Computation in Generative Grammar

‘There are a lot of promissory notes there when you talk about a generative grammar as being based on an operation of Merge that forms sets, and so on and so forth. That’s something metaphorical, and the metaphor has to be spelled out someday. Whether this is a high priority for today or not, I don’t know’ (Chomsky, 2011: 169).

This paper attempts to spell out what ‘computation’ means in the context of generative grammar. It does not present a theory of a particular computational architecture for grammar or an account of how a grammar is realised in the brain. These are matters for empirical (albeit theory-driven) research. My aim isn’t to translate every claim about generative syntax to a claim about computation. That would involve taking a stance on a wide range of theoretical issues beyond my ken (perhaps parallel-merge is blocked by fundamental properties of efficient computation, perhaps it isn’t). Instead, the paper aims to describe what is meant by the claim that a grammar is a function which is computed by the brain.

Generative grammar treats an individual's knowledge of language as a function-in-intension. It is a function because to know a language a person has to be able to grasp a mapping between sounds and meanings. It's intensional because, since there are infinitely many sound-meaning pairs in any given language and since the language user can’t simply know lists of pairings - they need some grasp of the internal structure of the function. Furthermore, since the set of pairings is infinite, the function must be recursively specified. Sentence is an abstract, technical notion meant to capture a particular mapping from sounds to meanings. Superficially, generative grammars describe a person's knowledge of language by recursively enumerating sentences and their structures.

The first section presents some of the problems arising from the notion of computation in generative grammar. Any account of computation in generative linguistics will have to address at least some of these. In the second section, I identify the fundamental source of confusion, an implicit commitment to computational realism in the work of Chomsky and others. In the third section, I argue that the notion of computation has fundamentally altered throughout the history of generative grammar as a result of the trend towards lexicalisation. I then argue that the form that computation that has emerged is much more aligned with Turing's original model of computation in Turing, 1936, than with any other view. Specifically, I argue that the operation which minimalists take to be the ‘simplest computational operation’ is the same as what Turing describes as the most general computational operation. This idea is articulated and defended in the final sections.

What results is a novel account of how generative grammar describes cognitive reality. I am not prepared to say whether this better or even coherent, though I do think it is more coherent than other presentations which is at least a start.

---

1 It may be best thought of as an application of the function mapping sounds to meanings, just as $5^2$ is an application of a function mapping numbers to numbers.

2 Sometimes a distinction is made between strong and weak generation and it is claimed that grammars are tasked with strongly generating the language - enumerating sentence structures rather than sentences. The need for this distinction, as we'll see, is grounded in a symbolic conception of what a sentence is; a relic of a time when sentences were understood as strings rather than functions. If the sentence enumerated correctly maps between sounds and meanings, it de facto has the 'correct' structure.
In this section, I will give a brief survey of some of the problems facing the notion of computation in generative linguistics. One of the primary sources of contention is the fact that generative grammars purport to describe a state, the state of mind corresponding to knowledge of language, in terms of a process for recursively enumerating syntactic structures while at the same time denying that such a process temporally occurs.

‘[A] generative system involves no temporal dimension. In this respect, generation of expressions is similar to other recursive processes such as construction of formal proofs. Intuitively, the proof ‘begins’ with axioms and each line is added to earlier lines by rules of inference or additional axioms. But this implies no temporal ordering. It is simply a description of the structural properties of the geometrical object ‘proof’” (Chomsky, 2007: 6).

Since the processes described by grammars don’t actually occur, their outputs do not exist; a fact embraced by generative theorists.

‘We can discuss the set of expressions or derivations generated by a grammar but in doing so ‘no new entities are postulated in these usages beyond FL [the faculty of language], its states L [some language], and their properties. Similarly, a study of the solar system could introduce the notion HT = {possible trajectories of Halley’s Comet within the solar system}, and the studies of motor organization or visual system could introduce the notions plans for moving the arm or visual images for cats (vs. bees). But these studies do not postulate weird entities apart from planets, comets, neurons, cats, and the like’ (Chomsky, 2001: 41-41).

Whatever the term ‘computation’ denotes in generative grammar, it is not a spatio-temporal process. Before we examine the implications of this fact, though it’s worth highlighting a couple of problems with the comet analogy. Firstly, a comet does actually follow one of these trajectories. In other words, the set of trajectories is a description of paths the comet might take. They constitute a modal claim about the possible behaviour of the comet. In contrast, the set of derivations a grammar generates is not a description of the grammars possible activity. Second, we do not characterise a comet as a device recursively enumerating its trajectories. Trajectories of a comet are determined by things like mass and velocity, properties of the comet which constrain its possible behaviour and apply to other objects as well. Recursivity is not a force like gravity (I am not aware of any other natural object that is characterised as a device for recursively enumerating its possible outputs). Finally, it is often claimed that the outputs of grammars are involved in mediating the interface between systems of phonology and semantics. One might reasonably think that such entities would have to exist in order to do this.

