Parsing By Chunks

Steven Abney
Bell Communications Research

In: Robert Berwick, Steven Abney and Carol Tenny (eds.), *Principle-Based Parsing,* Kluwer Academic Publishers, 1991

abney@sfs.nphil.uni-tuebingen.de
http://www.sfs.nphil.uni-tuebingen.de/~ abney/
Wilhelmstr. 113, 72074 Tübingen, Germany

Parsing By Chunks

Steven P. Abney
Bell Communications Research

November 10, 1994

0 Introduction

I begin with an intuition: when I read a sentence, I read it a chunk at a time. For example, the previous sentence breaks up something like this:

(1) [I begin] [with an intuition]: [when I read] [a sentence], [I read it] [a chunk] [at a time]

These chunks correspond in some way to prosodic patterns. It appears, for instance, that the strongest stresses in the sentence fall one to a chunk, and pauses are most likely to fall between chunks. Chunks also represent a grammatical watershed of sorts. The typical chunk consists of a single content word surrounded by a constellation of function words, matching a fixed template. A simple context-free grammar is quite adequate to describe the structure of chunks. By contrast, the relationships *between* chunks are mediated more by lexical selection than by rigid templates. Co-occurence of chunks is determined not just by their syntactic categories, but is sensitive to the precise words that head them; and the order in which chunks occur is much more flexible than the order of words within chunks.

The work I would like to describe is an attempt to give content to these intuitions, and to show that parsing by chunks has distinct processing advantages, advantages that help explain why the human parser might adopt a chunk-by-chunk strategy.

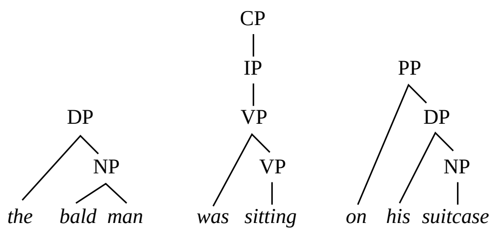
1 Chunks

There is psychological evidence for the existence of chunks. Gee and Grosjean 1983 examine what they call *performance structures*. These are structures of word clustering that emerge from a variety of types of experimental data, such as pause durations in reading, and naive sentence diagramming. Gee and Grosjean argue that performance structures are best predicted by what they call ϕ*-phrases*. ϕ-phrases are created by breaking the input string after each syntactic head that is a content word (with the exception that function words syntactically associated with a preceding content word—in particular, object pronouns—group with the preceding content word). The chunks of sentence (1) are ϕ-phrases.

Unfortunately, Gee and Grosjean must make some undesirable syntactic assumptions. For example, they assume that prenominal adjectives do not qualify as syntactic heads—otherwise, phrases like *a big dog* would not comprise one chunk, but two. Also, Gee and Grosjean do not assign syntactic structure to chunks. To remedy these deficiencies, I assume that a chunk has syntactic structure, which comprises a connected subgraph[1] of the sentence's parse-tree, and I define chunks in terms of *major heads*. Major heads are all content words except those that appear between a function word f and the content word that f selects.[2,3] For example, *proud* is a major head in *a man proud of his son*, but *proud* is not a major head in *the proud man*, because it appears between the function word the and the content word *man* selected by *the*.

The parse tree segments associated with some sample chunks are illustrated in (2).

(2)



They are determined as follows. Let h be a major head. The root of the chunk headed by h is the highest node in the parse tree for which h is the s-head, that is, the 'semantic' head. Intuitively, the s-head of a phrase is the most prominent word in the phrase. For example, the verb is the s-head of a sentence, the noun is the s-head of a noun phrase or prepositional phrase, and the adjective is the s-head of an adjective phrase. The s-head is not necessarily the same as the syntactic head. In GB, for example, an abstract element Infl, not the verb, is taken to be the head of the sentence, and the complementizer (C) is often taken to be the head of an embedded sentence (CP). (See Chomsky

[1] A chunk's structure is in fact a tree, but it is not necessarily a subconstituent of the global parse-tree. In particular, the chunk's root node may have some descendants in the global tree that are absent from the chunk's parse-tree.

[2] I consider the relation between a function word (e.g., determiner) and associated content-word head (e.g., noun) to be one of selection. See Abney 1987 for arguments.

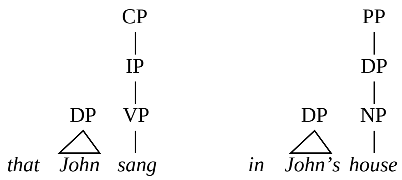
[3] There is one case that this definition does not handle. We wish to say that a pronoun that heads a prepositional phrase is a major head, despite being a function word, not a content word. I have no more elegant solution at present than to add a disjunctive clause to the definition of major head: "a major head is any content word that does not appear between a function word f and the content word f selects, OR a pronoun selected by a preposition."

1986, for example.) P is generally taken to be the head of PP, not the noun. And under the DP-analysis (Abney 1987), which I adopt, the determiner is the head of the noun phrase, and a degree element, not the adjective, is the head of the adjective phrase. S-heads can be defined in terms of syntactic heads, however, as follows. If the syntactic head h of a phrase P is a content word, h is also the s-head of P. If h is a function word, the s-head of P is the s-head of the phrase selected by h.

