Chapter 8

Combinators and Grammars

"I once asked Bravura whether there were any Kestrels in his forest. He seemed somewhat upset by the question, and replied in a strained voice: 'No! Kestrels are not allowed in this forest!'"
Raymond Smullyan, *To Mock a Mockingbird*

What does the theory presented in the earlier chapters actually tell us? Why should natural grammars involve combinatory rules, rather than the intuitively more transparent apparatus of the λ-calculus? Why are the combinators in question apparently confined to Smullyan's Bluebird, Thrush, and Starling—that is, to composition, type-raising, and substitution? Why are the syntactic combinatory rules further constrained by the Principles of Consistency and Inheritance? What expressive power does this theory give us? How can grammars like this be parsed?

8.1 Why Categories and Combinators?

There is a strong equivalence between (typed and untyped) combinatory systems and the (typed and untyped) λ-calculi, first noted by Schönfinkel (1924), elaborated by Curry and Feys (1958), and developed and expounded by Rosenbloom (1950), Stenlund (1972), Burge (1975), Barendregt (1981), Smullyan (1985, 1994), and Hindley and Seldin (1986). Even quite small collections of combinators of the kind already encountered are sufficient to define applicative systems of expressive power equal to that of the λ-calculus, as will be demonstrated below.

The difference between the λ-calculi and the combinatory systems is that the latter avoid the use of bound variables. One interest of this property lies in the fact that bound variables can be a major source of computational overhead—for example in the evaluation of expressions in programming languages related to the λ-calculus, such as LISP. The freedom that their users demand to use the same identifier for variables that are logically distinct in the sense of having distinct bindings to values in distinct environments means that all the various bindings must be stored during the evaluation. This cost is serious

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

enough that considerable ingenuity is devoted to minimizing it by the designers of such "functional" programming languages. One tactic, originating with Turner (1979b), is to avoid the problem entirely, by compiling languages like LISP into equivalent variable-free combinatory expressions, which can then be evaluated by structural, graph reduction techniques akin to algebraic simplification. We will see that there some rather striking similarities between the combinatory system that Turner proposes and the one that is at work in natural languages.

However, it seems quite unlikely that a pressure to do without variables for reasons of computational efficiency is at work in natural language interpretation.[1] The computational advantage of the combinatory systems is highly dependent upon the precise nature of the computations involved, and it is far from obvious that these particular types of computation are characteristic of linguistic comprehension (although the extensive involvement of higher-order functions in CCG is one property that does exacerbate the penalties incurred from the use of bound variables). Furthermore, the wide acceptance of the idea that the pronoun in sentences like *Every farmer in the room thinks he is a genius* is semantically a bound variable, as assumed in the analysis of such phenomena in section 4.4 in chapter 4, suggests that there is no overall prohibition against such devices at the level of Logical Form or predicate-argument structure. The binding conditions, and in particular Condition C, which are discussed in terms of CCG in Chierchia 1988 and Steedman 1997, are also phenomena that are most naturally thought of in terms of scope (although it has to be said that they do not look much like the properties of the *usual* kind of variables).[2]

It seems more likely that natural grammars reflect a combinatory semantics because combinator-like operations such as composition are themselves cognitively primitive and constitute a part of the cognitive substrate from which the language faculty has developed. Such primitive and prelinguistic cognitive operations as learning how to reach one's hand around an obstacle to a target have many of the properties of functional composition, if elementary movements are viewed as functions over locations. The onset of the ability to construct such composite motions appears to immediately precede the onset of language in children (Diamond 1990, 653–655). Similarly, a notion very like type-raising seems to be implicit in the kind of association between objects and their characteristic roles in actions that is required in order to use those objects as tools in planned action. (The idea that tool use and motor planning are immediate precursors of language goes back to de Laguna's (1927) observa-

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

tions on Köhler's's (1925) work on primate tool use and has been investigated more recently by Bruner (1968), Greenfield, Nelson and Saltzman (1972), Greenfield (1991), and Deacon (1988, 1997), among many others.)

To the extent that languages adhere to the Principle of Head Categorial Uniqueness and project unbounded dependencies from the same categories that define canonical word order, the presumed universal availability of combinatory operations in principle allows the child to acquire the full grammar of the language on the basis of simple canonical sentences alone, on the assumption of chapter 2, that the child has access (not necessarily error-free, and not necessarily unambiguously) to their interpretations. (We will return briefly to the problems induced by exceptions to Head Categorial Uniqueness in chapter 10.)

To see whether this hypothesis is reasonable, we must begin by examining the specific combinators that have been identified above—composition, type-raising, and Schönfinkel's **S**—and ask what class of concepts can be defined using them.

8.2 Why Bluebirds, Thrushes, and Starlings?

The equivalence between combinatory systems and the λ-calculus is most readily understood in terms of a recursive algorithm for converting terms in the λ-calculus into equivalent combinatory expressions. Surprisingly small collections of combinators can be shown in this way to completely support this equivalence. One of the smallest and most elegant sets consists of three combinators, **I**, **K**, and the familiar **S** combinator. The algorithm can be represented as three cases, as follows:[3]

(1)
$$\begin{aligned} \lambda x.x &= \mathbf{I} \\ \lambda x.y &= \mathbf{K}y \\ \lambda x.AB &= \mathbf{S}(\lambda x.A)(\lambda x.B) \end{aligned}$$
where x is not free in y

The combinators **I** and **K** have not been encountered before, but their definitions can be read off the example: **I** is the identity operator, and **K**, Smullyan's Kestrel, is vacuous abstraction or the definition of a constant function. This algorithm simply says that these two combinators represent the two ground conditions of abstracting over the variable itself and abstracting over any other variable or constant, and that the case of abstracting over a compound term consisting of the application of a function term A to an argument B is the Starling combinator **S** applied to the results of abstracting over the function and

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

over the argument. (Given the earlier definition of **S**, it is easy to verify that this equivalence holds.) Since the combinator **I** can in turn be defined in terms of the other two combinators (as **SKK**), the algorithm (attributed in origin to Rosser (1942) in Curry and Feys 1958, 237) is often referred to as the "**SK**" algorithm. It is obvious that the algorithm is complete, in the sense that it will deliver a combinatory equivalent of any λ-term. It therefore follows that any combinator, including composition and type-raising, can be defined in terms of **S** and **K** alone.