The standard response to these problems is to point out that generative grammar models ‘competence’, not performance. ‘Competence’ is not simply an idealised view of performance but a separate module which ‘underlies and accounts for’, ‘determines’ or ‘is put to use by’ performance (each of these phrases occur without further elaboration in Chomsky, 1964). At other times, it is said that competence is ‘embedded’ in performance systems. None of this terminology suggests that competence is performance viewed at

---

3 In some cases, the temporal aspect of a description is clearly eliminable. For example, reference to syntactic movement as though this were a process by which deeply embedded syntactic objects travel [cyclically] from inner phrases outwards makes no temporal claims. The language of ‘movement’ or ‘successive cyclic extraction’ projects onto the single temporal dimension relations that don’t fit into the two dimensions of hierarchy and linear order. It is a metaphor that allows us to speak of multidimensional syntactic structures while maintaining the intuitiveness of the two-dimensional tree framework (see Rogers, 2003).
a higher-level of abstraction (or that the devices described by competence-theories are higher-level descriptions of devices used for building structures in performance).

This vagueness remains a core feature [or 'bug' depending on your opinion] of the generative enterprise though this isn't to say that some people haven't taken stances on the issue. Some have claimed that competence must be causally implicated in performance (Rey, Barber and Fodor, Bever & Garrett), others reject a causal role for grammars (Collins) while others (Chomsky, Higginbotham) embrace the vagueness and say that the status of the theory should be left to further empirical enquiry.

When not being compared to comets, grammars are compared to the Peano axioms:

"One of the properties of Peano's Axioms PA is that PA generates the proof P of ‘2 + 2 = 4’ but not the proof P’ of ‘2 + 2 = 7’ (in suitable notation). We can speak freely of the property 'generable by PA' of holding of P but not P’, and derivatively of lines of generable proofs (theorems) and the set of theorems without postulating any entities beyond PA and its properties" (Chomsky, 2001: 41-42).

This excerpt has been a constant source of debate over the last two decades and I won't recapitulate the controversy here. The axiom analogy does help clarify matters to an extent. Steps in a proof or derivation can be ordered and when we think about them as mathematical objects there is no need to view that order as temporal rather than structural. The connections between computation and deduction are relatively well understood (see Kripke, 2017 for a defence of the claim that 'computation is a special form of deduction') and whether a set of axioms and associated rules system generate a proof does not depend upon their actually being used to generate that proof.

However, the analogy raises as many questions as it answers. The axiomatic view of a grammar, while aligning with the parsing-as-deduction approach to grammar formalisms (e.g. Shieber et al. 1995) commits us to some internal system of representation, typically a formal language. If this is the case, then it merely pushes our question back a level since we would need a grammar for this language as well. Likewise, axioms only 'say' something if they are interpreted (e.g. their variables are mapped on to some domain by a recursively defined interpretation function) or if there is a corresponding proof system. Without either of these, they are vacuous.

You could opt for a more 'abstract' conception of the relation between axioms and theorems according to which the axioms could realised in some way in the brain, perhaps as constraints on parsing. The theorems derived from them might be understood as abstract properties of the parser. If this is the case, it is unlikely the 'grammar' in use will be generative in any sense.

The abstractness of these claims has understandably lead to difficulties for generative-sympathetic researchers in psychology and cognitive neuroscience: 'The tendency in generative syntax, for example, is to speak as if the computations proposed in syntactic analyses need not be regarded as computations

---

4 ‘Knowledge of language guides/ provides the basis for actual use, but does not completely determine use’ (Boeckx, 2009: 134).

5 ‘A'ill that I am assuming is that whatever one does when parsing and producing take place, grammatical knowledge as characterized by grammars is involved’ (Hornstein, 2009: 171). 'The grammar as a function specifies a recursively enumerable set of structures that encode constraints on the processes that deal with production, perception, and general comprehension of linguistic material' (Collins, 2017: 229).

6 In some cases, the temporal element of derivation is clearly eliminable. Talk of ‘movement' simply projects onto the dimension of time, structural properties of sentences which don't fit into the two-dimensional hierarchical structure of dominance and linear order.

6 One may claim that these axioms are embodied rather than represented. See Pereplyotchik, 2017 for a helpful overview of some of these issues.
that are performed in real time. But why should the null hypothesis be that there is some notion of grammar that is not computed in the brain in real time? This assumption simply makes the link between linguistics and neuroscience harder to bridge, for reasons that are ultimately historical, and not necessarily principled' (Poeppel & Embick, 2005: 12). As Poeppel and others have pointed out, it is in no way clear how the primitive notions identified within generative grammar can be reduced to the units of explanation within neuroscience. Generative grammar appears to be 'ontologically incommensurable' (Poeppel, 2017) with other fields.

One might assume that the talk about functions above means that generative grammars were providing computational level theories in David Marr's sense, that is, characterisations of a function computed by some lower-level algorithmic system. However, Chomsky has made it clear that grammars should in no way be understood as higher-level descriptions of performance systems which is exactly what Marr's computational level of description provides.7

This all leaves generative grammar with a spate of questions which still, half a century after its introduction, require answering. If the process doesn't happen and the outputs don't exist, how is any of this explanatory? How can we decide between competing descriptions of an operation that never actually occurs? What is the difference between a binary operation that doesn't actually produce branching structures and a non-binary operation that also doesn't actually produce branching structure? And at a fundamental level, what is the difference between a rock which doesn't actually implement a process of merge and a brain which also doesn't actually implement a process of merge?

