adopt a syntactic analysis. However, the ultimate goal of parsing is, of course, to determine the appropriate interpretation that should be assigned to a string of words and to integrate that interpretation with discourse context and general knowledge. Thus, researchers have asked how people use context to decide on the appropriate interpretation for expressions – for example how people interpret elliptical phrases, how they should interpret a pronoun or other referring expression, and so on. It is clear that theories of parsing need to be fully integrated into more general accounts of language comprehension. This is likely to be a major focus of future research.

Further Reading

Altmann GTM (1998) Ambiguity in sentence processing. *Trends in Cognitive Sciences* **2**: 146–152.

Crocker MW (1999) In: Garrod S and Pickering M (eds) *Language Processing*. Brighton and Cambridge, MA: Psychology Press/MIT Press. [Covers basic computational issues.]

Crocker MW, Pickering MJ and Clifton C Jr (eds) *Architectures and Mechanisms for Language Processing*. Cambridge, UK: Cambridge University Press.

Frazier L and Rayner K (1982) Making and correcting errors during sentence comprehension: eye movements in the analysis of structurally ambiguous sentences. *Cognitive Psychology* **14**: 178–210.

Gibson E and Pearlmutter NJ (1998) Constraints on sentence processing. *Trends in Cognitive Sciences* **2**: 262–268.

Harley TA (2001) *The Psychology of Language*, 2nd edn, chap 9. Hove, UK: Psychology Press.

Haberlandt K (1994) Methods in reading research. In: Gernsbacher MA (ed.) *Handbook of Psycholinguistics*. San Diego, CA: Academic Press.

Mitchell DC (1994) Sentence parsing. In: Gernsbacher MA (ed.) *Handbook of Psycholinguistics*. San Diego, CA: Academic Press.

Pickering MJ (1999) Sentence comprehension. In: Garrod S and Pickering MJ (eds) *Language Processing*. Brighton and Cambridge, MA: Psychology Press/MIT Press.

Tanenhaus MK and Trueswell JC (1995) Sentence comprehension. In: Miller J and Eimas P (eds) *Speech, Language, and Communication*, vol. 11, pp. 217–262. San Diego, CA: Academic Press.

Trueswell JC, Tanenhaus MK and Garnsey S (1994) Semantic influences on parsing: use of thematic role information in syntactic disambiguation. *Journal of Memory and Language* **33**: 283–318.

Parsing: Overview

Introductory article

Florian Wolf, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA
Edward Gibson, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA

CONTENTS

Parsing is the task of determining how the words in a sentence combine to yield a structured representation of the sentence meaning, given a grammar.

INTRODUCTION

To understand a sentence, one has to determine how the words in the sentence are combined to arrive at a meaning for the sentence. Consider the following examples:

a. The dog bites the man.
b. The man bites the dog. (1)

Sentences (1a) and (1b) have the same words. However, the words are combined differently, resulting in two very different sentence meanings. The set of rules governing how words are combined for a given language is called a *grammar* of that language. Sentences (1a) and (1b) are acceptable English sentences, whereas the asterisk preceding (2) indicates that this sentence is unacceptable.

Sentences (1a) and (1b) follow the rules of English, whereas sentence (2) does not.

*The dog man bites the. (2)

The task of *parsing* is to determine how the words in a sentence combine to yield a structured representation of the sentence meaning, given a grammar. In contrast to a *parser*, a *recognizer* merely determines whether a sentence is grammatically correct or not, without producing a structured representation for the input sentence. A further contrast holds between *parsers* and *generators*. Whereas a parser takes a sentence as input and provides a representation of the meaning of the sentence as output, a *generator* takes some representation of meaning as input and provides a sentence as output.

This article discusses the process of parsing. The process of parsing is not just retrieving a representation of the sentence's meaning that is already stored in memory. If that were so, we would not be able to understand sentences that we have not heard before. Furthermore, there are an infinite number of possible sentences. Storing a representation of the meaning of each sentence in memory would require infinite memory resources.

This article will give a basic introduction to some parsing strategies that were developed in computational linguistics. It will also describe a method that deals efficiently with *local* (or *temporary*) and *global* ambiguity. Human languages are highly ambiguous. For example, sentence (3) contains both local and global ambiguity:

The man saw the woman on the hill with the telescope. (3)

The word 'saw' is locally ambiguous. That is, without disambiguating context, it could be either a tool or the past tense form of the verb 'see'. Sentence (3) is also globally ambiguous. It has five different readings, because the *prepositional phrases* (*PP*s) in (3), 'on the hill' and 'with the telescope', can modify either the *noun phrase* (*NP*) 'the woman' or the *verb phrase* (*VP*) 'saw the woman'. 'With the telescope' can also modify the *NP* 'the hill'. One of these readings would be equivalent to 'The man used the telescope in order to see the woman who was on the hill'. Another possible reading would be equivalent to 'The man saw the woman who was on the hill and who had a telescope'.

The number of readings of such ambiguities grows exponentially with the number of modifying phrases, PPs in this case. Such ambiguities are very frequent in human languages, but they usually do not present a problem to humans. This is in large part because humans use their knowledge of the world in order to rule out less likely possibilities as they are processing sentences word by word. For instance, humans usually do not get a reading of (3) in which 'with the telescope' modifies 'the hill', because without further context or assumptions we assume that it is unlikely that hills have telescopes.

PARSING STRATEGIES

The task in parsing is to discover how the words of a sentence can combine, using the rules in the grammar. A very simple grammar is presented in Figure 1 (where *S* = sentence; *NP* = noun phrase; *VP* = verb phrase; *Det* = determiner). The grammar indicates that a sentence (S) expands to a noun phrase (NP) and a verb phrase (VP), and that an NP expands to a determiner (Det) and a Noun, etc. We will refer to the symbol to the left of the arrow as the *left-hand side* (*LHS*) of a rule, and the right side of the arrow as the *right-hand side* (*RHS*) of a rule. The first symbol of the RHS is called the *left corner* of an RHS. The categories on the LHS of rules are sometimes called *nodes* in the grammar. Nodes that expand directly to a word (e.g. Det, Noun and Verb) are called *pre-terminals*. Nodes that do not expand directly to a word (e.g. S, NP and VP) are called *non-terminals*.

