1

Classical Categorial Grammars: AB Grammars

Summary. This first chapter deals with material from the late fifties and early sixties, but which nevertheless introduces the design of categorial grammars, which are lexcalized grammars, as opposed to the phrase structure grammars like context-free grammars that were introduced afterwards.

Although the success of phrase structure grammars went far beyond that of categorial grammars, their lexicalization was in fact an attractive feature, another one being their connection to logical semantics.

We end with more recent results: a learning algorithm for categorial grammars, which was proved to converge at the end of the nineties. Having a learning algorithm for a class of grammars which can describe (small parts of) natural language is, we think, quite an important feature of categorial grammars. It comes from their lexicalization and logical formulation, which will be further studied in the next chapter.

1.1 Semantic Categories and Ajdukiewicz Fractions

Though many of the ideas behind categorial grammars can be traced to the work of Husserl, Frege and Russell, we begin this introduction to categorial grammars with the work of Ajdukiewiz. For the history of categorial grammars, we refer the reader to (Casadio, 1988; Morrill, 2007). In 1935 Ajdukiewicz defined a calculus of fractions to test the correctness of logical statements (Ajdukiewicz, 1935):

> *The discovery of antinomies, and the method of their resolution have made problems of linguistic syntax the most important problems of logic (provided this word is understood in a sense that also includes meta-theoretical considerations). Among these problems that of syntactic connection is of the greatest importance for logic. It is concerned with the specification of the conditions under which a word pattern constituted of meaningful words, forms an expression which itself has a unified meaning (constituted, to be sure, by the meaning of the single words belonging to it). A word pattern of this kind is called syntactically connected.*

R. Moot and C. Retoré: The Logic of Categorial Grammars 2012, LNCS 6850, pp. 1–22, 2012.

His paper deals with both the formal language of logic and natural language, but is actually more concerned with the language of propositional and predicate logic.

> *If one applies this index symbolism to ordinary language, the semantic categories which we have assumed (in accordance with Leśniewski) will not always suffice, since ordinary languages are richer in semantic categories.*

Each word (or lexical entry) is provided with an index[1] which is a category. Categories are defined inductively as follows:

Basic categories. The two primitive types n (for entities or individuals or first order terms) and s (for propositions or truth values) are categories

Fractions. Whenever N is a category and $D_1, \ldots, D_p$ is a sequence or multiset of categories, then $\frac{N}{D_1 \cdots D_p}$ is itself a category. These complex categories are called functor categories or fractions.

If we formalize the definitions in his article, syntactically connected expressions and their exponents[2] are recursively defined as follows:

- a word or lexical entry is syntactically connected, and its exponent is its index.
- given
  - n syntactically connected expressions $d_1, \ldots, d_n$ of respective exponents $D_1, \ldots, D_n$
  - an expression f of exponent $\frac{N}{D_1 \cdots D_n}$

  the expression $f d_1 \cdots d_n$ (or any permutation of it) is syntactically connected and has exponent N.

This in particular entails that sequences of fractions reduce to a single index using the usual simplifications for fractions. It should be observed that in this "commutative setting" the simplification procedure of the fractions is not that simple if the bracketing corresponding to subexpressions is not given. As Ajdukiewicz is mainly concerned with the language of logic where one can use the Polish notation, word order is not really a problem for him.

1.2 Classical Categorial Grammars or AB Grammars

In 1953, that is a bit before Chomsky introduced his hierarchy of Phrase Structure Grammars (Chomsky, 1955), Bar-Hillel defines bidirectional categorial grammars (Bar-Hillel, 1953), refining Ajdukiewicz types to take constituent order into account. Therefore, his grammars are more adequate for modeling natural language, where

[1] Ajdukiewicz uses the word "index" interchangeably with the word "category".

[2] Ajdukiewicz uses the word "exponent" to mean the *result* category of an expression after reduction rules. For example, the exponent of $\frac{s}{n}$ n is s, which should be reminiscent of the simplification of a fraction which is multiplied by its denominator in elementary mathematics: $\frac{A}{B} \times B$ simplifying to A.

word order is crucial. We will use Lambek's notation for types (Lambek, 1958), following Bar-Hillel in his later work on categorial grammars.

In the literature, these grammars are called *AB grammars*, *classical categorial grammars* or *basic categorial grammars*.

Types or fractions are defined as follows:

$$\mathsf{L} \quad ::= \quad \mathsf{P} \quad | \quad (\mathsf{L} \setminus \mathsf{L}) \quad | \quad (\mathsf{L} / \mathsf{L})$$

where P is the set of primitive types, which we will also call atomic types or basic categories, which usually contains S (for sentences) np (for noun phrases) and n (for nouns), and may include pp (for prepositional phrase) inf (for infinitives) etc.

We will often omit the outer brackets when this does not lead to confusion and write $np \setminus S$ instead of $(np \setminus S)$ and $(np \setminus S) / np$ instead of $((np \setminus S) / np)$.

A note on the terminology used throughout this book (and much of the literature on categorial grammars): we often use the terms category, formula and (syntactic) type interchangeably. Since we will discuss *semantic* types only in Chapter 3, unless otherwise indicated, the word 'type' will mean *syntactic* type (or category or formula, that is a member of L).

It is usual to talk about a formula of type $B \setminus A$ or A / B as a *functor*, the formula B as its *argument* and the formula A as its *result*. Speaking colloquially, we will say a formula $B \setminus A$ or A / B (the functor) selects a B (the argument) to form an A (the result). As we will see in Chapter 3, talking about functions and arguments is not just a convenient way to refer to syntactic combinations, the function/argument distinction has direct semantic import. Finally, types of the form $A \setminus A$ or A / A are often called *modifiers* or *A-modifiers*. They take an argument of type A to produce a result of the same type.

The grammar is defined by a lexicon, that is a function Lex which maps words or terminals to finite sets of types (a set of types is needed, since in natural language a single word may admit various constructions: "*eat*" may ask for an object or not, for instance).

