Memory Requirements and Local Ambiguities of Parsing Strategies

Steven P. Abney
Bell Communications Research

Mark Johnson
Brown University

Introduction

Memory requirements and local ambiguities are two considerations that have played an important role in shaping the parsers adopted in the psycholinguistic and computational linguistic literatures. To the best of our knowledge, however, memory requirements and local ambiguities of psycholinguistically interesting classes of parsing strategies have never been explicitly defined, much less rigorously explored. This paper is an initial attempt to remedy the situation.

It is widely assumed that the human parser operates under the strictest constraints on space resources. For example, the classical explanation for the uninterpretability of center-embedded examples is stack overflow: parsing a sentence like *the rat the cat the dog chased bit ate the cheese* is difficult because it requires holding on to too many incomplete substructures (Chomsky & Miller 1963). Since that example involves a maximum of only three or four incomplete constituents, the stack-overflow explanation entails limits on space resources that are Draconian indeed. Even if we assume that additional factors are involved in center embedding, there are well-established limits on short-term memory that are often taken to imply corresponding limits on the space available to the parser. It seems safe to assume that a very small constant number of short-term memory units are available, where one memory unit is sufficient space for one parse-tree node.

A property of parsers that interacts with their space requirements is the number of *local ambiguities* they face for a given grammar. If we change our parsing strategy so as to reduce the amount of space required, we often increase the number of local ambiguities the parser faces. An increase in local ambiguities generally has undesirable consequences. Depending on how we deal with them, additional local ambiguities may increase error rate, or they may increase overall time and space usage, because of the overhead required for backtracking or pursuing alternative parses in parallel.

People often speak of local ambiguities as if they were intrinsic to the grammar of the language being parsed. For example, the Marcus parser (Marcus 1980) is said to exploit a "wait-and-see" strategy, in that when it encounters a local ambiguity, it delays making a decision in hopes of finding disambiguating information. However, local ambiguities are actually determined by the parser, not the grammar. They consist in the points at which a given parser cannot choose with certainty among multiple legal next actions leading to different parses. The local ambiguities the Marcus parser avoids are only local ambiguities relative to some other parsing algorithm which we take as an implicit standard. It would be equally fair to say that the other parsing algorithm pursues an "overly eager" strategy, and perceives local ambiguities where there are none in fact (taking the Marcus parser as standard).

These two properties of parsing strategies – space requirements and local ambiguities – have played an important role in shaping the parsing algorithms adopted in the literature. For example, Frazier (1978) emphasizes the necessity of minimizing space requirements, hence she adopts a basically top-down parsing strategy in which the parser never has to keep track of more than one parse-tree fragment. On the other hand, Marcus (1980) emphasizes the need to avoid local ambiguities, and therefore adopts a parsing strategy that is closer to bottom-up, and imposes less stringent constraints on space usage.

Nonetheless, these properties have not received careful examination, and assumptions sometimes made about them strike us as unrealistic. For example, Frazier assumes that a node only takes up space in short-term memory until it has been attached to some parent node – hence the parser attaches a node immediately, in the first position that suggests itself, in order to keep the load on short-term memory at a minimum. In other words, the parser's (short-term) space requirements are determined by the number of nodes without a parent. We consider it more realistic to include all nodes that the parser may need to refer to later in the parse, i.e., all nodes whose parent has not been identified, *and* all nodes that are missing one or more children. Of course, if we permit parse-tree nodes to be shuffled between short-term and long-term memory, short-term memory load cannot be equated with the parser's (undifferentiated) space requirements. We will concern ourselves only with undifferentiated space requirements.

We adopt the postulate, common to much (though by no means all) psycholinguistic research, that the human parser constructs a syntactic parse tree of the utterances it processes, and that it does so in an incremental, node-by-node and arc-by-arc fashion. We do not require that the parse tree exist in its entirety at any point – in fact, we assume that individual nodes are "garbage-collected" as soon as they are no longer needed, generally well before the end of the sentence has been reached. We also do not wish to make strong claims about how the human parser is implemented in the brain. We only require that speaking *as if* the human parser builds parse trees permits us to characterize accurately its space requirements and local ambiguities. Even this qualified assumption could be incorrect: as Stabler (1990) points out, semantic interpretation does not necessarily require the construction of a syntactic tree. Nevertheless many psycholinguistic models do assume that such trees are built "on-line", so it is interesting to study the general computational properties of algorithms for parse tree construction.

In this paper, we explore the range of possible parsers, their memory requirements, and the number of local ambiguities they face. We characterize parsers according to the *parsing strategies* they implement, where by *parsing strategy* we mean a way of enumerating the nodes and arcs of parse trees. For example, a top-down parser implements a top-down parsing strategy, in which each node is enumerated before any of its descendants. In a bottom-up parsing strategy, each node is enumerated after all its descendants. Many other parsing strategies are possible; indeed, *any* way of ordering the nodes of parse trees, even if it is not finitely specifiable, is a parsing strategy.

