cdot f, \alpha_1, \ldots, \alpha_k}{(\epsilon, \epsilon, t_s s t_h t_c) : \gamma, \alpha_1, \ldots, \alpha_k} \text{r1hopleft}$$

$$\frac{(\epsilon, s, \epsilon) :: f\texttt{=>}\gamma, \alpha_1, \ldots, \alpha_k \qquad (t_s, t_h, t_c) \cdot f\delta, \iota_1, \ldots, \iota_l}{(\epsilon, \epsilon, \epsilon) : \gamma, t_s t_h s t_c : \delta, \alpha_1, \ldots, \alpha_k, \iota_1, \ldots, \iota_l} \text{r3hopright}$$

$$\frac{(\epsilon, s, \epsilon) :: \texttt{<=}f\gamma, \alpha_1, \ldots, \alpha_k \qquad (t_s, t_h, t_c) \cdot f\delta, \iota_1, \ldots, \iota_l}{(\epsilon, \epsilon, \epsilon) : \gamma, t_s s t_h t_c : \delta, \alpha_1, \ldots, \alpha_k, \iota_1, \ldots, \iota_l} \text{r3hopleft}$$

This formulation of affix-hopping as a sort of string-inverse of head movement has the consequence that an affix can only "hop" to the head of a selected phrase, not to the head of the head selected by a selected phrase. That is, affix hopping can only take place in the configuration of selection.[1] It is now a simple matter to obtain a grammar G2 that gets simple inflected clauses.

[1] [242, 382] points out that the proposal in [49] for avoiding affix hopping also has the consequence that affixes on main verbs in English can only occur in the configuration where head movement would also have been possible.

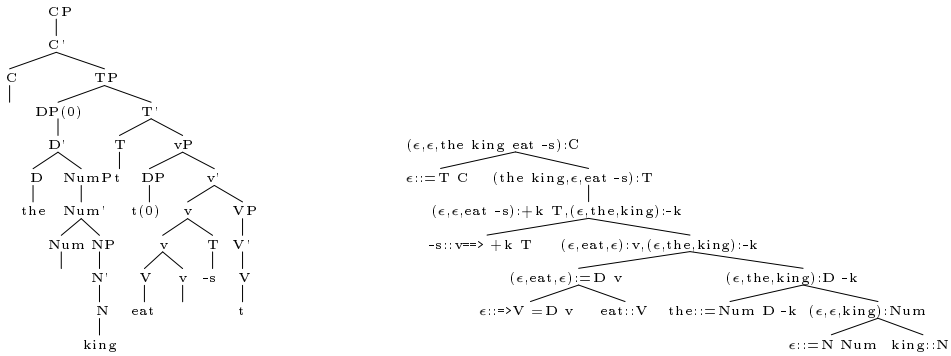
(17) We elaborate grammar G1 by adding a single lexical item:

```
-s::  v=> +k T
```

It is left as an exercise to verify that the set of strings of category `C` now allows main verbs to be inflected but not fronted, as desired:

a. the king eat -s the pie

b. *eat -s the king the pie



This kind of account of English clause structure commonly adds one more ingredient: do-support. Introductory texts sometimes propose that *do* can be attached to any stranded affix, perhaps by a process that is not part of the syntax proper. We return to this later.

15.2 Example: simple English

Here is the full grammar used to generate the examples at the beginning of this section:

ϵ:: =T C	ϵ:: =>T C	ϵ:: =>T +wh C	
-s:: =>Modal +k T	-s:: =>Have +k T	-s:: =>Be +k T	-s:: =v +k T
will:: =Have Modal	will:: =Be Modal	will:: =v Modal	
have:: =Been Have	have:: =v_{en} Have		
be:: =v_{ing} Be	been:: =v_{ing} Been		
ϵ:: =>V =D v	-en:: =>V =D v_{en}	-ing:: =>V =D v_{ing}	
eat:: =D +k V	laugh:: V		
the:: =N D -k	which:: =N D -k -wh		
king:: N	pie:: N		
-s:: v=> +k T			

The behavior of this grammar is English-like on a range of constructions:

(18) will -s the king laugh

(19) the king be -s laugh -ing

(20) which king have -s eat -en the pie

(21) the king will -s have been eat -ing the pie

(22) will -s the king have been eat -ing the pie

(23) * have the king will -s been eat -ing the pie

(24) * been the king will -s have eat -ing the pie

(25) * eat -ing the king will -s have been the pie

(26) the king laugh -s

(27) * laugh -s the king

We also derive

(28) -s the king laugh

This string is sometimes said to trigger "do-support," which will be discussed later.

15.3 Example: French clitics

Sportiche [242] reviews some of the difficulties in providing an account of French clitics. Following Kayne [139], Zwicky [278] and many others, he points out that they act like heads attached to a verb when, for example, they move with the verb in "complex inversion" (just as the verb and inflection move together in English questions, in the previous section):

(29) Jean [l'aurait]$_i$-il t_i connu?
John him-would-have-he known

'Would John have known him?'

But clitics are also related to argument positions, and these need not be adjacent to the verb:

(30) Il lui$_i$ donnera le chapeau t_i
He to-him will-give the hat

'He will give him the hat'

A clitic can be associated with the argument position of a verb other than the one it is attached to. In Italian, Spanish, Middle French, and perhaps some contemporary (but non-standard) dialects of French, one finds constructions like this:

(31) Jean la$_i$ veut manger t_i
John it wants to-eat

'John wants to eat it'

And finally, a verb can be associated with multiple clitics:

(32) Il le$_j$ lui$_i$ donnera t_j t_i
he it to-him will-give

'he will give it to him'

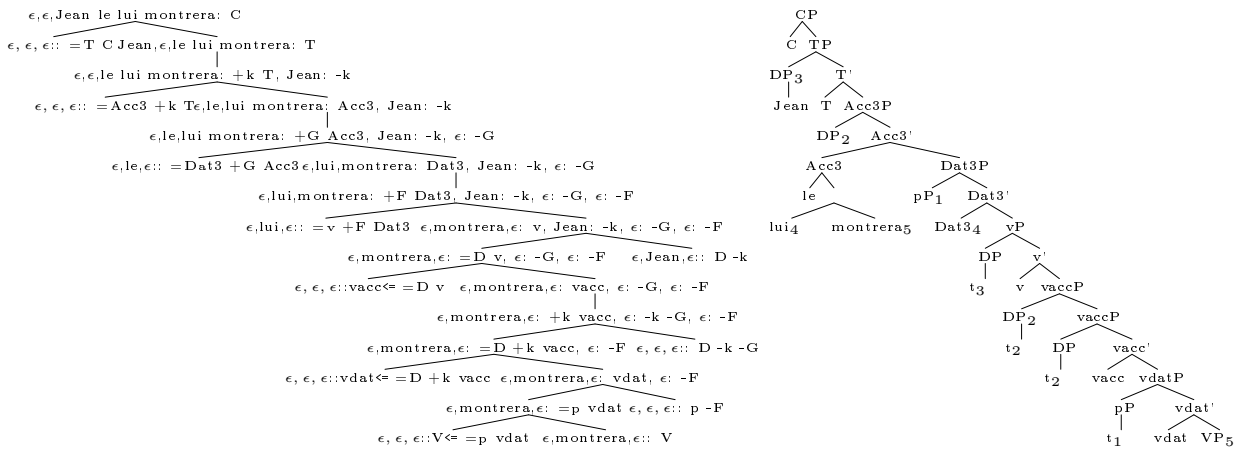
However, there is a limit on the number of clitics that can be associated with a verb in French. They are limited to at most one from each of the following sets, and at most one from the third and fifth sets together [242, 247]:

Nom	Neg	Refl12	Acc3	Dat3	Loc	Gen
{il}	{ne}	{me,te,se,nous}	{le,la,les}	{lui,leur}	{y}	{en}

One way to get the proper order of these clitics that is considered by Sportiche [241, 266ff] involves assuming that over V there is a projection for each of these sets. Then a Dat3 head can select V to form [lui V] by (right adjoining) head movement, which in turn can be selected by Acc3 to form [le lui V], and so on. And the association with argument positions can be accomplished by moving not the clitics themselves, but phrases ("operators," phonetically empty), so that the argument positions are properly "bound" by the clitics. If the binding of arguments by clitics in the 3rd and 5th set is accomplished by the same movement feature +F, then the (SMC) will prevent both from occurring at once, as desired. This approach is captured by a grammar like the following:

ϵ::=T C				
ϵ::=Refl12 +k T	ϵ::=Acc3 +k T	ϵ::=Dat3 +k T	ϵ::=v +k T	
se::=Acc3 +F Refl12	se::=Dat3 +F Refl12	se::=v +F Refl12		
le::=Dat3 +G Acc3	le::=v +G Acc3			
lui::=v +F Dat3				
ϵ::vacc<= =D v	ϵ::vdat<= =D +k vacc	ϵ::V<= =p vdat	montrera::V	
ϵ::P<= p	a::=D +k P	ϵ::p -F		
Jean::D -k	Marie::D -k	le::=N D -k	ϵ::D -k -F	ϵ::D -k -G
roi::N	livre::N			

With this grammar, we have derivations like the following, with the more conventional transformational depiction on the right. Notice that an element has moved from the lower P specifier of the vdatP argument of the verb to the Dat3P clitic phrase:



It is easy to check that with this grammar we also have, as desired:

(33) Jean montrera le livre à Marie

(34) Jean se montrera à Marie

(35) *Jean se lui montrera

This simple syntax of clitic constructions leaves out agreement and many other important phenomena, but many of the more sophisticated recent proposals similarly mix phrasal and head movement. Agreement is carefully considered in [242], and interesting accounts of stylistic inversion and subject clitic inversion in French are provided in [142, 208, 206]. It appears that all these fall easily in the scope of the mechanisms proposed here.

Maxime Amblard [9] has provided a semantics for these constructions, and points out various constructions that this analysis does not handle:

(36) Jean la laisse le lui donner
Jean her let it him to-give

15.4 Reflections

- This collection of rules is fairly complicated! And it is not complete yet, as we will see.
- How many rules are there here? Since they have disjoint domains we can take their union and get just one. Keenan & Stabler notice that taking unions like this is motivated when it increases the number of automorphisms, but I conjecture that this union won't.

  Would it look more like 1 rule if the move rule used copying? No.
- Chomsky says:

  > Unless some stipulation is added, there are two subcases of the operation Merge. Given A, we can merge B to it from outside A or from within A; these are external and internal Merge, the latter the operation called "Move," which therefore also "comes free," yielding the familiar displacement property of language. [52, p.12]

  At least, it is true that (external) merge and move (=internal merge) have structural similarities.

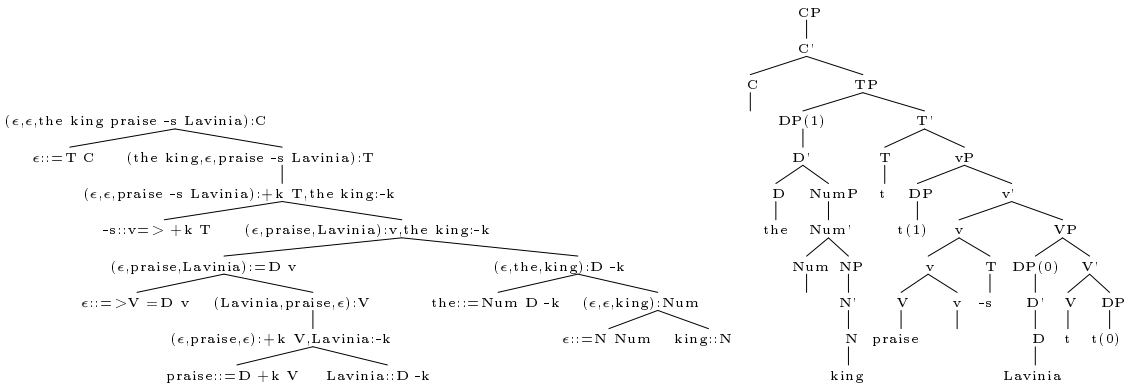
15.5 Verb classes and other basics

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:

praise::=D +k V ϵ::=>V =D v

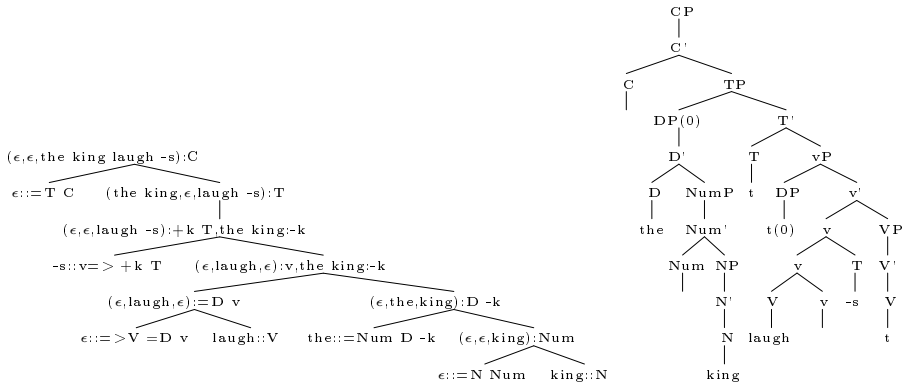
Here we see that the V selects a DP and then moves it to assign (accusative) case, forming a VP. This VP is then selected by v and the head V is left adjoined to the head v by head movement, and then the subject (the "external argument") of the verb is selected.



In contrast, an **intransitive verb** has a simpler lexical entry like this:

laugh::V

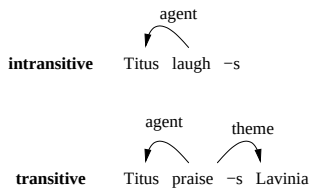
This verb selects no object and assigns no case, but it combines with v to get its subject in the usual way.



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.

15.5.1 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:

(37) * That Titus laughs

(38) Titus thinks that Lavinia laughs

(39) * Titus thinks which king Lavinia praises

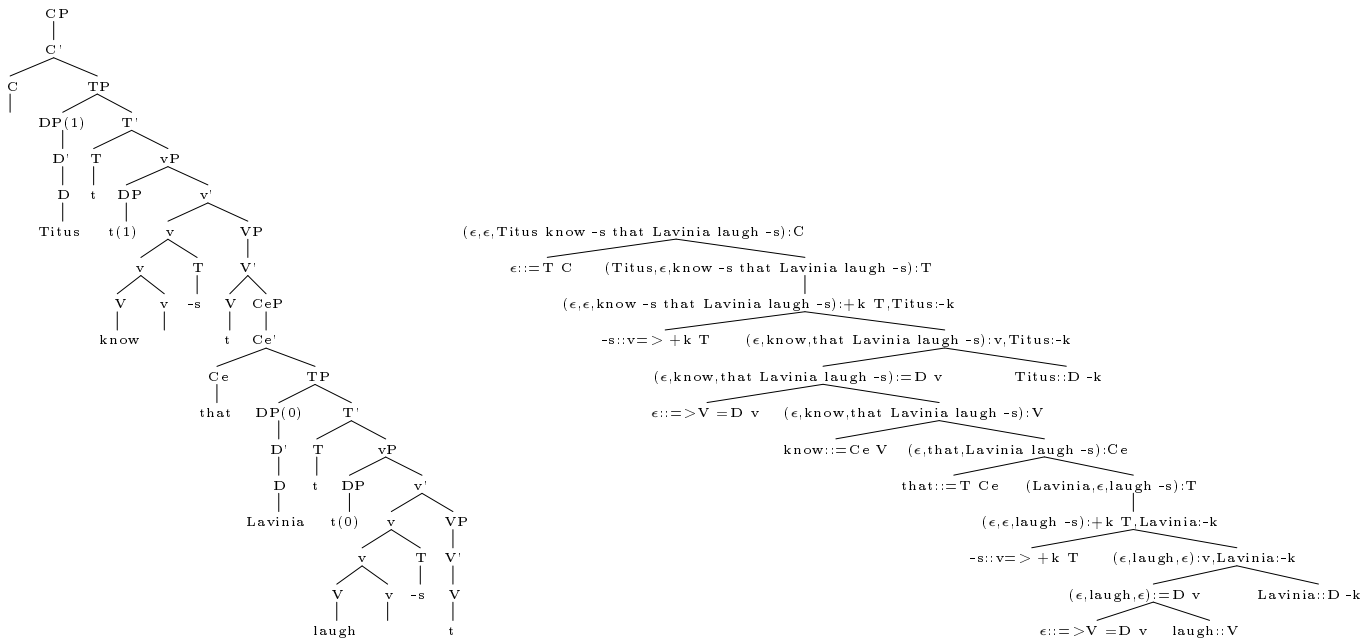
(40) * Titus wonders that Lavinia laughs

(41) Titus wonders which king Lavinia praises

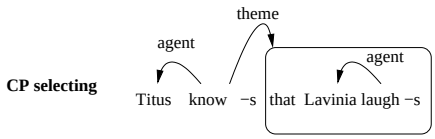
Verbs like *know* select both kinds of complements and can also occur in transitive and intransitive constructions. We can get these distinctions with lexical entries like this:

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):



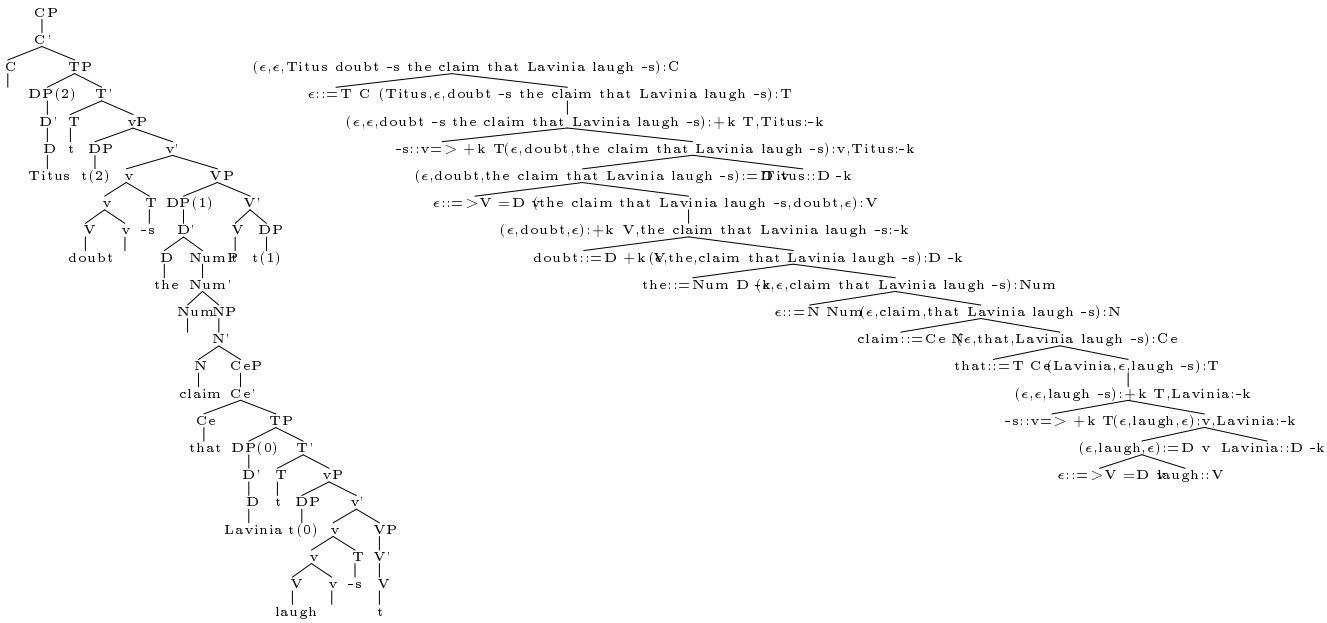
Semantically, the picture corresponds to the derivation as desired:



We can also add nouns that select clausal complements:

claim::=Ce N proposition::=Ce N

With these lexical entries we get trees like this:



15.5.2 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.