According to Geoffrey Pullum: 'When Chomsky & Lasnik (1977) start talking about the ‘computational system’ of human language (a mode of speaking that rapidly caught on, and persists in current ‘minimalist’ work), the ‘computation’ of which they spoke was one that takes place nowhere: no such computations are ever done, except perhaps using pencil and paper as a syntactic theorist tries to figure out how or whether a certain string can be derived' (Pullum, 2009: 14). Pullum is largely correct here in the claim that syntacticians may perform computations with pencil and paper to test a hypothesis about grammar. However, this practice isn't what is being described when computational properties or processes are ascribed to the faculty of language.

The only thing that is certain is that when Chomsky writes: `we are interested in the discovering the actual computational procedure' (Chomsky, 2016: 4), `actual computational procedure' does not correspond to any actual computational processes. The central questions facing any meta-theory for generative grammar is, how can we meaningfully characterise a grammar as a function if this function does not characterise a real-time process implemented by a lower-level algorithm? And how should we understand the surface appearance of grammars as algorithms for enumerating sentences when this algorithm does not describe `mental reality'? 

7 While appeal to Marr is common in generative linguistics, few researchers attempt to follow the analogy through and treat operations like merge as higher-level descriptions of cognitive processes. A notable exception to this is Neeleman and Van de Koots, 2010. While Chomsky once rejected the comparison to Marr's computational level: 'David Marr’s influential ideas about levels of analysis do not apply here at all, contrary to much discussion, because he too is considering input-output systems...' (Language and Nature p.12, 1995), recent work suggest that he endorses it (Berwick & Chomsky, 2016).

8 I think this problem goes some way to answering Ian Roberts when he asks ‘We are left with a serious question, one which I would like to think is of concern to both linguists and mainstream cognitive scientists: why is mainstream generative syntax overlooked in cognitive science as a whole?'(Roberts, 2014: 22).
Computational Realism

I think that one of the main reasons it has been hard to develop consensus concerning the nature of computation in generative grammar is a philosophical thesis implicit in a lot of work which has not been made explicit; computational realism. Commitment to computational realism enables researchers to separate the theoretical claims they make about computation in linguistics from any empirical claims about the kinds of computational systems actually implemented in the mind-brain. The argument for this position is something like the following:

The Church-Turing thesis holds that several systems for defining computable functions are mathematically equivalent (i.e. Turing machines, lambda calculus). Therefore, and this is a massive 'therefore', all of these systems are best understood as describing something beyond and independent of themselves, real platonic computation. Chomsky expresses this position at times: 'At that time it had come to be understood that there were a number of ways of characterizing recursive functions (theory of algorithms): Turing machine theory, Church’s lambda calculus, and others. All tried to capture the notion of mechanical procedure, and they were shown to be equivalent, so it was assumed – the assumption is called Church’s thesis – that there is only one such system and umpteen notations' (Chomsky, 2009: 389).

I won't say much about the therefore of this argument. Church’s thesis is typically understood as the equivalence thesis in the first premise. The idea that, if two systems are equivalent, there is a third system which they both reflect no doubt requires ancillary philosophical assumptions (we can call this Chomsky's Thesis). My interest in Chomsky’s Thesis stems from its implications for computational cognitive science. It helps to be clear about how this differs from other views. Marr explicitly insists that talk of algorithms requires a commitment to a particular system of representation. When stating what mathematical function is being computed by a system, you can ignore differences between frameworks but when describing the computational system involved, you have to commit yourself to a system of representation. The computational realist, in contrast, takes pure computation to be a notation independent phenomenon. The upshot of this is that computations described by linguists are not connected to any formalism.

I suspect that the implicit commitment to computational realism has done more to generate confusion about the generative programme than the ambiguity of the word ‘recursion’ - algorithm isn't clearly defined either, Moschovakis, 2014. A computational theory, according to this view, is not a theory about a particular computational system; if you think that different systems of representation are ultimately representing the same phenomenon, the details of the formalisms can be overlooked. Nor does it identify a function that is computed by some performance systems (as happens in Marrian computational level theories). It is, instead, something more abstract.

There is an important role for abstract, representation-independent questions about computation in cognitive science. For example, valuable work has been done in framing an appropriate notion of

---

9 Lobina follows Dean in calling it 'algorithmic realism'.

10 There are numerous formulations of Church's thesis (identifying Recursive function, lambda definable, and effectively calculable functions) and the Church-Turing thesis (identifying the latter with functions computable by a Turing machine).

11 This isn't the place to make this case in detail. For an overview of attempts to describe computation ‘in general’ rather than tied to particular formalisms or machines, see Gurevich, 2012. Moschovakis contrasts his view with the ‘standard view’ that ‘there are no algorithms, only implementations, variously called machines or models of computations' (Moschovakis, 1998: 71).
computational tractability computational cognitive science. While questions about computational complexity are typically formulated relative to the Turing machine model of computation, researchers do assume an invariance thesis; that is given a reasonable encoding of an input and a reasonable machine, the complexity of a problem will differ by, at most, a polynomial amount. The upshot of assuming this thesis (plausible but unproven) is that differences between systems of computation can be ignored (see van Rooij, 2008, for a detailed and informative discussion of how this works). However, these ability-ascriptions are typically claims about performance, not competence and the transferal of computational concerns from operations which do occur to operations that don’t is what raises the problems above.