Using a grammar like this (much more complex in real applications), a parsing algorithm then establishes a syntactic structure for an input sentence. One possible parsing strategy starts by looking at the rules and seeing what input one can find that is compatible with the rules. Such a strategy is called *top-down*. Alternatively, one might start by looking at the input, and seeing which rules in the grammar apply to that input. This is a *bottom-up* strategy. Still another possibility would be some combination of top-down and bottom-up. The following sections describe these different parsing strategies in more detail.

The parsing algorithms described below process a sentence one word at a time, from left to right,

S $\rightarrow$ *NP VP*	*Det* $\rightarrow$ *the*
NP $\rightarrow$ *Det Noun*	*Noun* $\rightarrow$ *man*
VP $\rightarrow$ *Verb NP*	*Noun* $\rightarrow$ *woman*
	Verb $\rightarrow$ *likes*
	Verb $\rightarrow$ *meets*

Figure 1. A very simple grammar.

similar to when people read or listen to language. A *stack* data structure (last in, first out) is used by the parser to keep track of the categories that the parser still needs to process to obtain a complete sentence structure. The parser also keeps a record of the structure that it has built so far.

Notice that the grammar in Figure 1 is unambiguous. That is, each left-hand side node has exactly one right-hand side. However, in more realistic grammars, this is not the case. Consider the possible expansions of the category VP in Figure 2. The first rule would be used for intransitive verbs such as 'walk', as in 'I walk'. The second rule would be used for transitive verbs such as 'like', as in 'I like the apple'. The third rule would be used for VPs that are modified by a prepositional phrase, for example, 'I see you with the telescope'. A parsing algorithm can handle ambiguity either by trying one choice at a time – a serial approach – or by following multiple alternatives at once – a parallel approach. Under a serial approach, the parser tries one of the rules first and pursues the resulting structure further, and backtracks and tries the other rule(s) only if the first rule fails, or if the goal is to find all possible parses. The ordering of the rules in the grammar determines which rule is applied first. Under a parallel approach, the parser works on all alternative rules and further pursues the resulting structures in parallel threads. A parallel approach requires some data structure that contains a set of parse trees, one tree for each combination of rules.

Top-down Parsing

In *top-down* parsing, one starts with the assumption that the input will eventually form a sentence. This involves initially positing an S-node and all of the extensions of the S-node specified by the grammar (in our case, only $S \rightarrow NP\ VP$). One keeps expanding nodes, following the rules of the grammar, until one finds some matching input. The stack in a top-down parser keeps track of what still needs to be found in order to get a sentence that is well-formed according to the grammar. The pseudocode in Figure 3 shows a top-down algorithm in a general form. Notice that in the case of ambiguity, that is, if a left-hand side of a rule has more than one possible right-hand side (cf. Figure 2), the parser does not specify an order in which these rules are applied. This is called *non-deterministic* parsing.

$$VP \rightarrow Verb$$
$$VP \rightarrow Verb\ NP$$
$$VP \rightarrow VP\ PP$$

Figure 2. Some possible expansions of a *VP*.

```
FUNCTION top-down-parse (SENTENCE)
  initialize STACK = [S]
  DO
    IF (top-element of STACK = non-terminal N) THEN
      Select a rule N → A B
      Pop N from STACK
      Push A B onto STACK
      Add A B below N in the structure for the input
    ELSE IF (top-element of STACK = pre-terminal P) THEN
      Find next word W in SENTENCE
      IF (there is a rule P → W) THEN
        Pop P from STACK
        Add W below P in the structure for the input
      ELSE
        Fail.
      ENDIF
    ENDIF
  UNTIL STACK = [] AND end of SENTENCE is reached.
  RETURN PARSE-TREES
END top-down-parse
```

Figure 3. Pseudocode for top-down parser.

In the following, we provide a step-by-step example of how a top-down parser would parse the sentence 'The man likes the woman', assuming the grammar from Figure 1 (cf. Figure 4):

- Step 1: Push *S* onto the stack. Stack = [*S*]
- Step 2: Apply the rules that have *S* on their LHS. There is only one such rule in our grammar, $S \rightarrow NP\ VP$. Pop *S* from the stack. Push *NP* and *VP* onto the stack. Stack = [*NP VP*]
- Step 3: Apply the rules that have *NP* on their LHS. Here, this is only $NP \rightarrow Det\ Noun$. Pop *NP* from the stack. Stack = [*Det Noun VP*]
- Step 4: Find 'the' in the input and incorporate it into the parse tree. Pop *Det* from the stack. Stack = [*Noun VP*]
- Step 5: Find 'man' in the input and incorporate it into the parse tree. Pop *Noun* from the stack. Stack = [*VP*]
- Step 6: Apply the rules that have *VP* on their LHS. Here, this is only $VP \rightarrow Verb\ NP$. Pop *VP* from the