We see this paper as laying the groundwork for an attack on the question, “For any given grammar, what parsing strategy has the optimal combination of space requirements and local ambiguities, and how can we implement that parsing strategy?” We do not attempt to answer that question here. But we do provide tools for evaluating the space requirements and local ambiguities of a given parsing strategy for a given grammar.

We also show that whatever the optimal parsing strategy for English, it is probably neither top-down nor bottom-up. Namely, if we ascribe the unacceptability of center-embedded structures to space limitations, then our parsing strategy must have the property that center-embedded structures require more space than either uniformly left- or right-embedded structures. This is not a property of either top-down or bottom-up parsing strategies.

Parsing Strategies

A *parsing strategy* is a way of enumerating the nodes and arcs of parse trees. Precisely, it is a function from parse-trees to enumerations of their nodes and arcs.

We will only be interested in certain classes of parsing strategies. In particular, we assume that nodes are not attached to one another until both of them exist, i.e., that an arc is enumerated after the two nodes it connects. Further, we assume that terminal nodes are enumerated exactly when the words they correspond to are read from the input stream. That is, if terminal node n_1 precedes terminal node n_2 (if n_1 is read from the input before n_2), then n_1 is enumerated before n_2.

We do not require preceding non-terminal nodes to be enumerated before succeeding non-terminals, but if they are, we say that the parsing strategy is *left-to-right*. If a strategy is left-to-right, then in a structure like the following, no node under Z (including Z) is enumerated until all the nodes under Y (including Y) have been enumerated.

Two well-known classes of parsing strategies are *top-down* and *bottom-up* strategies. In top-down parsing strategies, a node is enumerated before any of its descendants, and in bottom-up strategies, a node is enumerated after all its descendants. Top-down strategies vary with respect to the order in which nodes unrelated by dominance are enumerated, and with respect to the order in which arcs are enumerated. Left-to-right top-down strategies vary only with respect to the order in which arcs are enumerated. That is, the combination of left-to-right and top-down constraints is sufficient to determine a unique ordering on the nodes of a parse-tree. (Likewise for left-to-right bottom-up strategies.)

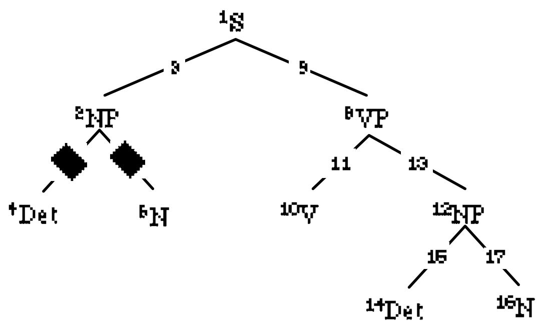
To illustrate, consider the grammar G_1:

S $\rightarrow$ NP VP

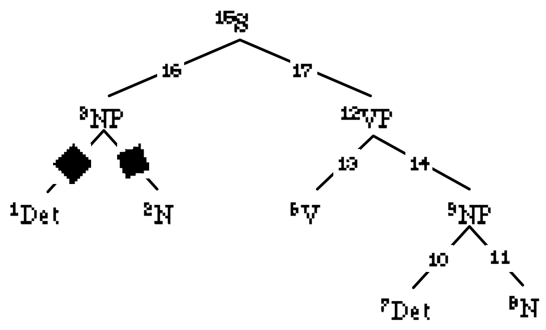
NP $\rightarrow$ Det N

VP $\rightarrow$ V NP

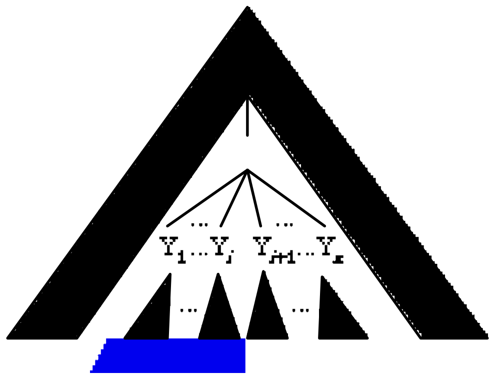
L(G_1) is the single sentence s = *Det N V Det N.* A left-to-right top-down node-enumeration strategy enumerates the parse-tree of s as follows (adopting a fairly standard ordering for the arcs)[1]:



The left-to-right bottom-up strategy enumerates the parse tree of s as follows:



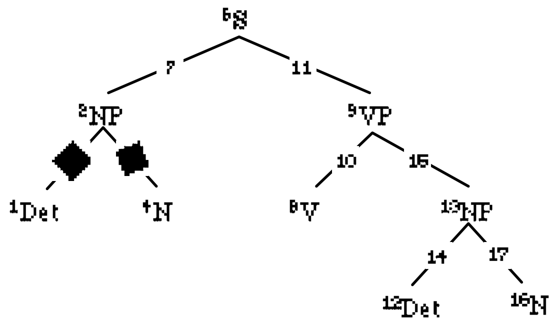
These two strategies delimit the end-points in a family of strategies we call *syntax-directed* parsing strategies, adapting a term from Nijholt (1980). A syntax-directed parsing strategy can be characterized by placing an "announce point" in every grammar rule, such that the children that precede the announce point (and all their descendants) are enumerated first, then the parent node is enumerated, and then the children after the announce point are enumerated (and all their descendants). More precisely, consider an arbitrary node with label X and with children labelled $Y_1 \ldots Y_{n_r}$, corresponding to the grammar rule $r = X \rightarrow Y_1 \ldots Y_{n_r}$:



In a syntax-directed node-enumeration strategy, there is a unique *announce point* i_r in each rule r, such that node X is enumerated after children $Y_1 \ldots Y_{i_r}$ and all their descendants, but before $Y_{i_r+1} \ldots Y_{n_r}$ and all their descendants. In the preceding diagram, the nodes in the shaded region are enumerated first, then X, and then the descendants of X in the unshaded region. Note that a corollary of this definition is the following. For each node y, at the point that y's parent is enumerated, either *all* of the subtree rooted at y (including y itself) has been enumerated (this is the case for $Y_1 \ldots Y_{i_r}$), or *none* of the subtree rooted at n has been enumerated (this is the case for $Y_{i_r+1} \ldots Y_{n_r}$).

Every syntax-directed strategy is left-to-right. Conversely, the left-to-right top-down and bottom-up strategies are both syntax-directed. In the top-down strategy, $i_r = 0$ for all rules r: a node is enumerated before any of its children or their descendants. In the bottom-up strategy, $i_r = n_r$ for all rules r: a node is enumerated after all its children, and all their descendants. An important, but less well-known strategy is the *left corner* strategy, in which a node is built immediately after its first child and that child's descendants, but before the remaining children or any of their descendants, as illustrated in (1):

(1)

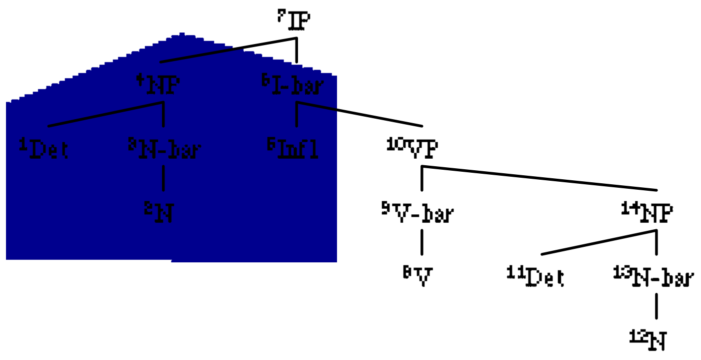


In the left corner strategy, $i_r = \min(1, n_r)$ for all rules r.

Top-down, bottom-up, and left-corner parsing strategies also happen to be examples of *uniform* syntax-directed strategies. A uniform syntax-directed strategy is one in which the announce point is the same in every rule. More precisely, a uniform syntax-directed strategy is one in which there is some j, fixed for the strategy, such that either $i_r = \min(j, n_r)$ for all rules r (all announce points are at the same point, counting from left to right), or else $i_r = \max(j, 0)$ for all rules r (all announce points are at the same point, counting from right to left).

Syntax-directed parsing strategies correspond systematically to the *generalized left corner* (GLC) parsers (Demers 1977), a family that includes some of the most important parsers in the computer science literature. Syntax-directed strategies are especially well suited for implementation by pushdown automata, using a construction modelled closely after that of Demers.

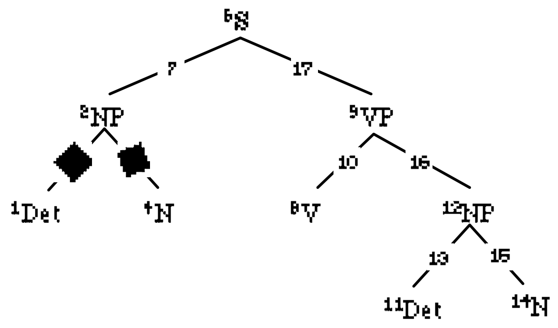
By contrast, strategies of linguistic interest are sometimes not syntax-directed. For example, a strategy that is popular in Government-Binding circles is head-driven parsing, a left-to-right strategy in which each non-terminal node has a head, and is enumerated immediately after its head. The following is a typical enumerated parse tree (ignoring arcs for simplicity's sake):



The shaded region marks the nodes that have been enumerated at the point just before the parent of I-bar is enumerated. Note that some of the subtree rooted at I-bar has been enumerated, but not all of it. Therefore, head-driven parsing is not syntax-directed.