The parse-tree T_C of a chunk C is a subgraph of the global parse-tree T. The root r of T_C is the highest node whose s-head is the content word defining C. For example, in (2), *man*, *sitting*, and *suitcase* are the major heads. $r =$ DP is the highest node whose s-head is *man*; $r =$ IP is the highest node whose s-head is *sitting*; and $r =$ PP is the highest node whose s-head is *suitcase*. T_C is the largest subgraph of T dominated by r that does not contain the root of another chunk. In (2), the parse-tree of *man*'s chunk is the subtree rooted in DP. The parse-tree of *sitting*'s chunk is the subtree rooted in CP (i.e., the entire global parse-tree) with the subtrees under DP and PP excised. The parse-tree of *suitcase*'s chunk is the subtree rooted in PP.

There is a single special case. Terminal nodes are excluded from a chunk if their inclusion would cause the chunk to have a discontinuous frontier. Examples of such 'orphan nodes' are complementizers, where the subject intervenes between the complementizer and the rest of the verb chunk, and some prepositions, where they are separated from the rest of the noun chunk by an intervening possessor. For example:

(3)



ϕ-phrases are generated from chunks by sweeping orphaned words into an adjacent chunk. As a consequence, ϕ-phrases, unlike chunks, do not always span connected subgraphs of the parse tree. In (3), for example, *that John* constitutes a ϕ-phrase; but syntactically, the phrase *that John* contains two unconnected fragments. The correspondence between prosodic units and syntactic units is not direct, but mediated by chunks. ϕ-phrases are elements in a prosodic level of representation. Chunks and global parse-trees are elements of two different levels of syntactic representation. Global parse-trees and ϕ-phrases are both calculated from chunks, but neither global parse-trees nor ϕ-phrases are calculated from the other.

A final issue regarding the definition of chunks is the status of pronouns. On the one hand, we would like a clean division between the grammar of chunks and the grammar of interchunk relations. Since pronouns function syntactically like noun chunks—in particular, they can fill subject and object positions—we would like to consider them chunks. On the other hand, they are generally stressless, suggesting that they not be treated as separate chunks (we did not treat them as separate chunks in (1), for example). A reasonable solution is to consider them to be lexical noun phrases, and assign them the same status as orphaned words. At the level of chunks, they are orphaned words, belonging to no chunk. At the level of ϕ-phrases, they are swept into an adjacent chunk. And at the level of syntax, they are treated like any other noun phrase.

We are now in a position to be more specific about *which* adjacent chunk orphaned words are swept into. If the orphaned word takes a complement, it is swept into the nearest chunk in the direction of its complement (i.e., the following chunk). Otherwise it is swept into the nearest chunk in the direction of its syntactic governor. For example, pronouns are function words that do not take complements. (To the best of my knowledge, they are the only function words that do not take complements.) Subject pronouns are swept into the following chunk, and object pronouns are swept into the preceding chunk.

The reader can verify that the units marked in (1) are ϕ-phrases (not chunks), in accordance with the definitions given in this section.

2 Structure of the Parser

A typical natural language parser processes text in two stages. A tokenizer/morphological analyzer converts a stream of characters into a stream of words, and the parser proper converts a stream of words into a parsed sentence, or a stream of parsed sentences. In a *chunking parser*, the syntactic analyzer is decomposed into two separate stages, which I call the *chunker* and the *attacher*. The chunker converts a stream of words into a stream of chunks, and the attacher converts the stream of chunks into a stream of sentences.

The attacher's name is derived from the manner in which it assembles chunks into a complete parse tree. It *attaches* one chunk to another by adding missing arcs between parse-tree nodes. In (2), for example, the attacher must add an arc from the IP node to the DP node dominating *the bald man*, and it must add an arc from the lower VP node to the PP node.

To illustrate the action of these three stages, the following are the streams output by each when parsing the sentence

> *The effort to establish such a conclusion of course must have two foci, the study of the rocks and the study of the sun.*
> (taken from Williams 1986)
>
> Words:
>
> $\{[_{\text{Det}}$ the$]\}$ $\{[_{\text{N}}$ effort$]\}$ $\{[_{\text{Inf-To}}$ to$]$ $[_{\text{P}}$ to$]\}$ $\{[_{\text{V}}$ establish$]\}$ $\{[_{\text{Predet}}$ such$]$ $[_{\text{Det}}$ such$]$ $[_{\text{Pron}}$ such$]\}$ $\{[_{\text{Det}}$ a$]\}$ $\{[_{\text{N}}$ conclusion$]\}$ $\{[_{\text{Adv}}$ of course$]\}$ $\{[_{\text{N}}$ must$]$ $[_{\text{V}}$ must$]\}$ $\{[_{\text{V}}$ have$]\}$ $\{[_{\text{Num}}$ two$]\}$ $\{[_{\text{N}}$ foci$]\}$ $\{[_{\text{Comma}}$,$]\}$
> ...

Words are sets of *readings*. Readings, but not words, have unique syntactic categories, feature-sets, etc. There is no one-one correspondence between words and pieces of text separated by whitespace. For example, we permit words with embedded whitespace, such as *of course*.

Chunks:

$[_{\text{DP}}$ $[_{\text{Det}}$ the] $[_{\text{NP}}$ $[_{\text{N}}$ effort]]]
$[_{\text{CP-Inf}}$ $[_{\text{IP-Inf}}$ $[_{\text{Inf-To}}$ to] $[_{\text{VP}}$ $[_{\text{V}}$ establish]]]]
$[_{\text{DP}}$ $[_{\text{Predet}}$ such] $[_{\text{Det}}$ a] $[_{\text{NP}}$ $[_{\text{N}}$ conclusion]]]
$[_{\text{CP}}$ $[_{\text{IP}}$ $[_{\text{AdvP}}$ $[_{\text{Adv}}$ of course]] $[_{\text{Modal}}$ will] $[_{\text{VP}}$ $[_{\text{V}}$ have]]]]
$[_{\text{DP}}$ $[_{\text{NP}}$ $[_{\text{Num}}$ two] $[_{\text{N}}$ foci]]]
$[_{\text{Comma}}$,]
$[_{\text{DP}}$ $[_{\text{Det}}$ the] $[_{\text{NP}}$ $[_{\text{N}}$ study]]]
$[_{\text{PP}}$ $[_{\text{P}}$ of] $[_{\text{DP}}$ $[_{\text{Det}}$ the] $[_{\text{NP}}$ $[_{\text{N}}$ rocks]]]]
...