An expression, that is a sequence of words or terminals $w_1 \cdots w_n$, is of type u whenever there exists for each w_i a type t_i in $\mathrm{Lex}(w_i)$ such that $t_1 \cdots t_n \longrightarrow u$ with the following reduction patterns:

$$\forall u, v \in \mathsf{L} \qquad \begin{array}{ll} u\ (u \setminus v) \longrightarrow v & (\setminus_e) \\ (v / u)\ u \longrightarrow v & (/_e) \end{array}$$

These rules are called elimination rules, or simplifications, or modus ponens.

An application of an elimination rule is defined as for context-free grammars: inside a sequence of types, we replace the left-hand side of an elimination rule by its right-hand side, in other words, if Γ and Γ' are lists of types then applications of the elimination rules rewrite $\Gamma\ u\ u \setminus v\ \Gamma'$ to $\Gamma\ v\ \Gamma'$ (for $\setminus_e$) and $\Gamma\ v / u\ u\ \Gamma'$ to $\Gamma\ v\ \Gamma'$ (for $/_e$). These rule applications should be compared to the "rewrites immediately" relation for context-free grammars which we will discuss below on page 6.

These rules provide the symbols $\setminus$ and $/$ with an intuitive meaning: an expression y is of type $A \setminus B$ whenever it needs an expression a of type A on its left to obtain an

expression ay of type B; symmetrically, an expression z is of type B/A whenever it needs an expression a of type A on its right to obtain an expression za of type B;

The set of sentences or the language generated by the grammar is the set of word sequences of type S.

The derivation tree is simply a binary tree whose leaves are the types t_i and whose nodes are labeled by rules $/_e$ and $\backslash_e$.

Example 1.1 (A Tiny AB Grammar)

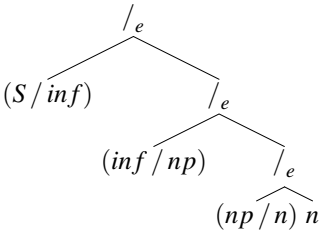
Consider the following lexicon:

Word	Type(s)	Translation
cosa	$(S/(S/np))$	*what*
guarda	(S/inf)	*he/she watches*
passare	(inf/np)	*passing by*
il	(np/n)	*the*
treno	n	*train*

The sentence '*guarda passare il treno*' (he/she watches the train passing by) belongs to the generated language:

$$\begin{array}{l} (S/inf) \quad (inf/np) \quad (np/n) \quad n \\ \longrightarrow (S/inf) \quad (inf/np) \quad np \\ \longrightarrow (S/inf) \quad inf \\ \longrightarrow S \end{array} \tag{1.1}$$

The derivation tree for this analysis can be written as:



(1.2)

A final way of presenting the same analysis is shown below. Compared to the analysis above, it is written upside-down. It uses explicit Lex rules and gives the result category of each rule application.

$$\dfrac{\dfrac{\text{guarda}}{S/inf}\,\text{Lex} \qquad \dfrac{\dfrac{\text{passare}}{inf/np}\,\text{Lex} \qquad \dfrac{\dfrac{\text{il}}{np/n}\,\text{Lex} \qquad \dfrac{\text{treno}}{n}\,\text{Lex}}{np}\,/_e}{inf}\,/_e}{S} \tag{1.3}$$

Though derivation 1.3 appears more detailed, it is actually equivalent to derivation 1.2: the rule name and the daughter categories suffice to deduce the result category in a bottom-up way, from the leaves to the root node. Derivation 1.3 corresponds to the natural deduction derivations we will see in Section 2.2.1. Derivation 1.2 corresponds to the derivation format used for the learning algorithms in Section 1.6. Finally, the flat structure of derivation 1.1 is close to the derivations of context-free grammars, a point we will explore further in the next section.

Returning to the lexicon above, we remark that the sentence '*cosa guarda passare*' (what is he/she looking passing by?) does not belong to the generated language: indeed the sequence

$$(S/(S/np))\ (S/inf)\ (inf/np)$$

does not contain anything that could be reduced.

It should be observed that AB grammars are lexicalized: that is to say, the grammar consists of the following two components:

1. a universal set of rules which is common to all languages ("universal grammar" in the terminology of (Chomsky, 1995); for AB grammars these are just the two reduction patterns $\backslash_e$ and $/_e$ above)
2. a lexicon, which is responsible for all differences between individual languages.

As such, the universal grammar — and its formal properties — becomes an object of study in itself. This lexicalist view is coherent with many modern linguistic theories, like the minimalist program of Chomsky (Chomsky, 1995) (language variation is only lexical), and with some formalisms for computational linguistics, like (Lexicalized) Tree Adjoining Grammars (Joshi et al, 1975; Joshi and Schabes, 1997) or Head-Driven Phrase Structure Grammars (Pereira and Shieber, 1987; Pollard and Sag, 1994).

Another observation is that the rules are like modus ponens, but in a logic where contraction and weakening are not allowed, and where the order of the hypothesis is taken into account (see Girard, 1995, for a clear and intuitive introduction to the structural rules seen from the perspective of linear logic). The relation between categorial grammars and (linear) logic will be a major theme of this book.

Let us state one of the first results on categorial grammars known as the Gaifman theorem of (Bar-Hillel et al, 1963) which is more or less equivalent to the existence of a Greibach normal form for context-free grammars:

Proposition 1.2. *Every AB grammar is equivalent to an AB grammar containing only types of the form*

$$p \qquad (p/q) \qquad ((p/q)/r)$$

where p, q, r stand for primitive types.

Proof. This theorem is an immediate consequence of Propositions 1.11 and 1.10 to be proved below using the well-known Greibach normal form theorem (Greibach, 1965). □

1.3 AB Grammars and Context-Free Grammars

1.3.1 Context-Free Grammars

Context-Free Grammars (CFGs) were introduced in (Chomsky, 1955) and a good introduction is provided in (Hopcroft and Ullman, 1979); we use the following standard notation:

- M^* stands for the set of finite sequences over the set M.
- M^+ stands for the set of finite non empty sequences over the set M.
- ε stands for the empty sequence of M^*.

