The Architecture of the Computation*

David Adger
Queen Mary University of London

February, 2020

1 The computational task

Most syntacticians, of whatever theoretical persuasion, will agree that a fundamental property of all human languages is that they pair meanings with sounds or signs (let's call these *forms*) in an unbounded fashion. The syntactic structures of a human language are not appropriately thought of as a finite list. Further, most syntacticians will agree that the form-meaning relationship in human languages is, at heart, hierarchical: forms do not syntactically combine linearly, rather larger form-meaning pairings are built out of smaller ones creating a self-similar hierarchy of phrases. It is the job of syntactic theory to give an account of this pairing and its unbounded nature.

One of Chomsky's earliest contributions is the recognition that a theory of the unbounded construction of hierarchical structures must incorporate a computational system that generates the structures (e.g. Chomsky 1959). The history of generative grammar is to a great extent the history of attempts to specify what that computational system is and how it interacts

*Thanks to Miriam Butt, Stefan Müller, Martin Haspelmath, Peter Svenonius and an anonymous referee for comments on an earlier draft.

with other systems that account for the details of form (phonology) and meaning (semantics and pragmatics).

Of course, syntactic theory shouldn't just provide an account of the fact that the link between linguistic form and linguistic meaning is unbounded. The theory also has to give an account of why *this* pairing, and not some other.

We can raise this question at two different levels. One level is more general and the questions are of the following sort: Are syntactic structures generally binary (Kayne 1994)? Are the syntactic rules that create structure sensitive to hierarchy rather than linear order (Chomsky 1975)? Are syntactic rules insensitive to the number of times they can apply (Adger 2019)? Are the basic units which syntax is composed of discrete rather than fuzzy (Newmeyer 1998)? Can syntactic dependencies span arbitrarily large structures (Chomsky 1977)? etc.

Generative theories of different stripes have tended to agree that the answers to these questions are positive, and that our theories should capture that. Of course, that said, different theories and approaches have taken different perspectives on many of these these issues. Perhaps syntactic structures can be ternary, with rules that can apply exactly twice to fuzzy categories at certain linearly defined points in a string (as could be the case in certain versions of construction grammar). Even at this general level, the correct characterisation of the phenomena is open to question.

The other level is more particular. Here, the questions revolve around specific linguistic phenomena and generalizations about them. I provide one brief example, which I return to in the final section, but this kind of question is the bread and butter of most work in syntax.

Take a simple sentence with three noun phrases, like (1):

(1) [Lilly]$_Z$ teased [the mouse]$_X$ before she devoured [her food.]$_Y$

I have labelled the noun phrases so it's easy to keep track of them.

We can of course ask a simple question about the Subject Z of (1), as follows:

(2) [Which cat]$_Z$ teased [the mouse]$_X$ before she devoured [her food]$_Y$?

We can also ask a question about the Object of *tease*. When we do this in English, there is no noun phrase in the X position, but the question is perfectly well-formed:

(3) What did [the cat]$_Z$ tease X before she devoured [her food]$_Y$?

The meaning can be roughly paraphrased as: what is the thing such that the cat teased that thing before she devoured her food.

In contrast, speakers of English reliably report that asking a parallel question about the Object of *devour*, leaving a gap in the Y position, is degraded:

(4) *What did [the cat]$_Z$ tease [the mouse]$_X$ before she devoured Y?

Here it appears to be impossible to associate the relevant meaning (What is the thing such that the cat teased the mouse before she devoured that thing?) with this form. Indeed, in this case it seems difficult to associate the form with any other meaning either, leading to unacceptability. This then is the question about how forms and meanings pair: why is the form in (4) not associable with a meaning analogously to (3)?

This discussion illustrates that the theory of syntax needs to explain:

(5)
a. the unbounded hierarchical nature of syntactic structure;
b. the general properties of that structure;
c. the specifics of the structure such that it pairs form and meaning in the particular way it does.

There are many possible sources of explanation for these three properties

of syntax: the structure-building system itself; the interface between that system and systems of sound (phrasal phonology, word formation, etc.); the interface to the systems of meaning (variable binding, reference tracking, quantifier scope, etc.); the interface between the computational system and the systems which put it to use in parsing and production; domain general effects of cognition impacting on the system; natural laws acting on the system; the input to the system in terms of the primary linguistic data that the child learner is exposed to, data which is itself affected by historical and sociolinguistic factors.

This paper will focus on the structure building system, what is sometimes called the computational system, as a source of explanation. In some sense it is the fundamental source of explanation in generative grammar, as it accounts for the central question of the unbounded hierarchical nature of the syntax of human language. It also, I will argue, can account for some of the general properties of this hierarchy, and for some aspects of the particulars of form-meaning pairing.

I called the system responsible for creating new pairings between form and meaning the *computational* system. What is the import of the term "computational" here?