Thomas Graf (Graf, 2013, 2017) has shown that, computationally speaking, the difference between features and constraints, is not significant. Every constraint statable in MSO (monadic second-order logic) can be replaced by subcategorisation features on lexical items and vice versa. However, Graf cautions against inferring from this equivalence that debates concerned the superiority of features or constraints are obsolete. This is an important point. The implicit commitment to computational realism doesn’t mean that inquiry into the nature of the computation involved in linguistic competence needs to stop. It would be unwise to conclude that, since computation is abstract and system-independent, it doesn’t matter what models of computation we rely on when developing our theories. There are still ways of viewing things which are more or less intelligible than others.

If generative grammar is to integrate with the other cognitive sciences, a commitment will have to be made to some kind of computational framework, or at the very least we will have to make claims about what aspects of the computational system underlying competence are supposed to be implemented. As Neeleman & van de Koot put it: ‘If competence theory has nothing to say about the computations performed by speakers and hearers, one may well ask what object the theory describes’ (Neeleman & van de Koot, 2010: 184).

In the rest of this paper, I will argue for a particular view of the object of linguistic computation. Key to this argument is the idea that the computational commitments of generative grammar have changed over the last 60 years in a way which has altered our conception of what a grammar and a language is.

**Computational Objects**

This section will present a brief overview of two technical changes which have occurred since the earliest days of generative grammar which have led to a reconceptualisation of the symbol. We’re focussing on the nature symbols here because there is a traditional view according to which ‘computation is just the manipulation of symbol tokens on the basis of their shapes’ (Harnad, 1995: 379). I will argue here that, while this model might cover early generative phrase-structure grammars, it is not an appropriate way to think about later work.

The first phrase-structure grammars were unambiguously rules for manipulating symbols on a page. Both Emil Post, who developed the formalism for representing the activity of string manipulation, and

---

12 Computational equivalence in this case is understood as ‘the ability to license exactly the same sets of derivations and output structures’ (Graf, 2017: 28).

13 They call this question ‘Ristad’s problem’: ‘What object is a competence grammar a theory of, given that it abstracts away from language computation?’ (Ibid)
Chomsky who adapted it to natural languages were explicit that the symbols invoked were written and rewritten.\textsuperscript{14} For example, A rule like:

\[
Z \rightarrow aZb
\]

states that the symbol \(Z\) can be rewritten as the symbol \(Z\) with the terminal symbols \(a\) and \(b\) occurring on either side of it. The upper-case symbols here were understood initially as variables ranging over sets of strings while the lower-case symbols were treated as terminals, strings of symbols over which the upper-case symbols ranged.

The traditional view is that a grammar comprised of such rules recursively enumerates the sentences [and syntactic structures] of a language. \[W\]hen the theory of transformational grammar is properly formulated, any such grammar must meet formal conditions that restrict it to the enumeration of recursive sets' (Chomsky, 1965: 208).\textsuperscript{15} A set is recursively enumerable if it is in the range of a general recursive function of a single variable or empty. Taking the empty condition, a standard tool for understanding recursive enumeration is to imagine a Turing machine presented with an empty tape which subsequently prints all the members of a set upon the tape. Such a machine implements a function for recursively enumerating a set of elements. If the elements enumerated are ordered pairs, then we can view the enumeration as an orthogonal definition of a function.\textsuperscript{16} While the idea that grammars literally manipulated marks on a page (or tape) was rejected early on in the generative project, the view of symbols as 'abstract marks' in some sense remained. This view treats lexical items as abstract, simple and inert outputs of grammars while grammars ground and explain which syntactic structures are in a language. It will be important to bear this in mind for later - the lexical objects are marks on the tape.

Two principally technical changes over the following 40 years changed the relations between these components; the introduction of features and the lexicalisation of syntactic information. Features were first introduced to generative grammar by the philosopher Gilbert Harman (Harman, 1963) but came into common use after \textit{Aspects of the Theory of Syntax}. The effect of the introduction of features was to render grammars intensional. By this, I mean that the variables which state the rules within the grammar are not interpreted over a set of strings but in terms of properties independent of the language. This was an important step in the process of detaching the concept of a grammar from the view of language as a set of sentences.

It was lexicalisation, though, which radically altered the concept of the symbol. This happened through a succession of stages. In the system of \textit{Aspects}, rather than understanding lexical items as bare, atomic

\begin{itemize}
  \item Suppose that we interpret each rule \(X \rightarrow Y\) of (13) as the instruction rewrite \(X\) as \(Y\)' (Chomsky, 1957:26) '\(i\)t seems reasonable to require for significance some guarantee that the grammar will actually generate a large number of sentences in a limited amount of time' (Chomsky, 1956: 118) Over time, this temporal element was 'idealised away' (ignored) and generation was understood as a logical notion.
  \item A couple of things are worth noting about this. i. We shouldn't mistake 'enumeration of recursive sets' with recursively enumerable sets (the former are a proper subset of the latter). Chomsky's point here is that grammars should be weaker than Turing machines. ii. This idea originally stems from Putnam, 1961. By the 80s when Chomsky was ascribing it to Dummett, Chomsky writes 'while languages may be recursive, there is no reason to suppose that this must be so' (Chomsky, 1980:122) and 'languages may not be recursively definable' (Ibid, 138). iii. Lobina takes this as evidence that 'recursive enumeration' is the primary sense in which generative grammars are recursive. 'A recursively enumerable set is a collection of items for which there is an algorithm that can list all its members... It is recursively enumerable sets that Chomsky has in mind, and it is worth pointing out that he has made reference to the influence of mathematical logic on his work on numerous occasions, and in a language clearly suggestive of the RC' (Lobina, 2017: 55). I think this is an entirely plausible reading of the first few decades of generative grammar.
  \item For example, the function \(g: X \rightarrow Y\) can be enumerated by the function \(f\), if \(f\) enumerates a set of ordered pairs comprised of elements from \(X\) and \(f(X)\).
\end{itemize}
symbols in a sequence, ‘each lexical entry is a set of features’ (Chomsky, 1965: 142). It was suggested that items in the lexicon contained their own subcategorisation frames which helped determine their combinatorial properties. With Remarks on Nominalisation and the development of X-Bar theory, category-specific rules were replaced with a single, universal form for clause structure (and clause construction). And later, with the ‘Projection Principle’, introduced in Chomsky, 1981, it was suggested that representations at each syntactic level ‘are projected from the lexicon in that they observe the subcategorization properties of lexical items’ (Chomsky, 1981: 29). In other words, lexical items project clausal structure.