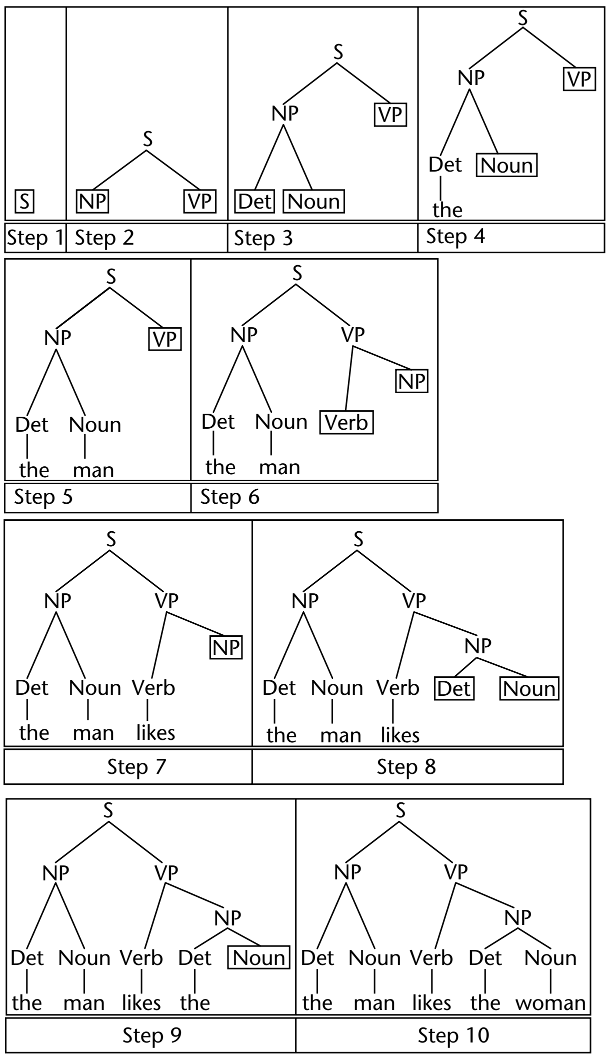

Figure 4. Top-down parsing step-by-step.

stack. Push *Verb* and *NP* onto the stack. Stack = [*Verb NP*]

- Step 7: Find 'likes' in the input and incorporate it into the parse tree. Pop *Verb* from the stack. Stack = [*NP*]
- Step 8: Apply the rules that have *NP* on their LHS. Here, this is only *NP* → *Det Noun*. Pop *NP* from the stack. Push *Det* and *Noun* onto the stack. Stack = [*Det Noun*]
- Step 9: Find 'the' and incorporate it into the parse tree. Pop *Det* from the stack. Stack = [*Noun*]
- Step 10: Find 'woman' in the input and incorporate it into the parse tree. Pop *Noun* from the stack. Stack = [ ]

One of the advantages of a top-down parser is that it never tries to form a structure that will never end up being an *S*, because it starts from *S*. One of the disadvantages of a top-down parser is that it can try to build trees that are inconsistent with the input. This did not happen with our extremely simplified grammar. However, if we also had a rule like *NP* → *Det Adj Noun* (*Adj* = Adjective), the parser could have predicted a structure that is inconsistent with the input, one that contains an Adjective.

Another problem with top-down algorithms is left-recursion. A grammar is left-recursive if it has some LHS that can be expanded through a series of rules such that the left corner of one of these expansions is the same LHS category. For example, if a grammar has a rule *VP* → *VP NP*, the parser could keep extending the left corner of its RHS, *VP*, and get caught in an endless loop, as shown in Figure 5.

Bottom-up Parsing

Whereas top-down parsing starts with the rules, *bottom-up* parsing first looks at the input and then tries to find rules in the grammar that apply to the input. The stack in a bottom-up parser keeps track of what has been found so far and what still has to be integrated in a parse tree. The bottom-up parser considered here consists of two basic steps – pushing categories on the stack that still need to be integrated into the input structure (*shift*), and applying rules in the grammar to the categories on the stack (*reduce*). This algorithm is therefore also called *shift-reduce* parsing. The pseudocode in Figure 6 shows a bottom-up shift-reduce algorithm in a general form.

In the following, we provide a step-by-step example of how a bottom-up shift-reduce parser would parse the sentence 'The man likes the woman', assuming the grammar from Figure 1 (cf. Figure 7):

- Step 1: Find 'the' and its lexical category, *Det*. Push *Det* onto the stack. Stack = [*Det*]
- Step 2: Find 'man' and its lexical category, *Noun*. Push *Noun* onto the stack. Stack = [*Noun Det*]

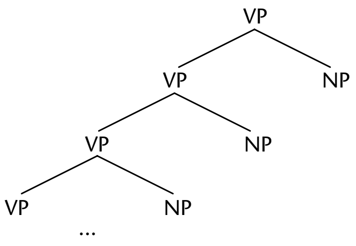

Figure 5. Endless loop due to left-recursion in a top-down algorithm.

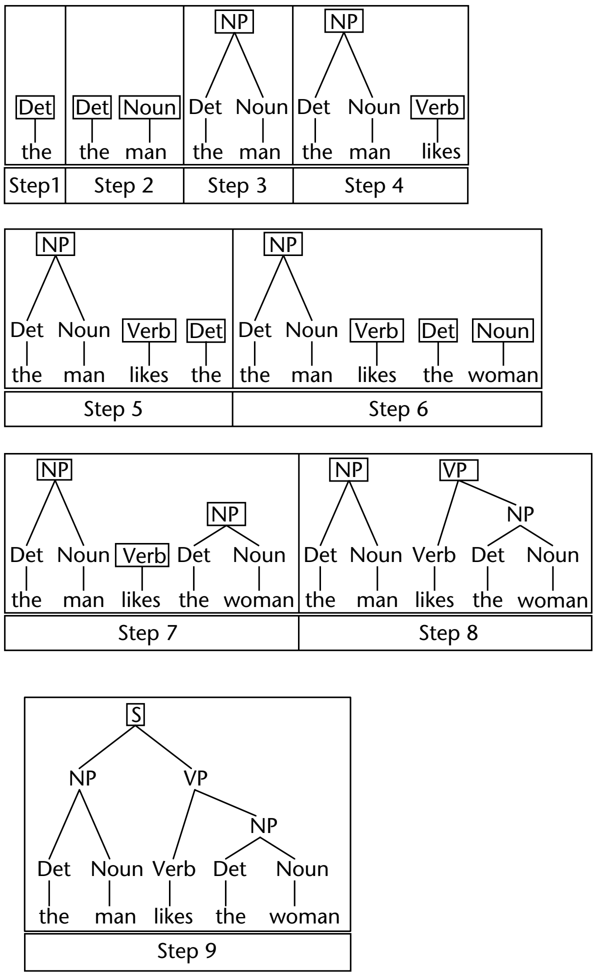

Figure 7. Bottom-up parsing step-by-step.

```
FUNCTION bottom-up-parse (SENTENCE)
  Initialize STACK = []
  DO
    Find next word W in SENTENCE
    IF (there is a rule P ⟶ W) THEN      /* shift */
      Push P onto STACK
    ENDIF
    IF (nodes on STACK = right-hand side of rule N ⟶ A B) THEN   /* reduce */
      Pop from STACK
      Push N onto STACK
      Add N above A B in the structure for the input
    ENDIF
  UNTIL STACK = [S] AND end of SENTENCE is reached
  RETURN PARSE-TREES
END bottom-up-parse
```

Figure 6. Pseudocode for bottom-up parser.