Lexical ambiguity is often resolvable within chunks, as seen here. Single-word chunks represent a common exception. A single word does not provide enough context to resolve lexical ambiguity.

Parse:

$[_{\text{CP}}$ $[_{\text{IP}}$ $[_{\text{DP}}$ the effort $[_{\text{CP-Inf}}$ to establish $[_{\text{DP}}$ such a conclusion]]]
    $[_{\text{VP}}$ of course must have $[_{\text{DP}}$ two foci]]
    $[_{\text{Appos}}$ $[_{\text{DP}}$ $[_{\text{DP}}$ the study $[_{\text{PP-of}}$ of the rocks]]
        $[_{\text{Conj}}$ and]
        $[_{\text{DP}}$ the study $[_{\text{PP-of}}$ of the sun]]]]]]

I have omitted chunk-internal nodes that are not themselves the roots of chunks, to make it clear what structure the attacher itself has built. In fact, though, there is no distinction in the final parse between nodes built by the chunker and nodes built by the attacher.

3 Chunker

The chunker is a non-deterministic version of an LR parser (Knuth 1965), employing a best-first search. I first give a brief description of LR parsing, for those unfamiliar with it. (A much more detailed discussion can be found in Aho and Ullman 1972.)

3.1 LR Parsing

An LR parser is a deterministic bottom-up parser. It is possible to automatically generate an LR parser for any of a large class of context-free grammars. The parser *shifts* words from the input string onto the stack until it recognizes a sequence of words matching the right-hand side of a rule from the grammar. At that point, it *reduces* the sequence to a single node, whose category is given in the left-hand side of the rule.

For example, consider the grammar

1. S → NP VP
2. NP → Det N
3. NP → N
4. VP → V NP

Suppose the input is N V N. The parser shifts the first N onto the stack. It recognizes the right-hand side of rule 3, and reduces N to NP. It continues as follows:

Stack	*Input*	*Action*
[ ]	N V N	SH N
[N]	V N	RE NP→N
[NP]	V N	SH V
[NP V]	N	SH N
[NP V N]		RE NP→N
[NP V NP]		RE VP→V NP
[NP VP]		RE S→NP VP
[S]		Accept

Control is mediated by *LR states*, which are kept on a separate control stack.[4] LR states correspond to sets of items. An item is a rule with a dot marking how much of the rule has already been seen. An example of an item-set is:

$$(4) \quad \begin{bmatrix} \mathrm{VP} \rightarrow \mathrm{V} \bullet \mathrm{NP} \\ \mathrm{VP} \rightarrow \mathrm{V} \bullet \mathrm{S} \\ \mathrm{NP} \rightarrow \bullet \mathrm{Det\ N} \\ \mathrm{NP} \rightarrow \bullet \mathrm{N} \\ \mathrm{S} \rightarrow \bullet \mathrm{NP\ VP} \end{bmatrix}$$

The kernel of an item-set is the set of items with some category preceding the dot. In (4), the kernel is [VP → V•NP, VP → V•S]. The rest of the item-set can be generated from the kernel, by adding items for every expansion of every category after a dot. In (4), we add NP → •N, NP → •Det N because of the NP after the dot in VP → V•NP. N and Det, which follow the dot in the new items, generate no new items, because they are terminals. VP → V•S generates

[4] In fact, the standard LR parser has only a control stack. Instead of building a parse tree, it outputs a string of rule-numbers, one for each reduction. Such a string encodes a parse-tree: reversed, it specifies a rightmost derivation of the input string. 'LR' stands for 'Left-to-right, Rightmost derivation.'

the item S $\rightarrow$ •NP VP. The NP after the dot generates the items NP $\rightarrow$ •N, NP $\rightarrow$ •Det N, but since they are already present, nothing changes.

Item-sets control the computation as follows. If a terminal symbol follows the dot in some item, a shift on a word of that category is legal. For example, if (4) is at the top of the control stack, we may shift on either Det or N. Suppose we shift an N. The kernel of the new item-set is determined by stepping over the N in any item in which an N follows the dot. In this case, the new kernel is [NP $\rightarrow$ N•]. (Since there are no non-terminals following any dot, this kernel happens to be the entire item-set.)

When the dot is at the end of some rule, a reduction is permitted on that rule. To continue our example, the item-set [NP $\rightarrow$ N•] calls for reduction of N to NP. We pop n elements off both the control stack and the parse-tree stack, where n is the number of children in the recognized rule. In this case, $n = 1$. This brings (4) back to the top of the control stack. Now we build an NP and push it onto the parse-tree stack, and we determine the kernel of the new state by stepping over NP in any items in (4) with an NP after the dot. The new kernel is [VP $\rightarrow$ V NP•, S $\rightarrow$ NP•VP]. We push the corresponding state onto the control stack. The configuration now is:

$$\begin{array}{cc} \text{Control Stack} & \text{Parse Stack} \\ \left[\begin{array}{l} \text{VP} \rightarrow \text{V}\bullet\text{NP} \\ \text{VP} \rightarrow \text{V}\bullet\text{S} \\ \text{NP} \rightarrow \bullet\text{Det N} \\ \text{NP} \rightarrow \bullet\text{N} \\ \text{S} \rightarrow \bullet\text{NP VP} \end{array}\right] \left[\begin{array}{l} \text{VP} \rightarrow \text{V NP}\bullet \\ \text{S} \rightarrow \text{NP}\bullet\text{VP} \\ \text{VP} \rightarrow \bullet\text{V NP} \\ \text{VP} \rightarrow \bullet\text{V S} \end{array}\right] & \text{V NP} \end{array}$$