Finally, there are two types of arc enumeration we will be concerned with. A parsing strategy is *arc-eager* when an arc is enumerated at the earliest point that the two nodes it connects have been enumerated. If two arcs become candidates for enumeration simultaneously, they are enumerated from left to right and bottom to top. A parsing strategy is *arc-standard* when an arc is enumerated at the earliest point such that (1) both nodes it connects have been enumerated, and (2) either none or all of the subtree under the child node has been enumerated. (Again, arcs that are not uniquely ordered by these constraints are enumerated left to right and bottom to top.) In slightly different words: under an arc-eager strategy, a node *n* is attached as soon as possible, i.e., as soon as both it and its parent have been created. Under an arc-standard strategy, *n* is attached as soon as possible, with the exception that, if any of *n*'s descendants already exist when *n* is created, *n*'s attachment is postponed until all of *n*'s descendants have been created and attached.

With a top-down or bottom-up node enumeration, eager and standard arc enumerations are equivalent. However, they diverge when the node enumeration is e.g. left corner. The following illustrates an arc-standard left corner strategy ((1) above illustrates the corresponding arc-eager enumeration):

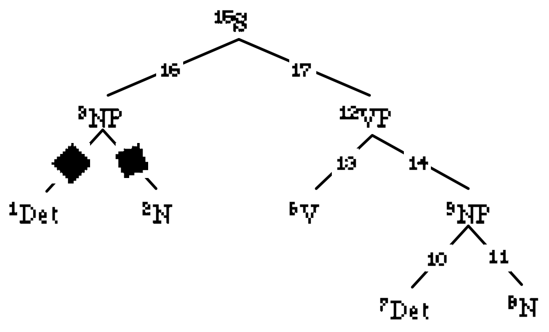


As we shall see in the next section, an arc-eager strategy may require less space than the corresponding arc-standard strategy, but never more. Since we are interested in minimal space requirements, we have used eager arc enumeration by default throughout the paper; standard arc enumerations appear only where explicitly noted.

Space Requirements

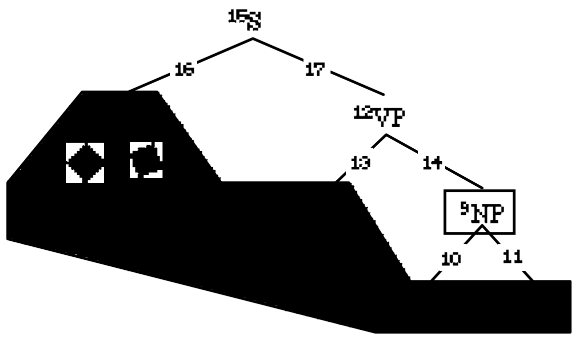
Now let us consider the space requirements of parsing strategies. The minimum space we need is one unit for each node that we may yet need to refer to. The nodes that we may yet need to refer to are those that are *incomplete* in the sense that either their parent, or some child, has not yet been constructed. Given an enumeration x_0 ... x_n of the nodes of a parse-tree, it is straightforward to calculate memory requirements. For each point i in the enumeration, $0 \leq i \leq n$, the incomplete nodes are those nodes y such that y has already been built (i.e., $y = x_j$, $j \leq i$), and either the parent or one of the children of y has not yet been built (i.e., for some $k > i$, x_k is either the parent or a child of y). The memory usage of the parse-tree enumeration x_0 ... x_n is the maximum number of incomplete nodes at any point i. The space required by a parsing strategy S, given a grammar G, is the maximum space required by any enumeration that S assigns to a parse-tree of G. This constitutes a lower bound on the space used by any parser that implements the strategy S. A parser that implements S cannot use less space than S requires, though it may use more.

To make the discussion more concrete, let us consider again the left-to-right, bottom-up, arc-eager enumeration of the parse-tree of G_1 (repeated below). There are 17 nodes and arcs. The memory requirements $f(i)$ at each point i in the enumeration are as shown.