The **SK** algorithm is extremely elegant, and quite general, but it gives rise to extremely cumbersome combinatory expressions. Consider the following examples, adapted from Turner 1979b. The successor function that maps an integer onto the integer one greater might be defined as follows in an imaginary functional programming language:

(2) $succ = \lambda x.plus\ 1\ x$

The obvious variable-free definition of this trivial function is the following:

(3) $succ = plus\ 1$

However, the **SK** algorithm produces the much more cumbersome (albeit entirely correct) expression shown in the last line of the following derivation:

(4)
$$\begin{array}{lll} succ & = & \lambda x.plus\ 1\ x \\ & \Rightarrow & \mathbf{S}\lambda x.plus\ 1 \lambda x.x \\ & \Rightarrow & \mathbf{S}(\mathbf{S}\lambda x.plus \lambda x.1)\mathbf{I} \\ & \Rightarrow & \mathbf{S}(\mathbf{SK}plus\mathbf{K}1)\mathbf{I} \end{array}$$

The following is the familiar recursive definition of the factorial function (where *cond A B C* means "if A then B else C"):[4]

(5) $fact = \lambda x.cond(equal\ 0\ x)1(times\ x(fact(minus\ x\ 1)))$

It yields the following monster:

(6)
$$\begin{array}{l} \mathbf{S}(\mathbf{S}(\mathbf{S}(\mathbf{K}\ cond)\ (\mathbf{S}(\mathbf{S}(\mathbf{K}\ equal)(\mathbf{K}\ 0))\mathbf{I}))(\mathbf{K}\ 1)) \\ \qquad (\mathbf{S}(\mathbf{S}(\mathbf{K}\ times)\mathbf{I})(\mathbf{S}(\mathbf{K}\ fact) \\ \qquad (\mathbf{S}(\mathbf{S}(\mathbf{K}\ minus)\mathbf{I})(\mathbf{K}\ 1)))) \end{array}$$

What is wrong with the **SK** algorithm is that it fails to distinguish cases in which either the function or the argument or both are terms in which the variable x does not occur (is not free) from the general case in which both function and argument are terms in x. It is only in the latter case that the combinator **S** is appropriate. Curry and Feys (1958, 190–194) offer the following alternative algorithm:[5]

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

(7) a. $\lambda x.x \;=\; \mathbf{I}$
b. $\lambda x.y \;=\; \mathbf{K}y$
c. $\lambda x.fx \;=\; f$
d. $\lambda x.fA \;=\; \mathbf{B}f(\lambda x.A)$
e. $\lambda x.Ay \;=\; \mathbf{C}(\lambda x.A)y$
f. $\lambda x.AB \;=\; \mathbf{S}(\lambda x.A)(\lambda x.B)$
where x is not free in f, y

This algorithm distinguishes the case (7c) (corresponding to η-reduction), in which the expression to be abstracted over consists of a function term that does not contain the variable and an argument term that *is* the variable. This case immediately preempts a great many applications of **K** (to constants), **I** (for the variable), and **S** (for the application). For example, it immediately gives us what we want for the successor function:

(8) *succ* $=$ *λx.plus 1 x*
$\Rightarrow$ *plus 1*

The new algorithm also distinguishes the cases (7d) and (7e), where either the argument term or the function term do not include the variable. These cases correspond to the familiar functional composition combinator **B**, and the "commuting" combinator **C**, which has not been encountered in natural syntax before, but whose definition is as follows:[6]

(9) $\mathbf{C}fxy \;\equiv\; fyx$

This algorithm gives rise to much terser combinatory expressions. For example, the earlier definition of factorial comes out as follows:

(10) **S**(**C**(**B***cond*(*equal 0*)) *1*)(**S***times*(**B***fact*(**C***minus 1*)))

Like the **SK** set, this set of combinators is complete with respect to the λ-calculi. This result is obvious, since it includes **S** and **K**. More interestingly, in includes other subsets that are also complete. The most interesting of these is the set **BCSI**. This set is complete with respect to the λ-calculus with the single exception that **K** itself is not definable. This set therefore corresponds to the λ-calculus without vacuous abstraction, which is known as the $\lambda_{\mathbf{I}}$-calculus (Church 1940), as distinct from the $\lambda_{\mathbf{K}}$-calculus. Vacuous abstraction is the operation that figured as an irrelevant side effect of Huet's unification algorithm in the discussion in chapter 7 of work by Dalrymple, Shieber, and Pereira (1991; see also Shieber, Pereira and Dalrymple 1996), who used it as an operation on predicate-argument structures, to recover interpretations for VP ellipsis. It is

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

therefore interesting to note the existence of calculi and combinatory systems that exclude it, corresponding to linear, relevance, and intuitionistic logics, and to recall that it is not represented among the syntactic combinatory rules either.

Turner (1979a,b) and others have proposed further cases to optimise and extend similar translations of λ-terms into combinatory equivalents, including combinators corresponding to **T**, the type-raising combinator (which is a natural partner to **C** in (7)), to Curry's Φ, the combinator that is implicit in the coordination rule proposed earlier, and to the "paradoxical" fixed-point combinators that are required to complete the combinatory definition of recursive functions like (10).

What then can we say concerning the nature and raison d'être of the combinatory system **BTS** that we have observed in natural language syntax? The most obvious question is whether this set of combinators is complete. To begin with, note that the linguistic combinatory rules, unlike the systems discussed in most of the literature cited above (but see Church 1940; Barendregt 1981, app. A; Hindley and Seldin 1986), are a *typed* combinatory system. That is to say, rules like the forward composition rule of chapter 3 are defined in terms of (variables over) the types of the domain and range of the input functions and the function that results. Indeed, the syntactic categories of a categorial grammar are precisely types, in that sense. So we are talking about completeness with respect to the simply typed λ^{τ}-calculi. Since mathematicians and computer scientists usually think of functions in this way, the typed λ-calculi are useful and interesting objects.

Interestingly, the paradoxical combinators such as Curry's **Y** and Smullyan's $\lambda x.xx$ are not definable in the typed systems. Since the existence of such fixed-point combinators is what allows the definition of recursion within the pure λ-calculus, recursive functions like *fact* cannot be defined within the pure λ^{τ}-calculi. There is also an interesting relation (discussed by Fortune, Leivant and O'Donnell 1983) to type systems in programming languages like PASCAL and ML.

Exactly the same correspondence holds between typed combinators and the typed λ-calculi as we have seen for the untyped versions. In particular, the **SK** system is complete with respect to the $\lambda^{\tau}_{\mathbf{K}}$-calculus. The **BCSI** system is similarly complete with respect to the $\lambda^{\tau}_{\mathbf{I}}$-calculus. Since the type-raising combinator **T** is equivalent to the combinatory expression **CI**, and since the linguistically observed set **BTS** includes **B** and **S**, it seems highly likely that **BTS** is related to **BCSI** and hence also to the $\lambda^{\tau}_{\mathbf{I}}$-calculus. Certainly **C** is

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

definable in terms of **T** and **B**, as shown by Church (see Smullyan 1985, 113).[7]

The only qualification to the correspondence that I have been able to identify is that the combinator **I** itself does not appear in general to be definable in terms of **BTS**. A combinator corresponding to a special case of **I**, of type $(\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)$, can be defined as **CT**. This is not the true **I** combinator, for it will not map an atom onto itself. Nevertheless, **CT** constitutes the identity functional for first-order functions and all higher types, so this does not seem a very important deviation. We may assume that the $\lambda_{\mathbf{I}}^{\tau}$-calculus constitutes an upper bound on the expressive power of the **BTS** system and that the two are essentially equivalent.[8]

It follows immediately that all of the important constraints on the system as a theory of natural grammars stem from directional constraints imposed upon syntactic combinatory rules by the twin Principles of Consistency and Inheritance, discussed in chapter 4. This observation raises the further question of the expressive or automata-theoretic power of CCG.