Now we have a conflict: we may either reduce by rule VP $\rightarrow$ V NP (because of the item VP $\rightarrow$ V NP•), or shift a V (because of the items VP $\rightarrow$ •V NP and VP $\rightarrow$ •V S). In this case, lookahead decides the conflict. We shift if the next word is a V, and reduce if there is no input left. In other cases, lookahead does not resolve the conflict, and we have a genuine next-action conflict. The *LR grammars* are those context-free grammars that do not generate next-action conflicts; they can be parsed deterministically by an LR parser.

3.2 Grammar

In the current implementation, I am using the following (toy) grammar for chunks:

$$
(5)\qquad
\begin{array}{rcl}
\text{PP} & \rightarrow & \text{P DP} \\
\text{DP} & \rightarrow & \text{Predet? D? NP} \\
\text{DP} & \rightarrow & \text{QPPron} \\
\text{NP} & \rightarrow & \text{(Num} \mid \text{QP)? (AP (Comma? Conj? AP)*)? N}^0 \\
\text{AP} & \rightarrow & \text{AdvP? A}^0 \\
\text{QP} & \rightarrow & \text{AdvP? Q} \\
\text{QPPron} & \rightarrow & \text{AdvP? QPron} \\
\text{DegP} & \rightarrow & \text{AdvP? Deg AP} \\
 & \mid & \text{AdvP? Deg AdvP} \\
\text{AdvP} & \rightarrow & \text{Adv? Adv} \\
\text{CP} & \rightarrow & \text{IP} \\
\text{IP} & \rightarrow & \text{(AdvP? Infl)? (VP} \mid \text{AuxP)} \\
\text{AuxP} & \rightarrow & \text{AdvP? Aux (VP} \mid \text{AuxP)} \\
\text{VP} & \rightarrow & \text{AdvP? V} \\
\text{PtcP} & \rightarrow & \text{AdvP? (Ing} \mid \text{En)} \\
 & & \\
\text{N}^0 & \rightarrow & \text{N}^0_{\text{3sg}}\text{* N} \\
\text{N}^0_{\text{3sg}} & \rightarrow & \text{(A} \mid \text{Ing} \mid \text{En} \mid \text{Num} \mid \text{N}_{\text{3sg}}\text{) Hyphen N}_{\text{3sg}} \\
\text{A}^0 & \rightarrow & \text{((N}_{\text{3sg}} \mid \text{Adv) Hyphen)? (A} \mid \text{Ing} \mid \text{En)}
\end{array}
$$

The lexicon includes *'s* and possessive pronouns in category D. Modals and *to* are in category Infl. Certain selectional constraints are imposed, though they are not represented in (5). For example, Aux imposes restrictions on its complement, and we must also guarantee that a DP whose determiner is *'s* does not appear in a PP chunk.

Grammar (5) is obviously incomplete; I present it here mostly for illustrative purposes. However, in its defense, it does contain the most common structures. Even though it represents only a small portion of a complete grammar for chunks, spotchecks of random text samples indicate that it covers most chunks occuring in natural text.

3.3 Non-Determinism in the Chunker

The chunker is a non-deterministic version of the LR parser just described. There are two sources of non-determinism in the chunker. First, the points at which chunks end are not explicitly marked in the word stream, leading to ambiguities involving chunks of different lengths. Second, a given word may belong to more than one category, leading to conflicts in which the chunker does not know e.g. whether to shift the following word onto the stack as an N or as a V. As a result, if we graph the computation path of the chunker on a given input—that is, let each node be a snapshot of the chunker, and each arc be a parsing action—the result is not a line, but a tree. The chunker performs a best-first search through this tree of legal computations.

The aim of using best-first search is to approach deterministic parsing without losing robustness. The success of the Marcus parser and similar deterministic natural-language parsers (e.g., Fidditch: Hindle 1983) gives one cause to believe that a deterministic or near-deterministic parser for English is possible. However, Marcus-style deterministic parsing has two related drawbacks. First, the complexity of grammar development and debugging increases too rapidly. I believe this results partly from the use of a production-rule grammar format, and partly from the fact that grammatical and heuristic information are folded together indiscriminately. Second, if the parser's best initial guess at every choice point leads to a dead end, the parser simply fails. It is much preferable to separate heuristic information from grammatical information, and use a non-deterministic architecture. As heuristics improve, we approach deterministic parsing on non-garden-path sentences. At the same time, sentences that are either genuine garden paths, or garden paths according to our imperfect heuristics, do not cause the parser to fail, but merely to slow down.

Non-determinism is simulated straightforwardly in the chunker. A *configuration* is a snapshot of a computation. From each configuration, there are some number of possible next actions. The chunker builds one *task* for each possible next action. A task is a tuple that includes the current configuration, a next action, and a *score*. A score is an estimate of how likely it is that a given task will lead to the best parse. Tasks are placed in a priority queue according to their score.