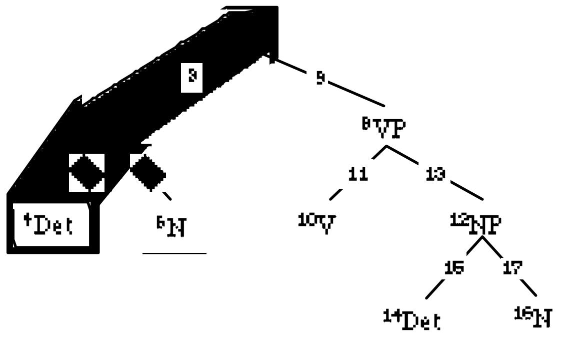


i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$f(i)$	1	2	3	2	1	2	3	4	5	4	3	4	3	2	3	2	0

The maximum comes at point 9, where there are five incomplete nodes, as illustrated in the following snapshot. The nodes and arcs that have been enumerated at point 9 are those in the outlined region; both NPs, V, Det, and N are all incomplete.



Using a top-down enumeration, by contrast, the maximum space required is three, at points 4 (dotted lines) and 6 (solid lines):

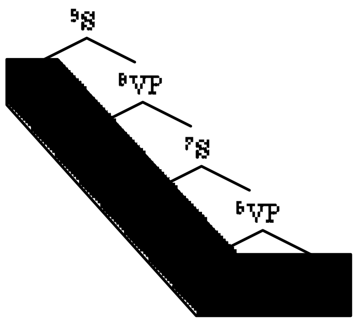


i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$f(i)$	1	2	2	3	2	3	1	2	1	2	1	2	1	2	1	2	0

In heavily right-branching languages, like English, a top-down strategy is far more space-efficient than a bottom-up strategy. Consider grammar G_2 =

$$\begin{aligned} S &\rightarrow NP\ VP \\ VP &\rightarrow V\ NP \\ &\ \ |\ \ V\ S \end{aligned}$$

and a sentence *NP V NP ... V NP*. Under the bottom-up strategy, no nonterminal nodes are constructed until the entire sentence has been read. For example, the following is a snapshot of the parse of *NP V NP V NP*, just after reading the final NP (ignoring arcs). All enumerated nodes are incomplete.



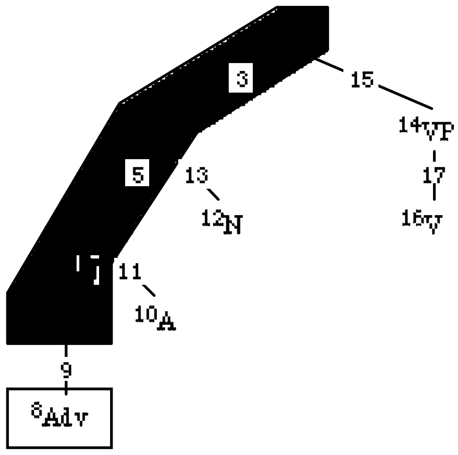
Hence space requirements are unbounded, inasmuch as no bound is placed on the length of the sentence. Under a top-down strategy, on the other hand, there are never more than two incomplete nodes at any point in the parse. For grammar G_2, the difference in space requirements for top-down and bottom-up strategies literally could not be greater.

We should emphasize here the importance of the difference between bounded and unbounded space. If the space requirements of a strategy are bounded, then the strategy is potentially implementable by a finite machine. If a strategy's space requirements are unbounded, then no finite machine can truly implement it. No matter how much space a machine has at its disposal, it will fail to parse some grammatical inputs (in fact, infinitely many), simply because it runs out of working memory.

Turning now to left-branching structures, we observe that bottom-up parsing is more space-efficient. Suppose we add the following rules to our grammar:

$$\begin{aligned} &\text{NP} \rightarrow \text{AP N} \\ &\text{AP} \rightarrow \text{AdvP A} \\ &\text{AdvP} \rightarrow \text{Adv} \end{aligned}$$

Parsing the sentence *Adv A N V*, the top-down strategy requires five units of memory. Just after reading *Adv*, we have the following structure, where the outlined nodes and arcs are those that have been enumerated. All outlined nodes are incomplete.



A bottom-up strategy requires only three memory units, which is the minimum for a bottom-up strategy on any non-trivial tree.

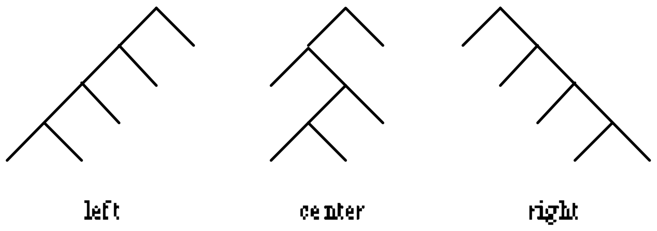
If the depth of left embedding is unbounded - e.g., if we add the rule NP → NP ’s N to our grammar – then a top-down strategy requires unbounded space. On sentences of form *N ’s N ... ’s N V*, the entire left branch from the root to the first N is enumerated before the N is enumerated. Since that branch consists of incomplete nodes, and there is no bound on its size, memory requirements are unbounded. (Not to mention the compounding problem that the structure is also locally infinitely ambiguous.)