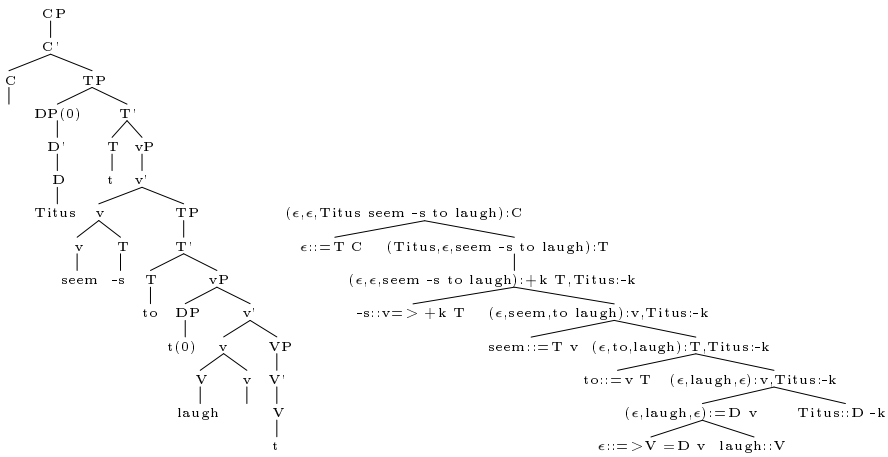
A simple version of this idea is implemented by the following lexical item for the raising verb *seem*

seem::=T v

and by the following lexical items for the infinitival *to*:

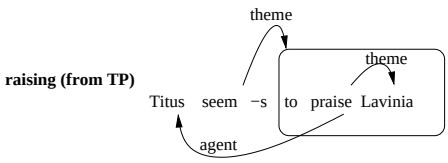
to::=v T to::=Have T to::=Be T

With these lexical entries, we get derivations like this:

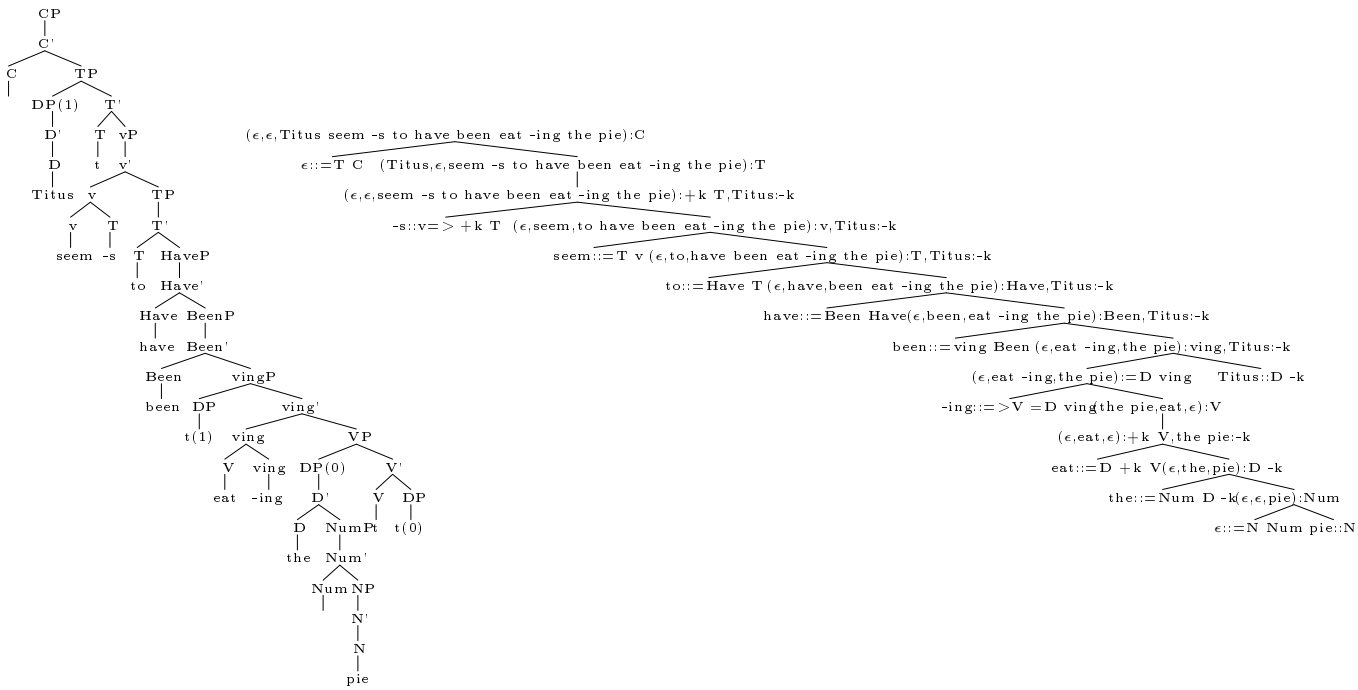


Notice that the subject of *laugh* cannot get case in the infinitival clause, so it moves to the higher clause. In this kind of construction, the main clause subject is not selected by the main clause verb!

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:

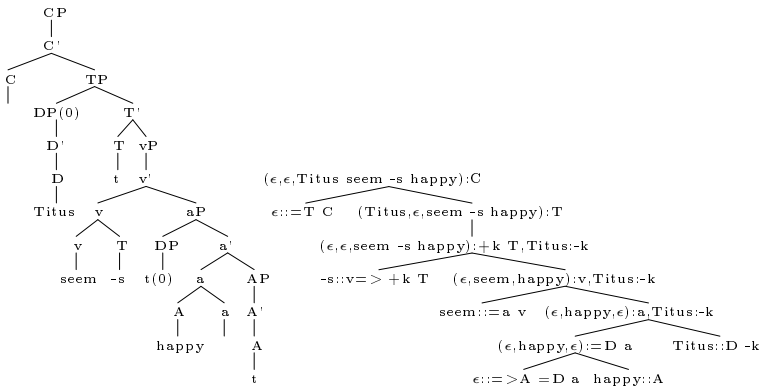


15.5.3 AP-selecting raising verbs

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:



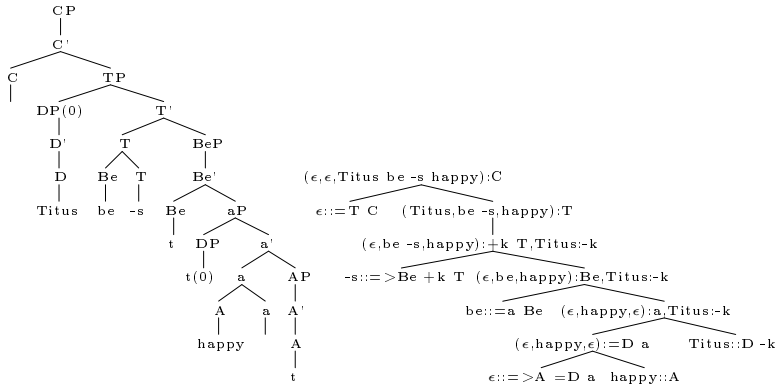
We obtain this derivation with these lexical items:

ϵ::=>A =D a. black::A	white::A
happy::A	unhappy::A
seem::=a v	

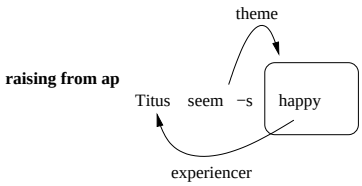
The verb *be* needs a similar lexical entry

be::=a Be

to allow for structures like this:



Semantically, the picture corresponds to the derivation as desired:



15.5.4 AP small clause selecting verbs, raising to object

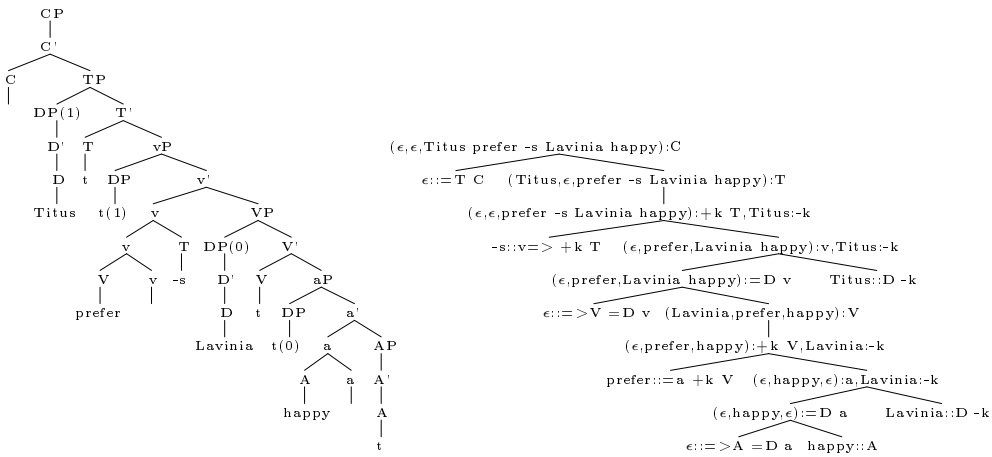
We get some confirmation for the analyses above from so-called "small clause" constructions like:

Titus considers Lavinia happy
He prefers his coffee black
He prefers his shirts white

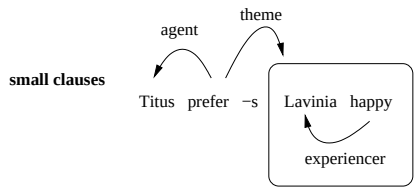
The trick is to allow for the embedded object to get case. One hypothesis is that this object gets case from the governing verb. A simple version of this idea is implemented by the following lexical items:

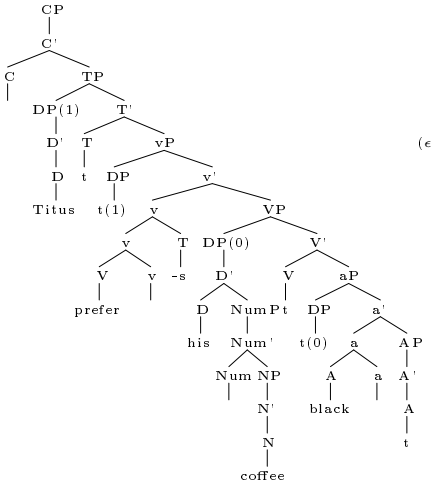
prefer::=a +k V	prefer::=T +k V
consider::=a +k V	consider::=T +k V

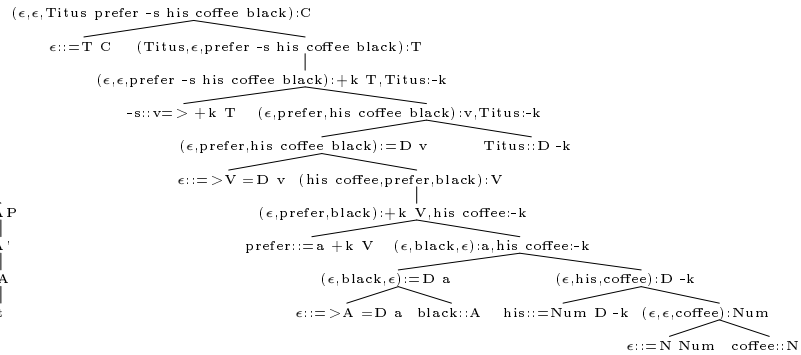
With these lexical items, we get derivations like this:

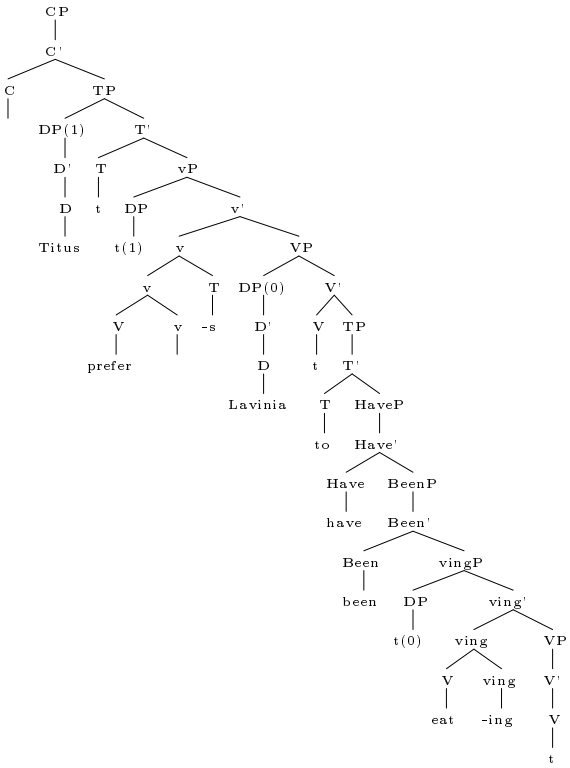


Semantically, the picture corresponds to the derivation as desired:









15.5.5 PP-selecting verbs, adjectives and nouns

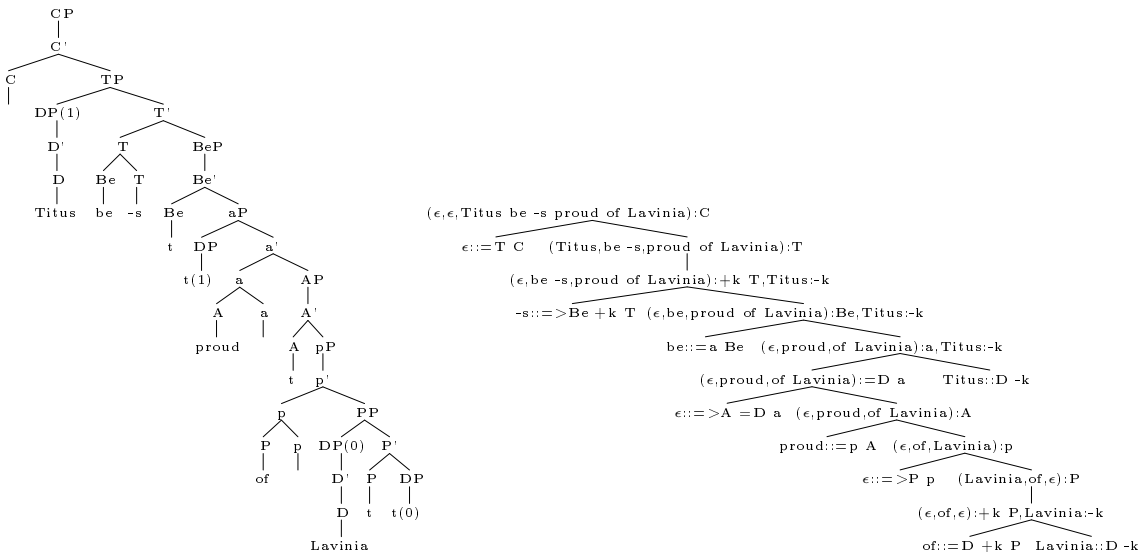
We have seen adjective phrases with subjects, so we should at least take a quick look at adjective phrases with complements. We first consider examples like this:

Titus is proud of Lavinia
Titus is proud about it

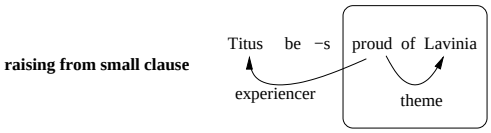
We adopt lexical items which make prepositional items similar to verb phrases, with a “little” p and a “big” P:

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:



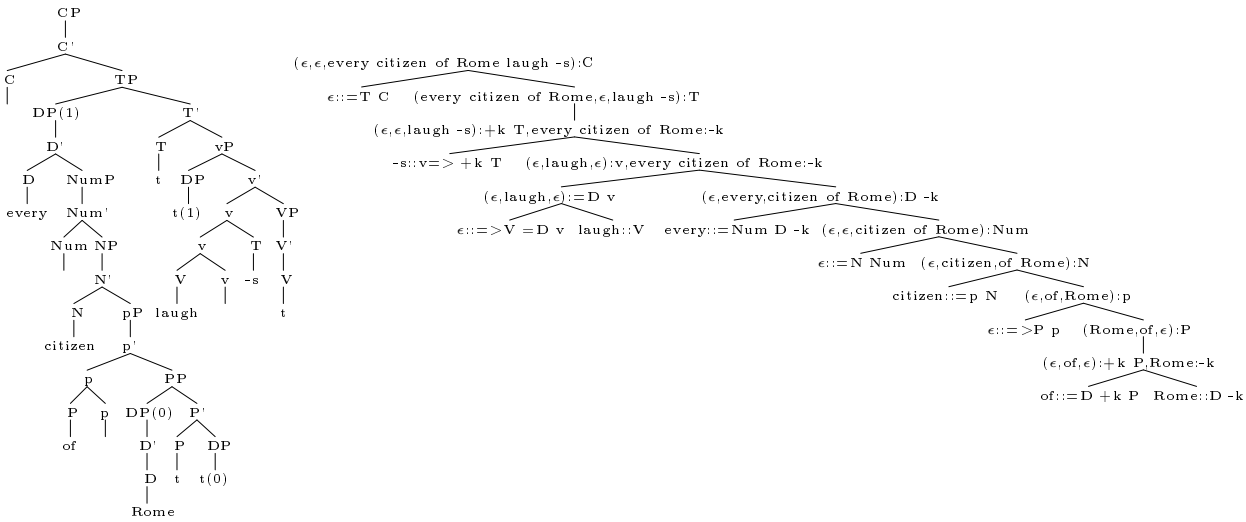
Semantically, the picture corresponds to the derivation as desired:

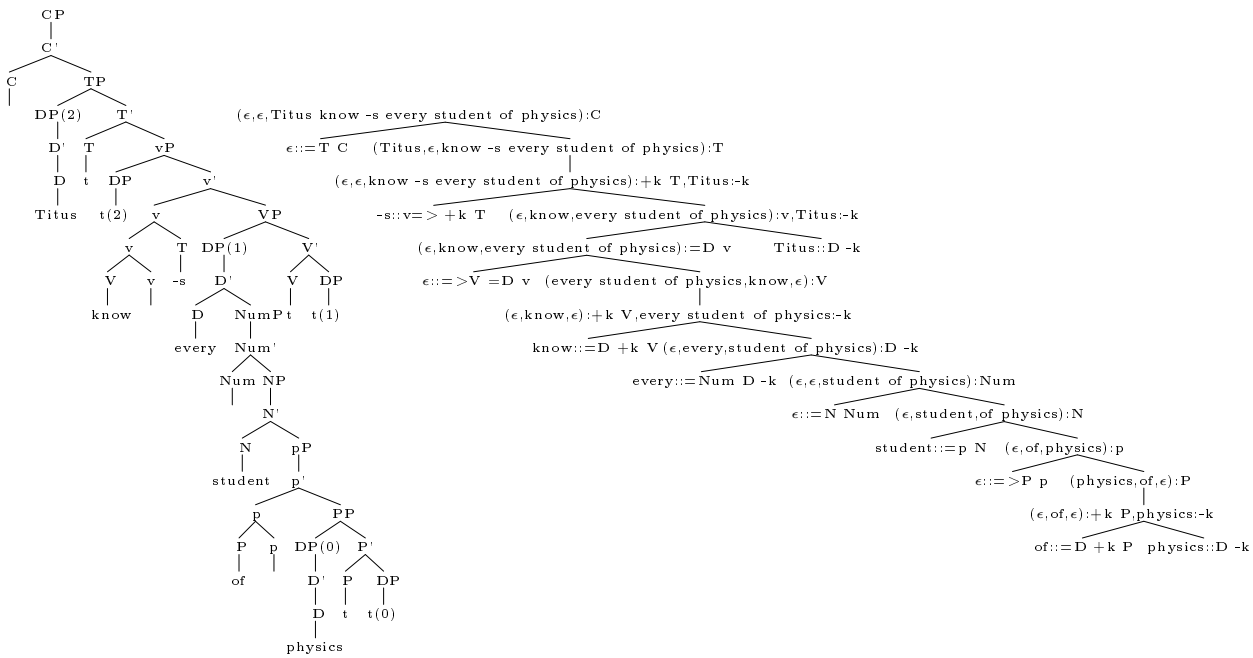


Similarly, we allow certain nouns to have PP complements, when they specify the object of an action or some other similarly constitutive relation:

student::=p N	student::N	physics::D -k
citizen::=p N	citizen::N	Rome::D -k

to get constructions like this:

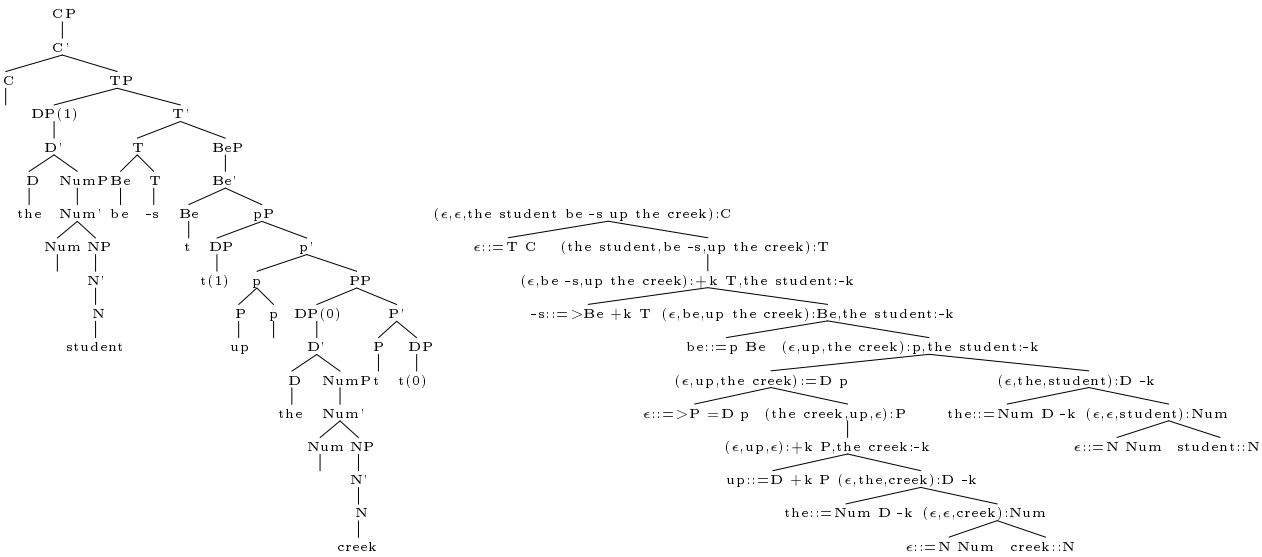




If we add lexical items like the following:

be::=p Be	seem::=p v
ϵ::=>P =D p	up::=D +k P
creek::N	

then we get derivations like this:



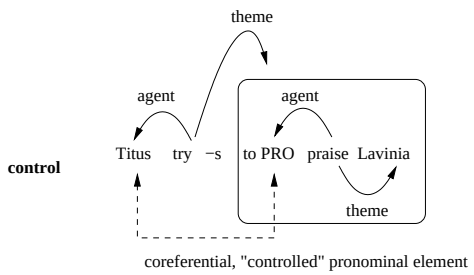
15.5.6 Control verbs

There is another pattern of semantic relations that is actually more common that 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:

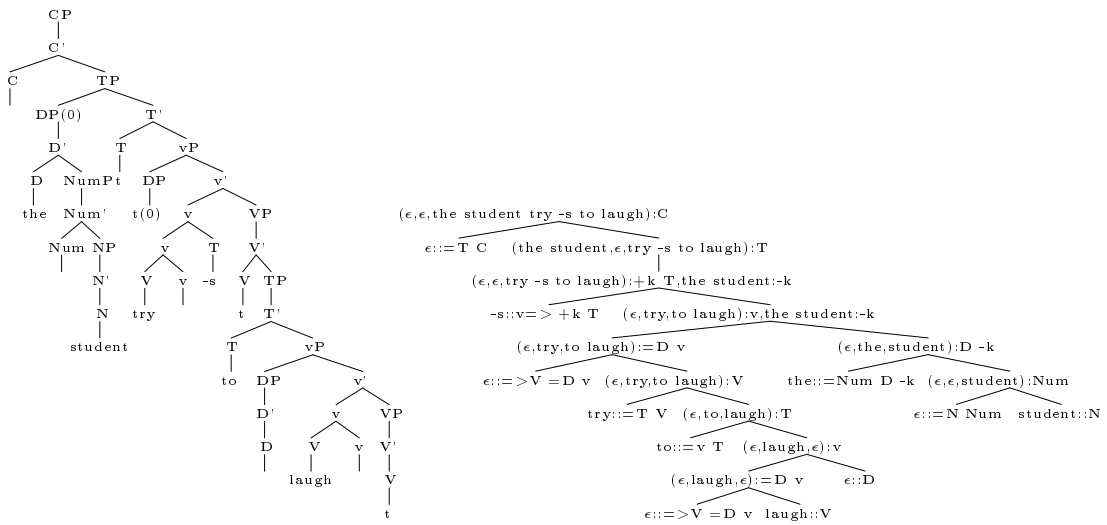


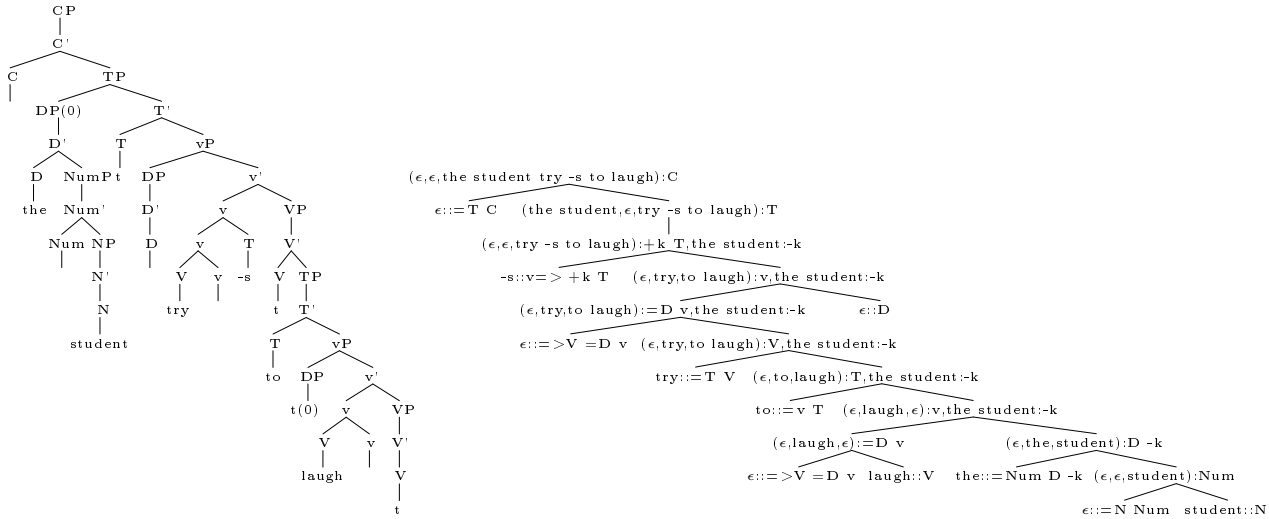
We <u>almost</u> succeed in getting a simple version of this proposal with just the following lexical items:

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:





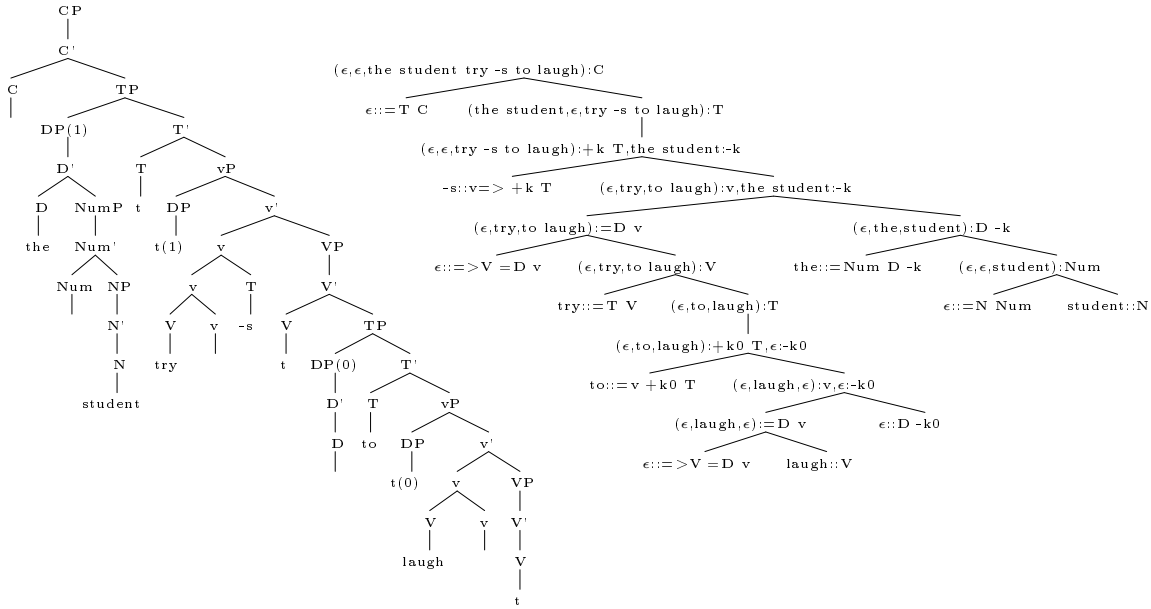
This second derivation is kind of wierd – it does not correspond to the semantic relations we wanted. How can we rule it out?

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" [54, 270, 170]. (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.

15.6 Summary

The rather complex range of constructions surveyed in the previous sections are all derived from a remarkably simple grammar. Here is the whole thing (containing some comments marked with %):

```
% complementizers
ϵ::=T C                ϵ::=>T C               ϵ::=>T +wh C           ϵ::=T +wh C
that::=T Ce            ϵ::=T Ce                                      % for embedded clauses
whether::=T Cwh        ϵ::=T +wh Cwh          ϵ::=>T +wh Cwh         % for embedded wh-clauses

% finite tense
-s::v=> +k T           % for affix hop
-s::=>Modal +k T       -s::=>Have +k T        -s::=>Be +k T          -s::=v +k T

% simple nouns
queen::N               pie::N                 human::N               car::N
coffee::N              shirt::N               language::N            king::N

% determiners
the::=Num D -k         every::=Num D -k       a::=Num D -k           an::=Num D -k
some::=Num D -k        some::D -k

% number marking (sg,pl)
ϵ::=N Num              -s::N=> Num

% names as lexical DPs – reconsider later!
Titus::D -k            Lavinia::D -k          Tamara::D -k           Saturninus::D -k
Rome::D -k             Sunday::D -k           physics::D -k          Rome::D -k

% pronouns as lexical DPs – treat Case marking later!
she::D -k he::D -k     it::D -k               I::D -k                you::D -k          they::D -k    % nom
her::D -k              him::D -k              me::D -k               us::D -k           them::D -k    % acc
my::=Num D -k          your::=Num D -k
her::=Num D -k         his::=Num D -k         its::=Num D -k         % gen

% wh determiners
which::=Num D -k -wh   which::D -k -wh
what::=Num D -k -wh    what::D -k -wh

% auxiliary verbs
will::=Have Modal      will::=Be Modal        will::=v Modal
have::=Been Have       have::=ven Have
be::=ving Be           been::=ving Been

% little v
ϵ::=>V =D v            -en::=>V =D ven        -ing::=>V =D ving
-en::=>V ven           -ing::=>V ving

% DP-selecting (transitive) verbs - select an object and take a subject too (via v)
praise::=D +k V        sing::=D +k V          eat::=D +k V           have::=D +k V

% intransitive verbs - select no object but take a subject
laugh::V               sing::V                charge::V              eat::V

% CP-selecting verbs
know::=Ce V            know::=Cwh V           know::=D +k V          know::V
doubt::=Ce V           doubt::=Cwh V          doubt::=D +k V         doubt::V
                       think::=Ce V                                                     think::V
                                              wonder::=Cwh V                            wonder::V

% CP-selecting nouns
claim::=Ce N           proposition::=Ce N     claim::N               proposition::N
```

```
% raising verbs - select only propositional complement no object or subject
seem::=T v

% infinitival tense
to::=v T          to::=Have T       to::=Be T         % NB: to does not select modals

% little a
ϵ::=>A =D a

% simple adjectives
black::A          white::A          human::A          mortal::A
happy::A          unhappy::A

% verbs with AP complements: predicative be seem
be::=a Be         seem::=a v

% adjectives with complements
proud::=p A       proud::A          proud::=T a

% little p (no subject?)
ϵ::=>P p

% prepositions with no subject
of::=D +k P       about::=D +k P    on::=D +k P

% verbs with AP & TP complements: small clause selectors as raising to object
prefer::=a +k V   prefer::=T +k V   prefer::=D +k V
consider::=a +k V consider::=T +k V consider::=D +k V

% nouns with PP complements
student::=p N     student::N
citizen::=p N     citizen::N

% more verbs with PP complements
be::=p Be         seem::=p Be       ϵ::=>P =D p       up::=D +k P   creek::N

% control verbs
try::=T V         want::=T V        want::=T +k V

% verbs with causative alternation: using little v that does not select subject
break::=D +k V    break::=D v

% PRO with "null case" feature k0
ϵ::D -k0
to::=v +k0 T      to::=Have +k0 T   to::=Be +k0 T     % NB: to does not select modals
```

A couple of points to notice:

- This MG has 137 lexical items in it. The corresponding MCFG generated by our translator (discussed earlier) has 1413 rewrite rules.

  A similar but larger grammar was constructed by John Hale [109].
- We still have not handle adjunctions, successive cyclic movements, resumptions, and many many other kinds of constructions. (Some of these missing constructions will be considered later!)
- There are many regularities in the lexicon that we have not tried to capture yet.

  For example, every one of the determiners has a -k. This suggests that maybe we should simply regard the -k as part of the indication of a determiner, the part that triggers EPP. The literture is full of suggestions along these lines.
- We have not said anything yet about how semantic values of these complexes are determined by their parts. And we have not said anything about the binding requirements of DPs.

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. Extend the grammar to get these topicalized constructions in this way, and then write a brief linguistic assessment of this approach.

2. **Put.** 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

   One idea is that while transitive verbs have two parts, v and V, verbs like *put* have three parts which we could call v and V and VV, where VV selects the PP, V selects the object, and v selects the subject. Extend the grammar in this way so that it gets *the cook put -s the pie in the oven.* Make sure that your extended grammar does NOT get *the cook put -s the pie*, or *the cook put -s in the oven.* Then write a brief linguistic assessment of this approach.

3. **Copy raising.** Some English dialects (like mine) allow raising verbs to appear with certain finite clauses, as in these examples [10, 227]:

   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, and then write a brief linguistic assessment of your approach. (Is this a lexical regularity (analogous to the -k property of D mentioned in the conclusions just above) or something else?)

4. **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 [57, pp.96-104] uses this example in his argument for asymmetric feature checking and particularly for Chomsky's [50] story about ±interpretable features. Assess these arguments.

5. **Head vs. phrasal movement.** Dave Schueler points out that our formalization of head movement in the configuration of selection is similar to a suggestion recently made by Pesetsky & Torrego [201]:

   (5a) What did Mary buy?

   In (5a), [a feature] μT 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 <u>not</u> 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?

§16 MG parser implementation

(0) We saw how tree-based MGs can be replaced by tuple-based MGs, defined as follows [249], without changing the language or the shape of the derivations of each string in the language:

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

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

  `C, T, D, N, V, P,...` (selected categories)

  `=C, =T, =D, =N, =V, =P,...` (selector features)

  `+wh, +case, +focus,...` (licensors)

  `-wh, -case, -focus,...` (licensees)
- **chains** $C = \Sigma^* \times T \times F^*$
- **expressions** $E = C^*$
- **lexicon** $Lex \subset \Sigma^* \times \{::\} \times F^*$, a finite set

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(1) Once the parsing rules are given in this deductive format, it is trivial to specify a CKY-like method for recognizing MG languages. We add a first rule for the lexical input sequence, treating empty lexical items as elements going from i to $j = i$, for every $0 \leq i \leq n$ where n is the length of the string.

$$\frac{\text{for each word } w \text{ from } i \text{ to } j \text{ such that } w :: \gamma \in Lex}{(i,j) : \gamma} \text{ lex}$$

$$\frac{(i,j) :: =f\gamma \qquad (j,k) \cdot f, \alpha_1, \ldots, \alpha_k}{(i,k) : \gamma, \alpha_1, \ldots, \alpha_k} \text{ merge1: lexical item selects a non-mover}$$

$$\frac{(j,k) : =f\gamma, \alpha_1, \ldots, \alpha_k \qquad (i,j) \cdot f, \iota_1, \ldots, \iota_l}{(i,k) : \gamma, \alpha_1, \ldots, \alpha_k, \iota_1, \ldots, \iota_l} \text{ merge2: derived item selects a non-mover}$$

$$\frac{(i,j) \cdot =f\gamma, \alpha_1, \ldots, \alpha_k \qquad (k,l) \cdot f\delta, \iota_1, \ldots, \iota_l}{(i,j) : \gamma, \alpha_1, \ldots, \alpha_k, (k,l) : \delta, \iota_1, \ldots, \iota_l} \text{ merge3: any item selects a mover}$$

$$\frac{(j,k) : +f\gamma, \alpha_1, \ldots, \alpha_{i-1}, (i,j) : -f, \alpha_{i+1}, \ldots, \alpha_k}{(i,k) : \gamma, \alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k} \text{ move1: final move of licensee}$$

$$\frac{(i,j) : +f\gamma, \alpha_1, \ldots, \alpha_{i-1}, (k,l) : -f\delta, \alpha_{i+1}, \ldots, \alpha_k}{(i,j) : \gamma, \alpha_1, \ldots, \alpha_{i-1}, (k,l) : \delta, \alpha_{i+1}, \ldots, \alpha_k} \text{ move2: nonfinal move of licensee}$$

(2) We modify the system above slightly as follows: rather than imposing the SMC on the move step, we will not produce any result which has two -f chains, for any f.

(3) Harkema shows that this MG recognition requires no more than $\mathcal{O}n^{4k+4}$ steps, where k is the number of licensors and n is the length of the input [111].

(4) Various MG recognizer implementations have been written, parsing MGs directly [246, 190, 109] or by translating them into MCFGs or closely related formalisms [108, 3, 164] but we describe a very direct approach.[1]

(5) Now to begin coding the implementation, we can declare these feature constructors as type constructors:

Cat	(category)	
Sel	(selected)	which appears in the literature as =
Pos	(licensor, positive trigger for move)	which appears in the literature as +
Neg	(licensee, negative trigger for move)	which appears in the literature as -

[1] These and some other related implementations are linked at `http://www.linguistics.ucla.edu/people/stabler/epssw.htm`.

We define these types so that they apply to integers. So a particular category will be *Cat n*, a 'mover' will be *Pos n*, and so on, for particular integers n.
We number the categories and the 'movers' separately,

$$0, 1, 2, \ldots$$

The numbering can be arbitrary otherwise.

(6) **Example.** Our grammars will be given in this form.

```
(*          file: g0.ml
        created: Thu Feb 28 09:17:43 PST 2008
   first author: E Stabler stabler@ucla.edu

   Example grammar given by (i) feature to string translations (moverString, catString)
                            (ii) size parameters (moverMax,catMax)
                            (iii) the list of lexical items, and
                            (iv) some examples (at least zero of them! so that the function is de

   We could write a program to generate the numbering of categories
   and these other parameters, but for now we do it by hand.
*)
let moverString f = List.assoc f [(0,"k"); (1,"wh")];;
let moverMax = 1;;      (* so the movers are 0,1 *)

let catString f = List.assoc f [(0,"C"); (1,"T"); (2,"D"); (3,"N"); (4,"V"); (5,"v")];;
let catMax = 5;;        (* so the categories are 0,1,2,3,4,5 *)

let grammar =[
   ([], [Sel 1; Cat 0]);                ([], [Sel 1; Pos 1; Cat 0]);
   ([], [Sel 5; Pos 0; Cat 1]);         ([], [Sel 4; Sel 2; Cat 5]);
   (["eat"], [Sel 2; Pos 0; Cat 4]);    (["laugh"], [Cat 4]);
   (["the"], [Sel 3; Cat 2; Neg 0]);    (["which"], [Sel 3; Cat 2; Neg 0; Neg 1]);
   (["king"], [Cat 3]);                 (["pie"], [Cat 3]);
];;

(* Apply cky to example 1 with: (eg grammar 1). Use neg ints for examples that should fail.*)
let example = function  (* (startCategory, inputString) pairs *)
    1 -> (0,["the";"king";"laugh"])
  | 2 -> (0,["the";"king";"the";"pie";"eat"])
  | 3 -> (0,["which";"pie";"the";"king";"eat"])
  | -1 -> (0,["the";"king";"the";"pie";"laugh"])
  | -2 -> (0,["the";"king";"pie";"eat"])
  | -3 -> (0,["which";"pie";"the";"king";"eat";"the";"pie"])
  | _ -> raise (failwith "No such example");;
```

(7) Our implementation provides a function that will print out the grammar in its more readable string form:

```
# grammar;;
- : (string list * feature list) list =
[([], [Sel 1; Cat 0]); ([], [Sel 1; Pos 1; Cat 0]);
 ([], [Sel 5; Pos 0; Cat 1]); ([], [Sel 4; Sel 2; Cat 5]);
 (["eat"], [Sel 2; Pos 0; Cat 4]); (["laugh"], [Cat 4]);
 (["the"], [Sel 3; Cat 2; Neg 0]); (["which"], [Sel 3; Cat 2; Neg 0; Neg 1]);
 (["king"], [Cat 3]); (["pie"], [Cat 3])]

# printLex grammar;;
[] :: =T C
[] :: =T +wh C
```

```
[] :: =v +k T
[] :: =V =D v
eat :: =D +k V
laugh :: V
the :: =N D -k
which :: =N D -k -wh
king :: N
pie :: N
- : unit = ()
#
```

(8) For the chart, where x is the number of movement trigger types, we could use a $2 * (x + 1)$ dimensional hypercube of the input length, but when there are many movers this will be too big and, in almost all applications, mainly empty.

So we will use 2-d hypercube (i.e. an $n \times n$ square matrix) as we did in cf-cyk and simply store the ranges of the "moving elements" in the items. We will actually use 2 $n \times n$ matrices, one for non-moving Cat items and one for Sel items

Since the SMC prohibits more than one (Neg e) chain per constituent, for each e, we will use an array to hold these chains – the length of the array will be the number of different elements e such that (Neg e) appears anywhere in the lexicon.

When a cell in the $n \times n$ arrays is empty, we will use a simple "empty" type (Y in the non-moving Cat item arrays, and Z in the Sel item arrays)

(9) We write the structure building functions carefully, since each one can be used many thousands of times in parsing a single string.

The structure building functions (not including head movement functions) are these:

$$merge1, merge2, merge3, move1, move2$$

Since the merges are binary, we provide, in effect, "reverse" versions that seach for appropriate first arguments given the second one:

$$merge1r, merge2r, merge3r$$

These functions are applied in all possible ways in the main loop

(10) We implement $move1, move2$ by applying them automatically to each result of any merge that that begins with (Pos n). If an item begins with (Pos n) but move cannot apply, this item is useless, and nothing is added to either chart or agenda. If move applies (1 or more times), the final result goes into the chart and agenda if new. Intermediate results go into chart, but not onto the agenda.

(11) In this first implementation, we assume that lexical items are either empty or 1 symbol long. We also assume lexical items are well formed in the sense that their features are in

(selectors,licensors)* Cat licensees*

never beginning with a licensor.

(12) We could say

type cell = item list

but that would require that we search the whole cell to find what we are looking for.

So first, as described in (10), move applies automatically to reduce any item that triggers it, before that item is put into the searchable parts of the chart. So the chart only needs Cat items and Sel Items.

And notice that the position of the Cat items is only useful when they are non-moving, so only these will be placed in an nXn array like we have in cf-cyk. The non-moving Cat items will be kept in a separate array.

Under each of these classifications, to store the items we use category arrays – i.e. arrays indexed by the category feature.