8.3 Expressive Power

The way the Dutch cross-serial verb construction was captured in examples like (2) of chapter 6 suggests that CCG is of greater strong generative power than context-free grammar.[9] The Dutch construction intercalates the dependencies between arguments and verbs, rather than nesting them, and therefore requires this greater power, at least for strongly adequate capture. Whether standard Dutch can be shown on the basis of this construction not to be a weakly context-free *language* is of course another question. Huybregts (1984) and Shieber (1985) have shown that a related construction in related dialects of Germanic is not even weakly context-free. It is therefore clear that Universal Grammar has more than context-free power, and the further question of whether standard Dutch happens to exploit this power in a way that makes the *language* non-context-free (as opposed to the strongly adequate grammar) is of only technical interest.

The question is, how *much* more power do cross-serial dependencies demand and does CCG offer? An interesting class of languages to consider is the class of indexed grammars, which are discussed by Gazdar (1988) with reference to the Dutch construction. More recently Vijay-Shanker and Weir (1990, 1994) have argued that several apparently unrelated near-context-free grammar formalisms, including the present one, are weakly equivalent to the least powerful level of indexed grammars, the so-called linear indexed grammars.[10]

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

This section presents an informal version of their argument.

Indexed grammars (IGs) are grammars that, when represented as phrase structure rewriting systems, allow symbols on both sides of a production to be associated with features whose values are *stacks*, or unbounded pushdown stores. We can represent such rules as follows, where the notation [...] represents a stack-valued feature under the convention that the top of the stack is to the left, and where α and β are nonterminal symbols and W_1 and W_2 are strings of nonterminals and terminals, in the general case including nonterminals bearing the stack feature:

(11) $\alpha_{[...]} \quad \rightarrow \quad W_1 \ \beta_{[...]} \ W_2$

Such rules have the effect of passing a feature encoding arbitrarily many long-range dependencies from a parent α to one or more daughters β. The rules are allowed to make two kinds of modification to the stack value: an extra item may be "pushed" or added on top of the stack, or the topmost item already on the stack may be "popped" or removed. These two types of rule can be represented as similar schemata, as follows:

(12) a. "pushing:" $\alpha_{[...]} \quad \rightarrow \quad W_1 \ \beta_{[i,...]} \ W_2$
b. "popping:" $\alpha_{[i,...]} \quad \rightarrow \quad W_1 \ \beta_{[...]} \ W_2$

In general, IGs may include rules that pass stack-valued features to more than one daughter. The most restrictive class of indexed grammars, linear indexed grammars (LIGs), allows the stack-valued feature to pass to only one daughter; that is, W_1 and W_2 are restricted to strings of terminals and nonterminals *not* bearing the stack feature.

It is easy to show that Linear Indexed Grammar (LIG) offers a formalism that will express cross-serial dependencies. I will simplify the Dutch problem for illustrative purposes and assume that the goal is to generate a language whose strings all have some number of nouns on the left, followed by the same number of verbs on the right, with the dependencies identified by indices in the grammar. The following simple grammar (adapted from Gazdar 1988) will do this.

(13)
$$\begin{array}{llll} S_{[...]} & \rightarrow & n & S_{[v,...]} \\ S_{[...]} & \rightarrow & S'_{[...]} & \\ S'_{[v,...]} & \rightarrow & S'_{[...]} & v \\ S'_{[\,]} & \rightarrow & \varepsilon & \end{array}$$

The derivation tree for the string $n_1 \ n_2 \ n_3 \ v_1 \ v_2 \ v_3$ is shown in figure 8.1.

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

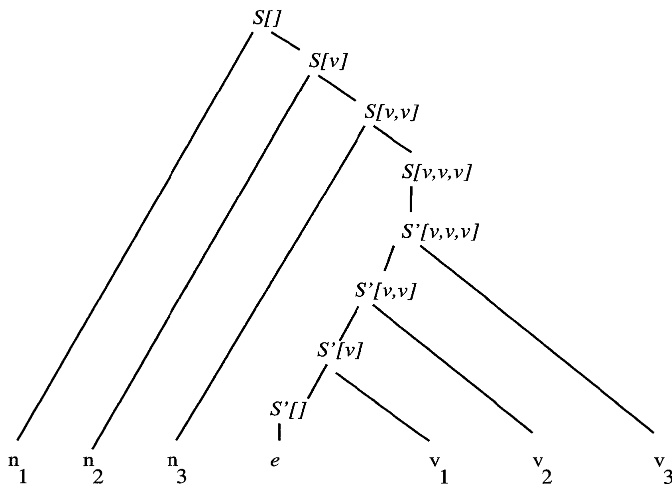

Figure 8.1
LIG derivation for n^3v^3

This is rather reminiscent of the structure produced by the (linguistically incorrect) CCG derivation using crossed composition but lacking type-raised categories shown in figure 6.2. While this particular grammar is weakly equivalent to a context free grammar (since a^nb^n is a context-free language, although a context-free grammar assigns different dependencies), it is equally easy to write a related grammar for the language $a^nb^nc^n$, which is not a context-free language.

Vijay-Shanker and Weir (1990, 1994) identify a characteristic automaton for these grammars, and show on the basis of certain closure properties that it defines what they call an "abstract family of languages" (AFL), just as the related pushdown automaton does. They provide polynomial time recognition and parsing results, of the order of n^6. These results crucially depend upon the linearity property, because it is this property that ensures that two branches of a derivation cannot share information about an unbounded number of earlier steps in the derivation (Vijay-Shanker and Weir 1994, 591–592). This fact both limits expressivity and permits efficient divide-and-conquer algorithms to apply.

Weir (1988) and Weir and Joshi (1988) were the first to observe that there is a close relation between linear indexed rules and the combinatory rules of CCG. Function categories like *give* and *zag helpen voeren* can be equated with indexed categories bearing stack-valued features, as follows:

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

(14) give := $(VP/NP_2)/NP_1 \equiv VP_{[NP_1,NP_2]}$
zag helpen voeren := $(((S\backslash NP_4)\backslash NP_3)\backslash NP_2)\backslash NP_1 \equiv S_{[NP_1,NP_2,NP_3,NP_4]}$

Note that the LIG categories no longer encode directionality—it is up to the LIG rules to do that.

Combinatory rules can be translated rather directly in terms of such categories into sets of LIG productions of the form shown on the right of the equivalences in (15) and (16). Since LIG categories do not capture directionality, the grammar for a particular language will be made up of more specific instances of these schemata involving just those categories that do in fact combine in the specified order for that language.[11]

(15) $X/Y \quad Y \quad \Rightarrow \quad X \quad \equiv \quad X'_{[\ldots]} \quad \rightarrow \quad X'_{[Y,\ldots]} \quad Y_{[\,]}$

(16) $X/Y \quad Y/Z \quad \Rightarrow \quad X/Z \quad \equiv \quad X'_{[Z,\ldots]} \quad \rightarrow \quad X'_{[Y,\ldots]} \quad Y_{[Z]}$

Rule (15) is forward application, realized as a binary LIG rule of the "push" variety. Rule (16) is first-order forward composition, **B**, and involves both pushing a Y and popping a Z. Crucially, the stack, represented as $\ldots$, is passed to only one daughter. The same is true for the substitution rule:

(17) $Y/Z \quad (X\backslash Y)/Z \quad \Rightarrow \quad X/Z \quad \equiv \quad X'_{[Z,\ldots]} \quad \rightarrow \quad Y_{[Z]} \quad X'_{[Z,Y,\ldots]}$

The same linearity property also holds for the rules corresponding to $\mathbf{B}^2, \mathbf{B}^3$ and so on, because the set of arguments of the function into Y is bounded. It would *not* hold for an unbounded schema for a rule corresponding to $\mathbf{B}^n$. This rule, which can be written in the present notation as follows, involves *two* stack-valued features, written $\ldots_1$ and $\ldots_2$:

(18) $X/Y \quad Y/Z\$ \quad \Rightarrow \quad X/Z\$ \quad \equiv \quad X'_{[\ldots_1,Z,\ldots_2]} \quad \rightarrow \quad X'_{[Y,\ldots_2]} \quad Y_{[\ldots_1,Z]}$

It is not currently known precisely what strong generative power such generalized rules engender. They may not take us to the full power of IGs, because the translation from CCG forces us to regard the left-hand side of the rule as bearing a *single* stack feature, which the production nondeterministically breaks into two stack fragments that pass to the daughters. This is not the same as passing the same stack to two daughters—crucially, the two branches of the derivation that it engenders do not share any information, and therefore seem likely to permit efficient divide-and-conquer parsing techniques.