Before going on to a discussion of center embedding, let us add a brief note about the effect of choice of arc-enumeration strategy on space requirements. Under an arc-eager strategy, a given arc may be enumerated earlier than under the corresponding arc-standard strategy, but never later. Enumerating an arc earlier may cause a node to become complete, thus decreasing space requirements, but it can never add to the number of incomplete nodes. Hence arc-eager strategies sometimes require less space than arc-standard strategies, but never more.

Center Embedding

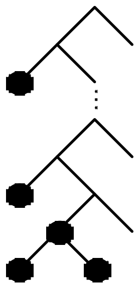
If we wish to explain the unparsability of center embedded constructions by appealing to memory limitations, then a property we desire of our parsing strategy is that its memory requirements be maximal on center-embedded constructions, not on left- or right-branching constructions. Given the popularity of both top-down and bottom-up strategies, then, it is perhaps surprising that neither of them have the desired property.

For the sake of simplicity, we consider only grammars that generate binary branching trees. A uniform left-branching structure involves recursive expansion of the left branch, right-branching involves expansion of the right branch, and uniform center-embedding involves alternating expansion of left and right branches:



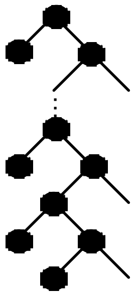
For binary-branching trees, there are three possible uniform syntax-directed, arc-eager parsing strategies: top-down, bottom-up, and left-corner.

The bottom-up strategy requires three units of space on uniformly left-branching trees, regardless of the size of the tree. On uniformly right-branching trees, the bottom-up strategy requires n+1 units of space, for n the number of terminal nodes. On uniformly center-embedded trees, the bottom-up strategy has maximum memory requirements just after building the first non-terminal node (incomplete nodes circled):



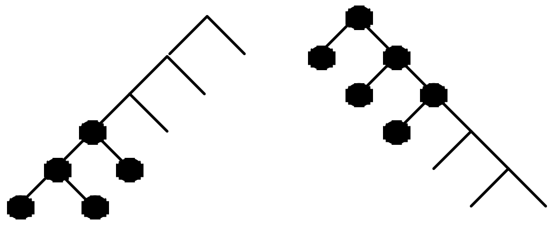
At that point, we require *m*+2 memory units, for *m* the number of terminal nodes on a left branch. If the topmost expansion is a left-branch expansion (as here), $m = \lfloor n/2 \rfloor$, else $m = \lceil n/2 \rceil$, for *n* the total number of terminal nodes. In short, the bottom-up strategy requires unbounded space on both center-embedding and right-branching structures. For sentences of a fixed length, the bottom-up strategy requires twice as much space for right-branching constructions as for center-embedded constructions. Hence the bottom-up strategy predicts (incorrectly) that right-branching structures should be at least as demanding as center-embedded constructions, with respect to space requirements, and even more demanding, if we take constant factors into account.

With a top-down strategy, uniformly left-branching trees require *n* units of space, for *n* the number of terminal nodes. Right-branching trees require constant space of 2 units. Center-embedded trees require maximum space at the point immediately after the last terminal on a left branch has been enumerated (dark circles are completed nodes, open circles are enumerated but incomplete nodes):

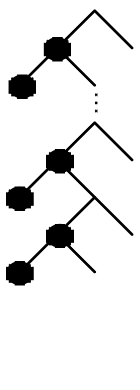


The top-down strategy requires m+1 units of space, for m the number of terminals on a right branch. If the topmost expansion is on a right branch (as here), $m = \lfloor n/2 \rfloor$, else $m = \lceil n/2 \rceil$, for n the total number of terminal nodes. The top-down strategy requires unbounded space on both left- and center-embedded constructions; and for sentences of fixed length, it requires twice as much space for left-embedded constructions as for center-embedded constructions. Like the bottom-up strategy, the top-down strategy fails to predict that center-embedding imposes the greatest load on memory.

The final alternative is the left-corner strategy. On left-branching structures, the left-corner strategy requires a constant 2 units of space. On right-branching structures, it requires a constant 3 units of space.[2] The following snapshots illustrate. (In the right-branching construction, the center incomplete node has just been enumerated, and neither of the arcs connecting it to its parent and child have been enumerated.)