- Step 3: Apply the rule *NP* → *Det Noun* to top of stack. Pop *Det* and *Noun* from the stack. Push *NP* onto the stack. Stack = [*NP*]
- Step 4: Find 'likes' and its lexical category, *Verb*. Push *Verb* onto the stack. Stack = [*Verb NP*]
- Step 5: Find 'the' and its lexical category, *Det*. Push *Det* onto the stack. Stack = [*Det Verb NP*]
- Step 6: Find 'woman' and its lexical category, *Noun*. Push *Noun* onto the stack. Stack = [*Noun Det Verb NP*]
- Step 7: Apply the rule *NP* → *Det Noun* to top of stack. Pop *Det* and *Noun* from the stack. Push *NP* onto the stack. Stack = [*NP Verb NP*]
- Step 8: Apply the rule *VP* → *Verb NP* to top of stack. Pop *Verb* and *NP* from the stack. Push *VP* onto the stack. Stack = [*VP NP*]
- Step 9: Apply the rule *S* → *NP VP* to top of stack. Pop *NP* and *VP* from the stack. Push *S* onto the stack. Stack = [*S*]

An advantage of bottom-up parsers is that they do not predict parse trees that are inconsistent with the input. Furthermore, unlike top-down parsers, bottom-up parsers cannot get caught in an endless loop if the grammar contains left-recursive rules. A disadvantage of bottom-up parsers is that they can generate structures that never result in an *S*.

Left-corner Parsing

Left-corner parsing combines some elements from both top-down and bottom-up parsing. In left-corner parsing, only rules consistent with the input are predicted. That is, a rule is only predicted (top-down) if the current input (bottom-up) matches the leftmost node (hence left-corner) of the RHS of a rule. In left-corner parsing, a stack is used to keep track of what input is still needed to complete a predicted rule.

Left-corner parsing is particularly interesting from a cognitive science point of view, because it mirrors human performance in parsing more closely than pure top-down or bottom-up parsers. Consider the following sentences:

John's brother's dog's tail fell off. (4)

Table 1. Performance of parsing algorithms and humans compared

	Sentence structure		
	Left-branching	*Center-embedded*	*Right-branching*
Stack size, top-down	unbounded	unbounded	bounded
Stack size, bottom-up	bounded	unbounded	unbounded
Stack size, left-corner	bounded	unbounded	bounded
Processing for humans	easy	hard	easy

```
FUNCTION left-corner-parse (SENTENCE)
  Initialize STACK = [S]
  DO
    Find next word W in SENTENCE
    IF (there is a rule P ⟶ W) THEN
      IF ((top-element of STACK = non-terminal N) AND ( P = left-corner
      of rule R ⟶ A B)) THEN
        Pop N from STACK
        Push B onto STACK
      ELSE IF ((STACK = []) AND (P = left-corner of rule R ⟶ A B)) THEN
        Push B onto STACK
      ENDIF
    ELSE
      Fail.
    ENDIF
  UNTIL STACK = [] AND end of SENTENCE is reached
  RETURN PARSE-TREES
END left-corner-parse
```

Figure 8. Pseudocode for left-corner parser.

The dog chased the cat that caught the mouse that squeaked. (5)

The mouse that the cat that the dog chased caught squeaked. (6)

Sentence (4) has a *left-branching* structure, (5) a *right-branching* structure, and (6) a *center-embedded* structure. Humans do not have difficulty understanding sentences with left-branching structures like (4) or with right-branching structures like (5). However, sentences with center-embedded structures like (6) are hard for humans to process, in spite of the fact that (6) has virtually the same meaning as (5). It is plausible that the difficulty in center-embedded structures like (6) has to do with the quantity of storage space that is required to parse them. Table 1 shows a comparison between the processing complexity, for humans, of different structural types, and the storage space for different parsing algorithms.

Table 1 shows that the stack size in a left-corner parsing algorithm mirrors human performance, but not in a top-down or a bottom-up algorithm. Thus, left-corner parsing algorithms are more psychologically plausible than top-down or bottom-up parsing algorithms.

The pseudocode in Figure 8 shows a left-corner algorithm in a general form.

Here is a step-by-step example parse of the sentence 'The man likes the woman' (cf. Figure 9):

- Step 1: Find 'the' and its lexical category, *Det*. Stack = [*S*]
- Step 2: Apply the rule *NP* → *Det Noun*. Push *Noun* onto the stack. Stack = [*S Noun*]
- Step 3: Find 'man' and its lexical category, *Noun*. Pop *Noun* from the stack. Stack = [*S*]
- Step 4: Apply the rule *S* → *NP VP*. Push *VP* onto the stack. Stack = [*VP*]
- Step 5: Find 'likes' and its lexical category, *Verb*. Pop *VP* from the stack. Stack = [ ]
- Step 6: Apply the rule *VP* → *Verb NP*. Push *NP* onto the stack. Stack = [*NP*]
- Step 7: Find 'the' and its lexical category, *Det*. Pop *NP* from the stack. Stack = [ ]
- Step 8: Apply the rule *NP* → *Det Noun*. Push *Noun* onto the stack. Stack = [*Noun*]
- Step 9: Find 'woman' and its lexical category, *Noun*. Pop *Noun* from the stack. Stack = [ ]

Head-corner Parsing

Head-corner parsing is a generalization of left-corner parsing. In left-corner parsing, one looks for the left-corner of the RHS of a rule to make top-down predictions. In head-corner parsing, one looks for the head of a rule. Roughly speaking, the