Definition 1.3 (Context-free grammar). *A context-free grammar is defined by:*

[Non Terminals] *a set NT of symbols called non terminals, one of them being the start symbol S.*

[Terminals] *set T of symbols, disjoint from NT, called terminals (or words according to the linguistic viewpoint)*

[Production rules] *a finite set of production rules of the form $X \longrightarrow W$ with $X \in NT$ and $W \in (T \cup NT)^*$*

We say a a context-free grammar is lexicalized *if each production rule contains a member of T on its right hand side.*

A sequence $V \in (T \cup NT)^*$ is said to rewrite immediately into a sequence $W \in (T \cup NT)^*$ whenever there exists $W', W'', W''' \in (T \cup NT)^*$ and a non terminal X such that

- $V = W'XW'''$
- $X \longrightarrow W''$ is a production rule.
- $W = W'W''W'''$

The relation $\longrightarrow$ is defined over sequences in $(T \cup NT)^*$ as the transitive closure of "rewrites immediately into". The language generated by a CFG is the smallest subset of T^* containing the sequences into which S rewrites.

Two grammars which generate the same languages are said to be weakly equivalent.

Whenever a non-terminal N rewrites into a sequence of terminals and non terminals $X_1, \ldots, X_n$ it is possible (as linguists often do) to denote the derivation tree by a term T representing this derivation tree as follows:

- a non terminal or a terminal is a derivation tree and its yield is itself.
- if $T_1, \ldots, T_n$ are derivation trees of $X_1, \ldots, X_n$ and if $X \longrightarrow X_1 \cdots X_n$ is a rule of the grammar then $[_X T_1, \ldots, T_n]$ is a derivation tree labeled X and its yield is the concatenation of the yields of $T_1, \ldots, T_n$.

Obviously a sequence of terminals $a_1 \cdots a_n$ is in the language if and only if there exists a derivation tree labeled S the yield of which is $a_1 \cdots a_n$. We denote by ε the empty sequence.

Two grammars which generate the same derivation *trees* are said to be strongly equivalent.

Definition 1.4. *A CFG is said to be ε-free whenever ε does not belong to the generated language.*

It is not difficult to decide whether a CFG is ε-free or not, and if it is not ε-free, the grammar can be written with the production rules of an ε-free CFG, together with the rule: $S \longrightarrow \varepsilon$.

Definition 1.5 (Chomsky normal form). *A CFG is said to be in Chomsky normal from whenever its production rules are of the form $X \longrightarrow Y\,Z$ or of the form $X \longrightarrow a$, with $X, Y, Z \in NT$ and $a \in T$.*

Proposition 1.6. *Any ε-free CFG can be transformed into a weakly equivalent CFG in Chomsky normal form and this transformation can be performed in polynomial time (see Chomsky, 1963; Hopcroft and Ullman, 1979).*

Definition 1.7 (Greibach normal form). *A CFG is said to be in Greibach normal form whenever its production rules are of the form: $X \longrightarrow a X_1 \cdots X_n$ with $a \in T$, $X, X_1, \ldots, X_n \in NT$. It is said to be in strong Greibach normal form whenever $n \leq 2$.*

Proposition 1.8. *Any ε-free CFG can be turned into a CFG in (strong) Greibach normal form, and these transformations can be performed in polynomial time (see Greibach, 1965; Harrison, 1978).*

Transforming a CFG into its Greibach normal form is a way of lexicalizing this grammar, since each rule of a grammar in Greibach normal form contains a terminal. While the derivation trees of a CFG and the ones of its Chomsky normal form are closely related, the derivation trees of the Greibach normal from of a CFG can be very different from the derivation trees of the original CFG: to lexicalize a CFG while preserving the analyses, one has to move to TAGs (Schabes and Waters, 1993; Joshi and Schabes, 1997).

1.3.2 From Context-Free Grammars to AB Grammars

The relationship between CFG and AB grammars was the subject of a detailed investigation in the early sixties (Bar-Hillel et al, 1963).

Proposition 1.9. *Every ε-free Context-Free Grammar in Greibach normal form is equivalent to an AB categorial grammar.*

Proof. Let us consider the following AB grammar:

- Its words are the terminals of the CFG.
- Its primitive types are the non terminals of the CFG.
- $\mathrm{Lex}(a)$, the finite set of types associated with a terminal a contains the formulae $((\cdots((X \,/\, X_n) \,/\, X_{n-1}) \,/\, \cdots) \,/\, X_2) \,/\, X_1$ such that there are non terminals $X, X_1, \ldots, X_n$ such that $X \longrightarrow a X_1 \cdots X_n$ is a production rule.

It is then easily observed that the derivation trees of both grammars are isomorphic. □

Proposition 1.10. *Each ε-free context-free grammar is weakly equivalent to an AB grammar containing only types of the form X or X / Y or $(X / Y) / Z$.*

Proof. Here we provide the reader with a simple "modern proof" using the existence of a Greibach normal from: indeed the Gaifman theorem first published in (Bar-Hillel et al, 1963) was proved before the existence of Greibach normal form for CFGs (Greibach, 1965), and these two theorems are actually more or less equivalent.

According to Proposition 1.8, any context-free grammar can be turned into a weakly equivalent CFG in strong Greibach normal form. As can be observed from the construction of an equivalent AB grammar in the previous proof, if the CFG is in strong Greibach normal form that is if rules are of the form: $X \longrightarrow aX_1 \cdots X_n$ with $0 \leq n \leq 2$, then the corresponding AB grammar only uses types of the form X, X / X_1, $(X / X_2) / X_1$. □

1.3.3 From AB Grammars to Context-Free Grammars

Proposition 1.11. *Every AB grammar is strongly equivalent to a CFG in Chomsky normal form.*

Proof. Let G be the CFG defined by:

- Terminals T are the words of the AB grammar.
- Non Terminals NT are all the subtypes of the types appearing in the lexicon of the AB grammar — a type is considered to be a subtype of itself.
- The production rules are of two kinds:
  - $X \longrightarrow a$ whenever $X \in \mathrm{Lex}(a)$
  - $X \longrightarrow (X / Z)\ Z$ and $X \longrightarrow Z\ (Z \setminus X)$ for all $X, Z \in NT$ — keep in mind that from the CFG viewpoint $(Z \setminus X)$ and (X / Z) are both non terminal symbols.