For example, suppose we have the simple grammar

Chunk → NP
Chunk → VP
NP → N
VP → V

If the first word in the sentence is *water*, the chunker creates two tasks:

$$(([\,], [\,], 0), [\text{SH } \mathit{water}_{\text{N}}\ [\text{NP} \rightarrow \text{N}\bullet]], s_1)$$

$$(([\,], [\,], 0), [\text{SH } \mathit{water}_{\text{V}}\ [\text{VP} \rightarrow \text{V}\bullet]], s_2)$$

The first element in each task is the current configuration, ([ ], [ ], 0)—i.e., the control and parse stacks are empty and the current word is word 0. The second element is the action to be performed: either shift *water* onto the stack as an N and go to state [NP → N•], or shift *water* onto the stack as a V and go to state [VP → V•]. The final element is the task's score. The two tasks are placed on the queue, with the best task first in the queue.

The chunker's main loop takes the best task from the queue, and makes that task's configuration be the current configuration. It executes the task's next action, producing a new configuration. Then a new set of tasks are computed for the new configuration, and placed on the priority queue, and the cycle repeats.

To continue our example, executing the first task yields configuration ([[NP $\rightarrow$ N•]], [N], 1). There is only one possible next action, [RE NP$\rightarrow$ N], producing a single new task. Assuming its score is better than s_2, the new queue is:

(([[NP $\rightarrow$ N•]], [N], 1), [RE NP$\rightarrow$ N], s_3)

(([ ], [ ], 0), [SH *water*$_V$ [VP $\rightarrow$ V•]], s_2)

The parser will execute the reduction task next.

Scores for tasks are determined by the following factors:

1. Lexical frequencies
2. General category preferences: e.g., prefer present participle to A, prefer N-N modification to adjectival modifier
3. LR-conflict resolution (E.g., prefer shift to reduce)
4. Agreement: disagreement does not produce ungrammaticality, but dispreference

A score is a vector of length 4, one position for each factor. Values range from 0 to negative infinity, and represent log frequency, for the lexical frequency factor, and number of violations, for the other factors. The natural order on scores is a partial order: $s_1 < s_2$ iff $f_i(s_1) < f_i(s_2)$, for every factor $f_1 \ldots f_4$. This partial order can be embedded in a total order by assigning weights to each factor: $s_1 < s_2$ iff $\sum_i w_i f_i(s_1) < \sum_i w_i f_i(s_2)$. The weights are currently assigned arbitrarily, though a method for fixing them empirically is clearly desirable.

As is desirable for best-first search, scores decrease monotonically as the parse proceeds. This guarantees that the first solution found is a best solution. Namely, each task represents a tree of possible computations. By making the scoring function monotonic decreasing, we assure that solutions derivable from some task t have scores no better than t's score. Since the first solution found has a score at least as good as that of any task still on the queue,[5] and every task on the queue has a score at least as good as any solution derivable from it, the first solution found has a score at least as good as that of any other solution.

3.4 Deciding Where a Chunk Ends

There is a problem with deciding where a chunk ends, inasmuch as the ends of chunks, unlike the ends of sentences, are not marked in text. Given that, in general, a single chunk will not cover the entire input, we would like to return the most highly-valued chunk that covers some prefix of the input. A straightforward way to do that is to pretend that every word has an alternate reading

[5] A solution's score is the same as the score of the 'Accept' task that generated it.

as an end-of-input marker. (LR parsers treat end-of-input as a grammatical category, albeit a special one.)

Hallucinating end-of-input markers at every position in the string sounds absurdly expensive, but in fact it is not. One piece of information that we must keep with a task, whether we hallucinate end-of-input marks or not, is which subset of the readings of the lookahead word the task is legal on. For example, suppose we have just shifted the word *many* onto the stack as a Q, and the current configuration is:

(6) [[QP $\rightarrow$ Q•]], [Q], 1

(That is, [QP $\rightarrow$Q•] is the sole LR state on the control stack, Q is the sole category on the parse stack, and the next word in the input is word 1.) The next word is *are*, let us say, which has two readings. It has a very common reading as a verb, and a very rare reading as a noun (a unit of measure of area). There is only one legal next action from configuration (6), namely, Reduce QP $\rightarrow$ Q. (This QP will ultimately be a modifier of the head noun *are*.) However, that reduction is legal only if the next word is a noun. Since the noun reading of *are* is rare, we should disprefer the task T calling for reduction by QP $\rightarrow$ Q. But we only know to disprefer T if we keep track of which subset of readings of the lookahead word T is legal on.

If we keep sets of lookahead readings with each task, we can slip fake end-of-input markers in among those lookahead readings. The only operations we perform which we would not have performed anyway are reductions that are legal *only* if the lookahead is a fake end-of-input marker. If we score such reductions relatively low (that is, if we prefer longer chunks to shorter chunks), it turns out that hallucinating end-of-input markers everywhere causes us to execute only a few tasks that we would not have executed otherwise.

The same technique is used for error recovery in the attacher. If it is not possible to get a parse for the entire sentence, the most highly-valued parse for some prefix of the input is returned. Since sentences often contain structures that were not anticipated in the grammar, and since we want to get as much information as possible even out of sentences we cannot completely parse, error recovery of this sort is very important.

4 Attacher

4.1 Attachment Ambiguities and Lexical Selection

The attacher's main job is dealing with attachment ambiguities. In basic construction, it is identical to the chunker. It simulates a non-deterministic LR parser, using the four heuristic factors given earlier. But in accordance with the importance of attachment ambiguity resolution, the attacher has two additional factors in its scores:

5. Prefer argument attachment, prefer verb attachment

6. Prefer low attachment

These factors are used to rate alternative attachment sites. Finding an attachment as argument is more important than finding an attachment as verb, so potential attachment sites are ranked as follows: attachment as verb argument (best), attachment as argument of non-verb, attachment as verb modifier, attachment as modifier of non-verb. The second factor is relative height of attachment sites, counted as number of sentence (IP) nodes below the attachment site in the rightmost branch of the tree at the time of attachment.