2 Computation and Syntax

Turing's (1937) analysis of computation takes it to be the composition of a series of elementary functions which are so simple they cannot be broken down into sub-components.

A function is a mathematical object that takes some input and, for that input, produces an output. It maps from a domain to a range, assigning to elements in the domain (possibly multiple elements) a value from the range. Though functions are mathematical objects, they can of course be physically realised by mechanisms in the world (Gallistel and King 2011), or understood

as models for them.

A very simple means of defining a function is via a look-up table. For example, we can define addition modulo 4 (that is, where the result is the remainder of the sum divided by 4) by a simple table that looks like this:

(6)

	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

Here the two numbers to be added are given as an element from the topmost row and an element from the leftmost column; the result of adding modulo 4 can be got by looking at the cell in the table where the relevant row and column intersect.

However, lookup tables are only useful for finite inputs. Addition in general cannot be given via a look-up table of this sort. Even a very simple function which has an infinite input, for example the function that gives you the next item in the sequence of binary numbers, can't be given by a look-up table, for obvious reasons: the table would have to be infinite. This is where the notion of computation comes into its own.

The classic insight here emerges from Turing's work (as well as that of other mathematicians such as Gödel, Church, Péter, and Post—see Kleene 1952), though some of the relevant concepts stretch back to Babbage and Lovelace (Haigh and Priestley 2015). The ideas that emerged from this research provided a basis for understanding how a finite system consisting of a few fundamental concepts can be used to rigorously define an unbounded outcome. Turing formalized the intuitive notion of an *effective procedure* via an abstract device that could carry out a set of elementary operations on symbols: reading a symbol, writing a symbol and changing the state of the device. A symbol is simply an inscription on some medium with a particular

shape—a purely formal entity. Writing a symbol, and then subsequently reading that same symbol requires that the device have a memory. It is the memory that gives Turing's approach its power, as memory allows the composition of functions: the stored output of previously computed functions can feed, as input, into those same, or other, functions.

Other approaches to the same issue were developed at round about the same time. Simplifying a great deal, in Church's work, functions are composed by a general mechanism that substitutes symbols for other symbols; in Post's earlier approach the system moves and deletes symbols; his later approach rewrites symbols in a way familiar to linguists from phrase structure grammars. These approaches are (extensionally equivalent) ways of defining what we now think of as a computational procedure (again, see Kleene 1952 for discussion and references).

The term "procedure" here is somewhat vexed. It has different meanings, depending on the discipline it is used in. Within computational linguistics, it is usually used to refer to the procedure that analyses a string and assigns a syntactic structure to it. That is not the use intended here. Following the use of the term in computability theory (e.g. Minsky 1967), generative grammar takes (part of) an individual's knowledge of language to be possession of such a computational procedure (Chomsky 1995, 14-15). The term procedure in this sense refers to a particular means of specifying knowledge that ranges over an unbounded domain (cf. Chomsky 1962, 539.

As mentioned earlier, one of the core insights of early generative grammar was that the infinite range of human languages could be best captured by such concepts. Chomsky's approach to this was influenced most by work stemming from Post's ideas, which differed from Turing's in that it formalized procedures for generating sets, as opposed to computing functions (Soare 2013), making it well suited to answer the questions of how to define an unbounded set of hierarchically structured expressions. Chomsky showed how such an approach could be related to (restricted) Turing style machines

to provide an account of the kinds of computational device that could define formal languages, and evaluated the extent to which such mechanisms could be used to give an account of natural languages (Chomsky 1956, Chomsky 1959).

On the surface, one might think that human languages could be viewed as sets of strings of symbols. However, another of Chomsky's early contributions was to highlight the differences between the formal languages Post and others studied and human languages, and to argue that one cannot insightfully treat the latter simply as sets of strings (Chomsky 1963, Chomsky 1980). A fundamental aspect of understanding the syntax of human languages is precisely the pairing of form and meaning, and generating string sets does not give any theoretical purchase on that issue, leaving untouched crucial properties such as ambiguity, sameness of meaning, structure dependency of rules, etc. Generative linguistics therefore takes syntax to be characterised by the *ways* that discrete elements can combine (potentially unboundedly) into hierarchical structures. That means that what is important for the syntax of human language is not the sets of forms defined, but the procedure that defines them and relates them to meaning.

It will be useful to tease out six distinct computational issues when we turn to thinking about the properties of the computational system of human language.

(7)
- a. The nature of the device (the basic operations);
- b. The nature of the procedure;
- c. The structure of memory in the device;
- d. The nature of the symbols;
- e. The relationship between abstract specification of a function, and procedures to compute it;
- f. The difference between procedures that compute a function, and procedures that use it.