This defines a CFG because the lexicon is finite, so there are only finitely many subtypes of types in the lexicon, hence finitely many production rules. The derivation trees in both formalisms are isomorphic. □

1.4 Parsing AB Grammars

Theorem 1.12. *A sentence of n words can be analyzed using an AB grammar in $O(n^3)$ time using $O(n^2)$ space.*

Proof. (easy exercise) Following the relation between AB grammars and CFGs in Chomsky normal form, it is not difficult to adapt the Cocke Kasami Younger algorithm (see e.g. Sikkel and Nijholt, 1997) to AB grammars. □

1.5 Limitations of AB Grammars

In an AB grammar one is not able to derive (t / v) from (t / u) and (u / v). Consider for instance the Italian sentence '*Cosa guarda passare?*' we've seen in Example 1.1. One is not able to derive it with the simple type assignments given there. We would need transitivity of / to obtain it:

$$(S / (S / np)) \quad (S / inf) \quad (inf / np) \quad \overset{(trans.)}{\longrightarrow} \quad (S / (S / np)) \quad (S / np) \quad \longrightarrow \quad S$$

We would also like to model the behavior of an object relative pronoun like *that/whom*, by providing it with the type $(n \setminus n) / (S / np)$ but unfortunately this too requires transitivity — unless a transitive verb also has the type $np \setminus (S / np)$, but it is rather unusual to analyze English verbs as combining first with their subjects and then with their objects, and, in our view, it is rather unnatural to require the *verb* to have different types when combining with a subject relative pronoun $(n \setminus n) / (np \setminus S)$ and when combining with an object relative pronoun $(n \setminus n) / (S / np)$.

On the mathematical side, one would like to interpret categories by subsets of a free monoid (the intended one being sequences of words), so that the subset of sequences of type S are precisely the correct sentences. This is indeed impossible: one may view the elimination rules as modus ponens, but then what is lacking are introduction rules to get completeness of the calculus with respect to this natural monoidal interpretation. This is solved by the Lambek calculus which we will study in the next chapter.

1.6 Learning AB Grammars

We will end our study of AB grammars with an interesting property: they enjoy good learning algorithms from positive examples, at least when examples are structured. This learning question is important for the following two reasons:

- It models, although very roughly, the process of language acquisition and more precisely of syntax acquisition (Gleitman and Newport, 1995; Pinker, 1995) extensively discussed in generative linguistics; indeed, it is the main justification for the existence of a *universal grammar* see e.g. (Chomsky, 1995).
  - The similarity with natural language acquisition by human beings, is that we only learn from positive examples, and that structure is needed for the language learner.
  - The main difference is that the sequence of languages which converges to the target language is increasing — meaning the learner starts with a set of rules which generate a subset of the target language, seen as a set of sentences, and generalizes this set of rules step by step — while in natural language acquisition the sequence of languages is decreasing — meaning the learner starts with a set of rules which generate a superset of the target language and constrains this set of rules.

- This learning algorithm provides a method for the automated construction of a grammar (that is a lexicon) from a corpus, which also can be viewed as an automated method for completing an existing grammar/lexicon.

1.6.1 Grammatical Inference for Categorial Grammars

Learning (also called grammatical inference) from positive examples is the following problem: define a function Learn from finite sets of positive examples to grammars of a given class $\mathscr{G}$, such that:

- Given a grammar G of the class $\mathscr{G}$ and an enumeration $s_1, s_2, \ldots$ of the sentences G generates, letting $\mathrm{Ex}_i = \{s_1, \ldots, s_i\}$, there exists an N such that for all $n \geq N$ the grammar $\mathrm{Learn}(\mathrm{Ex}_n)$ is constant and exactly generates the sentences produced by G.
- The following is not mandatory, but one usually asks for this extra property: for every set of sentences Ex the grammar $\mathrm{Learn}(\mathrm{Ex})$ generates all the examples in Ex

This definition is the so-called identification in the limit introduced by Gold in 1967 (Gold, 1967). The grammars we are to consider are of course AB grammars, but what will the positive examples be? In our definition the term "sentence" is left vague. Actually we shall use this definition not with mere sequences of words, but we will rather consider the derivation trees produced by the grammar, and so our examples will be derivation trees in which the types of the words are absent: this is not absolutely unrealistic, because the learner of a language has access to some information related to the syntactic structure of the sentences like prosody or semantics; nevertheless it is unrealistic, because the complete syntactic structure is not fully known.

The lexicalization of categorial grammars is extremely helpful for this learning question: indeed we have no rules to learn, but only the types of the words to guess. Observe that we need to bound the number of types per word; otherwise each new occurrence of a word may lead to the introduction of a new type for this word, and this process cannot converge.

As this presentation is just meant to give an idea of learning algorithms, we only present here the simplest case of learning from structures: the algorithm RG of Buszkowski and Penn. The AB grammars considered are rigid, that is to say there is exactly one type per word.

1.6.2 Unification and AB Grammars

The algorithm makes use of type-unification, and this kind of technique is quite common in grammatical inference (see Nicolas, 1999), so let us briefly define it and explain its relation to AB grammars. First, we will consider an extended formula language for AB which has a countable number of type variables, we will use x, y, x_1, x_2, …, y_1, y_2, … to denote type variables. These variables will play much the

same role as the atomic formulas, with the exception that we can substitute formulas for type variables. Given a substitution σ', a function from variables to types, we can extend σ' to a substitution σ, a function from types to types as follows (in the definition below, x denotes any type variable in the formula language).

$$\begin{aligned}\sigma(S) &= S\\ \sigma(x) &= \begin{cases}\sigma'(x) & \text{if } \sigma'(x) \text{ is defined}\\ x & \text{otherwise}\end{cases}\\ \sigma(A \setminus B) &= \sigma(A) \setminus \sigma(B)\\ \sigma(B / A) &= \sigma(B) / \sigma(A)\end{aligned}$$

Given a substitution σ, we can apply it to the lexicon of an AB grammar. If a sentence is generated by an AB grammar defined by a lexicon Lex then it is also generated by the AB grammar defined by the lexicon $\sigma(\text{Lex})$.[3]

A substitution is said to unify a set of types T if for all types A, B in T one has $\sigma(A) = \sigma(B)$. For such kinds of formulae, whenever a unifier exists, there exists a most general unifier (mgu) that is a unifier σ_u such for every unifier τ there exists a substitution σ_τ such that $\tau = \sigma_\tau \circ \sigma_u$.