The attacher also has special machinery, in addition to the basic machinery that it shares with the chunker. Unlike the chunker, the attacher must deal with words' selectional properties. (Indeed, the fact that lexically-specified complements are frequently optional is precisely the source of most attachment ambiguities the attacher faces.) The lexical selectional properties of a head determine which phrases can co-occur with that head. A given word has a frameset, that is, a set of subcategorization frames. Each frame contains a list of slots, representing the arguments the head takes. There is a good deal of freedom in the order in which arguments appear, but there are also some constraints. For example, direct objects must appear first, and sentential complements must appear last. The current implementation of the attacher recognizes two positional constraints, namely, 'only appears first' (annotation on slot: '<') or 'only appears last' ('>'). Arguments are also marked as obligatory (no extra annotation), optional ('?'), or iterable ('*'). For example, a typical subcategorization frame would be [DP<?, PP*, CP>], meaning that the word in question takes an optional direct object (which must be the first complement, if it appears at all), any number of PP's, and an obligatory final embedded clause.

In addition to frames, a frameset contains a specification of the adjuncts that can appear with the head in question. A 'fleshed-out' frame includes each of those adjuncts, in addition to the slots explicitly stored with it.

It is possible to convert a set of subcategorization frames into a set of context-free rules of the form XP $\rightarrow$ Y <args&modifiers>. Y is a specific lexical item of category X, and <args&modifiers> is some permutation of some fleshed-out frame of Y that does not violate any of the slot constraints.[6] Where there are n arguments and modifiers, there are 2^n such sequences, less those that violate some constraint. Thus, for a given word with m frames, there correspond $m2^n$ context-free rules, less the ones that violate a constraint. In the worst case, we would have $pm2^n$ such rules, where p is the number of words in the dictionary.

The actual number is much smaller, because words do fortunately group themselves into classes with respect to their subcategorization frames. However, even if the grammar that results from this approach is substantially smaller

[6] Actually, it is more complicated than a simple sequence of categories, because of the presence of iterative categories in frames. However, the complications introduced by iterative categories do not compromise the argument made here, so I ignore them.

than $pm2^n$, it almost certainly represents a larger grammar than is practical. It appears a better space-time tradeoff in this case to process subcategorization frames at run time, not at compile time. For this reason, in addition to LR machinery for handling rules that are insensitive to subcategorization, the attacher also has special facilities for dealing with subcategorization frames at run time.

Those facilities are as follows. Consider a word w with subcategorization frames. When we shift a chunk headed by w onto the stack, we suspend processing and look for complements of w. The general idea is to build a subgrammar on the fly that is looking for any category that could be the first complement of w, and parse that subgrammar. When we finish parsing the subgrammar—that is, when we execute an Accept action—the top node on the stack will be w's first complement. We attach it to w. Then we calculate the set of possible categories for the next complement of w, build another subgrammar, and parse it to get the next complement. We continue in this manner until there are no more potential complements, or until we (non-deterministically) decide to quit collecting complements. Then we resume where we had left off before collecting w's complements.

In more detail, we first calculate frameset-first(f), where f is w's frameset. Frameset-first(f) is the set of categories that can be the first category in some frame in f. For example, if w's frames are [DP<? PP*] and [PP* CP>?], frameset-first(f) = {DP, PP, CP}. We push a new initial LR state onto the stack, of the form [Start $\rightarrow \bullet X_1$, ..., Start $\rightarrow \bullet X_n$], where the X_i are the categories in frameset-first(f). If all frames contain only optional constituents, we also include an item Start $\rightarrow \bullet$. In our example, the new initial state is

$$(7) \quad \begin{bmatrix} \text{Start} \rightarrow \bullet \text{DP} \\ \text{Start} \rightarrow \bullet \text{PP} \\ \text{Start} \rightarrow \bullet \text{CP} \\ \text{Start} \rightarrow \bullet \end{bmatrix}$$

After pushing (7) onto the stack, we continue parsing. When we come to the point of executing an Accept action, the configuration is of form:

$$\begin{array}{llll} \text{control:} & \ldots\text{suspended parse}\ldots & [\text{Start} \rightarrow \bullet X_i] & [\text{Start} \rightarrow X_i \bullet] \\ \text{parse:} & \ldots & w & X_i \end{array}$$

At this point, instead of accepting, we attach X_i to w. That is, we pop X_i and w from the stack, make a copy of w that differs only in having X_i as new rightmost child, and push the copy of w back on the stack. We pop the top two states from the control stack, bringing us back to the configuration we were in before we suspended parsing to collect w's complements. Then we push a new initial state onto the stack that is generated as follows.

First, we calculate frameset-next(f, X_i), that is, a new frameset representing what remains of the frames after the slot of category X_i has been filled. We consider one frame at a time. If there is no slot for X_i in the frame, the frame is removed. If there is a slot, and it is not iterable, it is removed from the frame; if it is iterable, it remains in the frame. After the first slot is filled, all initial slots are removed. If a final slot is filled, the frame becomes empty. In our example, frameset-next(f, PP) = {[PP*], [PP*, CP>?]}, frameset-next(f, DP) = {[PP*]}, and frameset-next(f, CP) = {[ ]}.