Weir and colleagues treat type-raising as internal to the lexicon, rather than as a rule in syntax, However, Hoffman (1993, 1995b) has pointed out that a

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

similar increase in power over LIG follows from the involvement of type-raised categories like $\mathrm{T}/(\mathrm{T}\backslash NP)$ if T is regarded as a true variable, rather than a finite schematization. To a first approximation, the indexed category corresponding to a type-raised category looks like this:

(19) $\mathrm{T}/(\mathrm{T}\backslash Y) \equiv X'_{[X'_{[Y,\ldots 1]},\ldots 1]}$

Again, the LIG category does not capture the order information, and in particular the order-preserving character, of the original. That has to be captured in the LIG productions, in such facts as that for every instance of rule (15) there is a rule like the following:

(20) $X'_{[\ldots 1]} \quad \rightarrow \quad X'_{[X'_{[Y,\ldots 1]},\ldots 1]} \quad X'_{[Y,\ldots 1]}$

The LIG equivalent of a raised category has two copies of the stack $\ldots_1$. However, as far as functional application goes, it is simply a function like any other—that is, an instance of $\alpha_{[\beta,\ldots]}$. It follows that this rule is simply another instance of (15). Again, no information is shared across branches of the derivation.

However, by the same reasoning, when *two* of the raised categories compose, even via the first-order composition rule (16), so that Y is $T_{[X,\ldots]}$, their two distinct stack variables give rise to a nonlinear production, as follows:

(21) $X_{X_{[Z,\ldots 1,\ldots 2]}} \quad \rightarrow \quad X_{[X_{[Y,\ldots 2]},\ldots 2]} \quad X_{[Z,X_{[Y,\ldots 1]},\ldots 1]}$

This composition has the characteristic noted earlier of nondeterministically partitioning a single stack feature on the left-hand side into two fragments, passed as stack features to the daughters. In effect, the variable transforms bounded composition into the unbounded variety. Again no information is shared across the two branches of the derivation.

Hoffman shows how the language $a^n b^n c^n d^n e^n$ (which is outside the set of linear-indexed languages) can be defined by exploiting this behavior of variables in type-raised categories. It is therefore known that if this property is allowed in CCGs, it raises their power strictly beyond LIG. What is not currently known is how *much* beyond LIG it takes us, or whether CCLs so defined are a subset of IL, the full set of indexed languages.

Alternatively, we can, as suggested earlier, confine ourselves to LIG power by eschewing the general interpretation of composition and type-raising and by interpreting the variables involved in each as merely finite schemata. Such a limitation allows all derivations encountered in parts I and II of the book and

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

keeps CCG weakly equivalent to LIG.[12]

The advantages of keeping to such a limitation are potentially important, as Vijay-Shanker and Weir (1990, 1994) show. As noted earlier, because LIGs pass the stack to only one branch, they limit expressive power and allow efficient algorithms to apply. As a result, Vijay-Shanker and Weir have been able to demonstrate polynomial worst-case complexity results for recognizing and parsing CCGs and TAGs, which are also weakly equivalent to LIGs. It is currently unclear whether similar advantages obtain for the more general class of CCGs. The important fact that neither generalised composition nor variables in type-raised categories pass any one stack feature to more than one daughter gives reason to suppose that they too may be polynomially recognizable using divide-and-conquer techniques.

As Gazdar (1988) has pointed out, it is not clear that the linguistic facts allow us to keep within either of these bounds. The full generality of the Dutch verb-raising construction in noncoordinate sentences can be captured with weakly LIG-equivalent rules, but they allow functions of arbitrarily high valency to be grown. If such functions can coordinate, then we need the full power of IG. This result follows immediately from the fact that the unbounded coordination combinator Φ^n corresponds to a production that passes the same stack feature to two daughters:

(22) $X_{[...1]} \quad \rightarrow \quad X_{[...1]} \quad CONJ \quad X_{[...1]}$

The crucial cases for Dutch are those in which unboundedly long sequences of nouns or verbs of unbounded valency coordinate. However, once the valency or number of arguments gets beyond four, the limit found in the Dutch and English lexicon, the sentences involved become increasingly hard to process, and hard to judge.

Rambow (1994a) makes a similar argument for the translinear nature of scrambling in German. However, this argument depends on the assumption that unbounded scrambling is complete to unbounded depth of embedding. Because these sentences also go rapidly beyond anything that human processors can handle, any argument that either kind of sentence is grammatical depends on assumptions about what counts as a "natural generalization" of the construction, parallel to a famous argument of Chomsky's (1957) concerning the non-finite-state nature of center embedding.

Joshi, Rambow and Becker (to appear) have made the point that this analogy may not hold. They note that all such arguments—including Chomsky's—fall if a lesser automaton or AFL that covers all and only the acceptable cases

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

is ever shown to exist. The status of any residual marginal cases is then decided by that automaton. It is only because no one has yet identified such a finite-state automaton that Chomsky's claim that context-free grammars constitute a lower bound on competence still stands, and is unlikely ever to be overthrown.[13]

It follows that if LIG alone can be shown to be of sufficient power to provide strongly adequate grammars for the core examples, or alternatively if unbounded composition rules and variable-based type-raising are indeed of lesser power than IG, and if a class of automata characterizing an AFL can be identified, the question of whether that lesser power provides an upper bound on natural complexity comes down to the question of whether some exceedingly marginal coordinations and scramblings are acceptable or not. If an automaton exists that is strongly adequate to recognize all and only the sentences that we are certain about, then we might well let that fact decide the margin, in the absence of any other basis for claiming a natural generalization. This is a question for further research, but however it turns out, CCG should be contrasted in this respect with multimodal type-logical approaches of the kind reviewed by Moortgat (1997), which Carpenter (1995) shows to be much less constrained in automata-theoretic terms.

8.4 Formalizing Directionality in Categorial Grammars

In chapter 4, I claimed that the Principles of Adjacency, Consistency, and Inheritance are simple and natural restrictions for rules of grammar. In chapters 6 and 7, I claimed that a number of well-known crosslinguistic universals follow from them. We have just seen that low automata-theoretic power and a polynomial worst-case parsing complexity result also follow from these principles. So quite a lot hinges on the claim that these principles *are* natural and nonarbitrary.