The relation between two rigid AB grammars with respective lexicons Lex and Lex$'$ defined by *there exists a substitution* σ *such that* $\text{Lex}' = \sigma(\text{Lex})$ defines an order which is a complete lattice, and the supremum of a family corresponds to the least general grammar generating all the trees of all the grammars in the family.

1.6.3 The RG Algorithm

We present here the RG algorithm (learning Rigid Grammars) introduced by W. Buszkowski and G. Penn in (Buszkowski, 1987; Buszkowski and Penn, 1990) and which has been further studied by M. Kanazawa (Kanazawa, 1998).

To illustrate this algorithm, let us take a small set of positive examples:

(1.4) $[_{\setminus e}[_{/ e}$ a man] swims]
(1.5) $[_{\setminus e}[_{/ e}$ a fish]$[_{\setminus e}$ swims fast]]

Typing. As the examples are assumed to be correct sentences, we know the root should be labeled by the type S which is the only type fixed in advance, a constant.

Each time there is a $\setminus_e$ (resp. $/_e$) node labeled y, we know the argument node, the one on the left (resp. on the right) should be x while the function node the one on the right (resp. on the left) should be $x \setminus y$ (resp. y / x)

So by assigning a new variable to each argument node we have typed the whole tree, and so words have been provided with a type (involving the added variables and S).

[3] This is a slight abuse of notation, but its intended meaning should be clear: we apply the substitution σ to the elements of the *range* of the function Lex, so, for any word w, if $\text{Lex}(w)$ is $\{f_1, \ldots f_n\}$ then $\sigma(\text{Lex})(w)$ is $\{\sigma(f_1), \ldots, \sigma(f_n)\}$.

We can do so on our examples; to denote the resulting type, we add it on top of the opening bracket.

(1.6) $[^{S}_{/_e}[^{x_2}_{\backslash_e}$ a:(x_2 / x_1) man:$x_1]$ swims:$(x_2 \backslash S)]$

(1.7) $[^{S}_{\backslash_e}[^{y_2}_{/_e}$ a:(y_2 / y_3) fish:$y_3][^{(y_2 \backslash S)}_{\backslash_e}$ swims:y_1 fast:$(y_1 \backslash (y_2 \backslash S))]]$

Unification. The previous steps give us several types per word. For instance the examples above yield:

Example 1.13

word	*type*1	*type*2
a:	x_2 / x_1	y_2 / y_3
fast:		$y_1 \backslash (y_2 \backslash S)$
man:	x_1	
fish:		y_3
swims:	$x_2 \backslash S$	y_1

We now have to unify the set of types associated with a single word, and the output of the algorithm is the grammar/lexicon in which every words gets the single type which unifies the original types, collected from each occurrence of a word in each example. If these sets of types can be unified, then the result of this substitution is a rigid grammar which generates all the examples, and can be shown to be the least general grammar to generate these examples.
In our example, unification succeeds and leads the most general unifier σ_u defined as follows:

Example 1.14

$$\begin{aligned}\sigma_u(x_1) &= z_1\\ \sigma_u(x_2) &= z_2\\ \sigma_u(y_1) &= z_2 \backslash S\\ \sigma_u(y_2) &= z_2\\ \sigma_u(y_3) &= z_1\end{aligned}$$

which yields the rigid grammar/lexicon:

Example 1.15

a: z_2 / z_1
fast: $(z_2 \backslash S) \backslash (z_2 \backslash S)$
man: z_1
fish: z_1
swims: $z_2 \backslash S$

Convergence of the RG Algorithm

This algorithm converges in the sense we defined above, as shown by (Kanazawa, 1998). The technique also applies to learning rigid Lambek grammars from natural deduction trees (Bonato, 2000) and we follow his presentation.

For the proof of convergence, we make use of the following notions and notational conventions:

- $G \subset G'$. This reflexive relation between G and G' holds whenever every lexical type assignment $a : T$ in G is in G' as well — in particular when G' is rigid, so is G, and both grammars are identical. Note that this is just the normal subset relation for each of the words in the lexicon G': $\text{Lex}_G(a) \subset \text{Lex}_{G'}(a)$ for every a in the lexicon of G', with $\text{Lex}_G(a)$ non-empty. Keep in mind that in what follows we will also use the subset relation symbol to signify inclusion of the generated *languages*; the intended meaning should always be clear from the context.
- *size of a grammar.* The size of a grammar is simply the sum of the sizes of the occurrences of types in the lexicon, where the size of a type is its number of occurrences of base categories (variables or S).
- $G \sqsubset G'$. This reflexive relation between G and G' holds when there exists a substitution σ such that $\sigma(G) \subset G'$ which does not identify different types of a given word, but this is always the case when the grammar is rigid.
- *FA-structure.* An FA-structure is a binary tree whose leaves are labeled with words (terminals) and internal nodes with names of the rules, namely $/_e$ and $\backslash_e$. An analysis in an AB grammar, once the types are erased, is an FA structure, and, conversely, for every type T, every FA structure can be labeled with types in order to obtain an analysis of the sequence of words as having category T — that's what the typing algorithm does, with $T = S$. The positive examples we are using for the RG learning algorithm, see Examples 1.4 and 1.5, are FA-structures.
- $FL(G)$. Given a grammar G, $FL(G)$ is the tree language consisting of all the FA-structures with root S derived from G.
- $GF(D)$. Given a set of FA-structures D, $GF(D)$ is the lexicon obtained by collecting the types of each word in the various examples of D — as in Example 1.13 above.
- $RG(D)$. Given a set of examples D, $RG(D)$ is, whenever it exists, the rigid grammar/lexicon obtained by applying the most general unifier to $GF(D)$ — as in Example 1.15 above.

Proposition 1.16. *Given a grammar G, the number of grammars H such that $H \sqsubset G$ is finite.*

Proof. There are only finitely many grammars which are included in G, since G is a finite set of assignments. Whenever $\sigma(H) = K$ for some substitution σ the size of H is smaller or equal to the size of K, and, up to renaming, there are only finitely many grammars smaller than a given grammar.