After calculating the new frameset f', we build a new initial state from frameset-first(f') as before, and push it on the stack. For example, if we have just attached a DP to w, w's new frameset is [PP*], and the new initial state is [Start $\rightarrow$ •PP, Start $\rightarrow$ •].

When the new frameset contains only empty frames, or if we choose to Close (that is, reduce by Start $\rightarrow e$), we are finished collecting w's complements. Instead of pushing a new initial state on the stack, we resume the parse we had suspended.

Attachment ambiguities show up as Shift-Close conflicts. Suppose we are parsing a sentence of form DP VP DP PP, where the PP may be attached either to the immediately preceding DP, or to the VP. At the point of conflict, the configuration is:

(8) ctrl: ... [IP $\rightarrow$ DP•VP] [Start $\rightarrow$ •DP, •] [Start $\rightarrow$ •PP, •]
parse: VP DP

The conflict is whether to Close the DP, or to Shift the following PP. In general, Close is dispreferred, with the effect that low attachments are preferred. In certain cases, however, a higher attachment is preferred. For example, if VP's frameset permits a PP after the DP, Close is the preferred action from configuration (8), inasmuch as attachment to a verb is preferable to attachment to a noun. In general, to determine whether there is a more preferable high attachment, we need only look back through the stack for initial states. If the node corresponding to an initial state is a legal attachment site, and a more highly valued attachment site, then Close is preferred to Shift.

4.2 Attachment Ambiguities in the Chunker

I have asserted that the extra machinery for dealing with lexical selection and attachment ambiguities is only needed by the attacher. However, there are apparent examples of attachment ambiguity that arise within chunks, and it is important to explain why they do not require the machinery I have developed for the attacher.

For example, noun compounds have the property that any binary tree over the string of nouns is a valid parse, as far as syntactic constraints go. This is a hallmark of attachment ambiguities (cf. Church and Patil, 1982). Also, conjunction of prenominal adjectives can lead to similar ambiguities. However, these cases differ from inter-chunk attachment ambiguities in an important way. The chunker can simply treat noun sequences and adjective conjunction as iterative structures, e.g. [$_{\mathrm{DP}}$ a [$_{\mathrm{NP}}$ [$_{\mathrm{N+}}$ cherry picker exhaust manifold]]], and leave it to the semantics to figure out the interrelationships. (The treatment of noun compounds in grammar (5) above is only slightly more elaborate.) The phrase [$_{\mathrm{N+}}$ cherry picker exhaust manifold] represents the set of possible binary trees over the four nouns, but the ambiguity is a semantic ambiguity; the syntactic representation is unambiguous.

It may appear that the attacher could do the same for e.g. PP attachment. For example, if we are concerned only about VP's of the form V NP PP*, we could assume a flat VP expansion, generating structures like

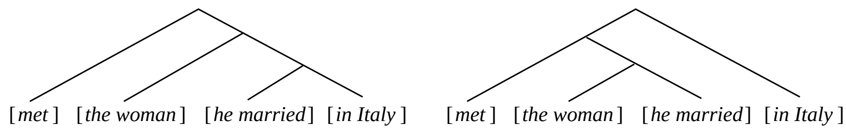
[$_{\mathrm{VP}}$ [place] [the ball] [in the box] [on the table] ...]

Such a structure could be interpreted as representing every possible binary tree over the bracketted phrases (as suggested by Church and Patil, 1982). But unfortunately, the ambiguity cannot be localized to a single VP. Consider *John met the woman he married in Italy.* To avoid a decision on the attachment of *in Italy*, we must assume a structure like

[$_{\mathrm{IP}}$ John [$_{\mathrm{VP}}$ [met] [$_{\mathrm{NP}}$ the woman] [$_{\mathrm{CP}}$ he married] [$_{\mathrm{PP}}$ in Italy]]]

In order to guarantee a syntactically unambiguous, flat structure, we must assume that embedded VP's (*married*, here) expand only to V. If we group [$_{\mathrm{CP}}$ he married] and [$_{\mathrm{PP}}$ in Italy] in the semantics, as in the first tree in (9), we interpret *in Italy* as a modifer of married. If the grouping in the semantics is as in the second tree in (9), we interpret *in Italy* as a modifer of met.

(9)



However, this approach is unsatisfactory, because virtually all chunks following the matrix verb are left unassembled, and a considerable amount of syntactic information that constrains the assembly of those chunks is ignored. That is, unlike with noun compounds, it is not true that every binary tree over the chunks in (9) is syntactically admissable. In particular, the relative clause cannot be a modifier of the verb, and the PP cannot be a modifier of the NP. In effect, by pushing ambiguity out of the syntax and into the semantics, we also end up requiring the semantics to do much of the work of the syntax. Concisely, there is an intermingling of syntactic and semantic constraints in interchunk relations that is not found within chunks.

5 Comparison to Related Models

5.1 The Chunker and Chart Parsing

An issue I have skirted to now is ambiguity in the output of the chunker. There are at least two sorts of ambiguity that arise that cannot be satisfactorily resolved by the heuristics we have discussed. First, it is possible for the same stretch of words to be analyzed as a chunk in more than one category. This arises especially with single-word chunks. For example, *described* may represent either a single-word VP or a single-word PtcP, and the ambiguity can be resolved only in context. In this case, both readings are passed to the attacher.

A more difficult type of ambiguity arises when it is not clear where to end a chunk. Consider the following sentences:

> In Manhattan apartments with low rent are in great demand
>
> In Manhattan apartments with low rent, rats are a serious problem

Neither is a hopeless garden path like "the horse raced past the barn fell," so we would like the parser to be able to handle either. However, it is not possible to decide whether the first PP chunk ends before or after *apartments* using only immediate context. We must pass both possibilities to the attacher, and let it choose based on right context.