The idea that structure was ‘projected’ from the lexicon rather than generated by the application of syntactic rules played a key role in the shift from broadly ‘top-down’ to ‘bottom-up’ views of structure building. The subcategorisation features of lexical ‘heads’ rather than rules for building phrase-structure, were now understood to determine the structure of phrases. This isn’t to say that structure-building in the reign of Government and Binding Theory, was confined to projection alone. Movement rules were also required to produce structures with wh-fronting, topicalisation, inversion and other phenomena.

It is this process of lexicalisation that rendered the Minimalist Program possible. Minimalism attempts to reduce syntactic structure-building to a single computational operation, merge, which takes two syntactic objects and makes a new syntactic object containing them. This operation is said to be ‘the simplest computational operation, embedded in some manner in every relevant computational procedure’ (Chomsky, 2013: 40) (more on this claim later).

Hornstein, 2017, breaks the definition of merge into two parts, an inductive definition of syntactic object and a statement of merge.

(1) a. If $\alpha$ is a lexical item then $\alpha$ is an SO 
b. If $\alpha$ is an SO and $\beta$ is an SO then Merge($\alpha,$ $\beta$) is an SO

(2) For $\alpha, \beta$, SOs, Merge($\alpha,$ $\beta$) $\rightarrow \{\alpha,$ $\beta\}$

The output of merge is a syntactic object which may be more structured depending on how many times merge has operated upon it.

The main accomplishment of the Minimalist Program is to collapse the distinction between the phrase-building part of a grammar and the movement part. Hierarchical structures are formed by merging syntactic objects into sets while movement is carried out by merging sets with their members (e.g. the syntactic object $\{\alpha, \beta\}$ can be merged with the $\beta$ contained within it to form a new object $\{\beta, \{\alpha, \beta\}\}$). What enables minimalists to explain syntactic diversity from such a simple operation is the fact that all the complexity is built into the lexical items themselves. In other words, syntactic objects drive computations. Whether syntactic objects merge with one another is determined by their features and

---

17 Structure-building, during the reign of government and binding theory, was not confined to projection alone. Movement rules were also required to produce structures with wh-fronting, topicalisation, inversion and other constructions.

18 There are advantages and disadvantages to this idea. On the one hand, lexical items have to be learned anyway so this relieves some of the burden on universal grammar. On the other hand, we are left with the question of where such internal complexity comes from if not merge. For a discussion of some of the challenges here, see Boeckx, 2010.
any structures that emerge are ‘trivial outputs of lexically encoded information combined with general
principles of constituent-structure building’ (Borer, 2017: 122).

It may look as though we are dealing here with two different methods of recursively enumerating
syntactic structures. A language could be viewed as the set of objects enumerated by either an early
phrase-structure or transformation grammar or by a lexicon closed under the operation merge.
However, the superficial similarity between these systems disguises a profound change in the underlying
conception of what a grammar is. While early generative grammars could be understood as sets of
rules for manipulating, combining and moving symbols, later minimalist grammars provide descriptions
of the internal properties of those symbols, lexical and functional items. Indeed, it’s inappropriate to
view the lexical items within later grammars as symbols of any kind as the items upon which merge
operates are themselves complex computational objects. Lexical items are not ‘operated upon’ in the
way that symbols are operated on in rewrite systems. They are not the subject of ‘rules’.

In the next section, I am going to argue that it would be wrong to view the objects manipulated within
a minimalist grammar by analogy with symbols on a tape and that they are instead more like the states
of a Turing machine.

**Turing on Computation**

In this section, I will introduce Turing machines and to encourage a particular way of thinking about
'states'. There are two important ideas I want to get across. The first is that a Turing machine formalism
describes *states* rather than processes. The second is that a Turing machine always 'implements' two
functions, the program function and the machine (or transition) function.