By definition, if $H \sqsubset G$ then there exist $K \subset G$ and a substitution σ such that $\sigma(H) = K$. Because there are only finitely many K such that $K \subset G$, and for every K there are only finitely many H for which there could exist a substitution σ with $\sigma(H) = K$ we conclude that there are only finitely many H such that $H \sqsubset G$. □

Proposition 1.17. *If $G \sqsubset G'$ then $FL(G) \subset FL(G')$.*

Proof. $G \sqsubset G'$ means that there exists σ such that $\sigma(G) \subset G'$. Let T be an FA-structure in $FL(G)$, hence T comes from an analysis A of a sequence of words $m_1 \cdots m_n$. If we apply σ to A we obtain an analysis of the same sequence of words in G'. Indeed for a word whose assignment is T in G we have the assignment $\sigma(T)$ which is its assignment in G', and the types obtained inside the tree match the rules since $\sigma(A \setminus B) = \sigma(A) \setminus \sigma(B)$ and $\sigma(A / B) = \sigma(A) / \sigma(B)$ So $\sigma(A)$ is an analysis in G' of $m_1 \cdots m_n$. Hence, by definition, the FA-structure underlying $\sigma(A)$ is in $FL(G')$, and this underlying FA-structure is $\sigma(T)$. □

Proposition 1.18. *If $GF(D) \sqsubset G$ then $D \subset FL(G)$.*

Proof. By construction of $GF(D)$, we have $D \subset FL(GF(D))$. In addition, because of Proposition 1.17, we have $FL(GF(D)) \subset FL(G)$. □

Proposition 1.19. *If $RG(D)$ exists then $D \subset FL(RG(D))$.*

Proof. By definition $RG(D) = \sigma_u(GF(D))$ where σ_u is the most general unifier of all the types of each word. So we have $GF(D) \sqsubset RG(D)$, and applying Proposition 1.18 with $G = RG(D)$ we obtain $D \subset FL(RG(D))$. □

Proposition 1.20. *If $D \subset FL(G)$ then $GF(D) \sqsubset G$.*

Proof. By construction of $GF(D)$, there is exactly one occurrence of a given type variable x in a tree of D typed as done in the example. Now, viewing the same tree as a tree of $FL(G)$ at the place corresponding to x there is a type label, say T. Doing so for every type variable, we can define a substitution by $\sigma(x) = T$ for all type variables x: indeed because x occurs once, such a substitution is well defined. When this substitution is applied to $GF(D)$ it yields a grammar which only contains assignments from G — by applying the substitution to the whole tree, it remains a well-typed tree, and in particular the types on the leaves must coincide. □

Proposition 1.21. *When $D \subset FL(G)$ with G a rigid grammar, the grammar $RG(D)$ exists and $RG(D) \sqsubset G$.*

Proof. By Proposition 1.20 we have $GF(D) \sqsubset G$, so there exists a substitution σ such that $\sigma(GF(D)) \subset G$.

As G is rigid, σ unifies all the types of each word. Hence there exists a unifier of all the types of each word, and $RG(D)$ exists.

$RG(D)$ is defined as the application of most general unifier σ_u to $GF(D)$. By the definition of a most general unifier, which works as usual even though we unify sets of types, there exists a substitution τ such that $\sigma = \tau \circ \sigma_u$.

Hence $\tau(RG(D)) = \tau(\sigma_u(GF(D))) = \sigma(GF(D)) \subset G$;
thus $\tau(RG(D)) \subset G$, hence $RG(D) \sqsubset G$. □

Proposition 1.22. *If $D \subset D' \subset FL(G)$ with G a rigid grammar then $RG(D) \sqsubset RG(D') \sqsubset G$.*

Proof. Because of Proposition 1.21 both $RG(D)$ and $RG(D')$ exist. We have $D \subset D'$ and $D' \subset FL(RG(D'))$, so $D \subset FL(RG(D'))$; hence, by Proposition 1.21 applied to D and $G = RG(D')$ (a rigid grammar) we have $RG(D) \sqsubset RG(D')$. □

Theorem 1.23. *The algorithm RG for learning rigid AB grammars converges in the sense of Gold (see Section 1.6.1).*

Proof. Take $D_i, i \in \omega$ an increasing sequence of sets of examples in $FL(G)$ enumerating $FL(G)$, in other words $\cup_{i\in\omega} D_i = FL(G)$:

$$D_1 \subset D_2 \subset \cdots D_i \subset D_{i+1} \cdots \subset FL(G)$$

Because of Proposition 1.21 for every $i \in \omega$ $RG(D_i)$ exists and because of Proposition 1.22 these grammars define an increasing sequence of grammars w.r.t. $\sqsubset$ which by Proposition 1.21 is bounded by G:

$$RG(D_1) \sqsubset RG(D_2) \sqsubset \cdots RG(D_i) \sqsubset RG(D_{i+1}) \cdots \sqsubset G$$

As they are only finitely many grammars below G w.r.t. $\sqsubset$ (Proposition 1.16) this sequence is stationary after a certain rank, say N, that is, for all $n \geq N$ $RG(D_n) = RG(D_N)$.

We have $FL(RG(D_N)) = FL(G)$:

$FL(RG(D_N)) \supset FL(G)$ Let T be an FA-structure of $FL(G)$. Since $\cup_{i\in\omega} D_i = FL(G)$ there exists a p such that $T \in FL(D_p)$.

- If $p < N$, because $D_p \subset D_N$, $T \in D_N$, and by Proposition 1.19 $T \in FL(RG(D_N))$.
- If $p \geq N$, we have $RG(D_p) = RG(D_N)$ since the sequence of grammars is stationary after N. By Proposition 1.19 we have $D_p \subset FL(RG(D_p))$ hence $T \in FL(RG(D_N)) = FL(RG(D_p))$.

In all cases, $T \in FL(RG(D_N))$.

$FL(RG(D_N)) \subset FL(G)$ Since $RG(D_N) \sqsubset G$, by Proposition 1.17 we have

$$FL(RG(D_N)) \subset FL(G)$$ □

1.6.4 Other Cases

The learning problem covered by the RG algorithm is very simple and restricted.