We assume that the I-language procedure needs a device to run on. That device will have a particular architecture. Given the abstract architecture of some version of a Turing machine, there is a *processor* that can read and write symbolic material to and from *memory* following a *procedure*. Evidence as to the particular architecture of that device will come from what the best theory is of linguistic and other evidence. Generative linguistics assumes that human beings have, or have the wherewithal to develop, such a device as part of their overall cognitive architecture.

A computational procedure is simply a set of statements, defining an unbounded set of pairings. The procedure itself is fundamentally static, (part of) an individual's tacit knowledge of language. It is not a procedure for processing language.

When, in generative syntax, we use temporal metaphors (e.g."the Verb combines with the Object before the VP combines with the Subject"), this should be understood as a statement about the *logical* structure of the procedure for pairing sound and meaning, not as a claim about how a particular sequence of sounds is processed in real time by a hearer or constructed by a speaker.

Let's examine a simple example to emphasize the point.

Take a Phrase Structure grammar that can generate a sentence like (8).

(8) The cat scratched Anson

(9) a. S → NP VP
    b. NP → Det N
    c. NP → PrN
    d. VP → V NP
    e. V → scratched
    f. PrN → Anson
    g. Det → the
    h. N → cat

These production rules are a generative procedure. Assuming a device with the requisite read/write architecture, and a memory upon which to inscribe and inspect symbols, we can run a computation (more commonly called, in linguistics, a derivation) as follows:

(10)
a. S
b. NP VP (by 9a)
c. Det N VP (by 9b)
d. Det N V NP (by 9d)
e. Det N V PrN (by 9c)
f. the N V PrN (by 9g)
g. the cat V PrN (by 9h)
h. the cat V Anson (by 9f)
i. the cat scratched Anson (by 9e)

A derivation/computation is a sequence of structures (organised symbols—in the case of (10), strings) related by operations. The operations in (10) follow the procedure in (9), and involve substitution of one symbol for another via scanning symbols, writing symbols, and moving the system into new states. In the specification of the procedure itself in (9), there is no temporal dimension.

A computation that links (8) with (9) can be run in a number of ways. (10) gives one of these, but there are other computations which derive the same result. We could, for example, have expanded the Object NP symbol before the Subject one, or inserted the lexical items for the Subject before expanding the VP, etc. These different derivations lead to the same outcome as far as deriving the possible pairings between meaning and form is concerned. For such equivalent computations in this Phrase Structure Grammar system, the procedure defines the structure of the sentence irrespective of which computational steps take place 'first'. In this sense, then, the different routes that the computation can take are logically, though not computationally, equiva-

lent.

We could also run the computation in a different direction entirely. The derivation in (10) starts with the symbol S and proceeds to operate on that symbol. We could equally start with the terminal symbols *scratched*, *the*, *Anson*, *cat*, and run a computation that derives S from these. The specification of the procedure would be the same, but the computation that the device runs would be different. So whether the computation runs top down as in (10), or bottom up, as just suggested, is irrelevant as far as the procedure itself is concerned.

There is no temporal dimension in the procedure then. There is, however, a sense in which there is a dimension of logical precedence. In (10), we expanded NP on line (b) rather than VP, and we expanded VP on line (c). But we cannot expand the NP within VP without first expanding the VP itself. The procedure does not allow this. So here we see the logical precedence inherent in the procedure (the rules or operations) realised as a constraint on the ordering of the lines in the derivation. Metaphorically, the operations have an abstract temporal dimension, and this is reflected in the common way of discussing derivations ("The Verb combines with the Object before the VP combines with the Subject"). However, the procedure itself has no temporal structure (it is a static set of statements).

Since a computation is a composition of a series of functions from symbols to symbols via states of the device, each symbol must be held within some medium (a memory). This is the third computational concept relevant to human language. Each line of the derivation takes up some space, and has access to some of the previous lines (for the case in hand, only access to the immediately preceding line is necessary). To draw on an obvious metaphor, the symbols have an abstract spatial dimension.

So metaphorical time and space are relevant to a computational procedure, inasmuch as the derivations the procedure defines may have dependencies based on rule order and on positions in memory. One might imagine that

the procedure itself may be constructed so as to ensure certain properties of this abstract time and space are dealt with in particular ways. For example, if access to only the preceding line of a derivation is possible, the class of derivations is restricted; or if all non-terminal categories must be expanded before any terminal is, information about terminals would be unavailable for rules expanding categories. For human language, then, it may be that the memory architecture of the device restricts the range of derivational types available to it, and hence the range of functions it can compute.

The question of the architecture of the device, and especially its memory architecture, have barely been touched upon in recent research (but see Adger 2017). Memory for computations, whether in man-made computers or in natural computational systems such as DNA, is generally not flat and uni-dimensional as is a Turing machine tape. Richer memory structures, for example hierarchical memory structures, make for smaller and more efficient computations (Cook and Reckhow 1973). The mechanism that the device uses to fetch items from memory and to place them there may also be richer in structure (e.g. Berwick and Chomsky 2016, 135–137). Memory architecture in the computational system of human language is very much an open question.