The universal claim further depends upon type-raising's being limited (at least in the case of configurational languages) to the following schemata:[14]

(23) $X \Rightarrow_{\mathbf{T}} \mathrm{T}/(\mathrm{T}\backslash X)$
$X \Rightarrow_{\mathbf{T}} \mathrm{T}\backslash(\mathrm{T}/X)$

If the following patterns (which allow constituent orders that are not otherwise permitted) were allowed, the regularity would be unexplained. In the absence of further restrictions, grammars would collapse into free order:

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

(24) $X \Rightarrow_{\mathbf{T}} \mathrm{T}/(\mathrm{T}/X)$
$X \Rightarrow_{\mathbf{T}} \mathrm{T}\backslash(\mathrm{T}\backslash X)$

But what are the principles that limit combinatory rules of grammar, to include (23) and exclude (24)? And how can we move type-raising into the lexicon without multiplying NP categories unnecessarily?

The intuition here is that *we want to make type-raising sensitive to the directionality of the lexically defined functions that it combines with.* However, the solution of combining type-raising with the other combination rules proposed by Gerdeman and Hinrichs (1990) greatly expands their number.[15]

The fact that directionality of arguments is inherited under the application of combinatory rules, according to the Principle of Inheritance, strongly suggests that directionality is a property of arguments themselves, just like their categorial type, *NP* or whatever, as suggested in Steedman 1987, and as in Zeevat, Klein and Calder 1987 and Zeevat 1988.

Our first assumption about the nature of such a system might exploit a variant of the notation used in the discussion of LIGs above (cf. Steedman 1987), in which a binary feature marks an argument of a function as "to the left" or "to the right." In categorial notation it is convenient to indicate this by subscripting the symbol $\leftarrow$ or $\rightarrow$ to the argument in question. Since the slash in a function will now be nondirectional, both $\backslash$ and $/$ can be replaced by a single nondirectional slash, also written $/$, so that for example the transitive verb category is written as follows:[16]

(25) enjoys := $(S/NP_{\leftarrow})/NP_{\rightarrow}$

(The result S has no value on this feature until it unifies with a function as its argument, so it bears no directional indication. It is just an unbound variable.)

In this notation the (noncrossed) forward composition rule is written as follows:

(26) *Forward composition*
$X/Y_{\rightarrow} \quad Y/Z_{\rightarrow} \quad \Rightarrow_{\mathbf{B}} \quad X/Z_{\rightarrow} \quad (>\mathbf{B})$

The forbidden rule (6) of chapter 4, which violates the Principle of Inheritance, would be written as follows:

(27) $X/Y_{\rightarrow} \quad Y/Z_{\rightarrow} \quad \not\Rightarrow \quad X/Z_{\leftarrow}$

However, given the definition of directionality as a feature of Z, this is not a rule of composition at all. As long as the combinatory rules are limited to operations like composition, only rules obeying the Principle of Inheritance are permitted.

Steedman, Mark. *The Syntactic Process*.
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

The feature in question does not have the equally desirable effect of limiting type-raising rules to the order-preserving kind in (23). Those rules are now written as follows:

(28) $X \Rightarrow_{\mathbf{T}} \mathrm{T}/(\mathrm{T}/X_{\leftarrow})_{\rightarrow}$
$X \Rightarrow_{\mathbf{T}} \mathrm{T}/(\mathrm{T}/X_{\rightarrow})_{\leftarrow}$

Since the input to the rule, X, is unmarked on this feature, there is nothing to stop us from writing the order-*changing* rules in (24):

(29) $X \Rightarrow_{\mathbf{T}} \mathrm{T}/(\mathrm{T}/X_{\rightarrow})_{\rightarrow}$
$X \Rightarrow_{\mathbf{T}} \mathrm{T}/(\mathrm{T}/X_{\leftarrow})_{\leftarrow}$

This is very bad. Although we can easily exclude the latter rules to define grammars for languages like English, we could with equal ease exploit the same degree of freedom to define a language in which *only* order-changing type-raising is allowed, so that the directionality of functions in the lexicon would be systematically overruled. Thus, we could have a language with an SVO lexicon, but OVS word order. Worse still, we could equally well have a language with one of each kind of type-raising rule—say, with an SVO lexicon but a VOS word order. Such languages seem unreasonable, and would certainly engender undesirably cynical attitudes toward life in any child faced with the task of having to acquire them.

Zeevat, Klein and Calder (1987) and Zeevat (1988) offer an ingenious, but partial, solution to this problem. Of the two sets of rules (28) and (29), it is actually the order-*changing* pair in (29) that looks most reasonable, in that the raised function can at least be held to inherit the *same* directionality as its argument. That is, both rules are instances of a schema in which the directionality value is represented as a variable, say, D. In the present notation they can both be conveniently represented by the following single rule:

(30) $X \Rightarrow_{\mathbf{T}} \mathrm{T}/(\mathrm{T}/X_{D})_{D}$

This rule has the attractive properties of being able to combine with either rightward- or leftward-combining arguments and of inheriting its own directionality from them. Since it therefore *only* combines to the left with leftward arguments and to the right with rightward ones, it offers a way around the problem of having multiple type-raised categories for arguments. We can simply apply this rule across the board to yield one type for NPs. In fact, we can do this off-line, in the lexicon, as Zeevat, Klein, and Calder propose.

However, there is a cost in theoretical terms. As noted earlier, since this is the direction-changing rule, the lexicon must reverse the word order of the

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

language. An SVO language like English must have an OVS lexicon. This is in fact what Zeevat, Klein and Calder (1987) propose (see Zeevat 1988, 207–210).

Despite this disadvantage, there is something very appealing about this proposal. It would be very nice if there were a different treatment of the directionality feature that preserved its advantages without implicating this implausible assumption about the lexicon. Of course, as a technical solution we might encode the values $\rightarrow$ and $\leftarrow$ as list structures $[0,1]$ and $[1,0]$, and write a similar single order-preserving rule as follows, using variables over the elements:

(31) $X \Rightarrow_{\mathbf{T}} \mathrm{T}/(\mathrm{T}/X_{[x,y]})_{[y,x]}$

But such a move explains nothing, for we could equally well exploit this device to write the order-changing rule or, by using constants rather than variables, define any mixture of the two. What is wrong is that directionality is being represented as an abstract feature, without any grounding in the properties of the string itself. If instead we define the feature in question in terms of string positions, in a manner that is familiar from the implementation of definite clause grammars (DCGs) in logic programming, we can attain a more explanatory system, in which the following results emerge:

1. The Principle of Inheritance is explained as arising from inheritance of this feature under unification of categories.
2. A single order-preserving type-raised category combining either to the right or to the left can be naturally specified.
3. No comparable single order-*changing* type-raised category can be specified (although a completely order-free category can).

Since these matters are somewhat technical, and since they impinge very little upon linguistics, this whole discussion is relegated to an appendix to the present chapter, which many readers may wish to skip entirely. Since the notation becomes quite heavy going, it is emphasised here that *it is not a proposal for a new CCG notation.* It is a semantics for the metagrammar of the *present* CCG notation.