Firstly, the class of grammars we are learning is quite limited:

1. They are AB grammars and not richer categorial grammars.
2. They are rigid, that is each word has only a single type of syntactic behavior. This limitation is not too difficult to overcome: different occurrences of the same word corresponding to different syntactic behaviors can be distinguished. This is sound when the occurrences correspond to words which are really different, such as *that* as a demonstrative and *that* as a complementizer, but it is less convincing when the word is the same like the transitive use of *eat* (*I ate an apple.*) and the absolutive use of *eat* (*I already ate*).

Secondly, we are using input structures which are not so easy to obtain, and which are probably too close to the output that we are looking for:

3. Parse structures (or FA structures) are much too precise; instead of having the complete tree structure labeled by rule applications, it would make more sense to have an *unlabeled* tree structure, partial information about the tree structure of the sentence or even just strings as input to the learning algorithms.

The base algorithm that we presented can be adapted in order to go beyond the limitations enumerated above.

1. A first extension is to learn Lambek grammars, which are discussed in the next chapter, from parse structures. Bonato (2000) shows that Lambek grammars are learnable from parse structures. The same learning mechanism works for minimalist grammars when they are viewed as categorial grammars, though with some complications (Bonato and Retoré, 2001). A strong generalization of these results has been proved: reversible regular tree languages (and dependency grammars too) are learnable from positive examples (Besombes and Marion, 2001).
2. An orthogonal extension is to consider k-valued grammars: in this later case, one has to try to unify types in all possible manners in order to have less than k types per word. This has was studied by Kanazawa (Kanazawa, 1998).
3. Regarding the input structures, the simplest generalization is to consider unlabeled trees: then one has to try all possible labeling with $\backslash_e$ and $/_e$. Going even further one can learn from unstructured sentences that are simply sequences of words: once again this is done by considering all possible structures on such sentences. This extension was investigated by Kanazawa (Kanazawa, 1998).

Each of these extensions increases the complexity of the algorithm considerably, as one can imagine, but nevertheless the existence of learning algorithms for categorial grammars is a good property which is shared by few other formalisms for natural language syntax.

1.7 Concluding Remarks

In this chapter, we introduced the core of categorial grammars, AB-grammars, common to the logical and combinatorial approaches. The salient features that distinguish this kind of grammar from phrase structure grammar are the following:

- A finite set of rules acting on categories, which do not depend on the particular language (this one of the reasons for learnability).
- The lexicon, which associates each word with a category describing its syntactic behaviour.
- As opposed to non-terminals, categories themselves have an internal structure which encodes how a word of this category interacts with words of other categories.

These differences explain the learnability in the sense of Gold for various restricted classes of categorial grammars.

The class of languages generated by AB-grammars is the same as the class of languages generated by Lambek grammars, though it is less simple in AB-grammars to give a simple account for syntactic constructions like (peripheral) extraction (as we have seen for our Example 1.1 with *cosa guarda passare*).

In the forthcoming chapters, we therefore develop a connection with logic: categories are formulas and rules are deduction rules, following an idea of Lambek. This also gives a correspondence with semantics, which already works (in a restricted way) for AB-grammars.

Exercises for Chapter 1

Exercise 1.1. Define an AB grammar for $a^n b^n$

Exercise 1.2. Define an AB grammar which generates the language of well-bracketed expressions such as $(((\,)(\,))$

Exercise 1.3. Consider the following context-free grammar.

$$S \to \wedge SS \mid \neg S \mid p \mid q \mid r$$

It generates formulae in a minimal logical language containing only the propositions p, q and r, the unary logical symbol $\neg$ and the binary logical symbol $\wedge$. It has the property that it generates expressions in Polish prefix notation. For example, $\neg \wedge \neg p \neg q$ is an expression generated by this grammar corresponding to the more usual infix notation $\neg(\neg p \wedge \neg q)$, which would be generated by the context-free grammar below.

$$S \to (S \wedge S) \mid \neg S \mid p \mid q \mid r$$

An advantage of Polish prefix notation is that it does not require any brackets, though people not used to it tend to find it hard to read.

1. Give an AB lexicon which generates the language of the Polish prefix context-free grammar at the start of this exercise.
2. Give a lexicon which generates the language of the infix notation context-free grammar. *Hint:* assign types to the brackets as well as to the logical symbols.

Exercise 1.4. Consider the following AB lexicon for English:

Word	Type(s)
Amy	np
Ben	np
Chris	np
London	np
some	$(np\,/\,n)$
all	$(np\,/\,n)$
a	$(np\,/\,n)$
students	n
homework	n
dislikes	$((np\setminus S)\,/\,np)$
visited	$((np\setminus S)\,/\,np)$
went	$((np\setminus S)\,/\,pp)$
to	$(pp\,/\,np)$

1. Create, for each of the types in the above lexicon, a derivation which uses this type. Try to find some odd or ungrammatical sentences which are derivable using the above lexicon.

2. Extend the lexicon above in such a way that the following sentences become derivable. Show your lexicon is correct by providing derivations of all sentences.
   a) All intelligent students do their homework.
   b) No lazy students do their homework.
   c) Amy returned from Paris.
   d) Amy returned from Paris yesterday.
   e) Amy just returned from Paris.
   f) Ben considers Chris boring.
   g) Chris dislikes all students who listen to Mozart.
3. In sentences 2c and 2d from the previous exercise, (at least) two type assignments are possible for the constituent "from Paris": one where the constituent is assigned the type *pp* and one where it is assigned the type $(np\backslash s)\backslash(np\backslash s)$. Give a derivation for 2c and 2d corresponding to each of these solutions. How could you argue in favor of one or of the other type assignment?
4. Extend the above lexicon to derive the following sentences.
   a) Ben and Amy visited Paris.
   b) Ben visited and disliked London.
   c) Amy went to England and visited London.
   d) Amy went to Paris and to London.
   e) All students and professors like the new library.
   f) Chris read all new and interesting books.

   What can you say about the different types assigned to "and"?
5. How would you analyze the following sentence?
   a) All students who Amy likes listen to Mozart.

Exercise 1.5. The following exercise is inspired by a remark from Quine (1961). Natural language expressions permit (derivational) ambiguities which are not present in the tiny logical language of the previous exercise.

We have, for example that "Amy and Ben or Chris" is ambiguous between "(Amy and Ben) or Chris" and "Amy and (Ben or Chris)". The words "both" and "either" can help disambiguate between the two readings.