We keep a list of "other items" are just for tracing, etc – items with move triggers

So the overall structure of our "chart" looks like this, where n = succ length of input.

(nXnNonmoverCat,	nXnSel,	moverCats,	other)
|	|	|	|
each cell i, j an nmItem array indexed by cat with no feats	each cell i, j an item array indexed by cat, with first feats removed	each cell cat a sitItem list with first feats removed	sitItem list
(simple,movers)	(simple,fs,movers)	(i,j,simple,fs,movers)	(i,j,simple,fs,movers)

Note that what we call movers here are "inside" each constituent, each item. They have type (int * int * feature list) array, with length (succ moverMax). In these elements the first (Neg f) feature of the feature list can be removed since it is encoded by its array position f.

In the nXnNonmoverCat array, position is encoded by the coordinates i,j of the cell and the category is encoded by position in the cell contents, and as non-movers there are no other features, so the leaves are lists of (simple,movers) pairs. That is, leaf contents have type: $(bool * chain\ array)$ list

In the nXnSel array, the leaves are lists of (simple,feats,movers) pairs. That is, leaf contents have type: $(bool * feature\ list * chain\ array)$ list

So the initial chart (for input length 3, catMax=5) is

$$
\begin{array}{l}
([|[|Y;Y;Y;Y|];[|Y;Y;Y;Y|];[|Y;Y;Y;Y|];[|Y;Y;Y;Y|]|],\\
[|[|Z;Z;Z;Z|];[|Z;Z;Z;Z|];[|Z;Z;Z;Z|];[|Z;Z;Z;Z|]|],\\
[|[];[];[];[];[];[]|],\\
[]).
\end{array}
$$

To insert a first new result $(3, 4, (false, [Sel1; Cat2], [|Em; Em|]))$, we go to cell 3,4 of selCoordChart, and change the initial contents Z to $(B\ x)$, where x is an array of length $(succ\ catMax)$ with all values [], and then we change $x.(1)$ from [] to $[(false, [Cat2], [|Em; Em|])]$. Note that we remove the feature $(Sel\ 1)$ since that is encoded already by the chart structure.

Since we build only in the upper triangle of the coordCharts, half will always stay Z or Y, respectively.

(13) Recap:

```
type feature = Sel of int | Cat of int | Pos of int | Neg of int;;
type chain = Em | Tu of (int * int * feature list);;

type nmItem = bool * chain array;;              (* non-movers have no features left *)
type selItem = bool * feature list * chain array;;
type sitItem = int * int * bool * feature list * chain array;;

type nmCatArray = Y | A of nmItem list array;;  (* length A arrays=(succ catMax) *)
type selArray = Z | B of selItem list array;;   (* length B arrays=(succ catMax) *)

type nmCatCoordChart = nmCatArray array array;; (* a square matrix of cells *)
type selCoordChart = selArray array array;;     (* a square matrix of cells *)

type chart = nmCatCoordChart * selCoordChart * sitItem list array * sitItem list;;
type agenda = sitItem list;;
```

(14) In lexical insertion, it suffices to put just the selector items into the agenda

```
main loop:
  given (agenda,chart), pop top item from agenda;
  compute (newAgenda,newChart) as detailed below, and repeat with newAgenda,
  until the current agenda is empty.

NB: remember again that move1, move2 are composed with merge steps,
    applying automatically to each result that triggers them.
```

```
      If move is triggered but cannot apply, nothing is added to either chart or agenda.
      If move applies (1 or more times), the final result goes into chart and agenda if new.
        Intermediate results go into (thirdOf3 chart), but not onto the agenda.

if pop agenda = (i, j, (true, (Sel f)::alpha, movers))
  then getall merge1,merge3 results --
        for merge1, search ranges with left edge j
          (and not overlapping with anything else in moving chains)
        for merge3, search ranges not overlapping with i,j
          (and not overlapping with anything else in moving chains)
if pop agenda = (i, j, (false, (Sel f)::alpha, movers))
  then getall merge2,merge3 results --
        for merge2, search ranges with right edge i
          (and not overlapping with anything else in moving chains)
        for merge3, search ranges not overlapping with i,j
          (and not overlapping with anything else in moving chains)
if pop agenda = (i, j, (_, (Cat f)::[], movers))
  then getall merge1r,merge2r results --
        for merge2, search ranges with right edge i
          (and not overlapping with anything else in moving chains)
        for merge1, search ranges with left edge j
          (and not overlapping with anything else in moving chains)
if pop agenda = (i, j, (_, (Cat f)::NONEMPTY, movers))
  then getall merge3r results -- searching all selectors in chart
        for merge3, search ranges not overlapping with i,j (or anything else in movers)
if pop agenda = (i, j, (_, (Pos f)::alpha, _)) NEVER HAPPENS -- SEE NB ABOVE
if pop agenda = (i, j, (_, (Neg f)::alpha, _)) NEVER HAPPENS
```

NB: overlap check are done only when string is actually concatenated, in $merge1, merge2$, or $move1$. Doing them in every step seems not worth the effort – too costly.

16.0 Parsing and tree collection

The recognizer above may produce $(0, n)$: *Start* with no indication of how that item was built. To facilitate recovery of the derivation history, sometimes the the chart items are annotated with their the sources (sometimes in an auxiliary data structure).

The first argument of merge is always a member of nXnSel, where each element can be specified by 4 integers if we keep track of the 'position' of each element in its list:

(leftEdge, rightEdge, category, position).

The second argument of merge is either an element of nXnNM or of movers, and in the latter case we can just let leftEdge=rightEdge=-1. So the presence of -1 in those fields unambiguously indicates elements from the movers array. (For lexical items, we will put -1 in the category and position fields as well.) Finally, since the operation requires (Sel category) and (Cat category) to have the same category, we need only specify that category once. In this way, each source of each element can be represented by a 7-tuple of integers:

(selLeftEdge, selRightEdge, category, selPosition, catLeftEdge, catRightEdge, catPosition).

This specification of sources removes the need for search even when move steps are involved in producing the result.

Given a list of the possible sources of each item, whole derivations can be quickly reconstructed with a backtracking strategy (if there are no cycles in the parent relations). Alternatively, we could try to define some semantic process that would incrementally build the most probable parse, in some sense of 'most probable'.

16.1 MGH parsing, etc

The CKY-like methods extend easily to MGs with head-movement, adjunction, and certain other kinds of operations [247, 176, 93, 98, 97].

16.2 MGs with copying (MGCs)

(15) Notice that we do not move by performing copy+delete. Rather, a movement is just a certain kind of possibly discontinuous dependency – a instance in which a constituent is collected from non-contiguous parts of the input.

Is there any reason to assume that *move* should be copy+delete? Besides various conceptual arguments based on theory internal and controversial assumptions about theoretical simplicity, various other kinds of arguments and evidence have been mustered:

partial movement: Some languages like German, Hindi and Hungarian allow constructions in which the matrix clause has a reduced wh-element and an intervening operator position holds the regular question word [81, 122]:

Was$_j$ hast Du t_j geglaubt [wen$_i$ sie t_i gesehen hat]$_j$?
what have you thought who she seen has

'who do you think she has seen'

partial movement in acquisition: We find similar constructions even among learners of adult languages that do not admit partial movement [257, 174]:

(child English) Which animal do you think what really says woof-woof?

covert copying of syntactic phrases in ellipsis? [146, 272, 128, 83, 110]

a. This is the book which$_i$ Max read t_i before knowing that Lucy did e.

b. * This is the book which$_i$ Max read t_i before hearing the claim that Lucy did e.

c. Dulles suspected everyone who Angleton believed that Philby did e

d. * Dulles suspected everyone who Angleton wondered why Philby did e

overt copying of syntactic phrases: [177, 248, 152, 193] Mandarin Chinese:

Zhangsan like play basketball not like play basketball
Zhangsan ai da lanqiu (*,) bu ai da lanqiu

'Does Zhangsan like to play basketball?'

Yoruba:

wiwa kata wo igi lu le ni Omole wa kata wo igi lu le
RED-drive tractor fell tree hit ground Foc builder drive tractor fell tree hit ground

(16) Let's use the notation $-\hat{y}$ to indicate a trigger for copy-movement, a copy-licensee, and add the following rules, where in the new merge3c and move2c rules, δ begins with $-\hat{y}$ for some y. (The original rules could be eliminated, or could be restricted to instances of non-copy-movers.)[2]

$$\frac{s :: =f\gamma, \alpha_1, \ldots, \alpha_k \qquad t \cdot f\delta, \iota_1, \ldots, \iota_l}{st : \gamma, \alpha_1, \ldots, \alpha_k, t : \delta, \iota_1, \ldots, \iota_l} \text{ merge3c1: lexical item selects a copy-mover}$$

$$\frac{s : =f\gamma, \alpha_1, \ldots, \alpha_k \qquad t \cdot f\delta, \iota_1, \ldots, \iota_l}{ts : \gamma, \alpha_1, \ldots, \alpha_k, t : \delta, \iota_1, \ldots, \iota_l} \text{ merge3c2: derived item selects a copy-mover}$$

$$\frac{s : +f\gamma, \alpha_1, \ldots, \alpha_{i-1}, t : -\hat{f}, \alpha_{i+1}, \ldots, \alpha_k}{ts : \gamma, \alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k} \text{ move1c: final move of copy-mover}$$

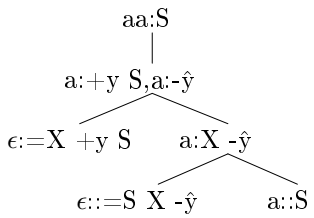
$$\frac{s : +f\gamma, \alpha_1, \ldots, \alpha_{i-1}, t : -\hat{f}\delta, \alpha_{i+1}, \ldots, \alpha_k}{ts : \gamma, \alpha_1, \ldots, \alpha_{i-1}, t : \delta, \alpha_{i+1}, \ldots, \alpha_k} \text{ move2c: nonfinal move of copy-mover}$$

Kobele [152, p.188] points out that the following grammar derives the non-mildly-context-sensitive language a^{2^n}:

[2] A number of details about the interactions between copy movement and other operations need to be decided here. For fuller discussion of some possibilities, see [152]. Kobele considers, among many other things, a formalism that marks the licensors rather than the licensees, which we mark here.

a::S ϵ:: =S X -ŷ ϵ:: =X +y S

Consider the following 3-step derivation which uses merge3c1 in the second step (we count from the bottom, since we are drawing trees linguist-style with the root on top).



Obviously, if we repeat these steps again using this derived aa:S instead of a::S, we will derive aaaa:S.

(17) How can these rules be implemented? Consider merge3c1 for example. Note that when $k \neq l$ in the following rule, we must have $i \neq i'$, where (i', i) is the same string as (k, l):

$$\frac{(i,j) :: =f\gamma, \alpha_1, \ldots, \alpha_k \qquad (k,l) \cdot f\delta, \iota_1, \ldots, \iota_l}{(i',j) : \gamma, \alpha_1, \ldots, \alpha_k, (k,l) : \delta, \iota_1, \ldots, \iota_l} \text{ merge3c1: lexical item selects a copy-mover}$$

A simple idea is that the side condition that (i', i) is a copy of (k, l) be enforced simply as a condition on the strings, "at PF." Note that we could avoid some useless checking of this condition based on the fact that in bottom-up CYK parsing, if (i', i) is a string copy of (k, l), then these two spans will also share the same syntactic analyses. So we need only check the string identity of (i', i) and (k, l) when they already have identical syntactic analyses. Kobele considers other alternatives [152, pp.187-212].

16.3 MG parsing one derivation at a time

A number of different 'all paths at once' strategies are defined in [112], but there has been very little work on 'one path at a time' strategies. This is true not only for MGs, but for all the other 'mildly context sensitive' grammar formalisms (TAGs, CCGs, etc). Some first steps were taken in [265, 131, 266, 269], 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.

