Artificial Intelligence and Language Processing

Peter Friedland Editor

A Finite and Real-Time Processor for Natural Language

People process natural language in real time and with very limited short-term memories. This article describes a computational architecture for syntactic performance that also requires fixed finite resources.

Glenn David Blank

Natural languages are amazingly versatile, but not infinitely so. I start with the premise that syntactic performance requires only fixed finite resources. I have found a performance processor that needs only a static state space. The processor operates in $O(n)$ or real time. This is a significant improvement on the $O(n^3)$ time available for context-free languages [10, 25, 32]. Efficient performance also depends on holding down grammar size. The processor presented here represents syntactic versatility without incurring combinatorial redundancy in the number of transitions or rules. It avoids both excess grammar size and excessive computational complexity.

I view the purpose of syntax as chiefly to expedite and simplify the processing of semantic and discourse structures. Excessive computational complexity obscures this function. Some researchers have advocated ignoring syntax altogether. But syntax always shows up somewhere, if not in a module that observes generalities efficiently, then in redundancies spread throughout a lexicon or linguistic knowledge base. Instead, I believe that the fixed finite resources hypothesis is a practical design principle for syntactic engines.

Most modern syntactic formalisms, calling for unbounded resources, are prone to intractability [1]. The designers of such formalisms accept Chomsky's argument that finite automata cannot model the *competence* of native speakers. A generation ago in this publication, Woods cited Chomsky to justify the unbridled recursion and manifold tests and registers of Augmented Transition Networks (ATNs). These facilities make his scheme "equivalent to a Turing machine in power," because "the actions which it performs are 'natural' ones for the analysis of language [34, p. 600].

Nevertheless, there are well-known limitations on human capabilities for processing syntax. Miller and Chomsky recognized "the obvious fact that [speakers and hearers] are limited finite devices" [22, p. 465]. There is a small but long-standing tradition of work modeling syntax with finite resources, for embedding [9, 18, 19, 27, 35] and also for keeping track of ambiguity [11, 20, 23]. One model [8] tries to be finite in both of these dimensions.[1] Exploiting these performance limitations leads to a more psychologically realistic and computationally efficient model of language processing.

Let me emphasize this point. Evidence that suggests that syntactic competence requires more than context-free power does not necessarily rule out performance with finite resources. A performance model may explicitly include *memory limitations* as part of its design. It is entirely a question of whether syntactic performance ever calls for unbounded resources. For example, Pullum [24] affirms Chomsky's view that natural languages cannot be regular, citing evidence from Central Sudanic languages where center-embedding is apparently more common and acceptable. The empirical question is, do speakers of Sudanic languages generate sentences with unboundedly or even unusually deep center-embedding? Bresnan et al. [5] claim that cross-serial dependencies in Dutch argue against describing this language in terms of context-free rules. The empirical question is, can Dutch speakers generate unboundedly or even unusually wide cross-serial dependencies? If not, then the fixed finite resources hypothesis is tenable. If one can count up the dependencies on one's fingers, than surely one can model them with finite resources. Indeed, Pullum does acknowledge that hearers may indeed process sentences "as if they were finite automata" [24, p. 114].

Register Vector Grammars (RVG) are equivalent to finite state automata (FSA). Implied in the name are

[1] Church's model is not a fixed finite system, however: it expands and shrinks as a function of input. He uses a heuristic to prevent it from ever actually requiring unbounded resources.

two major innovations that allow RVG to be far more efficient and compact than simple FSA, with respect to natural languages. First the *vectors*. Simple FSA represent states and categories as simple symbols—nodes and arcs in transition diagrams. Most modern syntactic formalisms abstract over category-symbols, with nonterminals and tree structures. RVG instead abstracts over state-symbols—with vectors of ternary-valued features. As we shall see, this technique helps to eliminate a great deal of redundancy from grammars. It keeps grammar size quite small.

Second are the *registers*, which keep track of alternative states. RVG is able to guarantee linear time because it pre-allocates only a small number of these registers. The processor supports reanalysis of structural ambiguities by backtracking to system states in these registers. However, the number of registers never grows; instead, RVG reuses them, thus systematically forgetting many (but not all) ambiguities. This approach puts a tight leash on nondeterminism; it also mimics human behavior.

This article begins by introducing the RVG architecture, comparing it first with its cousin, the nondeterministic finite automaton, and eventually with more computationally complex formalisms. This is followed by a description of how to rein in nondeterminism with a small number of registers, and an outline of planned enhancements to the basic architecture, with respect to handling subcategorization, agreement, conjunction, and canonical structures. The article will conclude by presenting an analysis of complexity demonstrating that the algorithm is indeed linear, and empirical results for small fragments of English.

REGISTER VECTOR GRAMMAR

As the fixed finite resources hypothesis implies, RVG is equivalent to a nondeterministic FSA. The RVG automaton is a 5-tuple (S, C, I, F, T), where S is a finite set of states, C is a finite set of input symbols (called categories), I is the initial state, F is a set of final states, and T is a transition relation mapping $S \times C$ to S. This definition is of course the same as that of a nondeterministic FSA. The difference lies in the nature of RVG's states and transition relation.