A Turing machine is formally represented as a quintuple \( \{Q, \Sigma, \Gamma, \delta, q_0\} \) comprised of a set of states
\( Q \), an alphabet of symbols \( \Sigma \) which the machine can produce, an alphabet which the machine can read
on the tape \( \Gamma \), a transition function \( \delta \), and a unique start state \( q_0 \). The complexity of the actions that the
machine is able to perform determines its computational power. These actions are stated by the
transition function. For example, the transition function for a Finite State Automaton maps a state-
symbol pair to a state i.e. \( Q \times \Sigma \rightarrow Q \). This tells us that the machine, in a state, can read a symbol and
change its state. In the case of a push-down automaton, the transition function maps a triplet, a state, a
tape-symbol, and whatever symbol is at the top of the memory stack, to a pair, a state and symbol on
the top of the stack i.e. \( Q \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{\lambda\}) \rightarrow Q \times (\Gamma \cup \{\lambda\}) \). That is, the machine, in a state,
can read a symbol and, depending upon what is at the top of the stack, it can change its state and push
a symbol onto the top of its memory stack. This more complex combination of actions as
characterised by the transition function provides the machine with a greater computational power.

A Turing machine is a set of states each of which grounds one or more actions. What counts as a
symbol or a ‘tape’ is relative to the machine and not too important for the description. The physical

---

19 These were originally confined to morphological features ‘derivations are driven by morphological properties to which syntactic
variation of languages is restricted’ (Chomsky, 1993: 194). At times, it has been necessary to preserve this simplicity by proposing
more complex features (e.g. EPP features).

20 It may be the case that grammars do not generate languages at all. Given the epiphenomenal nature of the notion of ”language"
this would not be a particularly disturbing discovery. It would mean that the real systems that are mental represented do not happen to
specify recursively enumerable languages' (Chomsky, 1980: 122-123). Lobina.
realisation of these states is irrelevant for the description of the machine as is how the actions are carried out; whether one is actually printing a tape, marking a disk, pouring a pint, or implementing a Turing machine in some other Turing machine. All that is required is that the states ground the actions which are stated by the transition function. Consider the following very simple machine:

\[
\begin{align*}
Q_0 & : R Q_0 Q_0 : 0 \rightarrow Q_1 \\
Q_1 & : L Q_1 Q_1 : 0 \rightarrow Q_2 \\
Q_2 & : 1 \rightarrow Q_2 Q_2 : 0 \rightarrow Q_3
\end{align*}
\]

This machine has 4 states. Each line in the description characterises the dispositional properties of those states. It tells us the stimuli to which the state is sensitive \{0,1\} and the actions it performs \{1, 0, L, R\}. The key to computation is that two actions are involved, an action on the tape \{printing, moving\} and a change of state. A relation like 0:1 characterises how the state is disposed to respond in the face of certain input. If it sees a 0, it will print a 1. While a relation like Q_0 ... Q_1 characterises a state-change pair. A simple single-tape, binary machine like the one above has 24 distinct stimulus-response pairs (transductions) and 6 state-change pairs. The program which the machine runs is determined by the states and the order they come in while a run is determined by what is on the tape.

The relationship between states and the actions they ground matters more for the description of the machine than their temporal succession in any given run. A mere description of the performance of a run of the machine, or any number of runs of the machine would not tell us canonically what the actual transition function is. This structure is not merely linear as states will connect to different states depending on what is on the tape (and so it is best not to think of the machine as describing a temporal order).

This characterisation of a Turing machine as a set of states which ground a set of dispositions to act is somewhat non-standard but is closer to how Turing’s original proposal. Regardless of whether you are dealing with a finite-state automaton or a full Turing machine, one thing is constant across models of computation, the internal states of the machine must be able to connect with one another. This point was highlighted by Turing when introducing his a-machines:

‘The most general single operation must therefore be taken to be one of the following: (A) A possible change (a) of symbol together with a possible change of state of mind. (B) A possible change (b) of observed squares, together with a possible change of state of mind. The operation actually performed is determined, as has been suggested on p. 250, by the state of mind of the computer and the observed symbols. In particular, they determine the state of mind of the computer after the operation is carried out’ (Turing, 1936: 251).

‘The most general single operation’ is actually two operations happening at once; a change on the tape (either the symbol or the location of the tape-head) and a change of state. We can think of this as an external change and an external change. Loops demonstrate that it is possible to have an external change without an internal change. It is best, though, not to regard this as an ’operation’ as the term

---

21 It is possible though to dissemble this set of states and recombine them into a different program. For example, if our machine started with Q_2 merged with Q_1 followed by Q_0 and then Q_3, it would compute the binary operation from <x, y> to <x - 1, y>.

22 ‘The behaviour of the computer at any moment is determined by the symbols which he is observing, and his ‘state of mind’ at that moment’ (Turing, 1936: 250).
'operation' suggests that there is some further machine carrying out the operation of shifting states. States are not inert entities to be manipulated by some higher process. A state is a functionally characterised atom of computation which can be connected to other states. And typically, in a machine, states are selective of what other states they connect to. The most general operation tells us this fact. It is necessary to describe the transition function of the machine but not a process which runs on the machine.

The most general operation is, in a sense, another description of the transition function. The transition function tells us the relation between a state and its features and some other state and its features:

\[ Q \times \left( \sum U \{ \lambda \} \right) \times \left( \Gamma U \{ \lambda \} \right) \rightarrow Q \times \left( \Gamma U \{ \lambda \} \right) \]

The most general operation, tells us that when certain properties are checked, the machine changes state. At a more abstract level, this operation connects states within a machine. It is best not to view it as an operation.

**Computation in Generative Grammar**

We are now in a position to give a rough formulation of the main claim of this paper; merge should be understood as the linguistic manifestation of Turing’s general operation rather than as any operation upon symbols. Once we take this approach, several of the problems we have discussed disappear and several other problems emerge. I’ll begin with the positives.