16.4 Appendix: listing

```
(*          file: mg-cky.ml
         created: Thu Feb 28 09:12:22 PST 2008
    first author: E Stabler stabler@ucla.edu
                  CKY-like recognizer for MGs, using a subset of OCaml.

    For final performance version, print routines can be removed, and we do not need to keep the 4th element of chart
*)

type feature = Sel of int | Cat of int | Pos of int | Neg of int;;
type chain = Em | Tu of (int * int * feature list);;  (* moving elements inside each item *)
type nmItem = bool * chain array;;                    (* non-movers, with cat coded by array address, have no features left *)
type selItem = bool * feature list * chain array;;
type sitItem = int * int * bool * feature list * chain array;; (* situated items for agenda+other *)
type nmCatArray = Y | A of nmItem list array;;        (* length A arrays=(succ catMax) *)
type selArray = Z | B of selItem list array;;         (* length B arrays=(succ catMax) *)
type nmCatCoordChart = nmCatArray array array;;       (* n X n matrix, for n=succ (length input) *)
type selCoordChart = selArray array array;;           (* n X n matrix, for n=succ (length input) *)
type chart = nmCatCoordChart * selCoordChart * sitItem list array * sitItem list;;

(* A grammar is needed to set size parameters moverMax, catMax, and specify categories.
   Just one grammar should be uncommented at a time!

#use "grammars/g0.ml";;     (* simplest SOV "the king laugh", "the king the pie eat" *)
#use "grammars/g4.ml";;     (* copy language aabaab *)
#use "grammars/g5.ml";;  (* generates 1^n2^n3^n4^n5^n as an s *)
#use "grammars/g-ne.ml";;  (* "titus praise -s lavinia" naive English SVIO, inspired by Mahajan 2000 *)
#use "grammars/g-nta.ml";;  (* "Beatrice Benedick criticize -s" naive Tamil SOVI, inspired by Mahajan 2000 *)
#use "grammars/g-nza.ml";;  (* "criticize -s Beatrice Benedick" naive Zapotec VISO, inspired by Mahajan 2000 *)
#use "grammars/tlingit.ml";; (* with a q-morpheme, inspired by Seth Cable *)

*)

#use "g0.ml";;     (* simplest SOV "the king laugh", "the king the pie eat" *)

(* BEGIN print functions for tracing, and some other utilities *)
let fstOf4 (a,_,_,_) = a;;  (* so (fstOf4 chart) = nmCatCoordChart *)
let sndOf4 (_,a,_,_) = a;;  (* so (sndOf4 chart) = selCoordChart *)
let thdOf4 (_,_,a,_) = a;;  (* so (thdOf4 chart) = movers *)
let fthOf4 (_,_,_,a) = a;;  (* so (fthOf4 chart) = other *)

let featString = function
    Sel b -> "="^(catString b)
  | Cat b -> catString b
  | Pos b -> "+"^(moverString b)
  | Neg b -> "-"^(moverString b);;

let printFeat f = Printf.fprintf stdout "%s " (featString f);;
let printFeats fs = List.iter printFeat fs;;

let printStrings ss =
  if ss=[]
  then Printf.fprintf stdout "[] " (* just to make the print presentation perfectly clear *)
  else List.iter (Printf.fprintf stdout "%s ") ss;;

let rec printLex grammar = match grammar with
    [] -> ()
  | (s,f)::xs ->
      begin
        printStrings s; Printf.fprintf stdout ":: ";
        printFeats f;  Printf.fprintf stdout "\n";
        printLex xs;
      end;;

let printChain simple = function
    Em -> ()
  | Tu (i,j,feats) ->
      begin
        Printf.fprintf stdout "(%i,%i)" i j;
        if simple then Printf.fprintf stdout ":: " else Printf.fprintf stdout ": ";
        List.iter printFeat feats;
      end;;

let printChainWithF simple f = function
    Em -> ()
  | Tu (i,j,feats) ->
      begin
        Printf.fprintf stdout "(%i,%i)" i j;
        if simple then Printf.fprintf stdout ":: " else Printf.fprintf stdout ": ";
        List.iter printFeat ((Neg f)::feats);
      end;;

(* to print nmItem, showing i,j coded into nXnNM array position, and Cat f coded into nmItem array position *)
let printNMItem i j f (simple,movers) =
  begin
    printChain simple (Tu (i,j,(Cat f)::[]));
    Array.iteri (printChainWithF false) movers;  Printf.fprintf stdout "\n";
  end;;

let printNMItemList i j f itemlist = List.iter (printNMItem i j f) itemlist;;

let printNMCatArray i j array = match array with
    Y -> ()
  | A catArray -> Array.iteri (printNMItemList i j) catArray;;

let rec printNMCatCoordChart coordChart =
  let n = pred (Array.length coordChart)
  in
    for i=0 to n do
      for j=0 to n do
        printNMCatArray i j coordChart.(i).(j)
      done
    done;;

let listArraySize array = Array.fold_right (function x -> (function y -> (List.length x) + y)) array 0;;
```

```
let nmCatArraySize catArray = match catArray with
    Y ->0
  | A array -> listArraySize array;;

let nmCatArrayArraySize catArray =
  Array.fold_right (function x -> (function y -> (nmCatArraySize x) + y)) catArray 0;;

let nmChartSize catArray =
  Array.fold_right (function x -> (function y -> (nmCatArrayArraySize x) + y)) catArray 0;;

(* to print selItem, showing i,j coded into nXnSel array position, and Sel f coded into selItem array position *)
let printSelItem i j f (simple,fs,movers) =
  begin
    printChain simple (Tu (i,j,(Sel f)::fs));
    Array.iteri (printChainWithF false) movers;  Printf.fprintf stdout "\n";
  end;;

let printSelItemList i j f itemlist = List.iter (printSelItem i j f) itemlist;;

let printSelArray i j array = match array with
    Z -> ()
  | B catArray -> Array.iteri (printSelItemList i j) catArray;;

let rec printSelCoordChart coordChart =
  let n = pred (Array.length coordChart)
  in
    for i=0 to n do
      for j=0 to n do
        printSelArray i j coordChart.(i).(j)
      done
    done;;

let selArraySize catArray = match catArray with
    Z -> 0
  | B array -> listArraySize array;;

let selArrayArraySize catArray =
  Array.fold_right (function x -> (function y -> (selArraySize x) + y)) catArray 0;;

let selChartSize catArray =
  Array.fold_right (function x -> (function y -> (selArrayArraySize x) + y)) catArray 0;;

(* to print sitItem in moverCats, insert Cat f that is coded into nmItem array position *)
let printMoverItem f (i,j,simple,fs,movers) =
  begin
    printChain simple (Tu (i,j,(Cat f)::fs));
    Array.iteri (printChainWithF false) movers;  Printf.fprintf stdout "\n";
  end;;

let printMoverItemList f itemlist = List.iter (printMoverItem f) itemlist;;

let printMoverArray = Array.iteri printMoverItemList;;

(* to print sitItem in other list *)
let printOtherItem (i,j,simple,fs,movers) =
  begin
    printChain simple (Tu (i,j,fs));
    Array.iteri (printChainWithF false) movers;  Printf.fprintf stdout "\n";
  end;;

let printOtherItemList itemlist = List.iter printOtherItem itemlist;;

let printChart (nXnNonmoverCat, nXnSel, moverCats, other) =
  begin
    printNMCatCoordChart nXnNonmoverCat;
    printSelCoordChart nXnSel;
    printMoverArray moverCats;
    printOtherItemList other;
  end;;

let chartStats (nXnNonmoverCat, nXnSel, moverCats, other) =
  let nmSize = nmChartSize nXnNonmoverCat in
  let selSize = selChartSize nXnSel in
  let moverSize = listArraySize moverCats in
  let otherSize = List.length other in
  let total = nmSize+selSize+moverSize+otherSize in
    Printf.fprintf stdout "Chart size (%i,%i,%i,%i) = %i total," nmSize selSize moverSize otherSize total;;

let rec printAgenda agenda =
  begin
    Printf.fprintf stdout "---agenda:\n";
    printOtherItemList agenda;
    Printf.fprintf stdout "---end agenda\n";
  end;;
(** END print functions for tracing **)

let rec ensureMember e list = match list with
    [] -> [e]
  | x::xs -> if e=x then x::xs else x::ensureMember e xs;;

(* put empty categories into *every string position* by calling: emptyInsert lex 0 (agenda0,chart0) lex. *)
let rec emptyInsert lex i stop (agenda,(nXnNM,nXnSel,movers,other)) remaining = match remaining with
    [] ->
      if i = stop
      then (agenda,(nXnNM,nXnSel,movers,other))
      else emptyInsert lex (succ i) stop (agenda,(nXnNM,nXnSel,movers,other)) lex
  | ([],(Cat x)::[])::moreLex ->  (* empty non-moving Cat -- i.e. just 1 feature! *)
      begin
        (match nXnNM.(i).(i) with
            Y -> let catArray = Array.make (succ catMax) []
            in
              begin
                catArray.(x) <- ((true,Array.make (succ moverMax) Em) :: []);
                nXnNM.(i).(i) <- A catArray;
              end
          | A catArray ->
              begin
                catArray.(x) <- ((true, (Array.make (succ moverMax) Em)) :: catArray.(x));
              end
        );
```

```
          emptyInsert lex i stop ((i,i,true,[Cat x],(Array.make (succ moverMax) Em))::agenda,(nXnNM,nXnSel,movers,other)) moreLex;
        end
    | ([],(Cat x)::fs)::moreLex ->  (* empty moving Cats *)
        begin
          if i=0 then movers.(x) <- (i,i,true,fs,Array.make (succ moverMax) Em)::movers.(x) else ();
          emptyInsert lex i stop ((i,i,true,(Cat x)::fs,Array.make (succ moverMax) Em)::agenda,(nXnNM,nXnSel,movers,other)) moreLex;
        end
    | ([],(Sel x)::fs)::moreLex ->  (* empty selector, add to chart **and to agenda** *)
        begin
          (match nXnSel.(i).(i) with
              Z -> let catArray = Array.make (succ catMax) []
              in
                begin
                  catArray.(x) <- ((true, fs, Array.make (succ moverMax) Em) :: []);
                  nXnSel.(i).(i) <- B catArray;
                end
            | B catArray ->
                begin
                  catArray.(x) <- ((true, fs, (Array.make (succ moverMax) Em)) :: catArray.(x));
                end
          );
            emptyInsert lex i stop ((i,i,true,(Sel x)::fs,Array.make (succ moverMax) Em)::agenda ,(nXnNM,nXnSel,movers,other)) moreLex
        end
    | _::moreLex -> emptyInsert lex i stop (agenda,(nXnNM,nXnSel,movers,other)) moreLex;;

(* add non-empty words to the chart *)
let rec lexInsert lex (agenda,(nXnNM,nXnSel,movers,other)) i input remainingG = match (input,remainingG) with
    ([],_) -> (agenda,(nXnNM,nXnSel,movers,other))
    | (w::ws,[]) -> lexInsert lex (agenda,(nXnNM,nXnSel,movers,other)) (succ i) ws lex
    | (w::ws,([word],(Cat x)::[])::moreLex) when w=word ->   (* non-moving Cat -- i.e. just 1 feature! *)
        begin
          (match nXnNM.(i).(succ i) with
              Y -> let catArray = Array.make (succ catMax) []
              in
                begin
                  catArray.(x) <- ((true,Array.make (succ moverMax) Em) :: []);
                  nXnNM.(i).(succ i) <- A catArray;
                end
            | A catArray ->
                begin
                  catArray.(x) <- ((true, (Array.make (succ moverMax) Em)) :: catArray.(x));
                end
          );
          lexInsert lex ((i,succ i,true,[Cat x],(Array.make (succ moverMax) Em))::agenda,(nXnNM,nXnSel,movers,other)) i input moreLex;
        end
    | (w::ws,([word],(Cat x)::fs)::moreLex) when w=word ->  let cell = movers.(x) in  (* moving Cat *)
        if List.mem (i,succ i,true,fs,Array.make (succ moverMax) Em) cell
        then lexInsert lex (agenda,(nXnNM,nXnSel,movers,other)) i input moreLex
        else
          begin
            movers.(x) <- (i,succ i,true,fs,Array.make (succ moverMax) Em)::cell;
            lexInsert lex ((i,succ i,true,(Cat x)::fs,Array.make (succ moverMax) Em)::agenda,(nXnNM,nXnSel,movers,other)) i input more
          end
    | (w::ws,([word],(Sel x)::fs)::moreLex) when w=word ->  (* selector, add to chart **and to agenda** *)
        begin
          (match nXnSel.(i).(succ i) with
              Z -> let catArray = Array.make (succ catMax) []
              in
                begin
                  catArray.(x) <- ((true, fs, Array.make (succ moverMax) Em) :: []);
                  nXnSel.(i).(succ i) <- B catArray;
                end
            | B catArray ->
                begin
                  catArray.(x) <- ((true, fs, (Array.make (succ moverMax) Em)) :: catArray.(x));
                end
          );
          let newAgenda = (i,succ i,true,(Sel x)::fs,Array.make (succ moverMax) Em)::agenda in
            lexInsert lex (newAgenda,(nXnNM,nXnSel,movers,other)) i input moreLex;
        end
    | (w::ws,_::moreLex) -> lexInsert lex (agenda,(nXnNM,nXnSel,movers,other)) i input moreLex;;

(** NOW UTILITIES FOR MAIN LOOP, COMPUTING THE CLOSURE **)
(* combine the moving elements of two constituents, if possible, returning (newArray,success) *)
let rec moverMergeVal array1 array2 newArray i stop =
  if i>=stop
  then (newArray,true)
  else if array1.(i)=Em then begin newArray.(i)<-array2.(i); moverMergeVal array1 array2 newArray (succ i) stop; end
  else if array2.(i)=Em then begin newArray.(i)<-array1.(i); moverMergeVal array1 array2 newArray (succ i) stop; end
  else (array1,false);;

(* update (agenda,chart) with each possible kind of item, after applying move if necessary *)
let rec update (i,j,simple,feats,moverArray) (agenda,(nXnNM,nXnSel,movers,other)) =
  let rec move (k,l,x,features,moverArray) agendaChart =                    (* that's the letter l *)
      (match moverArray.(x) with
          Em -> agendaChart
        | Tu (i,j,[]) -> (* move1: final move *)
            if j=k
            then let newMoverArray = Array.copy moverArray in
              begin
                newMoverArray.(x) <- Em;
                update (i,l,false,features,newMoverArray) agendaChart;
              end
            else agendaChart
        | Tu (i,j,(Neg f)::fs) -> (* move2: nonfinal *)
            if moverArray.(f) = Em
            then let newMoverArray = Array.copy moverArray in
              begin
                newMoverArray.(x) <- Em;
                newMoverArray.(f) <- Tu (i,j,fs);
                update (k,l,false,features,newMoverArray) agendaChart;
              end
            else agendaChart
        | Tu _ -> raise (failwith "move error: invalid licensee sequence")
      )
  in match feats with
      [] -> raise (failwith "update error: featureless result")
    | (Cat x) :: [] -> (* nonmoving Cat item *)
        (match nXnNM.(i).(j) with
            Y -> let catArray = Array.make (succ catMax) [] in
              begin
```

```
                    catArray.(x) <- ((false,moverArray) :: []);
                    nXnNM.(i).(j) <- A catArray;
                    ((i,j,false,(Cat x)::[],moverArray)::agenda,(nXnNM,nXnSel,movers,other));
                  end
            | A catArray ->
                  let cell = catArray.(x) in
                    if List.mem (false,moverArray) cell
                    then (agenda,(nXnNM,nXnSel,movers,other))
                    else
                      begin
(*                      Printf.fprintf stdout "adding: "; printOtherItem (i,j,false,(Cat x)::[],moverArray); *)
                        catArray.(x) <- ((false,moverArray)::cell);
                        ((i,j,false,(Cat x)::[],moverArray)::agenda,(nXnNM,nXnSel,movers,other));
                      end
          )
      | (Cat x) :: features -> (* moving Cat item *)
          let cell = movers.(x) in
            if List.mem (i,j,false,features,moverArray) cell
            then (agenda,(nXnNM,nXnSel,movers,other))
            else
              begin
(*              Printf.fprintf stdout "adding: "; printOtherItem (i,j,false,(Cat x)::features,moverArray); *)
                movers.(x) <- (i,j,false,features,moverArray)::cell;
                ((i,j,false,(Cat x)::features,moverArray)::agenda,(nXnNM,nXnSel,movers,other));
              end
      | (Sel x) :: features ->
          (match nXnSel.(i).(j) with
                Z -> let catArray = Array.make (succ catMax) []
                in
                  begin
(*                  Printf.fprintf stdout "adding: "; printOtherItem (i,j,false,(Sel x)::features,moverArray); *)
                    catArray.(x) <- ((false, features, moverArray) :: []);
                    nXnSel.(i).(j) <- B catArray;
                    ((i,j,false,(Sel x)::features,moverArray)::agenda,(nXnNM,nXnSel,movers,other));
                  end
            | B catArray ->
                  let cell = catArray.(x) in
                    if List.mem (false, features, moverArray) cell
                    then (agenda,(nXnNM,nXnSel,movers,other))
                    else
                      begin
(*                      Printf.fprintf stdout "adding: "; printOtherItem (i,j,false,(Sel x)::features,moverArray); *)
                        catArray.(x) <- ((false, features, moverArray) :: catArray.(x));
                        ((i,j,false,(Sel x)::features,moverArray)::agenda,(nXnNM,nXnSel,movers,other));
                      end
          )
      | (Pos x) :: features ->
(*        Printf.fprintf stdout "adding other: "; printOtherItem (i,j,false,(Pos x)::features,moverArray); *)
(*        let newOther = ensureMember (i,j,false,(Pos x)::features,moverArray) other *)
          let newOther = other
          in move (i,j,x,features,moverArray) (agenda,(nXnNM,nXnSel,movers,newOther))
      | (Neg x) :: features -> raise (failwith "update error: Neg result");;

(* for forward rules merge1, merge2, merge3, this function calls update on all matches with Cat items *)
let rec listAll agendaChart (i,k,fs,movers1) cell = match cell with
      [] -> agendaChart;
    | (simple,movers2)::more ->
        let result = moverMergeVal movers1 movers2 (Array.make (succ moverMax) Em) 0 (succ moverMax) in
          if (snd result)
          then listAll (update (i,k,false,fs,(fst result)) agendaChart) (i,k,fs,movers1) more
          else listAll agendaChart (i,k,fs,movers1) more;;

(* MERGE1: search nXnNM(j)(k) up to nXnNM(j)(stop) for matches *)
let rec getall1 (agenda,(nXnNM,nXnSel,movers,other)) (i,fs,movers1) x j k stop =
    if k > stop
    then (agenda,(nXnNM,nXnSel,movers,other))
    else match nXnNM.(j).(k) with
        Y -> getall1 (agenda,(nXnNM,nXnSel,movers,other)) (i,fs,movers1) x j (succ k) stop
      | A array ->
            let (newAgenda,newChart) = listAll (agenda,(nXnNM,nXnSel,movers,other)) (i,k,fs,movers1) array.(x) in
              getall1 (newAgenda,newChart) (i,fs,movers1) x j (succ k) stop;;

(* MERGE2: search nXnNM(i)(j) down to nXnNM(0)(j) for matches *)
let rec getall2 (agenda,(nXnNM,nXnSel,movers,other)) (k,fs,movers1) x i j =
    if i < 0
    then (agenda,(nXnNM,nXnSel,movers,other))
    else match nXnNM.(i).(j) with
        Y -> getall2 (agenda,(nXnNM,nXnSel,movers,other)) (k,fs,movers1) x (pred i) j
      | A array ->
            let (newAgenda,newChart) = listAll (agenda,(nXnNM,nXnSel,movers,other)) (i,k,fs,movers1) array.(x) in
              getall2 (newAgenda,newChart) (k,fs,movers1) x (pred i) j;;

(* MERGE3 *)
let rec getall3 agendaChart (i,j,fs1,movers1) cell = match cell with
      [] -> agendaChart
    | (k,l,simple,(Neg y)::fs2,movers2)::more ->
        if movers1.(y) = Em && movers2.(y)=Em
        then
            let (newMovers,ok) = moverMergeVal movers1 movers2 (Array.make (succ moverMax) Em) 0 (succ moverMax) in
              if ok
              then
                begin
                  newMovers.(y) <- Tu (k,l,fs2);
                  getall3 (update (i,j,false,fs1,newMovers) agendaChart) (i,j,fs1,movers1) more;
                end
              else getall3 agendaChart (i,j,fs1,movers1) more
        else getall3 agendaChart (i,j,fs1,movers1) more
    | _ -> raise (failwith "getall3: error");;

(* for backward rules merge1r, merge2r, this function calls update on all matches with Sel items *)
let rec listAllr agendaChart (i,k,movers1) cell = match cell with
      [] -> agendaChart;
    | (simple,fs,movers2)::more ->
            let result = moverMergeVal movers1 movers2 (Array.make (succ moverMax) Em) 0 (succ moverMax) in
              if (snd result)
              then listAllr (update (i,k,false,fs,(fst result)) agendaChart) (i,k,movers1) more
              else listAllr agendaChart (i,k,movers1) more;;

(* MERGE1r: search nXnSel(i)(j) down to nXnSel(0)(j) for matches *)
let rec getall1r (agenda,(nXnNM,nXnSel,movers,other)) (k,movers1) x i j =
```

```
    if i < 0
    then (agenda,(nXnNM,nXnSel,movers,other))
    else match nXnSel.(i).(j) with
        Z -> getall1r (agenda,(nXnNM,nXnSel,movers,other)) (k,movers1) x (pred i) j
      | B array ->
          let (newAgenda,newChart) = listAll1r (agenda,(nXnNM,nXnSel,movers,other)) (i,k,movers1) array.(x) in
            getall1r (newAgenda,newChart) (k,movers1) x (pred i) j;;

  (* MERGE2r: search nXnNM(j)(k) up to nXnNM(j)(stop) for matches *)
  let rec getall2r (agenda,(nXnNM,nXnSel,movers,other)) (i,movers1) x j k stop =
    if k > stop
    then (agenda,(nXnNM,nXnSel,movers,other))
    else match nXnSel.(j).(k) with
        Z -> getall2r (agenda,(nXnNM,nXnSel,movers,other)) (i,movers1) x j (succ k) stop
      | B array ->
          let (newAgenda,newChart) = listAll1r (agenda,(nXnNM,nXnSel,movers,other)) (i,k,movers1) array.(x) in
                getall2r (newAgenda,newChart) (i,movers1) x j (succ k) stop;;

  (* for backward rule merge3r, this function calls update on all matches with Sel items *)
  (* MERGE3r: category x of moving i,j matches all selector x at k,l in nXnSel(k)(l) *)
  let rec listAll3r agendaChart (i,j,k,l,fs1,movers1) cell = match cell with
      [] -> agendaChart;
    | (simple,fs2,movers2)::more -> (* first, make sure that movers1 and movers2 are compatible *)
        let result = moverMergeVal movers1 movers2 (Array.make (succ moverMax) Em) 0 (succ moverMax) in
            if (snd result)
            then (match fs1 with
                      (Neg y)::morefs1 ->     (* then, put new moving feature into result, if compatible *)
                        if (fst result).(y) = Em
                        then
                          begin
                            (fst result).(y) <- Tu (i,j,morefs1);
                            listAll3r (update (k,l,false,fs2,(fst result)) agendaChart) (i,j,k,l,fs1,movers1) more;
                          end
                        else listAll3r agendaChart (i,j,k,l,fs1,movers1) more
                    | _ -> raise (failwith "listAll3r: invalid feature in licensee sequence")
                  )
            else listAll3r agendaChart (i,j,k,l,fs1,movers1) more;;

  (* MERGE3r: category x of moving i,j finds selector x of k,l in nXnSel(k)(l), for any (non-overlapping) k,l *)
  (* The k<=l restriction to the upper triangle of the matrix is coded by going to (succ k) (succ k) when l > stop *)
  let rec getall3r (agenda,(nXnNM,nXnSel,movers,other)) (i,j,fs,movers1) x k l stop =   (* letter l *)
    if l > stop && k >= stop then (agenda,(nXnNM,nXnSel,movers,other))
    else if l > stop then getall3r (agenda,(nXnNM,nXnSel,movers,other)) (i,j,fs,movers1) x (succ k) (succ k) stop
    else if (l <= i || k >= j)
    then
        (match nXnSel.(k).(l) with
              Z -> getall3r (agenda,(nXnNM,nXnSel,movers,other)) (i,j,fs,movers1) x k (succ l) stop
            | B array ->
                let (newAgenda,newChart) = listAll3r (agenda,(nXnNM,nXnSel,movers,other)) (i,j,k,l,fs,movers1) array.(x) in
                  getall3r (newAgenda,newChart) (i,j,fs,movers1) x k (succ l) stop
        )
    else getall3r (agenda,(nXnNM,nXnSel,movers,other)) (i,j,fs,movers1) x k (succ l) stop;;

  (* MAIN LOOP TO COMPUTE THE CLOSURE OF THE CHART *)
  let rec main stop (agenda,chart) =
    begin
      (* printAgenda agenda;     for tracing, but often produces a lot of output! *)
      (* printChart chart;       for tracing, but often produces a lot of output! *)
      (match agenda with
      [] -> (agenda,chart)
    | (i,j,true,(Sel x)::fs,movers1)::moreAgenda ->  (* merge1, merge3 *)
        let (midAgenda,midChart) = getall1 (moreAgenda,chart) (i,fs,movers1) x j j stop in
        let (newAgenda,newChart) = getall3 (midAgenda,midChart) (i,j,fs,movers1) (thdOf4 chart).(x) in
          main stop (newAgenda,newChart)
    | (j,k,false,(Sel x)::fs,movers1)::moreAgenda ->   (* merge2, merge3 *)
        let (midAgenda,midChart) = getall2 (moreAgenda,chart) (k,fs,movers1) x j j in
            let (newAgenda,newChart) = getall3 (midAgenda,midChart) (j,k,fs,movers1) (thdOf4 chart).(x) in
              main stop (newAgenda,newChart)
    | (j,k,_,(Cat x)::[],movers1)::moreAgenda ->  (* merge1r, merge2r *)
        let (midAgenda,midChart) = getall1r (moreAgenda,chart) (k,movers1) x j j in
        let (newAgenda,newChart) = getall2r (midAgenda,midChart) (j,movers1) x k k stop in
          main stop (newAgenda,newChart)
    | (i,j,_,(Cat x)::fs,movers1)::moreAgenda ->  (* merge3r *)
        let(newAgenda,newChart) = getall3r (moreAgenda,chart) (i,j,fs,movers1) x 0 0 stop in
          main stop (newAgenda,newChart)
    | _ -> raise (failwith "main: invalid agenda item")
      );
    end;;

  let announceSuccess startCategory nXnNM inputLength = match nXnNM.(0).(inputLength) with
      Y -> Printf.fprintf stdout "No consitutent of category %s from %i to %i.\n" (catString startCategory) 0 inputLength
    | A array ->
        if array.(startCategory) <> []
        then Printf.fprintf stdout "There is a consitutent of category %s from %i to %i.\n" (catString startCategory) 0 inputLength
        else Printf.fprintf stdout "No consitutent of category %s from %i to %i.\n" (catString startCategory) 0 inputLength;;

  (* #load "unix.cma";; *)

  let cky lex (startCat,input) =
  (*  let time0 = (Unix.times()).Unix.tms_utime in *)
    let inputLength0 = List.length input in
    let (chart0:chart) =
      ( Array.make_matrix (succ inputLength0) (succ inputLength0) Y,
        Array.make_matrix (succ inputLength0) (succ inputLength0) Z,
        Array.make (succ catMax) [],
        []
      ) in
    let (fstAgenda,fstChart) = emptyInsert lex 0 inputLength0 ([],chart0) lex in
    let (sndAgenda,sndChart) = lexInsert lex (fstAgenda,fstChart) 0 input lex in
    let (newAgenda,newChart) = main inputLength0 (sndAgenda,sndChart) in
  (*  let time1 = (Unix.times()).Unix.tms_utime in *)
      begin
          printChart newChart;
        chartStats newChart;
  (*      Printf.printf " computed in %2.2f ms.\n" ((time1 -. time0)); *)
        announceSuccess startCat (fstOf4 newChart) inputLength0;
      end;;

  let eg g x = cky g (example x);;
```