The fourth relevant concept in (7) is the nature of the symbols. Computation is defined over symbols without meanings. However, the meanings of the symbols, that is, how they will be put to use, may impact very strongly on the nature of the symbol, and hence the nature of the computation. Thinking of 7 as 1111111 or as 7 or as 111 will impact the nature of the procedure which calculates the successor of 7. A procedure which simply adds a 1 to a sequence of 1s is different from a procedure that looks up the symbol 8, or from a procedure to calculate the successor for a binary number, giving, in this case, 1000.

This is also true for the computational procedure of human language. The specification of that procedure will depend on whether the atoms of the

computation are, for example, taken to be lexical items, with rich internal structure (as in the *Aspects* approach, Chomsky 1967), or as functional categories with minimal feature specification (as in many versions of Minimalism today, e.g. Ramchand and Svenonius 2014, Wiltschko 2014). Further, the richness of the specification of these symbols will be relevant to how the procedure is defined: are features privative, binary, or do they have richer attribute-value structure (Adger 2010)? Is the information carried by the symbols localized to the symbol itself, or connected to the symbol's structural context (cf. the discussion of inheritance in Chomsky 2008)? If so, how much of the structural context can the device running the procedure access (Citko 2014).

The fifth issue is the difference between a function (a set of pairs, etc.) and a procedure (a means for calculating the pairs), corresponding to the difference between the extension and intension of the function. We can see this as analogous to the linguistic abstraction ("the English that Anson speaks", an E-language concept, Chomsky 1986) and the natural object (Anson's I-language). The former is extensional, and, for language, essentially undefinable, while the latter is intensional, a computational procedure pairing meaning and form over an unbounded range, implemented in Anson's brain.

A major source of evidence as to the nature of particular I-languages (someone's English, Mandarin or Navajo) is patterns of form-meaning pairings in those languages, such as the phenomenon discussed in (4). Building on generalizations about these, we can develop theories about those aspects of the procedure that underlie all human languages. Thinking back to the three areas identified at the start of this paper, we bring together evidence from particular phenomena (restrictions on how questions can be constructed. etc.), generalizations across phenomena via the development of theoretical hypotheses (binarity of branching, structure dependency of rules, numerical insensitivity of rules, etc.), and the overarching unbounded nature of syntax. Unlike a programmer who designs a procedure, the linguist's aim is to dis-

cover the actual procedure for particular languages, using such evidence, and, ultimately, to discern what the architecture of the device that is common to all humans might be.

The sixth and final conceptual issue raised in (7) is the distinction between the grammar and the parser. An I-language is a procedure licensing derivations that pair forms and meanings in an unbounded fashion. A parser is a procedure that finds a structure that can be paired with a meaning when it encounters phenomenal data. The way that spoken human language is processed by human beings places particular restrictions on how to do this (for example, spoken language is encountered as a linear, though possibly multidimensional, sequence). Similar constraints, different in detail, hold for sign language. For neither is it obvious that the auditory or visual signals are presented in a way that is usefully connected to the form part of the form-meaning pair that is derived by the computational procedure of an I-language. This means that there may be all sorts of strategies and heuristics used by listeners in conjunction with their actual I-language to associate a form with a meaning. For example, a chart parser will create an ordered 2-dimensional array of the encountered units, and then appeal to the rules of the grammar (the computational procedure) to construct a structure which the grammar associates with the string of units. Human beings are probably not cognitively endowed with chart-parsers, but the basic point holds: the procedure for processing input data is logically distinct from the procedure that pairs forms and meanings. The same issues may hold in the other direction, for whatever device constructs a derivation that is relevant to a thought, though even less is known about this.

There is an alternative conception about the relationship between the parsing procedure and the generative procedure which takes them to be architecturally intimately intertwined, with the architecture of the grammar being dependent on its implementation as part of the processor (Lewis and Phillips 2015). This approach might lead one to expect a grammar sensi-

tive to the time-course of processing, with an architecture that is top-down, left-right. Early work in generative grammar rejected identifying the two procedures (Miller and Chomsky 1963) because of the existence of misalignments between grammatical knowledge and processing. Instead, Miller and Chomsky proposed one system that stores grammatical knowledge as a generative grammar (i.e. a computational procedure), and another that makes use of that knowledge in parsing and other cognitive processing. Both this two-system architecture, and an architecture where the grammar is implementationally dependent on the parser, are logically possible. It is straightforward to map a computational procedure (a grammar) into a procedure for parsing (e.g. Stabler 1991), and whether the grammar is usually presented through computations that have a bottom-up or top-down orientation is irrelevant for this (see also Stabler 2011 and more recently Hunter, Stanojević, and Stabler 2019). It is therefore incorrect to claim that grammatical architectures which typically present computations in a bottom-up fashion are incompatible with the time-course of parsing (e.g. Müller 2018, 523).