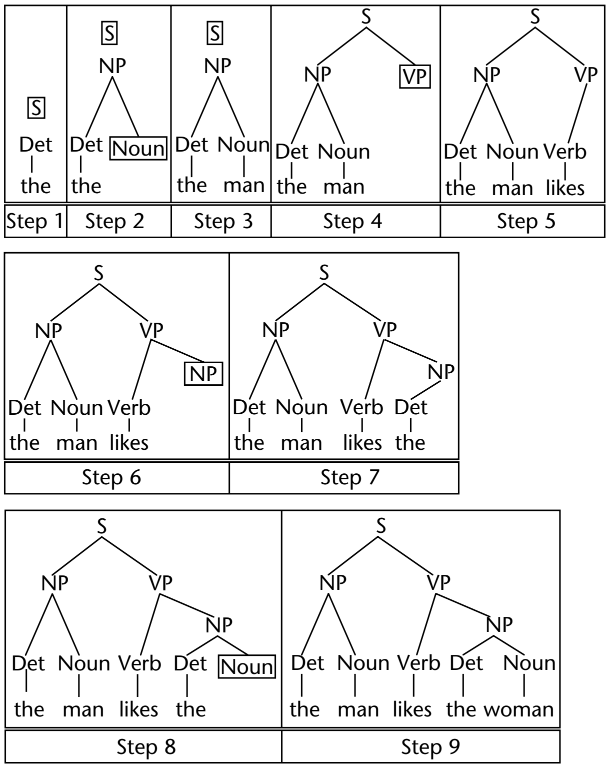

Figure 9. Left-corner parsing step-by-step.

head of a rule is the word at the semantic core of that rule. For example, in the rule *NP* → *Det Noun*, *Noun* is at the semantic core. Thus, Noun is the head of NP. The idea behind looking at the head of a rule first is that it contains a lot of useful information (such as subcategorization, thematic roles, etc.) which can allow for better top-down predictions. For instance, the lexical entry for a verb, which is the head of a VP, contains information about what NPs or PPs are needed to form a complete VP. As an example, consider the verb 'like'. Its lexical entry would contain the information that in order to form a complete VP, it needs an NP that denotes someone or something that is liked. These additional items are called *arguments*.

Having such information about head–argument relations available can improve the efficiency of parsing languages such as Japanese or German, whose word order is less constrained than that of English. Consider sentence (7):

The man likes the woman. (7)

In German, this sentence might also be written as

The woman likes the man. (8)

meaning, 'the man likes the woman'. The subject and object NPs are distinguished by morphological case marking in German. To be able to parse (7) as well as (8) correctly, a left-corner parsing algorithm would need two sets of rules. There would have to be one set of rules for (7) that includes $S \rightarrow NP\ VP$ and $VP \rightarrow Verb\ NP$. Another set of rules for (8) would have to include $S \rightarrow VP\ NP$ and $VP \rightarrow NP\ Verb$. A head-corner parsing algorithm does not require such a complex grammar. It is enough to have one set of rules that includes $S \rightarrow NP\ VP$ and $VP \rightarrow Verb\ NP$. Once a head-corner parser has found the head of the VP, 'likes', it can look for the argument of this head. This can be done bidirectionally – left-to-right or right-to-left – so that word order does not matter. The same applies to the rule $S \rightarrow NP\ VP$.

```
STRUCTURE EDGE {
    CATEGORY
    CHILDREN
    LEFT-END
    RIGHT-END
    REQUIRED-CATEGORIES
}
FUNCTION chart-parse (SENTENCE)            /* the top-level function */
    initialize-chart SENTENCE, returning CHART-ACTIVES, CHART-INACTIVES,
    PARSE-TREE
    IF ((CHART-INACTIVES contains node S) AND (distance (left-corner (node S)
    AND to right-corner (node S)) = length (SENTENCE))) THEN
        RETURN PARSE-TREES
    ENDIF
END chart-parse
FUNCTION initialize-chart (SENTENCE)
    FOR (all words W in SENTENCE) DO
        add-new-edge LEXICAL-ENTRY (W)
    END
    RETURN (PARSE-TREE, CHART-ACTIVES, CHART-INACTIVES)
END initialize-chart

FUNCTION add-new-edge (EDGE)
  IF (EDGE = inactive) THEN
    FOR (all CHART-ACTIVES) DO
      fundamental-rule EDGE, CHART-ACTIVES
      add EDGE to CHART-INACTIVES
      add-null-active-edges EDGE
    END
  ELSE IF (EDGE = active) THEN
    FOR (all CHART-INACTIVES) DO
      fundamental rule EDGE, CHART-INACTIVES
      add EDGE to CHART-ACTIVES
    END
  ENDIF
END add-new-edge
```

```
FUNCTION fundamental-rule (ACTIVE-EDGE, INACTIVE-EDGE)
  IF ((left-end of INACTIVE-EDGE matches right-end of ACTIVE-EDGE) AND
  (INACTIVE-EDGE satisfies first category requirement of ACTIVE-EDGE))
  THEN
    NEW-EDGE = INACTIVE-EDGE incorporated in ACTIVE-EDGE
    IF (NEW-EDGE = active) THEN
      add NEW-EDGE to CHART-ACTIVES
    ELSE IF (NEW-EDGE = inactive) THEN
        add NEW-EDGE to CHART-INACTIVES
      ENDIF
   ENDIF
END fundamental-rule

FUNCTION add-null-active-edges (INACTIVE-EDGE)
   FOR (each rule from GRAMMAR whose rhs is initiated by INACTIVE-EDGE) DO
      NEW-ACTIVE-EDGE = rule in GRAMMAR with rhs initiated by INACTIVE-
      EDGE
      add-new-edge NEW-ACTIVE-EDGE
   END
END add-null-active-edges
```

Figure 10. Pseudocode for chart parser.