16.5 Appendix: session

```
        Objective Caml version 3.09.3

# #use "mg-cky.ml";;
type feature = Sel of int | Cat of int | Pos of int | Neg of int
type chain = Em | Tu of (int * int * feature list)
type nmItem = bool * chain array
type selItem = bool * feature list * chain array
type sitItem = int * int * bool * feature list * chain array
type nmCatArray = Y | A of nmItem list array
type selArray = Z | B of selItem list array
type nmCatCoordChart = nmCatArray array array
type selCoordChart = selArray array array
type chart =
    nmCatCoordChart * selCoordChart * sitItem list array * sitItem list
val moverString : int -> string = <fun>
val moverMax : int = 1
val catString : int -> string = <fun>
val catMax : int = 5
val grammar : (string list * feature list) list =
  [([], [Sel 1; Cat 0]); ([], [Sel 1; Pos 1; Cat 0]);
   ([], [Sel 5; Pos 0; Cat 1]); ([], [Sel 4; Sel 2; Cat 5]);
   (["eat"], [Sel 2; Pos 0; Cat 4]); (["laugh"], [Cat 4]);
   (["the"], [Sel 3; Cat 2; Neg 0]);
   (["which"], [Sel 3; Cat 2; Neg 0; Neg 1]); (["king"], [Cat 3]);
   (["pie"], [Cat 3])]
val example : int -> int * string list = <fun>
val fstOf4 : 'a * 'b * 'c * 'd -> 'a = <fun>
val sndOf4 : 'a * 'b * 'c * 'd -> 'b = <fun>
val thdOf4 : 'a * 'b * 'c * 'd -> 'c = <fun>
val fthOf4 : 'a * 'b * 'c * 'd -> 'd = <fun>
val featString : feature -> string = <fun>
val printFeat : feature -> unit = <fun>
val printFeats : feature list -> unit = <fun>
val printStrings : string list -> unit = <fun>
val printLex : (string list * feature list) list -> unit = <fun>
val printChain : bool -> chain -> unit = <fun>
val printChainWithF : bool -> int -> chain -> unit = <fun>
val printNMItem : int -> int -> int -> bool * chain array -> unit = <fun>
val printNMItemList : int -> int -> int -> (bool * chain array) list -> unit =
  <fun>
val printNMCatArray : int -> int -> nmCatArray -> unit = <fun>
val printNMCatCoordChart : nmCatArray array array -> unit = <fun>
val printSelItem :
  int -> int -> int -> bool * feature list * chain array -> unit = <fun>
val printSelItemList :
  int -> int -> int -> (bool * feature list * chain array) list -> unit =
  <fun>
val printSelArray : int -> int -> selArray -> unit = <fun>
val printSelCoordChart : selArray array array -> unit = <fun>
val printMoverItem :
  int -> int * int * bool * feature list * chain array -> unit = <fun>
val printMoverItemList :
  int -> (int * int * bool * feature list * chain array) list -> unit = <fun>
val printMoverArray :
  (int * int * bool * feature list * chain array) list array -> unit = <fun>
val printOtherItem : int * int * bool * feature list * chain array -> unit =
  <fun>
val printOtherItemList :
  (int * int * bool * feature list * chain array) list -> unit = <fun>
val printChart :
  nmCatArray array array * selArray array array *
  (int * int * bool * feature list * chain array) list array *
  (int * int * bool * feature list * chain array) list -> unit = <fun>
val printAgenda :
  (int * int * bool * feature list * chain array) list -> unit = <fun>
val ensureMember : 'a -> 'a list -> 'a list = <fun>
val emptyInsert :
  ('a list * feature list) list ->
  int ->
  int ->
  (int * int * bool * feature list * chain array) list *
  (nmCatArray array array * selArray array array *
   (int * int * bool * feature list * chain array) list array * 'b) ->
  ('a list * feature list) list ->
  (int * int * bool * feature list * chain array) list *
  (nmCatArray array array * selArray array array *
   (int * int * bool * feature list * chain array) list array * 'b) =
  <fun>
val lexInsert :
  ('a list * feature list) list ->
  (int * int * bool * feature list * chain array) list *
  (nmCatArray array array * selArray array array *
   (int * int * bool * feature list * chain array) list array * 'b) ->
  int ->
  'a list ->
  ('a list * feature list) list ->
  (int * int * bool * feature list * chain array) list *
  (nmCatArray array array * selArray array array *
   (int * int * bool * feature list * chain array) list array * 'b) =
  <fun>
val moverMergeVal :
  chain array ->
  chain array -> chain array -> int -> int -> chain array * bool = <fun>
val update :
  int * int * bool * feature list * chain array ->
  (int * int * bool * feature list * chain array) list *
  (nmCatArray array array * selArray array array *
   (int * int * bool * feature list * chain array) list array *
   (int * int * bool * feature list * chain array) list) ->
  (int * int * bool * feature list * chain array) list *
  (nmCatArray array array * selArray array array *
   (int * int * bool * feature list * chain array) list array *
   (int * int * bool * feature list * chain array) list) =
  <fun>
val listAll :
  (int * int * bool * feature list * chain array) list *
  (nmCatArray array array * selArray array array *
   (int * int * bool * feature list * chain array) list array *
   (int * int * bool * feature list * chain array) list) ->
```

```
      int * int * feature list * chain array ->
      ('a * chain array) list ->
      (int * int * bool * feature list * chain array) list *
      (nmCatArray array array * selArray array array *
       (int * int * bool * feature list * chain array) list array *
       (int * int * bool * feature list * chain array) list) =
      <fun>
    val getall1 :
      (int * int * bool * feature list * chain array) list *
      (nmCatArray array array * selArray array array *
       (int * int * bool * feature list * chain array) list array *
       (int * int * bool * feature list * chain array) list) ->
      int * feature list * chain array ->
      int ->
      int ->
      int ->
      int ->
      (int * int * bool * feature list * chain array) list *
      (nmCatArray array array * selArray array array *
       (int * int * bool * feature list * chain array) list array *
       (int * int * bool * feature list * chain array) list) =
      <fun>
    val getall2 :
      (int * int * bool * feature list * chain array) list *
      (nmCatArray array array * selArray array array *
       (int * int * bool * feature list * chain array) list array *
       (int * int * bool * feature list * chain array) list) ->
      int * feature list * chain array ->
      int ->
      int ->
      int ->
      (int * int * bool * feature list * chain array) list *
      (nmCatArray array array * selArray array array *
       (int * int * bool * feature list * chain array) list array *
       (int * int * bool * feature list * chain array) list) =
      <fun>
    val getall3 :
      (int * int * bool * feature list * chain array) list *
      (nmCatArray array array * selArray array array *
       (int * int * bool * feature list * chain array) list array *
       (int * int * bool * feature list * chain array) list) ->
      int * int * feature list * chain array ->
      (int * int * 'a * feature list * chain array) list ->
      (int * int * bool * feature list * chain array) list *
      (nmCatArray array array * selArray array array *
       (int * int * bool * feature list * chain array) list array *
       (int * int * bool * feature list * chain array) list) =
      <fun>
    val listAll1r :
      (int * int * bool * feature list * chain array) list *
      (nmCatArray array array * selArray array array *
       (int * int * bool * feature list * chain array) list array *
       (int * int * bool * feature list * chain array) list) ->
      int * int * chain array ->
      ('a * feature list * chain array) list ->
      (int * int * bool * feature list * chain array) list *
      (nmCatArray array array * selArray array array *
       (int * int * bool * feature list * chain array) list array *
       (int * int * bool * feature list * chain array) list) =
      <fun>
    val getall1r :
      (int * int * bool * feature list * chain array) list *
      (nmCatArray array array * selArray array array *
       (int * int * bool * feature list * chain array) list array *
       (int * int * bool * feature list * chain array) list) ->
      int * chain array ->
      int ->
      int ->
      int ->
      (int * int * bool * feature list * chain array) list *
      (nmCatArray array array * selArray array array *
       (int * int * bool * feature list * chain array) list array *
       (int * int * bool * feature list * chain array) list) =
      <fun>
    val getall2r :
      (int * int * bool * feature list * chain array) list *
      (nmCatArray array array * selArray array array *
       (int * int * bool * feature list * chain array) list array *
       (int * int * bool * feature list * chain array) list) ->
      int * chain array ->
      int ->
      int ->
      int ->
      int ->
      (int * int * bool * feature list * chain array) list *
      (nmCatArray array array * selArray array array *
       (int * int * bool * feature list * chain array) list array *
       (int * int * bool * feature list * chain array) list) =
      <fun>
    val listAll3r :
      (int * int * bool * feature list * chain array) list *
      (nmCatArray array array * selArray array array *
       (int * int * bool * feature list * chain array) list array *
       (int * int * bool * feature list * chain array) list) ->
      int * int * int * int * feature list * chain array ->
      ('a * feature list * chain array) list ->
      (int * int * bool * feature list * chain array) list *
      (nmCatArray array array * selArray array array *
       (int * int * bool * feature list * chain array) list array *
       (int * int * bool * feature list * chain array) list) =
      <fun>
    val getall3r :
      (int * int * bool * feature list * chain array) list *
      (nmCatArray array array * selArray array array *
       (int * int * bool * feature list * chain array) list array *
       (int * int * bool * feature list * chain array) list) ->
      int * int * feature list * chain array ->
      int ->
      int ->
      int ->
      int ->
      (int * int * bool * feature list * chain array) list *
```

```
    (nmCatArray array array * selArray array array *
     (int * int * bool * feature list * chain array) list array *
     (int * int * bool * feature list * chain array) list) =
    <fun>
  val main :
    int ->
    (int * int * bool * feature list * chain array) list *
    (nmCatArray array array * selArray array array *
     (int * int * bool * feature list * chain array) list array *
     (int * int * bool * feature list * chain array) list) ->
    (int * int * bool * feature list * chain array) list *
    (nmCatArray array array * selArray array array *
     (int * int * bool * feature list * chain array) list array *
     (int * int * bool * feature list * chain array) list) =
    <fun>
  val announceSuccess : int -> nmCatArray array array -> int -> unit = <fun>
  val cky : ('a list * feature list) list -> int * 'a list -> unit = <fun>
  val eg : (string list * feature list) list -> int -> unit = <fun>

  # eg grammar 1;;
  adding: (2,3): =D v
  adding: (0,2): D -k
  adding: (2,3): v (0,2): -k
  adding other: (2,3): +k T (0,2): -k
  adding other: (2,3): +k T (0,2): -k
  adding other: (2,3): +k T (0,2): -k
  adding other: (0,3): +wh C
  adding: (0,3): C
  (0,3): C
  (0,3): T
  (1,2):: N
  (2,3):: V
  (2,3): v (0,2): -k
  (0,0):: =T +wh C
  (0,0):: =T C
  (0,0):: =V =D v
  (0,0):: =v +k T
  (0,1):: =N D -k
  (1,1):: =T +wh C
  (1,1):: =T C
  (1,1):: =V =D v
  (1,1):: =v +k T
  (2,2):: =T +wh C
  (2,2):: =T C
  (2,2):: =V =D v
  (2,2):: =v +k T
  (2,3): =D v
  (3,3):: =T +wh C
  (3,3):: =T C
  (3,3):: =V =D v
  (3,3):: =v +k T
  (0,2): D -k
  (2,3): +k T (0,2): -k
  (0,3): +wh C
  There is a consitutent of category C from 0 to 3.
  - : unit = ()

  # eg grammar (-1);;
  adding: (4,5): =D v
  adding: (2,4): D -k
  adding: (4,5): v (2,4): -k
  adding other: (4,5): +k T (2,4): -k
  adding other: (4,5): +k T (2,4): -k
  adding: (0,2): D -k
  adding: (1,3): D -k
  adding: (4,5): v (1,3): -k
  adding other: (4,5): +k T (1,3): -k
  adding other: (4,5): +k T (1,3): -k
  adding: (4,5): v (0,2): -k
  adding other: (4,5): +k T (0,2): -k
  adding other: (4,5): +k T (0,2): -k
  adding other: (4,5): +k T (0,2): -k
  adding other: (4,5): +k T (1,3): -k
  adding other: (4,5): +k T (2,4): -k
  adding other: (2,5): +wh C
  adding: (2,5): C
  (1,2):: N
  (2,5): C
  (2,5): T
  (3,4):: N
  (4,5):: V
  (4,5): v (0,2): -k
  (4,5): v (1,3): -k
  (4,5): v (2,4): -k
  (0,0):: =T +wh C
  (0,0):: =T C
  (0,0):: =V =D v
  (0,0):: =v +k T
  (0,1):: =N D -k
  (1,1):: =T +wh C
  (1,1):: =T C
  (1,1):: =V =D v
  (1,1):: =v +k T
  (2,2):: =T +wh C
  (2,2):: =T C
  (2,2):: =V =D v
  (2,2):: =v +k T
  (2,3):: =N D -k
  (3,3):: =T +wh C
  (3,3):: =T C
  (3,3):: =V =D v
  (3,3):: =v +k T
  (4,4):: =T +wh C
  (4,4):: =T C
  (4,4):: =V =D v
  (4,4):: =v +k T
  (4,5): =D v
  (5,5):: =T +wh C
  (5,5):: =T C
  (5,5):: =V =D v
  (5,5):: =v +k T
  (1,3): D -k
```

```
(0,2): D -k
(2,4): D -k
(4,5): +k T (2,4): -k
(4,5): +k T (1,3): -k
(4,5): +k T (0,2): -k
(2,5): +wh C
No consitutent of category C from 0 to 5.
- : unit = ()
#

Process caml-toplevel finished
```

§17 Semantics in the computational model

17.0 The interface with reasoning: first ideas

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 1986 [48, p68])

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. 1980 [85])

...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 – from the regularities governing our deployment of the sentences in which it appears... understanding a sentence consists, by definition, in nothing over and above understanding its constituents and appreciating how they are combined with one another. 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. If this is so, then compositionality is a trivial consequence of what we mean by "understanding" in connection with complex sentences. (Horwich 1998 [123, pp3,9])

17.1 Disambiguation in comprehension

(0) **The nature of the problem:** Disambiguation requires 'creativity', mentioned on page 5...

Hans Kamp agrees with Barbara Partee on the difficulty of determining pronoun reference:
"The strategies used in selecting the referents of anaphoric pronouns are notoriously complex... Efforts to understand these strategies have claimed much thought and hard work, but, in its general form at least, the problem appears to be much to complex to permit solution with the limited analytical tools available at the present time... Indeed, I incline to the opinion expressed, for example, in Partee [197, p.80], that all we can reasonably expect to achieve in this area is to articulate orders of preference among the potential referents of an anaphoric pronoun, without implying that the item which recieves the highest rating is in each and every case the referent of the pronoun." [133, p.39]

Hilary Putnam agrees with Jerry Fodor on the difficulty of deciding sameness of meaning:
"In short, it looks like the problem of mechanizing – 'computationalizing' – the notion of sameness of reference judgements, and hence sameness of meaning judgements, cannot be carried out without carrying out what [Jerry Fodor], I think rightly, calls the too Utopian program of computationalizing scientific explanation, the principle of abduction, or inference to the best explanation. There is an intimate link between the question of what is and is not the best explanation and the question 'Is he referring to the same thing?' or 'Has he stopped referring to the same thing?'." [211, p.222]

Why would a theory of ambiguity resolution be difficult to formulate?

A1: What an expression really refers to, or really means, is a normative question. This idea defended in the philosophical literature [64, 211, 70, 175]. Cf. [46, p.9], [86, pp.4-5] on 'cognizing'

A2: What a given individual in a given setting thinks an expression means does not depend on a delimited domain, as does, e.g., what syntactic structures the expression has. Fodor [86] defends this view, saying that syntax is 'informationally encapsulated' in the way perceptual systems are, and in the way general reasoning about matters of fact is not. Cf. Chomsky [48, p.14n.10], and critiques below

(1) **Psychological evidence of influences on (and rates of) ambiguity resolution:**

a. **Lexical priming** [65, 60, many others]

b. **Structural preferences on 'first pass'** [88, 91, many others]

c. **Pragmatics, discourse** [37, many others]

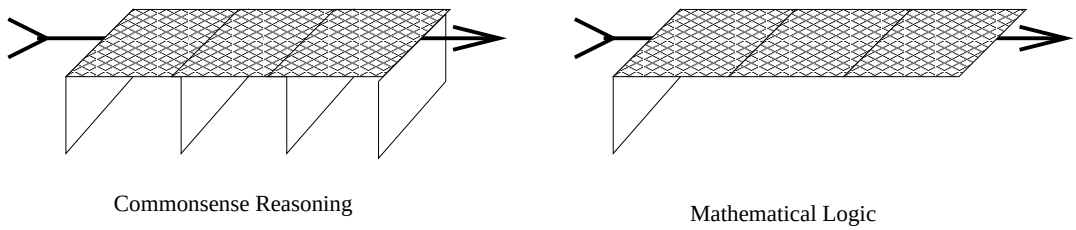
d. **Incremental interpretation** [62, 212, 233, 235, many others].

(2) **How 1 and 2 could be compatible?**

Long-standing controversy: Show lexical priming not 'cognitively penetrable', so distinct from general reasoning. [169, many others] Similarly, discourse influences do not eliminate 'garden paths' etc [217, many others]

Two basic ideas to allow (0) & (1) together:

first Reasoning is shallow. [180, 4].



This figure adapted from Minsky 1988 [180]

second Reasoning is narrow and task-specialized [34, 13] That is, there is a rather strict limit to the range of probabilistic dependencies that we keep track of at any one time. This is the essential idea behind Minksy'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 adjustments in support through the relevant propositions. This is also the basic idea behind Bayesian nets [198, 63].

17.2 Recognizing entailments

natural logics [25, 210, 229, 96, 184]

For example, trivially we judge pretheoretically that 2b below is true whenever 2a is.

2a. John is a linguist and Mary is a biologist.
2b. John is a linguist.

Thus, given that 2a,b lie in the fragment of English we intend to represent, it follows that our system would be descriptively inadequate if we could not show that our representation for 2a formally entailed our representation of 2b. (Keenan and Faltz [143, p2])

$$
\begin{array}{rll}
[\![\text{every}]\!] & = \{\langle p,q\rangle \mid p \subseteq q\} & \\
[\![\text{some}]\!] & = \{\langle p,q\rangle \mid (p \cap q) \neq \emptyset\} & \\
[\![\text{no}]\!] & = \{\langle p,q\rangle \mid (p \cap q) = \emptyset\} & \\
[\![\text{exactly N}]\!] & = \{\langle p,q\rangle \mid |p \cap q| = N\} & \text{for any } N \in \mathbb{N} \\
[\![\text{at least N}]\!] & = \{\langle p,q\rangle \mid |p \cap q| \geq N\} & \text{for any } N \in \mathbb{N} \\
[\![\text{at most N}]\!] & = \{\langle p,q\rangle \mid |p \cap q| \leq N\} & \text{for any } N \in \mathbb{N} \\
[\![\text{all but N}]\!] & = \{\langle p,q\rangle \mid |p - q| = N\} & \text{for any } N \in \mathbb{N} \\
[\![\text{between N and M}]\!] & = \{\langle p,q\rangle \mid N \leq |p \cap q| \leq M\} & \text{for any } N, M \in \mathbb{N} \\
[\![\text{most}]\!] & = \{\langle p,q\rangle \mid |p - q| > |p \cap b|\} & \\
[\![\text{the N}]\!] & = \{\langle p,q\rangle \mid |p - q| = 0 \text{ and } |p \cap q| = N\} & \text{for any } N \in \mathbb{N}
\end{array}
$$

For any binary quantifier Q we use $\uparrow Q$ to indicate that Q is (monotone) **increasing in its first argument**, which means that whenever $\langle p,q\rangle \in Q$ and $r \supseteq p$ then $\langle r,q\rangle \in Q$. Examples are *some* and *at least N*.

For any binary quantifier Q we use $Q \uparrow$ to indicate that $Q\ Q$ is (monotone) **increasing in its second argument** iff whenever $\langle p,q\rangle \in Q$ and $r \supseteq q$ then $\langle p,r\rangle \in Q$. Examples are *every, most, at least N, the, infinitely many,...*.

For any binary quantifier Q we use $\downarrow Q$ to indicate that Q is (monotone) **decreasing in its first argument**, which means that whenever $\langle p,q\rangle \in Q$ and $r \subseteq p$ then $\langle r,q\rangle \in Q$. Examples are *every, no, all, at most N,...*

For any binary quantifier Q we use $Q \downarrow$ to indicate that $Q\ Q$ is (monotone) **decreasing in its second argument** iff whenever $\langle p,q\rangle \in Q$ and $r \subseteq q$ then $\langle p,r\rangle \in Q$. Examples are *no, few, fewer than N, at most N,...*.

Since *every* is decreasing in its first argument and increasing in its second argument, we sometimes write $\downarrow$every $\uparrow$. Similarly, $\downarrow$no $\downarrow$, and $\uparrow$ some $\downarrow$.

It is now easy to represent sound patterns of inference for different kinds of quantifiers.

$$\frac{\texttt{B(Q(A))} \qquad \texttt{C(every(B))}}{\texttt{C(Q(A))}}[\text{Q}\uparrow] \quad \text{(for any Q}\uparrow\text{: all, most, the, at least N, infinitely many,...)}$$

$$\frac{\texttt{B(Q(A))} \qquad \texttt{B(every(C))}}{\texttt{C(Q(A))}}[Q\downarrow] \quad \text{(for any Q}\downarrow\text{: no, few, fewer than N, at most N,...)}$$

$$\frac{\texttt{B(Q(A))} \qquad \texttt{C(every(A))}}{\texttt{B(Q(C))}}[\uparrow Q] \quad \text{(for any }\uparrow\text{Q: some, at least N, ...)}$$

$$\frac{\texttt{B(Q(A))} \qquad \texttt{A(every(C))}}{\texttt{B(Q(C))}}[\downarrow Q] \quad \text{(for any }\downarrow\text{Q: no, every, all, at most N, at most finitely many,...)}$$

Example: Aristotle noticed that "Darii syllogisms" like the following are sound:

$$\frac{\text{Some birds are swans} \qquad \text{All swans are white}}{\text{Therefore, some birds are white}}$$

We can recognize this now as one instance of the $Q \uparrow$ rule:

$$\frac{\texttt{birds(some(swans))} \qquad \texttt{white(every(swan))}}{\texttt{white(some(birds))}} \quad [Q\uparrow]$$

The second premise says that the step from the property $[\![\text{bird}]\!]$ to $[\![\text{white}]\!]$ is an "increase," and since we know *some* is increasing in its second argument, the step from the first premise to the conclusion always preserves truth.

We could continue to develop a logic in this way (compare [96, 17, 229, 210]), and we could consider various ways to get alternative quantifier scopes (e.g. by covertly changing the quantifier positions, or by introducing more complex inference rules), but let's now turn our attention to aspects of meaning that are not explicitly given by our syntactic structures.

sorted and description logics [94, 154, 268, 186]

17.3 Recognizing probable consequences

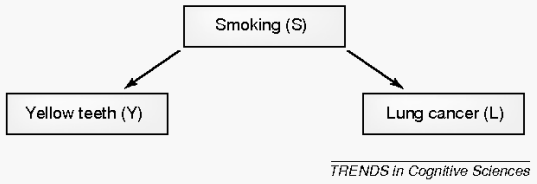
general reasoning with Bayesian models? [256, 198, 63, many others][1]

Problem 0: Bayesian networks are propositional, and so miss general relations among propositions. And on the other hand, it is not clear how to update a first order database, since even equivalence of first order formulas is undecidable, and even in the propositional case equivalence is intractable [189, 116, 195]

A standard response to this problem is to abandon the untenable assumption that human beliefs are consistent with respect to logical equivalence ("logical omniscience"), and instead associate degrees of belief with particular formulas, with the possibility of propogating belief via inferential steps [14]. This still leaves the intractable problem of conditioning general databases with new information [136]. One standard response to this problem is to calculate approximations [153]; another response is...