In a center-embedded construction, the left-corner strategy requires m+1 units of space, for m the number of terminals on a left branch:



If the topmost recursion is through the left branch, $m = \lfloor n/2 \rfloor$, else $m = \lceil n/2 \rceil$, for n the total number of terminal nodes. In either case, unbounded space is required for center-embedded constructions. Unlike top-down and bottom-up strategies, the left-corner strategy *does* correctly predict that center-embedded constructions require more space than either left- or right-branching constructions.

Local Ambiguities

An important additional factor that impinges on our choice of parsing strategy is the presence of local ambiguities. We have been speaking to now as if we knew in advance what the parse-tree is, but of course, in general, more than one parse-tree will be consistent with the input seen so far at any point in the parse. Local ambiguities require that we either guess and accept a certain error rate, or simulate a non-deterministic computation, with concomitant increase in space resources for keeping track of multiple possible parser states. In either case, the cost of local ambiguities can be very high, and they are generally to be avoided.

We can determine where local ambiguities arise, given a particular parsing strategy. The nodes and arcs that must be constructed after reading word w_i and before reading w_{i+1} are those nodes and arcs that are enumerated between the terminal nodes corresponding to w_i and w_{i+1}. We call the set of such nodes and arcs the *ith parse increment*. Local ambiguities arise where the parser is unsure which set of nodes and arcs to build: that is, at those points in the input where there is more than one parse increment consistent with the input seen so far and the next k words of lookahead.

For example, consider the grammar G_3 =

$S \rightarrow$ Adv NP VP

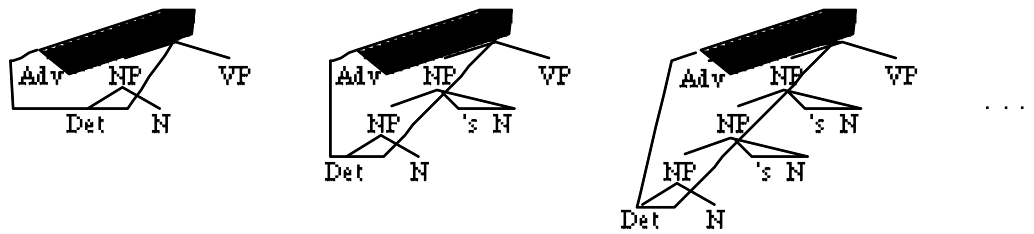
NP $\rightarrow$ NP 's N

NP → Det N

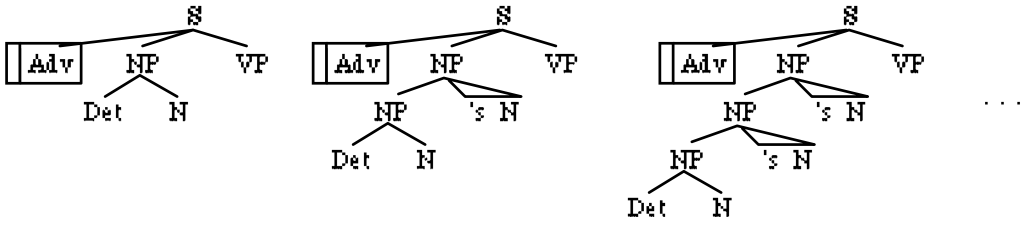
with a top-down parsing strategy, and $k = 1$. At the beginning of a sentence beginning "suddenly the man ...," lookahead tells us that the first category is Adv. We consider all parse trees generated by G_3 whose frontier begins with Adv; they are indicated in (2) below. The first parse increment of any given parse tree is the set of nodes and arcs enumerated before the first word; the first parse increments are outlined in solid lines. The first parse increments are all identical, so there is no local ambiguity at position 0 in the input.

After reading the first word from the input, we know that the first two categories in the frontier of the correct parse tree are *Adv Det* (one word read, one word of lookahead). We consider all parse trees of G_3 consistent with this input; this is the same set of parse trees as was consistent with input *Adv*, illustrated in (2). The second parse increment of any one parse tree consists of the nodes and arcs enumerated before the second word, but not before the first word. The second parse increments are outlined in dotted lines in (2). There are differences among the second parse increments, hence there is a local ambiguity just before the second word. Intuitively, using a top-down node enumeration and one word of lookahead, the parser does not know how many NP's to build.

(2)



With a bottom-up strategy, by contrast, there is no local ambiguity before either the first or second word. The first parse increment is the empty set (no nodes or arcs are enumerated before Adv), and the second parse increment consists of the sole node Adv:



Also with a left-corner strategy, there is no local ambiguity at either point. The first parse increment is the empty set, and the second parse increment consists of Adv, S, and the arc connecting them.