CHART PARSING

As mentioned in the introduction, human languages are extremely ambiguous. The parsing strategies discussed so far are very inefficient at handling ambiguity, whether they use a serial approach or a parallel approach. Consider the following sentence:

The tall man with brown hair saw the short woman with blond hair. (9)

Remember that 'saw' is ambiguous between a *Noun*-reading (a tool) and a *Verb*-reading (past tense of 'see'). This means that 'the tall man with brown hair' would have to be processed twice – once for the *Noun*- and once for the *Verb*-reading – although this segment gets the same structure in both parses. In realistic large-scale applications with big grammars, this can lead to efficiency problems that paralyse the whole application.

A way out of this situation is to give the parser a memory. With a memory, the parser can keep a record of all the parses it has attempted so far, and look them up instead of reparsing them. In parsing, such a memory is called *chart*. The chart keeps a record of partially as well as completely parsed rules from the grammar. In a chart, these rules are called *edges*. Partially parsed edges are called *incomplete* or *active*. Completely parsed edges are called *complete* or *inactive*. Once an edge is entered into the chart, it stays there. This is because we want to keep a record of all possible parses of a sentence, to facilitate backtracking. The following information about all edges (both active and inactive) is contained in a chart:

- the syntactic category of the edge, e.g. *NP*
- where in the sentence the edge begins (the *left end*)
- where in the sentence the edge ends (the *right end*)
- pointers to further inactive edges, e.g. to *Det Noun* for *NP*
- The following information is also included for active edges: a list of what categories are still needed to complete the active edge (to make it inactive).

A chart parser functions by combining active edges with inactive edges via the *Fundamental Rule*. The Fundamental Rule looks for matches between categories that are needed in an active edge and the set of inactive (complete) edges. In order to start a parse, one has to specify a top-down or bottom-up strategy. This is accomplished by

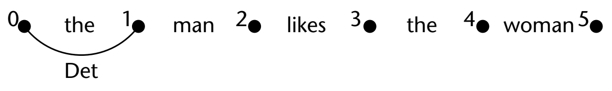

Figure 11. Chart parsing, step 1.

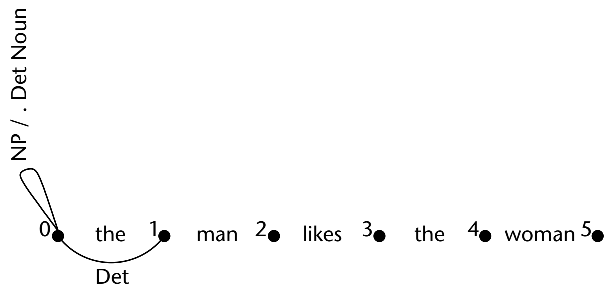

Figure 12. Chart parsing, step 2.

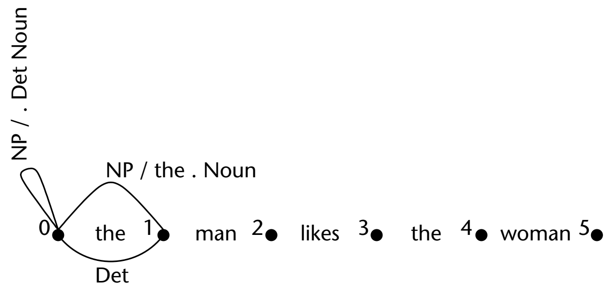

Figure 13. Chart parsing, step 3.

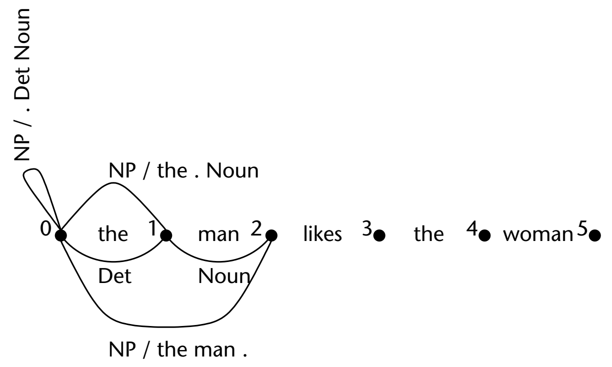

Figure 14. Chart parsing, step 4.

initializing the chart by adding either an empty active edge for an S (top-down) or inactive edges for the words in the sentence (bottom-up). The algorithm presented in Figure 10 uses a bottom-up strategy.

During the parse, when a match is found, a new edge is constructed by adding the inactive edge to

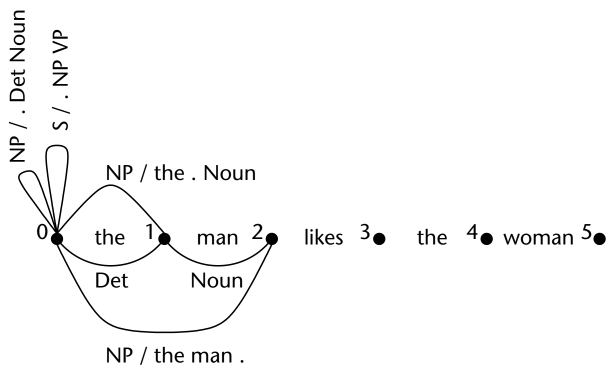

Figure 15. Chart parsing, step 5.

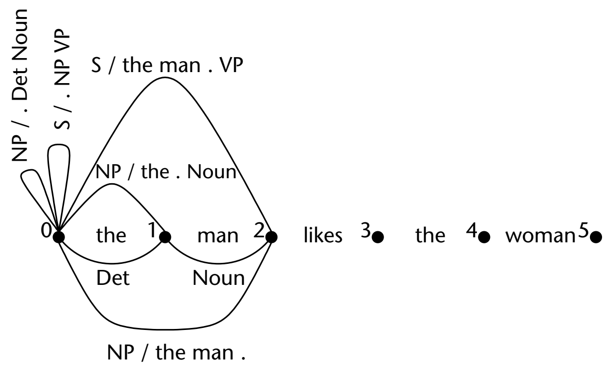

Figure 16. Chart parsing, step 6.

the contents of the active edge. The original edges are left in the chart to allow reanalysis. The edges in the chart are labeled as follows:

[category]/[what is already there]. [what is still needed]

Examples:

- *S/NP. VP*: category is S, an NP has been processed already, a VP is still needed, so the edge is active.
- *NP/Det Noun.* : category is NP, a Det and a Noun have already been processed, so the edge is inactive.