I've assumed throughout this discussion that the best way to define the unbounded set of pairs is via a computational procedure. There is an alternative, which would be to define the set of pairs via a specification of the pairs' properties in a more direct fashion. This would essentially amount to saying that any two symbols can be a pair as long as a series of constraints holds of the pair:

(11) $\lambda\langle x,y\rangle.\ P(x, y) \wedge Q(y) \wedge R(x) \wedge ...$

Here the various predicates P, Q and R act as constraints on the form or meaning or the form-meaning pair. There may be further relevant variables beyond form and meaning (for example, the grammatical content of the structures themselves). In such approaches, a grammar is not seen as involving a procedure at all, but is understood rather as a kind of definition as to what a well formed structure can look like (e.g. Rogers 1996, and versions of Lexical

Functional Grammar, Optimality Theoretic Syntax, and Head Driven Phrase Structure Grammar).

3 A Minimalist Architecture

Early generative grammar adopted a particular perspective on the nature of the device and the computational procedure. The procedure was seen to involve two distinct parts: phrase structure rules which license derivations from which phrase markers can be constructed, and transformational rules which map phrase markers to other phrase markers. The phrase structure rules deal in information about linear order, hierarchical clustering, and grammatical category, all aspects of grammar that were known to be important from research on linguistics over millennia. However, in addition to order, category and hierarchy, it became clear that there were non-local dependency relations that also formed part of the grammar of human languages (Harris 1957). To capture these relations, Chomsky, building on Harris's work, developed transformational rules within an explicitly computational (Post-style) framework. These rules delete, insert and permute symbols in the phrase marker derived from the initial computation.

The architecture of the computational system of early generative grammar involves a device which carries out elementary operations following a procedure that maps strings of symbols to strings of symbols in two passes (a Phrase Structure sub-procedure and a Transformational sub-procedure). The device's memory is simply the medium on which those symbols are inscribed. As the theory developed over the 1960s, the initial Phrase Marker came to be associated with certain aspects of semantic interpretation, and the terminal one with (eventually pronounced) surface structure. In this way, the architecture of the computational system is effectively correlated with the two poles of form and meaning. The initial phrase marker, created from the phrase structure rules, is a representation of those aspects of struc-

ture relevant to meaning, while the final phrase marker, generated from the initial one by transformational rules, is a representation of those aspects of structure relevant to pronounced form.

As the theory developed further, however, the form-meaning relationship and the two pass computation were decoupled. This was motivated by the discovery of surface structure effects on meaning, by the development of trace theory, and by issues of learnability. The transformational procedure itself was split in three: as well as mapping from the initial phrase marker to a surface structure, new sets of transformational rules were developed that applied to surface structures but had no effect on form, and only an effect on meaning. These rules dealt with issues such as quantifier scope. Similarly, rules were developed that mapped from surface structure to phonological structure, giving the classic Y-model of grammar (Chomsky and Lasnik 1977). Phrase structure rules create D-structure, and transformational rules map from this to S-structure, and from S-structure to both Logical and Phonetic Form (LF and PF).

The basic architecture of the computational system is, however, the same, with a phrase structure computation distinct from transformational computation. Further development of the theory reduced the amount of information in the rules, with generalizations being stated instead on the structures that the rules derived. Although generative grammar in the Government and Binding era still had a two pass architecture, the explanatory work done by the generative procedure itself was minimized, and empirical patterns were, more and more, handled by constraints (filters) on the phrase structure rules (X-bar theory, binarity) and on the structures created by the computation (ECP, Case Theory, Binding Theory, etc). However, there were also constraints imposed on the transformational component (Subjacency), so that the procedure itself was limited in certain ways. This led to a fairly complex architecture, with multiple sets of rules and a rich system of constraints on the structures created by those rules. This approach was empirically success-

ful, but the complexity of the computational system is obvious.

Since the early 1990s, in an attempt to reduce this complexity, the various separate rule systems were replaced by a single structure building rule, Merge, eliminating the distinction between the phrase structure procedure and the transformational one. Chomsky (1995) suggested that the computational procedure should be taken to be one which maps a set of lexical items into a hierarchically configured object which has the relevant structure to link form and meaning. The procedure does this in a way very similar to the earliest proposals, via a computation mapping symbols to new symbols. However, the mapping does not, unlike phrase structure rules, rewrite symbols: it merely adds information about how they are grouped.

In the 1995 system, the two operations that build structures are Merge and Move: the former takes two disconnected existing structures and combines them; the latter takes a single structure and alters it. But these are not two distinct rule blocks; the Merge and Move operations interweave, so the architecture is a single pass architecture, rather than involving distinct ordered rule blocks.