A strategy is more *eager* (or less *circumspect*) with respect to node enumeration if it builds non-terminal nodes at earlier points in the input string. Thus top-down strategies are more eager than left corner strategies, which are in turn more eager than bottom-up strategies. Choosing a less eager strategy sometimes reduces the number of local ambiguities the parser faces, as the previous example demonstrates. However, we saw earlier that choosing a strategy that is too circumspect – for example, moving from a left corner strategy to a bottom-up strategy – can increase space requirements. For grammar G_3, bottom-up and left-corner strategies appear to be equally well suited, requiring only constant space, and presenting no local ambiguities. If we added a right-recursive rule, such as VP $\rightarrow$ V S, the bottom-up strategy would require unbounded space, so the left-corner strategy would appear to be represent the optimal combination of space requirements and local ambiguities. Whether the left-corner strategy is an optimal strategy for all grammars, we must leave as a question for future research.

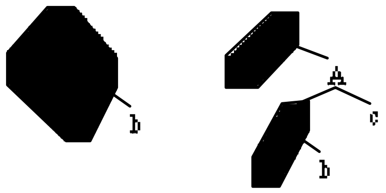
Differences in arc-enumeration order can also affect local ambiguities, even if we hold the node-enumeration order constant. Consider the grammar

$S \rightarrow a\ A \mid a\ B$

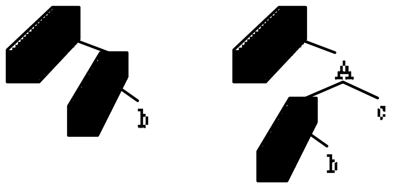
$A \rightarrow B\ c$

$B \rightarrow b\ b$

Let the parsing strategy be left-corner, arc-eager, and $k = 1$. Then after seeing input *ab*, with lookahead *b*, there are two possible parse-trees. The outlined subtrees represent the union of the first, second, and third parse increments:



The alternative parse-trees agree about which nodes should be constructed, but disagree as to whether B is a child of S or not. On the other hand, if the strategy is left-corner, arc-standard, then there is no local ambiguity. The arc between S and B is not enumerated until after the subtree under B is complete, so the parse increments up to the second *b* are identical:



We saw earlier that an arc-eager strategy may require less space than an arc-standard strategy, but never more. Here we see that the cost of diminished space requirements is sometimes an increase in local ambiguities.

Conclusion

In motivating hypotheses about the human parser, researchers have sometimes appealed to the need to minimize space requirements and local ambiguities. We feel that such arguments have been unpersuasive hitherto, for want of suitable means for evaluating a parser's space requirements and local ambiguities. We have shown how to evaluate the minimum space requirements of, and local ambiguities faced by, any parser that can be characterized as constructing phrase-structure trees. We have used these measures to obtain a surprising result. Namely, if human difficulty in processing center-embedded constructions is to be attributed to memory limitations, then the human parser must employ a strategy that has greater space requirements for center-embedded constructions than for left- or right-branching constructions. Neither top-down nor bottom-up strategies have that property. We hope that these techniques will prove generally useful in evaluating models of the human parser.

References

Chomsky, N., & G. Miller (1963) “Finitary Models of Language Users,” chapter 13 of R. Luce et al., eds., *Handbook of Mathematical Psychology* (vol II), John Wiley and Sons, New York.

Demers, A. (1977) “Generalized Left Corner Parsing,” *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages,* 1977 ACM SIGACT/SIGPLAN, pp. 170-182.

Frazier, L. (1978) *On Comprehending Sentences: Syntactic Parsing Strategies*, unpublished PhD dissertation, University of Connecticut.

Marcus, M. (1980) *A Theory of Syntactic Recognition for Natural Language*, The MIT Press, Cambridge, Massachusetts.

Nijholt, A. (1980) *Context-Free Grammars: Covers, Normal Forms, and Parsing*, Springer-Verlag, New York.

Stabler, E. (1990) “Avoid the pedestrian’s paradox,” in C. Tenny, ed., *The MIT Parsing Volume, 1988-1989*, Center for Cognitive Science, MIT, Cambridge, Massachusetts, pp. 83-100.

Footnotes

[1] In terminology we introduce below, the illustrated strategy can be characterized as either arc-eager or arc-standard; they are equivalent when node enumeration is top-down (or bottom-up) and left-to-right.

[2] Crucially, we assume eager arc enumeration. With standard arc enumeration, the left-to-right left-corner strategy requires unbounded space on right-branching constructions.

---

[1]In terminology we introduce below, the illustrated strategy can be characterized as either *arc-eager* or *arc-standard*; they are equivalent when node enumeration is top-down (or bottom-up) and left-to-right.

[2]Crucially, we assume eager arc enumeration. With standard arc enumeration, the left-to-right left-corner strategy requires unbounded space on right-branching constructions.