Appendix: Directionality as a Feature

This appendix proposes an interpretation, grounded in string positions, for the symbols $/$ and $\backslash$ in CCG. This interpretation is easiest to present using unification as a mechanism for instantiating underspecified categories and feature

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

value bundles, a mechanism that has been implicit at several points in the earlier discussion.

For a full exposition of the concept of unification, the reader is directed to Shieber 1986. The intuition behind the notion is that of an operation that amalgamates compatible terms and fails to amalgamate incompatible ones. The result of amalgamating two compatible terms is the most general term that is an instance of both the original terms. For example, the following pairs of terms unify, to yield the results shown:

(32)
$$\begin{array}{lll} x & a' & \Longrightarrow a' \\ f'(g'a') & x & \Longrightarrow f'(g'a') \\ f'x & f'(g'y) & \Longrightarrow f'(g'y) \\ f'a'x & f'yy & \Longrightarrow f'a'a' \end{array}$$

The following pairs of terms do not unify:

(33)
$$\begin{array}{lll} a' & b' & \Longrightarrow \mathit{fail} \\ f'x & g'y & \Longrightarrow \mathit{fail} \\ f'a'b' & f'yy & \Longrightarrow \mathit{fail} \end{array}$$

(Constants are distinguished from variables in these terms by the use of primes.)

Besides providing a convenient mechanism for number and person agreement, unification-based formalisms provide a convenient way of implementing combinatory rules in which X, Y, and so on, can be regarded as variables over categories that can be instantiated or given values by unification with categories like NP or $S\backslash NP$. This observation provides the basis for a transparent implementation of CCG in the form of definite clause grammar (DCG; Pereira and Warren 1980) in programming languages like Prolog (and its higher-order descendant λ-Prolog), and in fact such an implementation has been implicit at a number of points in the exposition above—for instance in the discussion of agreement in chapter 3. A example of a simple (but highly inefficient) program of this kind for use as a proof checker for the feature-based account of directionality that follows is given in Steedman 1991c.

The unification-based implementation has the important attraction of forcing the Principle of Combinatory Type Transparency to apply to combinatory rules interpreted in this way, because of the resemblance between a model-theoretic semantics for unification and the set-theoretic representations of categories (see van Emden and Kowalski 1976; Stirling and Shapiro 1986; Miller 1991, 1995).

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

One form of DCG equivalent of CFPSG rewrite rules like (34a) is the Prolog inference rule (34b), in which `:-` is the Prolog leftward logical implication operator, and `P0, P1, P` are variables over string positions. such as the positions 1, 2, and 3, in (34c) (see Pereira and Shieber 1987 for discussion):

(34) a. $S \rightarrow NP\ VP$
b. `s(P0,P) :- np(P0,P1),vp(P1,P).`
c. $_1$ dexter $_2$ walks $_3$

The Prolog clause (34b) simply means that there is a sentence between two string positions `P0` and `P` if there is an NP between `P0` and some other position `P1`, and a VP between the latter position and `P`. This device achieves the effect of declarativizing string position and has the advantage that, if lists are used to represent strings, the Prolog device of difference-list encoding can be used to represent string position implicitly, rather than explicitly as in (34c) (see Pereira and Warren 1980; Stirling and Shapiro 1986).

The basic form of a combinatory rule under the Principle of Adjacency is $\alpha\ \beta \Rightarrow \gamma$. However, this notation leaves the linear order of α and β implicit. We therefore temporarily expand the notation, replacing categories like *NP* by 4-tuples, of the form $\{\alpha, DP_\alpha, L_\alpha, R_\alpha\}$, comprising (a) a *type* such as *NP*; (b) a *distinguished position*, which we will come to in a minute; (c) a *left-end position*; and (d) a *right-end position.* The latter two elements are the exact equivalent of the DCG positional variables.

The Principle of Adjacency then finds expression in the fact that all legal combinatory rules must have the form in (35), in which the right-end of α is the same as the left-end of β:

(35) $\{\alpha, DP_\alpha, P_1, P_2\}\ \{\beta, DP_\beta, P_2, P_3\} \Rightarrow \{\gamma, DP_\gamma, P_1, P_3\}$

I will call the position P_2, to which the two categories are adjacent, the "juncture."

The distinguished position of a category is simply the one of its two ends that coincides with the juncture when it is the "canceling" term *Y*, which from now on we can refer to as the "juncture term" in a combination. A rightward combining function, such as the transitive verb *enjoy*, specifies the distinguished position of its argument (here underlined for salience) as being that argument's *left*-end. So this category is written in full as in (36a), using a *nondirectional* slash /:

(36) a. enjoy $:= \{\{VP, DP_{vp}, L_{vp}, R_{vp}\}/\{NP, \underline{L_{np}}, \underline{L_{np}}, R_{np}\}, DP_{verb}, L_{verb}, R_{verb}\}$
b. enjoy $:= \{VP/\{NP, \underline{L_{np}}, \underline{L_{np}}, R_{np}\}, _, L_{verb}, R_{verb}\}$

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

The notation in (36a) is rather overwhelming. When positional features are of no immediate relevance in such categories, they will be suppressed, either by representing the whole category by a single symbol or by representing anonymous variables whose identity and binding is of no immediate relevance as "_."[17] For example, when we are thinking of such a function *as* a function, rather than as an argument, we will write it as in (36b), where *VP* stands for $\{VP, DP_{vp}, L_{vp}, R_{vp}\}$ and the distinguished position of the verb is written _. It is important to note that although the binding of the NP argument's distinguished position to its left-end L_{np} means that *enjoy* is a rightward function, the distinguished position is *not* bound to the actual right-end of the verb, R_{verb}, as in the following version of (36b):

(37) $\text{*enjoy} := \{VP/\{NP, \underline{R_{verb}}, \underline{R_{verb}}, R_{np}\}, _, L_{verb}, \underline{R_{verb}}\}$

It follows that the verb can potentially combine with an argument elsewhere, just so long as it is to the right. This property was crucial to the earlier analysis of heavy NP shift. Coupled with the parallel independence in the position of the result from the position of the verb, it is the point at which CCG parts company with the directional Lambek calculus, as we will see.

In the expanded notation the rule of forward application is written as follows:

(38) $\{\{X, DP_x, P1, P3\}/\{Y, P2, P2, P3\}, _, P1, P2\} \ \ \{Y, P2, P2, P3\} \ \Rightarrow \ \{X, DP_x, P1, P3\}$

The fact that the distinguished position must be one of the two ends of an argument category, coupled with the requirement of the Principle of Adjacency, means that *only* the two order-preserving instances of functional application can exist, and only consistent categories can unify with those rules.

A combination under this rule proceeds as follows. Consider example (39), the VP *enjoy musicals*. (In this example the elements are words, but they could be any constituents.)