Chomsky (2004) makes a further theoretically significant proposal: Move is eliminated as a separate rule. Merge is redefined so that it applies to two existing structures, independent of whether they are syntactically disconnected, or whether one is part of the other. There is then a single mechanism that both builds and alters structure, entailing that there is no way of distinguishing structure-building from structure-transformation via different rule types.

This redefinition of Merge sharpens the issues of what the basic operations are and what they operate on. It also highlights the difference between a definitional view of grammar and a computational one, mentioned at the close of the last section. Take the following inductive definition of Merge (see e.g. Chomsky 1995, Chomsky 2005, p12; the version I give here is from Hornstein 2018):

(12) a. if α is a lexical item then α is a syntactic object (the lexicon)
b. If α is a syntactic object and β is a syntactic object then Merge(α, β) is a syntactic object.

(13) For α, β, syntactic objects, Merge(α, β)=$\{\alpha, \beta\}$

This definition of Merge certainly allows unbounded generation of hierarchically structured objects, but leaves a great deal open. Assume a set of syntactic objects which we can label A, ..., D, then the following are also syntactic objects, given (13):

(14) a. {A, B}
b. {A, {A, B}}

The (a) example is licensed by the second clause in (12) together with (13); the (b) example is licensed by taking {A, B} and the item A within {A, B} to be the relevant syntactic objects. It follows from the second clause in (12) that {A, {A, B}} is also a syntactic object. This latter means of creating new syntactic objects has come to be called Internal Merge, but it is simply an application of the definition.

However, as well as these, the definition also licenses the following:

(15) { {A, B} , {A, D} }

This example is got by assuming that a syntactic computation has already taken place and constructed the object { {A, B} , D }. We then take α and β in our definitions to be the objects A and D, deriving (15). This means of creating a new syntactic object has come to be called Parallel Merge (Citko 2005, see also the Sidewards Movement of Nunes 2001).

The purely definitional approach in (12) also allows a wide range of further types of derivation that may or may not be desirable. For example, assume we have the syntactic objects α=C, β=B, and we have already constructed

a syntactic object {A, {A, B}}. Nothing in the definition disallows Merge of C and B, which are both syntactic objects, but given that B is contained within {A, {A, B}}, we have the following result:

(16) { A, {A, {C, B} }}

This licensing of (16), given the previously defined syntactic objects, involves counter-cyclic Merge. Similarly, if we have {A, {C, B} }, then we could take A and B in {A, {C, B} } and derive,

(17) {A, {C, {A, B}} }

This would be an example of a lowering movement.

Treating Merge definitionally, then, is simple, but it is also potentially expansive, allowing in a wide range of operations on symbols and of derivational types. Some researchers have taken the expansiveness of Merge to be a positive, enriching the descriptive toolbox for syntactic theory (e.g. Richards 2001, Takahashi and Hulsey 2009 on countercyclic operations and Nunes 2001, Citko 2005 on Parallel operations).

An alternative perspective is pursued in Adger (2017) and in Chomsky, Gallego, and Ott (in press), where the range of derivational types is restricted by appeal to the architecture of the computational system. To see how this might work, consider, instead of (12), a procedure running on a particular computational architecture. Assume a device, with a set of states, and a memory, as well as the capacity to group symbols. This is the approach sketched by Berwick and Chomsky (2016):

> We can think of the computational process as operating like this. There is a workspace, which has access to the lexicon of atomic elements, and contains any new object that is constructed. To carry a computation forward, an element X is selected from the workspace, and then a second element Y is selected. X and Y

> can be two distinct elements in the workspace, ..., what is called External Merge. Or one can be part of the other, called Internal Merge, ... Berwick and Chomsky 2016

Here there is a device, which has as part of its architecture an operation Select. Select identifies an item X from a memory medium (the workspace), it then identifies another item Y either from memory, or within X itself. The second operation that the device can carry out is Merge, which groups the two elements that Select has identified, and writes the result back to memory (the workspace). As well as the workspace memory, there is a more static memory structure (the lexicon) which can be accessed by the device.

Much is left unspecified in this quote. Does the workspace have structure? How does Select identify the relevant elements? When the result of Merge is written to the workspace, does it replace X and Y, or simply add the result to the workspace? Indeed, we might ask if Select and Merge are correctly characterised, if they are empirically sufficient, and if the assumed memory architecture is correct.

I leave these questions open (though see the cited works above for discussion). I'd like to close this section with the question of what space we might find reasonable hypotheses as to answers to them.