Active edges are printed above the words, inactive edges below. Furthermore, there are solid circles that denote stages of the parser.

Here is a step-by-step example, parsing the sentence 'The man likes the woman' (cf. Figures 11–21; to keep the figures legible, only the more important edges are shown):

- Step 1: Find 'the' and its lexical category, *Det*. Push an inactive edge, [*Det* / the .], onto chart-inactives. Chart-actives = [ ], chart-inactives = [*Det* / the .]
- Step 2: Make null-active-edge, [*NP*/. *Det Noun*]. Chart-actives = [*NP*/. *Det Noun*], chart-inactives = [*Det* / the .]
- Step 3: Apply Fundamental Rule to inactive edge, [*Det* / the .], and incorporate it into null-active-edge,

[*NP* / . *Det Noun*]. Push active edge, [*NP* / the . *Noun*], onto chart-actives. Chart-actives = [[*NP* / the . *Noun*] [*NP* / . *Det Noun*]], chart-inactives = [*Det* / the .]

- Step 4: Find 'man' and its lexical category, *Noun*. Push an inactive edge, [*Noun* / man .], onto chart-inactives. Apply Fundamental Rule to inactive edge, [*Noun* / man .], and incorporate it into active edge, [*NP* / *Det* . *Noun*]. Push inactive edge, [*NP* / *Det Noun*.], onto chart-inactives. Chart-actives = [[*NP* / the . *Noun*] [*NP* / . *Det Noun*]], chart-inactives = [[*NP* / the man .] [*Noun* / man .] [*Det* / the .]]
- Step 5: Make null-active-edge, [*S* / . *NP VP*]. Chart-actives = [[*S* / . *NP VP*] [*NP* / the . *Noun*] [*NP* / . *Det Noun*]], chart-inactives = [[*NP* / the man .] [*Noun* / man .] [*Det* / the .]]
- Step 6: Apply Fundamental Rule to inactive edge, [*NP* / the man .], and incorporate it into active edge, [*S* / . *NP VP*]. Push active edge, [*S* / the man . *VP*], onto chart-actives. Chart-actives = [[*S* / the man. *VP*] [*S* / . *NP VP*] [*NP* / the . *Noun*] [*NP* / . *Det Noun*]], chart-inactives = [[*NP* / the man .] [*Noun* / man .] [*Det* / the .]]
- Step 7: Find 'likes' and its lexical category, *Verb*. Push an inactive edge, [*Verb* / likes .], onto chart-inactives. Chart-actives = [[*S* / the man . *VP*] [*S* / . *NP VP*] [*NP* / the . *Noun*] [*NP* / . *Det Noun*]], chart-inactives = [[*Verb* / likes .] [*NP* / the man .] [*Noun* / man .] [*Det* / the .]]
- Step 8: Make null-active-edge, [*VP* / . *Verb NP*]. Chart-actives = [[*VP* / . *Verb NP*] [*S* / the man . *VP*] [*S* / . *NP VP*] [*NP* / the . *Noun*] [*NP* / . *Det Noun*]], chart-inactives = [[*Verb* / likes .] [*NP* / the man .] [*Noun* / man .] [*Det* / the .]]
- Step 9: Apply Fundamental Rule to inactive edge, [*Verb* / likes .], and incorporate it into active edge, [*VP* / . *Verb NP*]. Push active edge, [*VP* / likes . *NP*], onto chart-actives. Chart-actives = [[*VP* / likes . *NP*] [*VP* / . *Verb NP*] [*S* / the man . *VP*] [*S* / . *NP VP*] [*NP* / the . *Noun*] [*NP* / . *Det Noun*]], chart-inactives = [[*Verb* / likes .] [*NP* / the man .] [*Noun* / man .] [*Det* / the .]]
- Step 10: Parse *NP* 'the woman', similar to *NP* 'the man' (cf. steps 1–4). Chart-actives = [[*NP* / the . *Noun*] [*NP* / . *Det Noun*] [*VP* / likes . *NP*] [*VP* / . *Verb NP*] [*S* / the man . *VP*] [*S* / . *NP VP*] [*NP* / the . *Noun*] [*NP* / . *Det Noun*]], chart-inactives = [[*NP* / the woman .] [*Noun* / woman .] [*Det* / the .] [*Verb* / likes .] [*NP* / the man .] [*Noun* / man .] [*Det* / the .]]
- Step 11: Apply Fundamental Rule to inactive edge, [*NP* / the woman .], and incorporate it into active edge, [*VP* / likes . *NP*]. Push inactive edge, [*VP* / likes the woman .], onto chart-inactives. Apply Fundamental Rule to inactive edge, [*VP* / likes the woman .], and incorporate it into active edge, [*S* / the man . *VP*]. Push inactive edge, [*S* / the man likes the woman .], onto chart-inactives. Chart-actives = [[*NP* / the . *Noun*] [*NP* / . *Det Noun*] [*VP* / likes . *NP*] [*VP* / . *Verb NP*] [*S* / the man . *VP*] [*S* / . *NP VP*] [*NP* / the . *Noun*] [*NP* / . *Det Noun*]], chart-inactives = [[*S* / the man likes the woman .] [*VP* / likes the woman .] [*NP* / the woman .] [*Noun* / woman .] [*Det* / the .] [*Verb* / likes .] [*NP* / the man .] [*Noun* / man .] [*Det* / the .]]

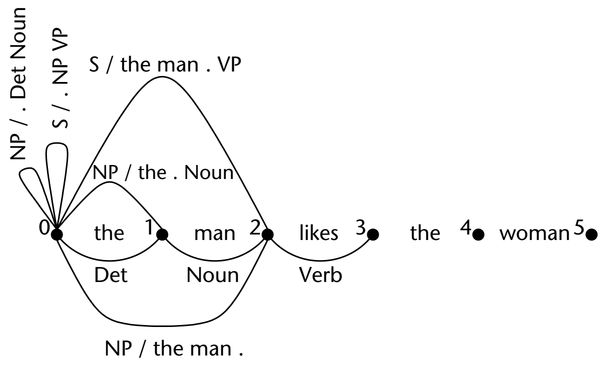

Figure 17. Chart parsing, step 7.