(39) 1 enjoy 2 musicals 3

$\{VP/\{NP, L_{arg}, L_{arg}, R_{arg}\}, _, L_{fun}, R_{fun}\}$ $\{NP, DP_{np}, L_{np}, R_{np}\}$

The derivation continues as follows. First the positional variables of the categories are bound by the positions in which the words occur in the string, as in (40), which in the first place we will represent explicitly, as numbered string positions:[18]

(40) 1 enjoy 2 musicals 3

$\{VP/\{NP, L_{arg}, L_{arg}, R_{arg}\}, _, 1, 2\}$ $\{NP, DP_{np}, 2, 3\}$

Next the combinatory rule (38) applies, to unify the argument term of the func-

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

tion with the real argument, binding the remaining positional variables including the distinguished position, as in (41) and (42):

(41) 1 enjoy 2 musicals 3

$$\{VP/\{NP,L_{arg},L_{arg},R_{arg}\},_,1,2\} \quad \{NP,DP_{np},2,3\}$$
$$\{X/\{Y,P2,P2,P3\},_,P1,P2\} \quad \{Y,P2,P2,P3\}$$

(42) 1 enjoy 2 musicals 3

$$\frac{\{VP/\{NP,2,2,3\},_,1,2\} \quad \{NP,2,2,3\}}{\{VP,1,3\}}$$

At the point when the combinatory rule applies, the constraint implicit in the distinguished position must actually hold. That is, the distinguished position must be adjacent to the functor.

Thus, the Consistency property of combinatory rules follows from the Principle of Adjacency, embodied in the identification of the distinguished position of the argument terms with the juncture $P2$, the point to which the two combinands are adjacent, as in the application example (38).

The Principle of Inheritance also follows directly from these assumptions. The fact that rules correspond to combinators like composition forces directionality to be inherited, like any other property of an argument such as being an NP. It follows that only instances of the two very general rules of composition shown in (43) are allowed, as a consequence of the three principles:

(43) a. $\{\{X,DP_x,L_x,R_x\}/\{Y,P2,P2,R_y\},_,P1,P2\} \quad \{\{Y,P2,P2,R_y\}/\{Z,DP_z,L_z,R_z\},_,P2,P3\}$
$\Rightarrow_{\mathbf{B}} \quad \{\{X,DP_x,L_x,R_x\}/\{Z,DP_z,L_z,R_z\},_,P1,P3\}$

b. $\{\{Y,P2,L_y,P2\}/\{Z,DP_z,L_z,R_z\},_,P1,P2\} \quad \{\{X,DP_x,L_x,R_x\}/\{Y,P2,L_y,P2\},_,P2,P3\}$
$\Rightarrow_{\mathbf{B}} \quad \{\{X,DP_x,L_x,R_x\}/\{Z,DP_z,L_z,R_z\},_,P1,P3\}$

To conform to the Principle of Consistency, it is necessary that L_y and R_y, the ends of the canceling category Y, be distinct positions—that is, that Y not be coerced to the empty string. This condition has always been explicit in the Principle of Adjacency (Steedman 1987, 405, and see above), although in any Prolog implementation such as that in Steedman 1991c it has to be explicitly imposed. These schemata permit only the four instances of the rules of composition proposed in Steedman 1987 and Steedman 1990, and chapter 4, repeated here as (44) in the basic CCG notation:

(44) *The possible composition rules*

a. $X/Y \quad Y/Z \quad \Rightarrow_{\mathbf{B}} \quad X/Z$ $(>\mathbf{B})$

b. $X/Y \quad Y\backslash Z \quad \Rightarrow_{\mathbf{B}} \quad X\backslash Z$ $(>\mathbf{B}_{\times})$

c. $Y\backslash Z \quad X\backslash Y \quad \Rightarrow_{\mathbf{B}} \quad X\backslash Z$ $(<\mathbf{B})$

d. $Y/Z \quad X\backslash Y \quad \Rightarrow_{\mathbf{B}} \quad X/Z$ $(<\mathbf{B}_{\times})$

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

"Crossed" rules like (44b,d) are still allowed (because of the nonidentity noted in the discussion of (36) between the distinguished position of arguments of functions and the position of the function itself). They are distinguished from the corresponding noncrossing rules by further specifying DP_z, the distinguished position on Z.[19] However, no rule violating the Principle of Inheritance, like (27), is allowed: such a rule would require a *different* distinguished position on the two Zs and would therefore not be functional composition at all.[20] This is a desirable result: as shown in the earlier chapters, the non-order-preserving instances (44b, d) are required for the grammar of English and Dutch. In configurational languages like English they must of course be carefully restricted with regard to the categories that may unify with Y.

The implications of the present formalism for the type-raising rules are less obvious. Type-raising rules are unary, and probably lexical, so the Principle of Adjacency does not obviously apply. However, as noted earlier, we only want the *order-preserving* instances (23), in which *the directionality of the raised category is the reverse of that of its argument.* But how can this reversal be anything but an arbitrary property?

Because the directionality constraints are defined in terms of string positions, the distinguished position of the subject argument of a predicate *walks*—that is, the right-edge of that subject—is equivalent to the distinguished position of the predicate that constitutes the argument of an order-preserving raised subject *Dexter*—that is, the *left*-edge of that predicate. It follows that both of the order-preserving rules are instances of the single rule (45) in the extended notation:

(45) $\underline{\{X, DP_{arg}, L_{arg}, R_{arg}\}}$

$$\Rightarrow \{\mathrm{T}/\{\mathrm{T}/\underline{\{X, \underline{DP_{arg}}, \underline{L_{arg}}, \underline{R_{arg}}\}}, \underline{DP_{arg}}, L_{pred}, R_{pred}\}, -, \underline{L_{arg}}, \underline{R_{arg}}\}$$

The crucial property of this rule, which forces its instances to be order-preserving, is that *the distinguished-position variable DP_{arg} on the argument of the predicate in the raised category is the same as that on the argument of the raised category itself.* (The two distinguished positions are underlined in (45).) Notice that this choice forces the raised NP and its argument to be string adjacent; it is exactly the opposite choice from the one that we took in allowing lexical categories like (36) to unify with arguments anywhere in the specified direction.[21] Of course, the position is unspecified at the time the rule applies, and it is simply represented as an unbound unification variable with an arbitrary mnemonic identifier. However, when the category combines with a predicate, this variable will be bound by the directionality specified in the

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

predicate itself. Since this condition will be transmitted to the raised category, *it will have to coincide with the juncture of the combination.* Combination of the categories in the nongrammatical order will therefore fail, just as if the original categories were combining without the mediation of type-raising.

Consider the following example. Under rule (45), the categories of the words in the sentence *Dexter walks* are as shown in (46), before binding.

(46) 1 Dexter 2 walks 3
$$\{S/\{S/\{NP,DP_g,L_g,R_g\},DP_g,L_{pred},R_{pred}\},_,L_g,R_g\} \quad \{S/\{NP,R_{np},L_{np},R_{np}\},DP_w,L_w,R_w\}$$

Binding of string positional variables yields the categories in (47).