Problem 1: In pursuit of tractability, Bayesian models stipulate limited propositional domains with the Markov assumption. [104, 103]

Box 1. Bayes nets, the Markov assumption and conditional independence

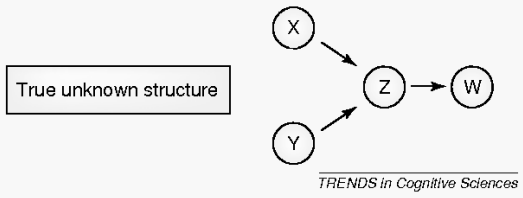


The graph above represents the claim that smoking is a cause of yellowed teeth and lung cancer, but that lung cancer does not cause yellowed teeth and yellowed teeth do not cause lung cancer. It also represents claims about the conditional probability relations among the three variables: for all values 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 = present|L = absent, S = 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



Bayesian learning

(1) Prior probability distribution Pr(G; θ) 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, θ) 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) ≥ Pr(G*|D)

These diagrams are from Glymour 2003 [104]

A standard response to this problem is that, while there is no principled limitation on the range of facts brought to bear on a hypothesis in science, in quick, human assessments of plausibility, there are of course limitations [34, 13]. But precisely defining those limitations remains an open problem.

Problem 2: Bayesian models provide no account of how new terms can be introduced to provide better, simpler, more believable models [145, 77]. Even in commonsense situations, humans can think of new kinds of possible explanations of the data, and we know very little about how this is done (... an important topic of learnability theory).

[1] There are many computational studies of this framework, including an Ocaml extension called **IBAL** that uses (recursive functions defined over) Bayesian networks to compute distributions over the possible values of probabilistic functions [202, 215, 203, 155].

Bibliography

[1] Åfarli, T. A. A promotion analysis of restrictive relative clauses. *The Linguistic Review 11* (1994), 81–100.

[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] Albro, D. M. *Studies in Computational Optimality Theory, with Special Reference to the Phonological System of Malagasy.* PhD thesis, UCLA, 2005.

[4] Alechina, N., and Ågnotes, T. The dynamics of syntactic knowledge. *Journal of Logic and Computation 17* (2006), 83–116.

[5] Allauzen, C., and Mohri, M. 3-way composition of weighted finite state transducers. *ArXiv.0802.1465v1* (2008).

[6] 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.

[7] Altmann, G. T. M., and Steedman, M. Interaction with context during human sentence processing. *Cognition 30* (1988), 191–238.

[8] Altmann, G. T. M., van Nice, K. Y., Garnham, A., and Henstra, J.-A. Late closure in context. *Journal of Memory and Language 38* (1998), 459–484.

[9] Amblard, M. Treating the semantics of French clitics with MG. In *Workshop on the Logic of Variation, Utrecht University* (2006). Slides available at http://www.labri.fr/perso/amblard/.

[10] Asudeh, A., and Toivonen, I. Perception and uniqueness: Evidence from English and Swedish copy raising. University of Canterbury, New Zealand, 2004.

[11] Baayen, H., and del Prado Martin, F. M. Semantic density and past-tense formation in three Germanic languages. *Language 81*, 3 (2005), 666–698.

[12] Bach, E. Questions. *Linguistic Inquiry 2* (1971), 153–166.

[13] Baker, A. G., Mercier, P., Vallée-Tourangeau, F., Frank, R., and Pan, M. Selective associations and causality judgements: Presence of a strong factor may reduce judgements of a weaker one. *Journal of Experimental Psychology: Learning, Memory and Cognition 19*, 2 (1993), 414–432.

[14] Barnett, O., and Paris, J. Maximum entropy inference with quantified knowledge. *Logic Journal of the Interest Group in Pure and Applied Logic 16* (2007), 85–98.

[15] Barss, A., and Lasnik, H. A note on anaphora and double objects. *Linguistic Inquiry 17* (1986), 347–354.

[16] Beesley, K. R., and Karttunen, L. *Finite State Morphology.* CSLI Publications, Stanford, California, 2003.

[17] Bernardi, R. *Reasoning with Polarity in Categorial Type Logic.* PhD thesis, University of Utrecht, Utrecht, 2002.

[18] Berwick, R. C., and Weinberg, A. S. *The Grammatical Basis of Linguistic Performance: Language Use and Acquisition.* MIT Press, Cambridge, Massachusetts, 1984.

[19] Bhatt, R. Adjectival modifiers and the raising analysis of relative clauses. In *Proceedings of the North Eastern Linguistic Society, NELS 30* (1999).

[20] Bhatt, R., and Joshi, A. Semilinearity is a syntactic invariant: a reply to Michaelis and Kracht. *Linguistic Inquiry 35* (2004), 683–692.

[21] Bikel, D. M. Intricacies of Collins' parsing model. *Computational Linguistics 30*, 4 (2004), 479–511.

[22] Bloomfield, L. *Language*. University of Chicago Press, Chicago, 1933.

[23] Booth, T. L., and Thompson, R. A. Applying probability measures to abstract languages. *IEEE Transactions on Computers C-22* (1973), 442–450.

[24] Borsley, R. D. Relative clauses and the theory of phrase structure. *Linguistic Inquiry 28* (1997), 629–647.

[25] Braine, M. The 'natural logic' approach to reasoning. In *Reasoning, Necessity and Logic*, W. Overton, Ed. Erlbaum, Hillsdale, NJ, 1990.

[26] Bresnan, J. Control and complementation. In *The Mental Representation of Grammatical Relations*, J. Bresnan, Ed. MIT Press, Cambridge, Massachusetts, 1982.

[27] Bresnan, J., Kaplan, R. M., Peters, S., and Zaenen, A. Cross-serial dependencies in Dutch. *Linguistic Inquiry 13*, 4 (1982), 613–635.

[28] Brody, M. *Lexico-Logical Form: A Radically Minimalist Theory*. MIT Press, Cambridge, Massachusetts, 1995.

[29] Brody, M. Mirror theory: a brief sketch. In *A Celebration*. MIT Press, Cambridge, Massachusetts, 1999.

[30] Brody, M. *Towards an Elegant Syntax*. Routledge, NY, 2003.

[31] Brosgol, B. M. *Deterministic Translation Grammars*. PhD thesis, Harvard University, 1974.

[32] Buckley, E. Integrity and correspondence in Manam double reduplication. In *Proceedings of the North Eastern Linguistic Society, NELS 27* (1997).

[33] Buckwalter, T. Buckwalter Arabic morphological analyzer version 1.0. In *Linguistic Data Consortium LDC2002L49*. University of Pennsylvania, 2002.

[34] Buehner, M. J., and Cheng, P. W. Causal learning. In *The Cambridge Handbook of Thinking and Reasoning*, K. J. Holyoak and R. G. Morrison, Eds. Cambridge University Press, NY, 2005.

[35] Buell, L. Swahili relative clauses. UCLA M.A. thesis, 2000.

[36] Cable, S. Q-particles and the nature of wh-fronting: Evidence from Tlingit. MIT manuscript, UCLA handout, 2007.

[37] Chambers, C. G., Tanenhaus, M. K., Eberhard, K. M., Filip, H., and Carlson, G. N. Circumscribing referential domains during real-time language comprehension. *Journal of Memory and Language 47* (2002), 30–49.

[38] Charniak, E., Goldwater, S., and Johnson, M. Edge-based best-first chart parsing. In *Proceedings of the Workshop on Very Large Corpora* (1998).

[39] Chen, S., and Goodman, J. An empirical study of smoothing techniques for language modeling. Tech. Rep. TR-10-98, Harvard University, Cambridge, Massachusetts, 1998.

[40] Chi, Z. Statistical properties of probabilistic context free grammars. *Computational Linguistics 25* (1999), 130–160.

[41] Chi, Z., and Geman, S. Estimation of probabilistic context free grammars. *Computational Linguistics 24* (1998), 299–306.

[42] Chomsky, N. Three models for the description of language. *IRE Transactions on Information Theory IT-2* (1956), 113–124.

[43] Chomsky, N. *Syntactic Structures.* Mouton, The Hague, 1957.

[44] Chomsky, N. Formal properties of grammars. In *Handbook of Mathematical Psychology, Volume II*, R. D. Luce, R. R. Bush, and E. Galanter, Eds. Wiley, NY, 1963, pp. 323–418.

[45] Chomsky, N. *Aspects of the Theory of Syntax.* MIT Press, Cambridge, Massachusetts, 1965.

[46] Chomsky, N. *Rules and Representations.* Columbia University Press, NY, 1980.

[47] Chomsky, N. *Lectures on Government and Binding.* Foris, Dordrecht, 1981.

[48] Chomsky, N. *Knowledge of Language.* Praeger, NY, 1986.

[49] Chomsky, N. A minimalist program for linguistic theory. In *The View from Building 20*, K. Hale and S. J. Keyser, Eds. MIT Press, Cambridge, Massachusetts, 1993.

[50] Chomsky, N. *The Minimalist Program.* MIT Press, Cambridge, Massachusetts, 1995.

[51] Chomsky, N. Minimalist inquiries: The framework. In *Step by Step: Essays on Minimalism in Honor of Howard Lasnik*, R. Martin, D. Michaels, and J. Uriagereka, Eds. MIT Press, Cambridge, Massachusetts, 2000, pp. 89–155.

[52] Chomsky, N. Three factors in language design. *Linguistic Inquiry 36*, 1 (2005).

[53] Chomsky, N., and Halle, M. *The Sound Pattern of English.* MIT Press, Cambridge, Massachusetts, 1968.

[54] 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.

[55] Cinque, G. *Adverbs and Functional Heads : A Cross-Linguistic Perspective.* Oxford University Press, Oxford, 1999.

[56] Cinque, G. The status of 'mobile' suffixes. In *Aspects of Typology and Universals*, W. Bisang, Ed. Akademie Verlag, Berlin, 2001, pp. 13–19.

[57] Collins, C. *Local Economy.* MIT Press, Cambridge, Massachusetts, 1997.

[58] Collins, M. *Head-driven statistical models for natural language parsing.* PhD thesis, University of Pennsylvania, Philadelphia, 1999.

[59] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms.* MIT Press, Cambridge, Massachusetts, 1991.

[60] Cronk, B. C. Phonological, semantic, and repetition priming with homophones. *Journal of Psycholinguistic Research 30*, 4 (2001), 365–378.

[61] Culy, C. The complexity of the vocabulary of Bambara. *Linguistics and Philosophy 8*, 3 (1985), 345–352.

[62] Dahan, D., and Tanenhaus, M. K. Continuous mapping from sound to meaning in spoken-language comprehension: Immediate effects of verb-based thematic constraints. *Journal of Experimental Psychology: Learning, Memory, and Cognition 30*, 2 (2004), 498–513.

[63] 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.

[64] Davidson, D. On the very idea of a conceptual scheme. *Proceedings and Addresses of the American Philosophical Association 47* (1974), 5–20. Reprinted in D. Davidson, *Inquiries into Truth and Interpretation*, Oxford: Clarendon Press, 2001.

[65] De Groot, A. M. B. The priming of word associations: A levels of processing approach. *Quarterly Journal of Experimental Psychology Human Experimental Psychology 39* (1987), 721–756.

[66] De Groot, A. M. B. Representational aspects of word imageability and word frequency as assessed through word association. *Journal of Experimental Psychology: Learning, Memory, and Cognition 15* (1989), 824–845.

[67] De Mori, R., Galler, M., and Brugnara, F. Search and learning strategies for improving hidden Markov models. *Computer Speech and Language 9* (1995), 107–121.

[68] 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.

[69] Deng, L., and Rathinavelu, C. A Markov model containing state-conditioned second-order non-stationarity: application to speech recognition. *Computer Speech and Language 9* (1995), 63–86.

[70] Dennett, D. *The Intentional Stance.* MIT Press, Cambridge, Massachusetts, 1989.

[71] Dimitrova-Vulchanova, M., and Giusti, G. Fragments of Balkan nominal structure. In *Possessors, Predicates and Movement in the Determiner Phrase*, A. Alexiadou and C. Wilder, Eds. Amsterdam, Philadelphia, 1998.

[72] Drake, A. W. *Fundamentals of Applied Probability Theory.* McGraw-Hill, NY, 1967.

[73] Dudas, K. *The Phonology and Morphology of Modern Japanese.* PhD thesis, University of Illinois, 1968.

[74] Dupont, P., Denis, F., and Esposito, Y. Links between probabilisitic automata and hidden Markov models. *Pattern Recognition 38* (2005), 1349–1371.

[75] Earley, J. *An Efficient Context-Free Parsing Algorithm.* PhD thesis, Carnegie-Mellon University, 1968.

[76] Earley, J. An efficient context-free parsing algorithm. *Communications of the Association for Computing Machinery 13* (1970), 94–102.

[77] Earman, J. *Bayes or Bust? A Critical Examination of Bayesian Confirmation Theory.* MIT Press, Cambridge, Massachusetts, 1992.

[78] Eisner, J., and Satta, G. Efficient parsing for bilexical context-free grammars and head automaton grammars. In *Proceedings of the 37th Annual Meeting, ACL'99* (1999), Association for Computational Linguistics.

[79] Ellis, C. A. *Probabilistic Languages and Automata.* PhD thesis, University of Illinois, 1970.

[80] Fabb, N. English suffixation is constrained only by selection restrictions. *Linguistics and Philosophy 6* (1988), 527–539.

[81] Fanselow, G., and Mahajan, A. Toward a minimalist theory of wh-expletives, wh-copying and successive cyclicity. In *Wh-Scope Marking*, U. Lutz, G. Müller, and von Stechow, Eds. John Benjamins, NY, 2000, pp. 195–230.

[82] Fedorenko, E., Gibson, E., and Rohde, D. The nature of working memory in linguistic, arithmetic and spatial integration processes. *Journal of Memory and Language 56* (2007), 246–269.

[83] Fiengo, R., and May, R. *Indices and Identity.* MIT Press, Cambridge, Massachusetts, 1995.

[84] Fine, K. Transparent grammars. In *Logic from Computer Science*, Y. N. Moschovakis, Ed. Springer-Verlag, NY, 1992, pp. 129–151.

[85] Fodor, J., Garrett, M., Walker, E., and Parkes, C. Against definitions. *Cognition 8* (1980), 263–367.

[86] Fodor, J. A. *The Modularity of Mind: A Monograph on Faculty Psychology.* MIT Press, Cambridge, Massachusetts, 1983.

[87] Forney, G. D. The Viterbi algorithm. *Proceedings of the IEEE 61* (1973), 268–278.

[88] Frazier, L. *On Comprehending Sentences: Syntactic Parsing Strategies.* PhD thesis, University of Massachusetts, Amherst, 1978.

[89] Frazier, L. Syntactic complexity. In *Natural Language Parsing*, D. Dowty, L. Karttunen, and A. Zwicky, Eds. Cambridge University Press, NY, 1985.

[90] Frazier, L., and Fodor, J. D. The sausage machine: a new two-stage parsing model. *Cognition 6* (1979), 291–325.

[91] Frazier, L., and Rayner, K. Making and correcting errors during sentence comprehension. *Cognitive Psychology 14* (1982), 178–210.

[92] 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.

[93] Frey, W., and Gärtner, H.-M. On the treatment of scrambling and adjunction in minimalist grammars. In *Proceedings, Formal Grammar'02* (Trento, 2002).

[94] Frisch, A. M. A general framework for sorted deduction. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning, KR'89* (San Mateo, California, 1989), R. Brachman, H. Levesque, and R. Reiter, Eds., Morgan Kaufmann.

[95] Fromkin, V., Ed. *Linguistics: An Introduction to Linguistic Theory.* Blackwell, Oxford, 2000.

[96] Fyodorov, Y., Winter, Y., and Francez, N. Order-based inference in natural logic. *Research on Language and Computation* (2003). Forthcoming.

[97] 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.

[98] 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.

[99] Ghomeshi, J., Jackendoff, R., Rosen, N., and Russell, K. Contrastive focus reduplication in English. *Natural Language and Linguistic Theory 22*, 2 (2004), 307–357.

[100] Gibson, E. *A Computational Theory of Human Linguistic Processing.* PhD thesis, Carnegie-Mellon University, 1991.

[101] Gibson, E. Linguistic complexity: Locality of syntactic dependencies. *Cognition 68* (1998), 1–76.

[102] Gleason, J. B. The child's learning of English morphology. *Word 14* (1958), 150–177.

[103] Glymour, C. Android epistemology for babies: Reflections on words, thoughts and theories. *Synthese 122* (2000), 53–68.

[104] Glymour, C. Learning, prediction and causal Bayes Nets. *Trends in Cognitive Science 7*, 1 (2003), 43–47.

[105] Gonzalez, R. C., and Thomason, M. G. *Syntactic Pattern Recognition.* Addison-Wesley, London, 1978.

[106] Greenberg, J. Some universals of grammar with particular reference to the order of meaningful elements. In *Universals of Human Language*, J. Greenberg, Ed. Stanford University Press, Stanford, California, 1978.

[107] Grenander, U. Syntax-controlled probabilities. Tech. rep., Brown University, Providence, Rhode Island, 1967.

[108] Guillaumin, M. Conversions between mildly sensitive grammars. UCLA and École Normale Supérieure. http://www.linguistics.ucla.edu/people/stabler/epssw.htm, 2004.

[109] Hale, J. *Grammar, Uncertainty, and Sentence Processing.* PhD thesis, Johns Hopkins University, 2003.

[110] Hardt, D. *Verb Phrase Ellipsis: Form, Meaning and Processing.* PhD thesis, University of Pennsylvania, 1993.

[111] Harkema, H. A recognizer for minimalist grammars. In *Sixth International Workshop on Parsing Technologies, IWPT'00* (2000).

[112] Harkema, H. *Parsing Minimalist Languages.* PhD thesis, University of California, Los Angeles, 2001.

[113] Harris, T. E. On chains of infinite order. *Pacific Journal of Mathematics 5* (1955), 707–724.

[114] Harris, T. E. *The theory of branching processes.* Springer, Berlin, 1963.

[115] Hay, J., and Baayen, R. Shifting paradigms: gradient structure in morphology. *Trends in Cognitive Sciences 9* (2005), 342–348.

[116] Hill, M., Paris, J., and Wilmers, G. Some observations on induction in predicate probabilistic reasoning. *Journal of Philosophical Logic 31* (2001), 43–75.

[117] Hoekstra, T. Parallels between nominal and verbal projections. In *Specifiers: Minimalist Approaches*, D. Adger, S. Pintzuk, B. Plunkett, and G. Tsoulas, Eds. Oxford University Press, Oxford, 1999, pp. 163–187.

[118] Hopcroft, J. E., Motwani, R., and Ullman, J. D. *Introduction to Automata Theory, Languages and Computation (2nd Edition).* Addison-Wesley, Reading, Massachusetts, 2000.

[119] Hopcroft, J. E., and Ullman, J. D. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[120] Hornby, A., Cowie, A., and Lewis, J. W. *Oxford Advanced Learner's Dictionary of Current English*, 3 ed. Oxford University Press, London, 1974.

[121] Hornstein, N. Movement and control. *Linguistic Inquiry 30* (1999), 69–96.

[122] Horvath, J. On the syntax of 'wh-scope marker' constructions: some comparative evidence. In *Wh-Scope Marking*, U. Lutz, G. Müller, and von Stechow, Eds. John Benjamins, NY, 2000, pp. 271–316.

[123] Horwich, P. *Meaning.* Oxford University Press, Oxford, 1998.

[124] Hudson, R. About 37% of word-tokens are nouns. *Language 70* (1994), 331–345.

[125] Huybregts, M. Overlapping dependencies in Dutch. Tech. rep., University of Utrecht, 1976. Utrecht Working Papers in Linguistics.

[126] Jelinek, F. Markov source modeling of text generation. In *The Impact of Processing Techniques on Communications*, J. K. Skwirzinksi, Ed. Nijhoff, Dordrecht, 1985, pp. 567–598.

[127] Jelinek, F., and Mercer, R. L. Interpolated estimation of Markov source parameters from sparse data. In *Pattern Recognition in Practice*, E. S. Gelsema and L. N. Kanal, Eds. North-Holland, NY, 1980, pp. 381–397.

[128] Johnson, K. What VP ellipsis can do, and what it can't, but not why. In *The Handbook of Contemporary Syntactic Theory*, M. Baltin and C. Collins, Eds. Blackwell, Oxford, 2001, pp. 439–479.

[129] Johnson, M. Finite state approximation of constraint-based grammars using left-corner grammar transforms. In *Proceedings of the Annual Meeting, ACL/COLING'98* (1998), Association for Computational Linguistics.

[130] Johnson, M. Pcfg models of linguistic tree representations. *Computational Linguistics 24*, 4 (1999), 613–632.

[131] Joshi, A. K. Processing crossed and nested dependencies: An automaton perspective on the psycholinguistic results. *Language and Cognitive Processes 5*, 1 (1990), 1–27.

[132] 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.

[133] Kamp, H. A theory of truth and semantic representation. In *Formal Methods in the Study of Language*, G. Groenendijk, T. Janssen, and M. Stokhof, Eds. Foris, Dordrecht, 1984.

[134] Kaplan, R., and Kay, M. Regular models of phonological rule systems. *Computational Linguistics 20* (1994), 331–378.

[135] Karp, R. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, R. Miller and J. Thatcher, Eds. Plenum, NY, 1972, pp. 85–103.

[136] Karp, R., and Luby, M. Monte-carlo algorithms for enumeration and reliability problems. In *Proceedings of Foundations of Computer Science (FOCS)* (1983), pp. 56–64.