Generative grammar has always taken there to be a computational system at the heart of the human linguistic capacity. The general theory of computational systems is built on certain ideas that have a minimizing effect. For example, there are three principles in classical recursive function theory which allow functions to compose (Kleene 1952): substitution; primitive recursion; and minimization (also called bounded search). These are all designed in a way that one might think of as computationally efficient: they reuse the output of earlier computations. Substitution replaces the argument of a function with another function (possibly the original one); primitive recursion defines a new function on the basis of a recursive call to itself, bottoming out in a previously defined function; minimization produces

the output of a function with the smallest number of steps.

One of the guiding themes of Minimalist Syntax has been the question of whether one can detect anything like this in the computational system of human language. Is the architecture of the computational system organized in such a fashion that it minimizes computations (Chomsky 1995)?

This question is not a question about processing complexity, the nature of human memory structure, or heuristics for extracting structures from signals. It is rather the question of whether the procedure that generates the set of sound-meaning pairs is structured so that the computations it defines are redundant, allow backtracking, use unneeded memory resources, etc., or whether they are structured so as to minimize operations, use minimal memory resources, define short overall computations etc. The intuition guiding minimalist syntax is that pursuing a theory that follows the latter course will give us better explanations for why syntactic phenomena are constrained in the ways they are.

Such an approach can provide a grounding for some of the more general level issues discussed in section 1. The core grouping operation Merge generates unbounded hierarchies without imposing linear order, so that rules appealing to linear order are impossible. Binary branching limits redundant computation over such hierarchies by reducing the range of possible computations. If backtracking is impossible, the system cannot count the number of times a rule has applied, and so human languages do not have numerical limits on rule applications.

It can also provide explanations for the more specific questions about limitations on the form-meaning pairings. Take for example the commonly agreed notion that apparent long-distance syntactic dependencies are actually local (Chomsky 1977). Why should this be? One possibility is to ground the explanation for certain locality effects in processing (e.g. working memory capacity as in Hofmeister and Sag 2010, though see Sprouse, Wagers, and Phillips 2012). Minimalist syntax provides an alternative, which is that min-

imization of memory in the system that generates the form meaning pairings itself is what is responsible for all dependencies being local. What accounts for apparent long distance dependencies raised in section 1 is that, in certain circumstances, maintenance of what is in the system's memory can be extended (see e.g. Keine to appear for empirical evidence). In certain circumstances such memory extension is impossible, limiting possible pairings of form and meaning.

4 Summary

Chomsky's early observations that the hierarchical, combinatorial nature of human syntax requires some kind of a computational procedure to be implemented in the human mind is as valid today as it was 60 years ago. The architecture of the mind requires a device, with a procedure that it follows, and a memory for symbol manipulation. The organization of these three computational components (the device, procedure and memory) changed little over the early history of generative grammar, but a major shift took place in the 1990s when the structure of the device and the procedure was simplified. Simplification of architecture can open up complexity in the range of computations allowed, so a different kind of simplification, minimization of computation, has come to the fore in recent years, and begun to provide approaches to the question of why grammars of human languages have the structure that they do.

References

Adger, David. 2010. A minimalist theory of feature structure. In Anna Kibort and Greville Corbett, eds., *Features: Perspectives on a Key Notion in Linguistics*, 185–218, Oxford: Oxford University Press.

Adger, David. 2017. A memory architecture for merge, http://ling.auf.net/lingbuzz/003440.

Adger, David. 2019. *Language Unlimited: The Science Behind Our Most Creative Power*. Oxford: Oxford University Press.

Berwick, Robert C. and Chomsky, Noam. 2016. *Why Only Us?*. Cambridge, MA: MIT Press.

Chomsky, Noam. 1956. Three models for the description of language. *IRE (now IEEE) Transactions on Information Theory* 2:113–124.

Chomsky, Noam. 1959. On certain formal properties of grammars. *Information and control* 2:137–167.

Chomsky, Noam. 1962. Explanatory models in linguistics. In Ernst Nagel, Patrick Suppes, and Alfred Tarski, eds., *Logic, Methodology and philosophy of science: Proceedings of the 1960 International Congress*, 528–550, Stanford, CA: Stanford University Press.

Chomsky, Noam. 1963. Formal properties of grammars. *Handbook of Math. Psychology* 2:323–418.

Chomsky, Noam. 1967. The formal nature of language. In Eric Lenneberg, ed., *Biological Foundations of Language*, 397–442, New York, N.Y.: Wiley and Sons.

Chomsky, Noam. 1975. *Reflections on language*. New York: Pantheon Books.

Chomsky, Noam. 1977. On *wh*-movement. In P. Culicover, T. Wasow, and A. Akmajian, eds., *Formal Syntax*, 71–132, New York: Academic Press.

Chomsky, Noam. 1980. On binding. *Linguistic Inquiry* 11:1–46.

Chomsky, Noam. 1986. *Knowledge of Language*. New York: Praeger Publications.

Chomsky, Noam. 1995. *The Minimalist Program*. Cambridge, MA: MIT Press.