(47) 1 Dexter 2 walks 3
$$\{S/\{S/\{NP,DP_g,1,2\},DP_g,L_{pred},R_{pred}\},_,1,2\} \quad \{S/\{NP,R_{np},L_{np},R_{np}\},DP_w,2,3\}$$

The combinatory rule of forward application (38) applies as in (48), binding further variables by unification. In particular, DP_g, R_{np}, DP_w, and $P2$ are all bound to the juncture position 2, as in (49):

(48) 1 Dexter 2 walks 3
$$\{S/\{S/\{NP,DP_g,1,2\},DP_g,L_{pred},R_{pred}\},_,1,2\} \quad \{S/\{NP,R_{np},L_{np},R_{np}\},DP_w,2,3\}$$
$$\{X/\{Y,P2,P2,P3\},_,P1,P2\} \quad \{Y,P2,P2,P3\}$$

(49) 1 Dexter 2 walks 3
$$\underline{\{S/\{S/\{NP,2,1,2\},2,2,3\},1,2\} \quad \{S/\{NP,2,1,2\},2,2,3\}}$$
$$\{S,1,3\}$$

By contrast, the same categories in the opposite linear order fail to unify with any combinatory rule. In particular, the backward application rule fails, as in (50):

(50) 1 *Walks 2 Dexter 3
$$\{S/\{NP,R_{np},L_{np},R_{np}\},_,1,2\} \quad \{S/\{S/\{NP,DP_g,2,3\},DP_g,L_{pred},R_{pred}\},_,2,3\}$$
$$\{Y,P2,P1,P2\} \quad \{X/\{Y,P2,P1,P2\},_,P2,P3\}$$

(Combination is blocked because 2 cannot unify with 3.)

On the assumption implicit in (45), the only permitted instances of type-raising are the two rules given earlier as (23). The earlier results concerning word order universals under coordination are therefore captured. Moreover, we can now think of these two rules as a single underspecified order-preserving rule directly corresponding to (45), which we might write less long-windedly as follows, augmenting the original simplest notation with a vertical "order-preserving" slash | to distinguish it from the undifferentiated nondirectional slash /:

(51) *The Order-preserving type-raising rule*
$$X \Rightarrow_{\mathbf{T}} \mathrm{T}|(\mathrm{T}|X)$$

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

The category that results from this rule can combine in either direction, but will always preserve order. Such a property is extremely desirable in a language like English, whose verb requires some arguments to the right and some to the left, but whose NPs do not bear case. The general raised category can combine in both directions, but will still preserve word order. Like Zeevat's (1988) rule, it thus eliminates what was earlier noted as a worrying extra degree of categorial ambiguity. As under that proposal, the way is now clear to incorporate type-raising directly into the lexicon, substituting categories of the form $\mathrm{T}|(\mathrm{T}|X)$, where X is a category like *NP* or *PP*, directly into the lexicon in place of the basic categories, or (more readably, but less efficiently), to keep the basic categories and the rule (51), and exclude the base categories from all combination. Most importantly, we avoid Zeevat's undesirable assumption that the English lexicon is OVS, thus ensuring continued good relations with generations of language learners to come.

Although the order-preserving constraint is very simply imposed, it is in one sense an additional stipulation, imposed by the form of the type-raising rule (45). We could have used a unique variable—say, DP_{pred}—in the crucial position in (45), unrelated to the positional condition DP_{arg} on the argument of the predicate itself, to define the distinguished position of the predicate-argument of the raised category, as in (52):

(52) $*\{X, DP_{arg}, L_{arg}, R_{arg}\} \Rightarrow$
$\{\mathrm{T}/\{\mathrm{T}/\{X, \underline{DP_{arg}}, L_{arg}, R_{arg}\}, \underline{DP_{pred}}, L_{pred}, R_{pred}\}, _, L_{arg}, R_{arg}\}$

However, this tactic would yield a completely unconstrained type-raising rule, whose result category could not merely be substituted throughout the lexicon for ground categories like *NP* without grammatical collapse. (Such categories immediately induce totally free word order—for example, permitting (50) on the English lexicon.)

Although it is conceivable that such non-order-preserving type-raised categories might figure in grammars for extremely nonconfigurational languages, such languages are usually characterized by the presence of *some* fixed elements. It seems likely that type-raising is universally confined to the order-preserving kind and that the sources of so-called free word order lie elsewhere.[22]

Such a constraint can therefore be understood in terms of the present proposal simply as a requirement for the lexicon itself to be consistent. It should also be observed that a single uniformly order-*changing* category of the kind proposed by Zeevat (1988) is not possible under this theory.

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

That is not to say that more specific order-changing categories cannot be defined in this notation. As noted earlier, in a non-verb-final language such as English the object relative-pronoun must have the category written in the basic notation as $(N\backslash N)/(S/NP)$, which is closely related to a type-raised category. In the extended notation, and abbreviating $N\backslash N$ as R, this category is the following:

(53) whom := $\{R/\{\mathrm{T}/\{X,\underline{L_{arg}},L_{arg},R_{arg}\},\underline{L_{pred}},L_{pred},R_{pred}\},\text{–},L_{arg},R_{arg}\}$

In fact, provided we constrain forward crossed composition correctly, as we must for any grammar of English, the following slightly less specific category will do for the majority dialect of English in which there is no distinction between subject and object relative-pronouns *who*, or for the un-case-marked relative-pronoun *that*:

(54) who/that := $\{R/\{\mathrm{T}/\{X,\underline{DP_{arg}},L_{arg},R_{arg}\},\underline{L_{pred}},L_{pred},R_{pred}\},\text{–},L_{arg},R_{arg}\}$

In the former category both the complement function and its argument are specified as being on the right. In the latter, the directionality of the complement argument is unspecified. Thus, we need look no further than the relative-pronouns of well-attested dialects of English to see exploited in full all the degrees of freedom that the theory allows us to specify various combinations of order-preserving and non-order-preserving type-raising in a single lexical category.

The account of pied-piping proposed by Szabolcsi (1989), to which the reader is directed for details, is also straightforwardly compatible with the present proposal.[23]

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25

Chapter 9

Processing in Context

[After the second word of Tom wanted to ask Susan to bake a cake*] we have in the semantics a function, which we might call (*Tom want*). ...If the parser is forced to make a choice between alternative analyses, it may make reference in this choice to semantics.*

John Kimball, "Predictive Analysis and Over-the-Top Parsing"

To account for coordination, unbounded dependency, and Intonation Structure, strictly within the confines of the Constituent Condition on Rules, we have been led in parts I and II of the book to a view of Surface Structure according to which strings like *Anna married* and *thinks that Anna married* are constituents in the fullest sense of the term. As we have repeatedly observed, it follows that they must also potentially be constituents of noncoordinate sentences like *Anna married Manny* and *Harry thinks that Anna married Manny*. For moderately complex sentences there will in consequence be a large number of nonstandard alternative derivations for any given reading.

We should continue to resist the natural temptation to reject this claim out of hand on the grounds that it is at odds with much linguistic received opinion. We have already seen in earlier chapters that on many tests for constituency—for example, the list cited in (1) of chapter 2—the combinatory theory does better than most. The temptation to reject the proposal on the basis of parsing efficiency should similarly be resisted. It is true that the presence of such semantic equivalence classes of derivations engenders rather more nondeterminism in the grammar than may have previously been suspected. Although this makes writing parsers a little less straightforward than might have been expected, it should be clear that this novel form of nondeterminism really is a property of English and all other natural languages and will be encountered by any theory with the same coverage with respect to coordination and intonational phenomena. It is also worth remembering that natural grammars show no sign of any pressure to minimize nondeterminism elsewhere in the grammar. There is therefore no a priori reason to doubt the competence theory on these grounds.

The only conclusion we can draw from the profusion of grammatical nondeterminism is that the mechanism for coping with it must be very powerful.

Steedman, Mark. *The Syntactic Process.*
E-book, Cambridge, Mass.: The MIT Press, 2000, https://hdl.handle.net/2027/heb08464.0001.001.
Downloaded on behalf of 18.118.154.25