If this view is correct, the initial concern about how an I-language function-in-intension is implemented while not being temporally computed is based on a category mistake. Our mistake had been to assume that a function can only be implemented by running on a machine, as though a functional characterisation must ultimately be an algorithmic characterisation. In reality, the function-in-intension corresponds to a description of the machine not the function that runs on it. While a Marrian computational level theory identifies the function that is computed by some algorithm implemented in the brain, a generative grammar is concerned with the transition function of a computational system. It describes the computational properties of the states of this machine (i.e. syntactic objects). Once we embrace the idea that a grammar is a description of a transition function, not a program function, then the question of when this function runs disappears.

This also shows the error in the idea that merge is a form of concatenation. While this claim is often repeated, it’s in obvious conflict with the claim that merge is ‘the simplest computational operation, embedded in some manner in every relevant computational procedure’ (Chomsky, 2013: 40). There is no reason to think that concatenation is the ‘simplest computational operation’, it took several states to describe it above, and concatenation-plus-something-else (e.g. labelling, projection etc.) isn’t going to be any simpler. Concatenation is not a necessary component of any other computational operation. Unlike Turing’s most general operation, concatenation is optional.

---

23 ‘Merge combines units by concatenating them into a set’ (Jackendoff, 2011: 601). ‘Algebraically, Merge works via the concatenation of two (structured) objects’ (Berwick et. al., 2011: 92). ‘The operation concatenation is an elementary operation shared by Merge and Move’ (Kitahara, 2002: 173). ‘Merge is a species of concatenation and hierarchy in language is the result of combining concatenation with endocentric labelling understood in a Bare Phrase Structure way’ (Hornstein, 2009: 16)
This issue is more than terminological, it reflects a profound difference in how one views a grammar. Concatenation is an operation upon expressions; a matter of combining on a tape. It takes us back to the idea that a grammar is a device for symbol manipulation which we have seen is inappropriate. An account of merge which considers it as a form of concatenation fails to capture either the simplicity of merge - it must be prior to concatenation - or the complexity of lexical items - features are what drive merge. Identifying merge with Turing's general operation accounts for both of these properties.

But what are we really getting at here? It might help to consider this all in the context of a challenge. Say that you want to characterise a cognitive system which can perform infinitely many different actions. You might think of this task in two different ways. You could either think that your job is to characterise a function that recursively enumerates infinitely many outputs. Or you can think that your job is to design a machine which can perform infinitely many functions (i.e. algorithms). In the first case, the actions are identified with runs on the machine or the outputs of those runs on the tape. In the second case, the actions are functions themselves. Taking seriously the idea that a sentence is a mapping from sound to meaning, that is, a function, we'll assume that our job is to characterise a machine rather than a function.

How do you characterise a machine that can compute infinitely many functions? What you'll be looking for is something like a Universal Turing Machine. Obviously, a grammar is not a Universal Turing Machine as it only maps phonological form to semantic content but it does need to be able to run unboundedly many sentences (to be clear, we're taking sentences to be algorithms that run on the grammar-machine, not strings the machine outputs). The key to a Universal Turing Machine is that it has a system for representing arbitrary Turing machines so that it can be fed a description of a machine on a tape, a numeration, and then proceed to implement the algorithm that this machine computes. Symbols on the tape must be understood as commands which are themselves a statement of the states and properties of the machine which is to be simulated. To capture the fact that symbols must be interpreted, a universal machine is stated as a series of quintuples just like the machine above. The exact details depend on the kinds of symbols the machine has access to and what kinds of computational operation the system can perform [does it have a memory stack or multiple tapes?].

Now imagine you want to reverse engineer a system like this. You know that there is a system which can perform unboundedly many functions and that there can be many thousands of input symbols to this system (in our case, these are roughly what we understand by lexical items). You would be aiming for an account of the computational properties of the states denoted by the symbols on the tape. But this can only be given in the context of a theory about the computational system involved (the notion

---

24 In a 2009 discussion Boeckx acknowledges that the idea that concatenation introduces linear order is 'unfortunately built in the notion of concatenation for some, but it's not what I intended, so if you don’t like 'concatenation', use 'combine' while Chomsky responds that “Concatenate’ means order, so it is more complex than Merge” (Boeckx, 2009: 52). While this terminological confusion is not as great as the ‘recursion’ debate it is unhelpful nonetheless. Merge is the simplest possible mode of recursive generation; you can’t get below it. Phrase structure grammar is much more complex, concatenation is more complex, and anything you can dream of is more complex (Chomsky, 2009 :393).

25 There is an ambiguity in the term ‘computational operation’. On the one hand, it can refer to an operation implemented computationally on some Turing machine like our addition function. Alternatively, it can mean an operation which is a component of the Turing machine architecture. Both merge and ‘the most general operation’ should be understood as the latter. Turing’s most general single operation is not an operation for manipulating symbols; it is the operation by which computational states are combined within a machine to produce algorithms. This is how we should think of merge (and correspondingly the complex computational object upon which merge operates).

26 I don't like the term instruction because instructions normally occur within sentences of a language and the symbols we are interested in here and smaller than this. The symbols in which we are interested might be something like α associated with the actions [1:0 Q:]}
of instruction or state only makes sense when you understand the kinds of actions the machine can carry out) and you don’t know this. You don’t even know if the system will ultimately have anything like Turing machine architecture.