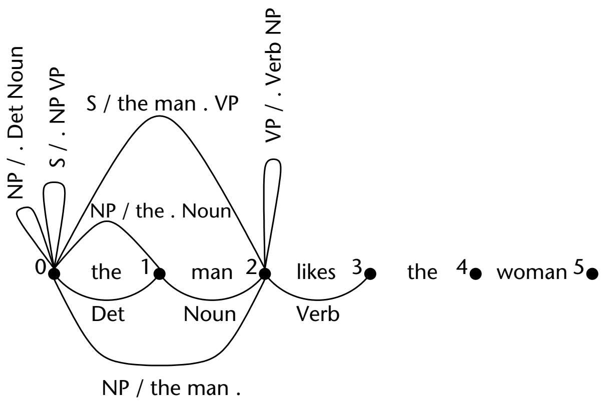

Figure 18. Chart parsing, step 8.

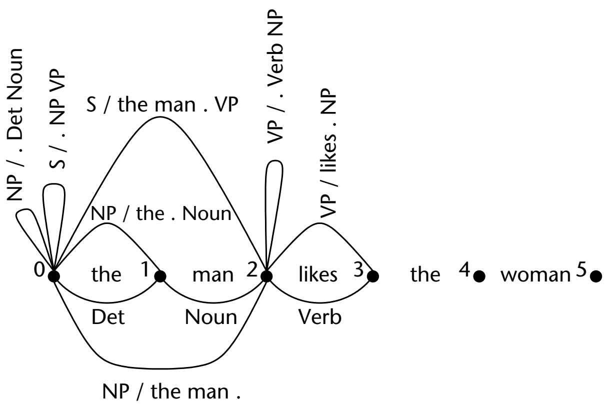

Figure 19. Chart parsing, step 9.

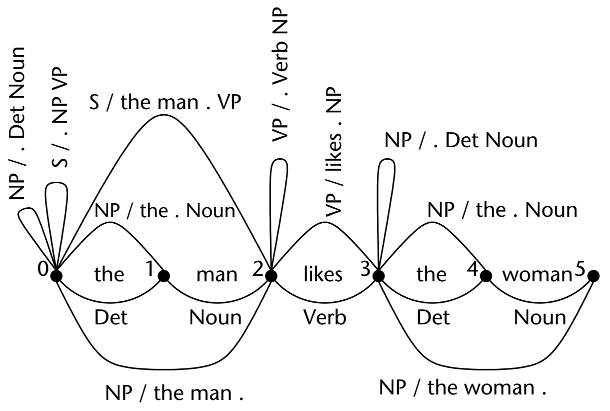

Figure 20. Chart parsing, step 10.

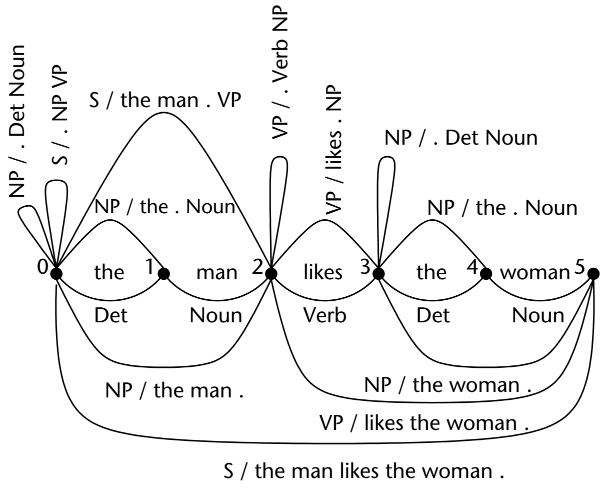

Figure 21. Chart parsing, step 11.

SUMMARY

Parsing – determining the structure of a sentence, given a grammar – is a crucial aspect of establishing meaning in language processing. There are two basic sets of constraints in parsing. One set of constraints is top-down: the rules in the grammar constrain which structures sentences can have in a given language. The other set of constraints is bottom-up: the input sentence to the algorithm constrains which rules from the grammar can apply. Parsing algorithms differ with respect to the constraints they use more prominently: bottom-up, top-down, or a combination of both (left- and head-corner).

An important issue in parsing is efficiently dealing with structural and lexical ambiguities. In an ambiguity, if the later disconfirmed alternative reading is pursued initially, a standard parser has to backtrack and re-parse parts of the input sentence. Re-parsing can be avoided by using a chart, a data structure that stores partially parsed input, so that it can be looked up during backtracking.

Further Reading

Abney SP and Johnson M (1991) Memory requirements and local ambiguities of parsing strategies. *Journal of Psycholinguistic Research* **20**(30): 233–250.

Aho AV and Ullman JD (1972) *The Theory of Parsing, Translation, and Compiling*, vol. 1: Parsing. Englewood Cliffs, NJ: Prentice-Hall.

Crocker M (1999) Mechanisms for sentence processing. In: Garrod SC and Pickering M (eds) *Language Processing*. London, UK: Psychology Press.

Dowty D, Karttunen L and Zwicky A (eds) (1985) *Natural Language Processing: Psychological, Computational and Theoretical Perspectives*. Cambridge, UK: Cambridge University Press.

Grosz BJ, Jones KS and Webber BL (eds) (1986) *Readings in Natural Language Processing*. Los Altos, CA: Morgan Kaufmann.

Jurafsky D and Martin JH (2000) *Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition, and Computational Linguistics*. Upper Saddle River, NJ: Prentice-Hall.

Manning CD and Schütze H (1999) *Foundations of Statistical Natural Language Processing*. Cambridge, MA: MIT Press.

Pereira F and Shieber SM (1987) *Prolog and Natural Language Analysis*. Cambridge, UK: Cambridge University Press.