[137] Karttunen, L. Computing with realizational morphology. In *Computational Linguistics and Intelligent Text Processing*, A. Gelbukh, Ed., Lecture Notes in Computer Science, Volume 2588. Springer Verlag, Heidelberg, 2003, pp. 205–216.

[138] 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.

[139] Kayne, R. S. Null subjects and clitic climbing. In *The Null Subject Parameter*, O. Jaeggli and K. Safir, Eds. Kluwer, Dordrecht, 1989.

[140] Kayne, R. S. *The Antisymmetry of Syntax.* MIT Press, Cambridge, Massachusetts, 1994.

[141] Kayne, R. S. Prepositions as probes. In *Structures and Beyond: The Cartography of Syntactic Structures Volume 3*, A. Belletti, Ed. Oxford University Press, Oxford, 2004.

[142] Kayne, R. S., and Pollock, J.-Y. New thoughts on stylistic inversion. Manuscript, New York University and CNRS, Lyon, 1999.

[143] Keenan, E. L., and Faltz, L. M. *Boolean Semantics for Natural Language.* Reidel, Dordrecht, 1985.

[144] Keenan, E. L., and Stabler, E. P. *Bare Grammar.* CSLI Publications, Stanford, California, 2003.

[145] Kelly, K. T., and Glymour, C. Why probability does not capture the logic of scientific justification. In *Contemporary Debates in Philosophy of Science*, C. Hitchcock, Ed. Wiley-Blackwell, NY, 2003.

[146] Kennedy, C. Ellipsis and syntactic representation. In *The Interfaces: Deriving and Interpreting Omitted Structures*, K. Schwabe and S. Winkler, Eds. John Benjamins, Amsterdam, 2003.

[147] Kenstowicz, M. *Phonology in Generative Grammar.* Blackwell, Cambridge, Massachusetts, 1994.

[148] Kimball, J. Seven principles of surface structure parsing in natural language. *Cognition 2* (1973), 15–47.

[149] Kiss, G. R., Armstrong, C., Milroy, R., and Piper, J. An associative thesaurus of English and its computer analysis. In *The Computer and Literary Studies.* University Press, Edinburgh, 1973.

[150] 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).

[151] Knuth, D. E. On the translation of languages from left to right. *Information and Control 8* (1965), 607–639.

[152] Kobele, G. M. *Generating Copies: An investigation into structural identity in language and grammar.* PhD thesis, UCLA, 2006.

[153] Koch, C., and Olteanu, D. Conditioning probabilistic databases. *ArXiv.0803.2212v1* (2008).

[154] Kohlhase, M. *A Mechanization of Sorted Higher-Order Logic Based on the Resolution Principle.* PhD thesis, Universität des Saarlandes, 1994.

[155] Koller, D., McAllester, D., and Pfeffer, A. Effective Bayesian inference for stochastic programs. In *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97)* (Menlo Park, 1997), AAAI Press, pp. 740–747.

[156] Koopman, H., and Szabolcsi, A. *Verbal Complexes.* MIT Press, Cambridge, Massachusetts, 2000.

[157] Kornai, A. Probabilistic grammars and languages. In *Proceedings of Mathematics of Language 10* (2007).

[158] Kuroda, S.-Y. Classes of languages and linear-bounded automata. *Information and Control 7* (1964), 207–223.

[159] Ladefoged, P. *A Course in Phonetics (3rd Edition).* Harcourt Brace Javonovich, NY, 1993.

[160] Lambek, J. The mathematics of sentence structure. *American Mathematical Monthly 65* (1958), 154–170.

[161] Lee, L. Fast context-free parsing requires fast Boolean matrix multiplication. In *Proceedings of the 35th Annual Meeting, ACL'97* (1997), Association for Computational Linguistics.

[162] Leroy, M., and Morgenstern, A. Reduplication before age two. In *Studies on Reduplication*, B. Hurch, Ed. Walter de Gruyter, Berlin, 2004, pp. 475–491.

[163] Lewis, R. L., Vasishth, S., and Dyke, J. V. Computational principles of working memory in sentence comprehension. *Trends in Cognitive Science 10*, 10 (2006), 447–454.

[164] Ljunglöf, P. Pure functional parsing: an advanced tutorial. Göteborg University, 2005.

[165] Magerman, D. M., and Weir, C. Efficiency, robustness, and accuracy in picky chart parsing. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics* (1992).

[166] Mahajan, A. Eliminating head movement. In *The 23rd Generative Linguistics in the Old World Colloquium, GLOW '2000, Newsletter #44* (2000), pp. 44–45.

[167] Manning, C. D., and Carpenter, B. Probabilistic parsing using left corner language models. In *Proceedings of the 1997 International Workshop on Parsing Technologies* (1997).

[168] Marcus, M. *A Theory of Syntactic Recognition for Natural Language.* MIT Press, Cambridge, Massachusetts, 1980.

[169] Marslen-Wilson, W., and Tyler, L. K. Against modularity. In *Modularity in Knowledge Representation and Natural-Language Understanding*, J. L. Garfield, Ed. MIT Press, Cambridge, Massachusetts, 1987.

[170] Martin, R. *A Minimalist Theory of PRO and Control.* PhD thesis, University of Connecticut, Storrs, 1996.

[171] 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.

[172] McCarthy, J. A prosodic theory of nonconcatenative morphology. *Linguistic Inquiry 12*, 3 (1981), 373–418. Reprinted in *Phonological Theory: The Essential Readings*, edited by John A. Goldsmith, Oxford: Blackwell, 1999.

[173] McCarthy, J., and Prince, A. Prosodic morphology and templatic morphology. *University of Massachusetts Occasional Papers in Linguistics 13* (1991).

[174] McDaniel, D., Chiu, B., and Maxfield, T. L. Parameters for wh-movement types: evidence from child English. *Natural Language and Linguistic Theory 13* (1995), 709–753.

[175] McGinn, C. Charity, interpretation, and belief. *Journal of Philosophy 74* (1977), 521–535.

[176] Michaelis, J. *On Formal Properties of Minimalist Grammars.* PhD thesis, Universität Potsdam, 2001. *Linguistics in Potsdam 13*, Universitätsbibliothek, Potsdam, Germany.

[177] 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.

[178] Miller, G. A., and Chomsky, N. Finitary models of language users. In *Handbook of Mathematical Psychology, Volume II*, R. D. Luce, R. R. Bush, and E. Galanter, Eds. Wiley, NY, 1963, pp. 419–492.

[179] Miller, G. A., and Gildea, P. How children learn words. *Scientific American 257*, 6 (1987), 94–99.

[180] Minsky, M. L. *The Society of Mind.* Simon and Schuster, NY, 1988.

[181] Mohri, M. Finite-state transducers in language and speech processing. *Computational Linguistics 23* (1997).

[182] Mohri, M., Pereira, F. C. N., and Riley, M. A rational design for a weighted finite-state transducer library. In *Automata implementation : second International Workshop on Implementing Automata, WIA'97*, D. Wood and S. Yu, Eds., Lecture Notes in Computer Science No. 1436. Springer, NY, 1998.

[183] 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.

[184] Moss, L. Natural language, natural logic, natural deduction. *Forthcoming* (2004). Indiana University.

[185] Munro, P., and Riggle, J. Productivity and lexicalization in Pima compounds. In *Proceedings of the Berkeley Linguistic Society, BLS* (Berkeley, 2004).

[186] Nardi, D., and Brachman, R. J. An introduction to description logics. In *The Description Logic Handbook*, F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, Eds. Cambridge University Press, NY, 2007.

[187] Nevins, A., and Vaux, B. Metalinguistic, shmetalinguistic: The phonology of shm-reduplication. In *Proceedings of the Chicago Linguistics Society* (2003).

[188] Nijholt, A. *Context Free Grammars: Covers, Normal Forms, and Parsing.* Springer-Verlag, NY, 1980.

[189] Nix, C., and Paris, J. A note on binary inductive logic. *Journal of Philosophical Logic 36* (2007), 735–771.

[190] Niyogi, S. A minimalist implementation of verb subcategorization. In *Seventh International Workshop on Parsing Technologies, IWPT'01* (2001).

[191] Nozohoor-Farshi, R. *LRRL(k) grammars: a left to right parsing technique with reduced lookaheads.* PhD thesis, University of Alberta, 1986.

[192] Ohala, J. J. The phonetics and phonology of aspects of assimilation. In *Papers in Laboratory Phonology I: Between the grammar and physics of speech*, J. Kingston and M. Beckman, Eds. Cambridge University Press, Cambridge, 1990.

[193] Oylaran, O. O. On the scope of the serial verb construction in Yoruba. *Studies in African Linguistics 13*, 2 (1982), 109–136.

[194] Papoulis, A. *Probability, Random Variables, and Stochastic Processes.* McGraw-Hill, NY, 1991.

[195] Paris, J. *The Uncertain Reasoner's Companion.* Cambridge University Press, NY, 1994.

[196] Parker, D. S. Schur complements obey Lambek's categorial grammar: another view of Gaussian elimination and LU decomposition. Computer Science Department Technical Report, UCLA, 1995.

[197] Partee, B. Bound variables and other anaphors. In *Theoretical Issues in Natural Language Processing*, D. Waltz, Ed. Association for Computing Machinery, NY, 1975.

[198] Pearl, J. *Probabilistic Reasoning in Intelligent Systems: Patterns of Plausible Inference.* Morgan Kaufmann, San Francisco, 1988.

[199] 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.

[200] Pesetsky, D. *Zero Syntax: Experiencers and Cascades.* MIT Press, Cambridge, Massachusetts, 1995.

[201] 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.

[202] Pfeffer, A. IBAL: A probabilistic rational programming language. In *Proceedings of the seventeenth International Conference on Artificial Intelligence (IJCAI-01)* (San Francisco, CA, Aug. 4–10 2001), B. Nebel, Ed., Morgan Kaufmann Publishers, Inc., pp. 733–740.

[203] Pfeffer, A. Functional specification of probabilistic process models. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA* (2005), M. M. Veloso and S. Kambhampati, Eds., AAAI Press / The MIT Press, pp. 663–669.

[204] 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.

[205] Pinker, S., and Prince, A. On language and connectionism: Analysis of a parallel distributed processing model of language acquisition. *Cognition 28* (1988), 73–193.

[206] Poletto, C., and Pollock, J.-Y. On the left periphery of Romance wh-questions. University of Padua, Université de Picardia à Amiens, 1999.

[207] Pollard, C., and Sag, I. *Head-driven Phrase Structure Grammar.* The University of Chicago Press, Chicago, 1994.

[208] Pollock, J.-Y., Munaro, N., and Poletto, C. Eppur si muove! In *A Celebration.* MIT Press, Cambridge, Massachusetts, 1999.

[209] Pullum, G. K. On syntactically mandated phrase reduplication. Presented at MIT, 2006.

[210] Purdy, W. C. A logic for natural language. *Notre Dame Journal of Formal Logic 32* (1991), 409–425.

[211] Putnam, H. Computational psychology and interpretation theory. In *Meaning and Cognitive Structure*, Z. Pylyshyn and W. Demopoulos, Eds. Ablex, New Jersey, 1986, pp. 101–116, 217–224.

[212] Pylkkänen, L., and McElree, B. The syntax-semantic interface: On-line composition of sentence meaning. In *Handbook of Psycholinguistics, Second Edition*, M. J. Traxler and M. A. Gernsbacher, Eds. Academic Press, NY, 2006, pp. 539–580.

[213] Quirk, R., Greenbaum, S., Leech, G., and Svartvik, J. *A Comprehensive Grammar of the English Language.* Longman, London, 1985.

[214] Rabiner, L. R. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE 77* (1989), 257–286.

[215] Ramsey, N., and Pfeffer, A. Stochastic lambda calculus and monads of probability distributions. In *Conference Record of 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'02.* ACM Press, New York, 2002, pp. 154–165.

[216] Ratnaparkhi, A. *Maximum Entropy Models for Natural Language Ambiguity Resolution.* PhD thesis, University of Pennsylvania, 1998.

[217] Rayner, K., Carlson, M., and Frazier, L. The interaction of syntax and semantics during sentence processing. *Journal of Verbal Learning and Verbal Behavior 22* (1983), 358–374.

[218] Riggle, J. Infixing reduplication in Pima and its theoretical consequences. UCLA. Publication forthcoming, 2003.

[219] Ristad, E., and Thomas, R. G. Hierarchical non-emitting Markov models. In *Proceedings of the 35th Annual Meeting, ACL'97* (1997), Association for Computational Linguistics.

[220] Ristad, E., and Thomas, R. G. Nonuniform Markov models. In *International Conference on Acoustics, Speech, and Signal Processing* (1997).

[221] Ritchie, G. Completeness conditions for mixed strategy context free parsing. *Computational Linguistics 25*, 4 (1999), 457–486.

[222] Rizzi, L. *Relativized Minimality.* MIT Press, Cambridge, Massachusetts, 1990.

[223] Rizzi, L. Reconstruction, weak island sensitivity, and agreement. Università di Siena, 2000.

[224] Roark, B., and Sproat, R. *Computational Approaches to Morphology and Syntax.* Oxford, NY, 2007.

[225] Roche, E., and Schabes, Y. Deterministic part-of-speech tagging with finite-state transducers. In *Finite-State Language Processing*, E. Roche and Y. Schabes, Eds. MIT Press, Cambridge, Massachusetts, 1997.

[226] Roche, E., and Schabes, Y. Introduction. In *Finite-State Language Processing*, E. Roche and Y. Schabes, Eds. MIT Press, Cambridge, Massachusetts, 1997.

[227] Rogers, A. Three kinds of physical perception verbs. In *Proceedings of the 7th Regional Meeting of the Chicago Linguistic Society* (1971), pp. 206–222.

[228] Sahin, N. T., Pinker, S., and Halgren, E. Abstract grammatical processing of nouns and verbs in Broca's area: Evidence from fMRI. *Cortex 42* (2006), 540–562.

[229] Sanchez-Valencia, V. *Studies on Natural Logic and Categorial Grammar.* PhD thesis, University of Amsterdam, Amsterdam, 1991.

[230] Satta, G. Tree adjoining grammar parsing and boolean matrix multiplication. *Computational Linguistics 20* (1994), 173–232.

[231] Schachter, P. Focus and relativization. *Language 61* (1985), 523–568.

[232] Schützenberger, M. P. A remark on finite transducers. *Information and Control 4* (1961), 185–196.

[233] Sedivy, J. C., Tanenhaus, M. K., Chambers, C. G., and Carlson, G. N. Achieving incremental semantic interpretation through contextual representation. *Cognition 71* (1999), 109–147.

[234] Shazeer, N. M., Linman, M. L., and Keim, G. A. Solving crossword puzzles as probabilistic constraint satisfaction. In *Proceedings of the 6th National Conference on Artificial Intelligence (AAAI-99); Proceedings of the 11th Conference on Innovative Applications of Artificial Intelligence* (Menlo Park, 1999), AAAI/MIT Press, pp. 156–162.

[235] Shieber, S., and Johnson, M. Variations on incremental interpretation. *Journal of Psycholinguistic Research 22* (1994), 287–318.

[236] Shieber, S. M. Evidence against the context-freeness of natural language. *Linguistics and Philosophy 8*, 3 (1985), 333–344.

[237] 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.

[238] 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.

[239] Sipser, M. *Introduction to the Theory of Computation.* PWS Publishing, Boston, 1997.

[240] Sportiche, D. Adjuncts and adjunctions. Presentation at 24th LSRL, UCLA, 1994.

[241] Sportiche, D. Sketch of a reductionist approach to syntactic variation and dependencies. In *Evolution and Revolution in Linguistic Theory*, H. Campos and P. Kempchinsky, Eds. Georgetown University Press, Washington, 1995. Reprinted in Dominique Sportiche, *Partitions and Atoms of Clause Structure: Subjects, agreement, case and clitics.* NY: Routledge.

[242] Sportiche, D. *Partitions and Atoms of Clause Structure : Subjects, Agreement, Case and Clitics.* Routledge, NY, 1998.

[243] 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.

[244] Stabler, E. P. *Computational Minimalism: Acquiring and parsing languages with movement.* Blackwell, Oxford, 1999. Forthcoming.

[245] Stabler, E. P. Remnant movement and complexity. In *Constraints and Resources in Natural Language Syntax and Semantics*, G. Bouma, E. Hinrichs, G.-J. Kruijff, and D. Oehrle, Eds. CSLI Publications, Stanford, California, 1999, pp. 299–326.

[246] Stabler, E. P. Minimalist grammars and recognition. In *Linguistic Form and its Computation*, C. Rohrer, A. Rossdeutscher, and H. Kamp, Eds. CSLI Publications, Stanford, California, 2001. (Presented at the SFB340 workshop at Bad Teinach, 1999).

[247] 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.

[248] Stabler, E. P. Varieties of crossing dependencies: Structure dependence and mild context sensitivity. *Cognitive Science 93*, 5 (2004), 699–720.

[249] Stabler, E. P., and Keenan, E. L. Structural similarity. In *Algebraic Methods in Language Processing, AMiLP 2000* (University of Iowa, 2000), A. Nijholt, G. Scollo, T. Rus, and D. Heylen, Eds. Revised version forthcoming in *Theoretical Computer Science*.

[250] Steriade, D. Positional neutralization. ms, 1995.

[251] Steriade, D. Alternatives to the syllabic interpretation of consonantal phonotactics. In *Proceedings of the 1998 Linguistics and Phonetics Conference*, O. Fujimura, B. Joseph, and B. Palek, Eds. Karolinum Press, Oxford, 1999, pp. 205–242.

[252] Stockall, L. *Magnetoencephalographic Investigations of Morphological Identity and Irregularity.* PhD thesis, Massachusetts Institute of Technology, 2004.

[253] Stolcke, A. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Linguistics 21* (1995), 165–201.

[254] Stowell, T. Empty heads in abbreviated English. GLOW 14 talk, 1991.

[255] Szymanski, T. G., and Williams, J. H. Noncanonical extensions of bottom-up parsing techniques. *SIAM Journal on Computing 5* (1976), 231–250.

[256] Tenenbaum, J. B., and Griffiths, T. L. Generalization, similarity , and bayesian inference. *Behavioral and Brain Sciences 24* (2001), 629–640.

[257] Thornton, R. *Adventures in Long-Distance Moving: The Acquisition of Complex Wh-Questions.* PhD thesis, University of Connecticut, 1990.

[258] Travis, L. *Parameters and effects of word order variation.* PhD thesis, Massachussets Institute of Technology, Cambridge, Massachusetts, 1984.

[259] Trueswell, J., and Gleitman, L. Children's eye movements during listening: Developmental evidence for a constraint-based theory of lexical processing. In *Interface of Language, Vision, and Action: Eye Movements and the Visual World*, J. M. Henderson and F. Ferreira, Eds. Psychology Press, NY, 2004.

[260] Urbanczyk, S. Double reduplications in parallel. In *The Prosody-Morphology Interface*, H. van der Hulst, R. Kager, and W. Zonneveld, Eds. Cambridge University Press, NY, 1999, pp. 390–428.

[261] Valiant, L. G. General context free recognition in less than cubic time. *Journal of Computer and System Sciences 10* (1975), 308–315.

[262] Valois, D. The internal syntax of DP and adjective placement in French and English. In *Proceedings of the North Eastern Linguistic Society, NELS 21* (1991).

[263] Vergnaud, J.-R. *Dépendances et Niveaux de Représentation en Syntaxe.* PhD thesis, Université de Paris VII, 1982.

[264] Vidal, E., Thollard, F., de la Higuera, C., Casacuberta, F., and Carrasco, R. C. Probabilistic finite-state machines, parts I and II. *IEEE Transactions on Pattern Analysis and Machine Intelligence 27*, 7 (2005).

[265] Vijayashanker, K. *A Study of Tree Adjoining Languages.* PhD thesis, University of Pennsylvania, 1987.

[266] Villemonte de la Clergerie, E. Parsing MCS languages with thread automata. In *Proceedings of the 6th International Workshop on Tree Adjoining Grammars and Related Frameworks, TAG+6* (2002).

[267] Viterbi, A. J. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory IT-13* (1967), 260–269.

[268] Walther, C. *A Many-Sorted Calculus Based on Resolution and Paramodulation.* Morgan Kauffman, Los Altos, California, 1987.

[269] Wartena, C. *Storage Structures and Conditions on Movement in Natural Language Syntax.* PhD thesis, Universität Potsdam, 1999.

[270] Watanabe, A. *Case Absorption and Wh Agreement.* Kluwer, Dordrecht, 1993.

[271] Wetherell, C. Probabilistic languages: A review and some open questions. *Computing Surveys 12(4)* (1980), 361–379.

[272] Winkler, S., and Schwabe, K. Exploring the interfaces from the perspective of omitted structures. In *The Interfaces: Deriving and Interpreting Omitted Structures*, K. Schwabe and S. Winkler, Eds. John Benjamins, Amsterdam, 2003.

[273] Wittgenstein, L. *Philosophical Investigations.* MacMillan, NY, 1958. This edition published in 1970.

[274] Younger, D. Recognition and parsing of context free languages in time $O(n^3)$. *Information and Control 10* (1967), 189–208.

[275] Zepeda, O. *A Papago Grammar.* University of Arizona Press, Tucson, 1983.

[276] Zuraw, K. Aggressive reduplication. *Phonology 19* (2002), 395–439.

[277] Zuraw, K. R. Floating phonotactics: Variability and infixation of Tagalog loanwords. M.A. thesis, UCLA, 1996.

[278] Zwicky, A. Clitics and particles. *Language 61* (1985), 283–305.

Index