What kind of theory might you develop? It can’t just list-off every algorithm that the system is able to run since there are infinitely many. What it must do is break them down into their component states. A theory that simply listed the symbol-state pairings would not be complete as it wouldn't tell us about the operations of the universal machine. The description would have a label for states (the coding that appears on the tape) and a description of the computational properties of the state which that label calls/denotes. However, these would not be like states in the deterministic Turing machine above as they would not occupy a single position in the program. They would be combinable to construct novel programs. They would be unboundedly combinable and the 'mechanism' of their combination would be Turing's most general operation, the operation which enables the change from state to state. The most important thing about computation is that states can combine/merge with one another to form algorithms. If the symbols in a numeration can be combined to form an algorithm for mapping sound to meaning, an operation like merge is required.

‘Each SO [syntactic object] generated enters into further computations. Some information about the SO is relevant to these computations. In the best case, a single designated element should contain all the relevant information: the label (the item `projected'' in X'-theories; the locus in the label-free system of Collins 2002). The label selects and is selected in EM [External Merge], and is the probe that seeks a goal for operations internal to the SO: Agree or IM [Internal Merge].’ (Chomsky (2008, 141))

Let's consider how this is different to a theory of parsing. ‘Syntactic parsing is the task of recognizing a sentence and assigning a syntactic structure to it' (Jurafsky & Martin 2018: 223). Viewed as a machine, a parser takes a string as input 'on the tape' and implements an algorithm for constructing a syntactic representation of it; identifying the parts of speech in the sentence and the hierarchical relations between them. It typically does this by consulting a stored representation of a grammar. For example, the classic CYK parser consults a CNF representation of a grammar and incrementally builds possible parses of the input. The details of different parsers though don't matter for us since what is important is the fact that this relies on a different view of grammar to that outlined above. In contrast, a grammar as we have described it does not implement one algorithm as a parser might (e.g. the CYK algorithm), but unboundedly many. Likewise, a description of a grammar is not a description of performance, i.e. the runs of a system. It is instead an account of the fundamental computational atoms a system implementing the language must have. As such, it is not concerned with the constraints of space or time.

What kind of theory is this?

This is a distinctive kind of cognitive computational theory. I don't think it should be either surprising that generative linguistics doesn't assimilate to more familiar forms of cognitive computational theorising; language is unlike any other cognitive system. In this sense, it is more like a psychological theory as defined by Chomsky as it 'is concerned, at the very least, with human capacities to act and to interpret experience, and with the mental structures that underlie these capacities and their
exercise’ (Chomsky, 1980: 1). The transition function of a machine doesn't describe the machine's performance in space and time but the underlying vehicles of computation.27

A grammar provides a description which is machine-oriented without being mechanical [causal] since a causal-mechanical theory would require information about implementation.28 Chomsky takes these to be performance considerations, independent of the description of the underlying states of machine: ‘The concept of an automaton is OK so long as we understand it to be organized in the manner of a control system for a much bigger, unbounded system’ (Chomsky, 1972: 41).29

I should clarify that I don't think the fact that Turing called this 'the most general single operation' is in any way the reason that Chomsky refers to merge as the 'simplest computational operation', not least because generality and simplicity are not the same thing. I'm not even claiming that these connections were intentional - reason can be cunning. I do, however, believe that they are both discussing the same operation because it is the operation which underlies all computation. Second, this does not mean that generative grammars describe Turing machines. All automata require the general operation. The question of what kind of automaton a grammar corresponds to remains open.

Conclusion

There are several advantages of this characterisation of grammar; it address why a grammar (viewed as a function-in-intension) doesn't 'run' on the mind in a temporal sense, it makes sense of the changes in generative linguistics which grounds linguistic properties in the features syntactic objects, and it makes sense of the claim that merge is the simplest computational operation (and connects this claim to Turing's work) in a way which is not possible if we view merge as a kind of concatenation.

---

27 ‘You know you have the capacity because if you add time and space - like a Turing machine - then you get the right answer’ (Chomsky, 2009: 391-392).

28 ‘The preceding account is not dispositional (it says little about what a person is disposed to say under particular circumstances), and it is not ‘causal' (neuropsychological). Furthermore, the account is not ‘functionalist'; it does not ‘regard psychology as given by a set of causal connections, analogous to the causal operations of a machine. But the account of ‘competence' is descriptive: It deals with the configuration and structure of the mind/brain and takes one element of it, the component L, to be an instantiation of a certain general system that is one part of the human biological endowment. We could regard this instantiation as a particular program (machine), although guarding against implications that it determines behaviour… Our assumption is that the person before us has a language with particular rules and principles along with other systems which interact with it as a matter of mental/physiological fact, and which we might think of as a particular machine program with a particular data structure, and so forth’ (Chomsky, 1986: 238-239).

29 The idea that the 'control unit' of a Turing machine (given by the transition function) corresponds to the rules of grammar is seldom made explicitly in generative work. When criticising connectionist approaches to nested dependencies and the need to rewrite models that could acquire 2 levels of nesting in order to capture 3 levels: ‘In Turing machine terms, the control unit has to be totally changed, which means you’re not capturing the rules' (Chomsky, 2009:392).