Chomsky, Noam. 2004. Beyond explanatory adequacy. In Adriana Belletti, ed., *Structures and Beyond: The Cartography of Syntactic Structures*, volume 3, 104–131, Oxford: Oxford University Press.

Chomsky, Noam. 2005. Three factors in language design. *Linguistic Inquiry* 104:1–61.

Chomsky, Noam. 2008. On phases. In Robert Freidin, Carlos P. Otero, and Maria Luisa Zubizarreta, eds., *Foundational Issues in Linguistic Theory*, 133–166, Cambridge, MA: MIT Press.

Chomsky, Noam, Gallego, Ángel, and Ott, Dennis. in press. Generative grammar and the faculty of language: Insights, questions, and challenges. *Catalan Journal of Linguistics* .

Chomsky, Noam and Lasnik, Howard. 1977. Filters and control. *Linguistic Inquiry* 8:425–504.

Citko, Barbara. 2005. On the nature of merge: External merge, internal merge, and parallel merge. *Linguistic Inquiry* 36:475–496.

Citko, Barbara. 2014. *Phase theory: An introduction*. Cambridge University Press.

Cook, Stephen A and Reckhow, Robert A. 1973. Time bounded random access machines. *Journal of Computer and System Sciences* 7:354–375.

Gallistel, Charles R and King, Adam Philip. 2011. *Memory and the computational brain: Why cognitive science will transform neuroscience*, volume 6. John Wiley & Sons.

Haigh, Thomas and Priestley, Mark. 2015. Where code comes from: Architectures of automatic control from babbage to algol. *Communications of the ACM* 59:39–44.

Harris, Zellig. 1957. Co-occurrence and transformation in linguistic structure. *Language* 33:283–340.

Hofmeister, Philip and Sag, Ivan A. 2010. Cognitive constraints and island effects. *Language* 86:366.

Hornstein, Norbert. 2018. On merge. In James McGilvray, ed., *The Cambridge Companion to Chomsky*, 69–86, Cambridge: Cambridge University Press.

Hunter, Tim, Stanojević, Miloš, and Stabler, Edward. 2019. The active-filler strategy in a move-eager left-corner minimalist grammar parser. In *Proceedings of the Workshop on Cognitive Modeling and Computational Linguistics*, 1–10.

Kayne, Richard S. 1994. *The Antisymmetry of Syntax*. Cambridge, MA: MIT Press.

Keine, Stefan. to appear. Locality domains in syntax: Evidence from sentence processing. *Syntax* .

Kleene, Stephen. 1952. *Introduction to Metamathematics*. Amsterdam: North-Holland Publishing Company.

Lewis, Shevaun and Phillips, Colin. 2015. Aligning grammatical theories and language processing models. *Journal of Psycholinguistic Research* 44:27–46.

Miller, George A. and Chomsky, Noam. 1963. Finitary models of language users. In D. Luce, ed., *Handbook of Mathematical Psychology*, 2–419, John Wiley & Sons.

Minsky, Marvin Lee. 1967. *Computation: Finite and Infinite Machines*. Prentice-Hall Englewood Cliffs.

Müller, Stefan. 2018. *Grammatical theory: From transformational grammar to constraint-based approaches*. Berlin: Language Sciences Press.

Newmeyer, Frederick. 1998. *Language Form and Language Function*. Cambridge, MA: MIT Press.

Nunes, Jairo. 2001. Sideward movement. *Linguistic Inquiry* 32:303–344.

Ramchand, Gillian and Svenonius, Peter. 2014. Deriving the functional hierarchy. *Language Sciences* 46:152–174.

Richards, Norvin. 2001. *Movement in Language: Interactions and Architectures*. Oxford: Oxford University Press.

Rogers, James. 1996. A model-theoretic framework for theories of syntax. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, 10–16, Association for Computational Linguistics.

Soare, Robert Irving. 2013. Interactive computing and relativized computability. *Computability: Turing, Gödel, church, and beyond* 203–260.

Sprouse, Jon, Wagers, Matt, and Phillips, Colin. 2012. A test of the relation between working-memory capacity and syntactic island effects. *Language* 82–123.

Stabler, Edward P. 1991. Avoid the pedestrian's paradox. In Robert C. Berwick, Stephen P. Abney, and Carol Tenny, eds., *Principle-based parsing*, 199–237, Springer.

Stabler, Edward P. 2011. Computational perspectives on minimalism. *Oxford handbook of linguistic minimalism* 617–643.

Takahashi, Shoichi and Hulsey, Sarah. 2009. Wholesale late merger: Beyond the a/$\bar{\text{a}}$ distinction. *Linguistic Inquiry* 40:387–426.

Turing, Alan Mathison. 1937. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society* 2:230–265.

Wiltschko, Martina. 2014. *The universal structure of categories: Towards a formal typology*. Cambridge: Cambridge University Press.