(1.8) Amy and Ben or Chris
(1.9) Both Amy and Ben or Chris
(1.10) Amy and either Ben or Chris

1. Give lexical entries for all words above and verify that there are two ways of deriving Sentence 1.8 above, but only one way to derive Sentence 1.9 and 1.10. *Hint:* assign atomic formulas *b* and *e* to "both" and "either" respectively.
2. Which of the two readings for Sentence 1.8 corresponds to Sentence 1.9 and which reading corresponds to Sentence 1.10?
3. Comment on the similarities between the solutions you provided here and your solution to Exercise 1.3.2.

Exercise 1.6. Give the context-free grammar which corresponds the lexicon of Exercise 1.4.

Exercise 1.7. Consider the following context-free grammar, which is already in Greibach normal form.

$$
\begin{aligned}
S &\to aA \\
S &\to bB \\
S &\to aSA \\
S &\to bSB \\
A &\to a \\
B &\to b
\end{aligned}
$$

1. What is the language generated by this grammar?
2. Transform the grammar into an AB grammar.

References

Ajdukiewicz, K.: Die syntaktische Konnexität. Studia Philosophica 1, 1–27 (1935); English translation in McCall, 207–231 (1967)

Bar-Hillel, Y.: A quasi arithmetical notation for syntactic description. Language 29, 47–58 (1953)

Bar-Hillel, Y., Gaifman, C., Shamir, E.: On categorial and phrase-structure grammars. Bulletin of the Research Council of Israel F(9), 1–16 (1963)

Besombes, J., Marion, J.Y.: Identification of reversible dependency tree languages. In: Popelínský and Nepil, pp. 11–22 (2001)

Bonato, R.: Uno studio sull'apprendibilità delle grammatiche di Lambek rigide — a study on learnability for rigid Lambek grammars. Tesi di Laurea & Mémoire de D.E.A. Università di Verona & Université Rennes 1 (2000)

Bonato, R., Retoré, C.: Learning rigid Lambek grammars and minimalist grammars from structured sentences. In: Popelínský and Nepil, pp. 23–34 (2001)

Buszkowski, W.: Discovery procedures for categorial grammars. In: van Benthem, J., Klein, E. (eds.) Categories, Polymorphism and Unification. Universiteit van Amsterdam (1987)

Buszkowski, W., Penn, G.: Categorial grammars determined from linguistic data by unification. Studia Logica 49, 431–454 (1990)

Casadio, C.: Semantic categories and the development of categorial grammars. In: Oehrle, R.T., Bach, E., Wheeler, D. (eds.) Categorial Grammars and Natural Language Structures, pp. 95–124. Reidel, Dordrecht (1988)

Chomsky, N.: The logical structure of linguistic theory (1955); revised 1956 version published in part by Plenum Press (1975); University of Chicago Press (1985)

Chomsky, N.: Formal properties of grammars. In: Handbook of Mathematical Psychology, vol. 2, pp. 323–418. Wiley, New-York (1963)

Chomsky, N.: The minimalist program. MIT Press, Cambridge (1995)

Girard, J.Y.: Linear logic: its syntax and semantics. In: Girard, J.Y., Lafont, Y., Regnier, L. (eds.) Advances in Linear Logic. London Mathematical Society Lecture Notes, vol. 222, pp. 1–42. Cambridge University Press (1995)

Gleitman, L., Liberman, M. (eds.): An invitation to cognitive sciences, Language, vol. 1. MIT Press (1995)

Gleitman, L., Newport, E.: The invention of language by children: Environmental and biological influences on the acquisition of language. In: Gleitman and Liberman, ch. 1, pp. 1–24 (1995)

Gold, E.M.: Language identification in the limit. Information and Control 10, 447–474 (1967)

Greibach, S.A.: A new normal-form theorem for context-free phrase structure grammars. Journal of the ACM 12(1), 42–52 (1965)

Harrison, M.A.: Introduction to Formal Language Theory. Addison Wesley (1978)

Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)

Joshi, A., Schabes, Y.: Tree adjoining grammars. In: Rozenberg and Salomaa, ch. 2 (1997)

Joshi, A., Levy, L., Takahashi, M.: Tree adjunct grammar. Journal of Computer and System Sciences 10, 136–163 (1975)

Kanazawa, M.: Learnable classes of categorial grammars. Studies in Logic, Language and Information. FoLLI & CSLI, distributed by Cambridge University Press (1998)

Lambek, J.: The mathematics of sentence structure. American Mathematical Monthly, 154–170 (1958)

McCall, S. (ed.): Polish Logic, 1920-1939. Oxford University Press (1967)

Morrill, G.: A chronicle of type logical grammar: 1935-1994. Research on Language and Computation 5(3), 359–386 (2007)

Nicolas, J.: Grammatical inference as unification. Rapport de Recherche RR-3632, INRIA (1999), http://www.inria.fr/

Pereira, F.C.N., Shieber, S.M.: Prolog and Natural-Language Analysis. CSLI Lecture Notes, vol. 10. University of Chicago Press, Chicago (1987)

Pinker, S.: Language acquisition. In: Gleitman and Liberman, ch. 6, pp. 135–182 (1995)

Pollard, C., Sag, I.A.: Head-Driven Phrase Structure Grammar. Center for the Study of Language and Information, Stanford (1994) (distributed by Cambridge University Press)

Popelìnskỳ, L., Nepil, M. (eds.): Proceedings of the third workshop on Learning Language in Logic. LLL 2001, FI MU Report series, FI-MU-RS-2001-08. Faculty of Informatics – Masaryk University, Strabourg (2001)

Quine, W.V.: Logic as a source of syntactical insights. In: Jakobson, R. (ed.) Proceedings of the Symposia in Applied Mathematics, Structure of Language and its Mathematical Aspects, vol. XII, pp. 1–5. American Mathematical Society (1961)

Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages. Springer, Berlin (1997)

Schabes, Y., Waters, R.C.: Lexicalized context-free grammars. In: Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics, Columbus, Ohio, pp. 121–129 (1993)

Sikkel, K., Nijholt, A.: Parsing of context-free languages. In: Rozenberg and Salomaa, ch. 2 (1997)