As a consequence, the output of the chunker is actually not a stream, properly speaking. Reading from the chunker at a given position yields a set of chunks, along with the positions at which they end. In the attacher, the input pointer for a configuration after a shift is not necessarily one greater than the previous input pointer.

This invites comparison to a chart. One of the advantages cited for a chart is that it does not require that all competing readings in cases of ambiguity cover the same segment of the input string. In effect, the chunker as revised outputs chart edges. I think that is a profitable way of viewing the architecture of the parser I have described. A chart parser introduces a cache point for every partial parse tree constructed, to avoid duplication of effort. Often, though, the added overhead involved in caching and checking caches is greater than the savings from avoiding repeated construction. In the parser I have described, chunks, and only chunks, are "cached"—in the sense that separate branches of the attacher's non-deterministic calculation can use the same chunks, without duplicating the effort of constructing them. This appears to be a good intermediate position between caching all trees, whether they are likely to be reused or not, and caching no trees.

5.2 The Chunker and the Sausage Machine

A brief note is in order comparing the chunker to Frazier and Fodor's Sausage Machine (Frazier and Fodor, 1978). Apart from having two levels, there is actually little similarity between a chunking parser and the Sausage Machine. Processing in both stages of the Sausage Machine are identical, whereas in a chunking parser, only the attacher is powerful enough to deal with lexical selection and attachment ambiguities. The 'chunks' that the first-stage processor builds, in the Sausage Machine, are determined entirely by what fits in an input buffer of arbitrarily-chosen size. In a chunking parser, by contrast, chunks have a detailed syntactic definition, which can be defended on syntactic grounds alone (cf. Abney, to appear). For the same reason, the correspondence between chunks and ϕ-phrases is lacking in the Sausage Machine model. Again, because of the heterogeneity of Sausage Machine 'chunks,' there is no basis for supposing that they constitute particularly good cache points in a nondeterministic parse. (In fact, the Sausage Machine model is deterministic, so the question does not arise.)

In brief, virtually none of the advantages of a chunking parser, which I summarize in the next section, accrue to the 'chunks' produced by the Sausage Machine.

6 Conclusion

By way of conclusion, I would like to reiterate the advantages of a chunking parser. First, one of the most difficult problems for context-free parsing techniques is attachment ambiguities. But within chunks, (syntactic) attachment ambiguities do not arise, and simple context-free parsing techniques are very effective. By having separate chunker and attacher, we can limit the use of expensive techniques for dealing with attachment ambiguities to the parts of grammar where they are really necessary—i.e., in the attacher.

Another motivation is modularity. Since the chunker is insensitive to the state of the attacher, we can develop and debug it separately from the attacher. The chunker also simplifies the task that the attacher faces: many lexical ambiguities can be resolved within chunks, relieving the attacher of that task, and there is less clutter to deal with at the level of chunks than at the level of words.

A related motivation is that the chunker-attacher division keeps attachment ambiguities from being multiplied with chunk ambiguities. The chunker evaluates chunks as well as it can on its own, instead of making decisions relative to one or another branch of the attacher's non-deterministic computation.

As we have seen, there is also some psychological evidence for chunks. Gee and Grosjean argue that the 'performance structures' that emerge from a range of diverse experiments are best predicted by what they call ϕ-phrases. Apart from their structure—that is, seen simply as strings—chunks and ϕ-phrases are nearly identical.

A fifth motivation is related to Gee and Grosjean's work. They show that ϕ-phrases are a good predictor of intonation. Given the correspondence between ϕ-phrases and chunks, there is a possibility of using the chunker in determining intonation, for speech synthesis.

And last, but not least, we can account for a range of otherwise inexplicable syntactic constraints if we assume the existence of chunks. For example, we can explain why **the proud of his son man* is odd, by observing that it involves a chunk, *of his son*, embedded in another chunk, *the proud man.* (See Abney, to appear.) If the chunks are produced in a stream, it is not possible to interleave them.

References

1. Abney, Steven (1987) *The English Noun Phrase in Its Sentential Aspect*, unpublished doctoral dissertation, MIT, Cambridge, MA.

2. Abney, Steven (to appear) "Syntactic Affixation and Performance Structures," in Denis Bouchard and Katherine Leffel, eds., *Views on Phrase Structure*, Kluwer.

3. Aho, Alfred V., and Jeffrey D. Ullman (1972) *The Theory of Parsing, Translation, and Compiling*, in two volumes, Prentice-Hall, Inc., Englewood Cliffs, NJ.

4. Chomsky, Noam (1986) *Barriers*, MIT Press, Cambridge, MA.

5. Church, Kenneth, and Ramesh Patil (1982) "Coping with Syntactic Ambiguity or How to Put the Block in the Box on the Table," *American Journal of Computation Linguistics*, 8.3-4, 139-149.

6. Frazier, L., and J.D. Fodor (1978) "The Sausage Machine: A new two-stage parsing model," *Cognition* 6, 291-325.

7. Gee, James Paul, and François Grosjean (1983) "Performance Structures: A Psycholinguistic and Linguistic Appraisal," *Cognitive Psychology* 15, 411-458.

8. Hindle, Donald (1983) "User manual for Fidditch, a deterministic parser," Naval Research Laboratory Technical Memorandum #7590-142.

9. Knuth, D.E. (1965) "On the translation of languages from left to right," *Information and Control* 8.6, 607-639.

10. Williams, George E. (1986) "The Solar Cycle in Precambrian Time," *Scientific American* 255.2, 88-97, New York, NY.