Ternary Feature Vectors

The states and transition relation of RVG are represented in terms of vectors of ternary-valued features. A feature vector f is denoted by the features $(f_1 \ldots f_n)$ Each feature may take on one of three possible values; "+", "−", or "?" ("on," "off" or "don't care"). A particular grammar involves vectors of some fixed length, k. If k is 9, then a state vector might look like this: ++?−−?−+.

The transition relation of RVG relies on two ternary vector operators. Given two vectors of ternary features, the **match** operator produces a Boolean result:

$$\text{match}(f_i, g_i) = \begin{cases}\text{TRUE} & \text{if } f_i = g_i \text{ or } f_i = ? \text{ or } g_i = ? \\ \text{FALSE} & \text{otherwise}\end{cases}$$

$$\text{match}(f, g) = \begin{cases}\text{TRUE} & \text{if match}(f_i, g_i) = \text{TRUE for all } i \\ \text{FALSE} & \text{otherwise.}\end{cases}$$

For example, **match** (+ − ?, + ? ?) is TRUE, but **match** (+ + +, + ? −) is FALSE, since a + opposes a − in the this position.

The **change** operator takes two vectors and produces a third. Definite values (+ or −) in the second vector override corresponding values in the first vector, but indefinite values (?) have no effect. Formally:

$$\text{change}(f_i, g_i) = \begin{cases} g_i & \text{if } g_i = + \text{ or } g_i = - \\ f_i & \text{if } g_i = ? \end{cases}$$

$$\text{change}(f, g) = (\text{change}(f_i, g_i))$$

For example, **change** (− + ? − + ? − + ?, − − − + + + ? ? ?) produces − − − + + + − + ?. Note that the **change** operator is *asymmetric*: the output vector gets the definite values (+ or −) of the second argument. Where the second argument has indefinite values (?), though, the values of the first argument "pass through."

RVG's transition relation is implemented by three data structures:

- A table of productions (the *Synindex*). Each production in the Synindex is a 5-tuple ⟨cat, cond, change, lexflag, actions⟩, where *cat* is a symbol, *cond* and *change* are ternary vectors, *lexflag* is a character differentiating different types of productions, and *actions* is a list of executable functions. A lexflag with value **I** designates one production as *InitFinal*. The initial state **I** is InitFinal's change vector. The set of final states F are just those that match InitFinal's cond vector.
- A list of lexical entries (the *Lexicon*). Each lexical entry is a 2-tuple ⟨morph, lexcat⟩, where *morph* is its morphological description and *lexcat* is a list of category labels.
- The current syntactic state register (*SynState*). The SynState holds a ternary-valued vector, which the transition function matches and updates.

We can now define RVG's transition relation. Given S, a set of state vectors, C, a set of category symbols (in the lexicon), and *Synindex*, a set of productions, there is a relation **R** on $S \times C$ to S. Relation ⟨SynState, lexcat, SynState⟩ is a member of **R** if there is a production ⟨cat, cond, change, lexflag, actions⟩ such that:

1) **match**(cond, SynState) is TRUE
2) cat = lexcat
3) SynState ⇐ **change**(SynState, change)

Simple Rigid and Free Order Languages

Figure 1 shows a RVG for a simple Subject-Verb-Object (SVO) language:

Each feature in this grammar is associated with a position in the left to right order of categories. The first feature, **S**, associates with the subject constituent position, feature **V** with the verb position, and **O** with the

object. Thus features can control the occurrence of categories. (They need not necessarily have to do with particular categories or positions, as we shall see.)

Here is how the above grammar recognizes **George loves Martha**. (The two vectors on each line are the state before and after each production fires.)

Word	SynState	Commentary
	+++	Initialize with CLOSE's change (**Init**Final)
George	+++ −++	Production SUBJ fires: match (+++, +??) → TRUE (matches **+S** change (+++, ??) → −++ (new SynState with **−S**)
loves	−++ −−+	VERB's cond requires **+V**; its change is **−V**.
Martha	−−+ −−−	OBJ's cond requires **+O**; its change is **−O**.
•	−−− +++	CLOSE's cond requires **−S** & **−V** (Init**Final**)

Figure 2 shows a simple nondeterministic finite state automaton (FSA) which is equivalent to the above RVG Synindex. The five nodes in this diagram are equivalent to the five states reachable by the grammar of Figure 1. Note that in the simple FSA representation, to represent the optionality of OBJ, we need to draw two CLOSE arcs. RVG avoids this redundancy because ternary values provide explicitly for optionality. The *cond* of CLOSE, −−?, matches either −−+ or −−−.

Another example better demonstrates the expressiveness of vectors. Suppose we want to model a partially *free-order* language. It allows SUBJ to occur freely, but requires VERB to appear before OBJ. Figure 3 illustrates the RVG version (with no change to the lexicon of Figure 1). The only difference between Figures 1 and 3 is that the latter relaxes one constraint. The *cond* of VERB which had **−S** now has **?S**. Relaxing **S** allows VERB to occur freely with respect to SUBJ.

Compare the above with the equivalent FSA diagram shown in Figure 4.

The simple FSA notation leads to much redundancy for this language. For example, the diagram shows explicitly that SUBJ may occur first, or follow VERB, or follow both VERB and OBJ. Though the number of constituents is the same, the number of transitions have multiplied. The RVG, on the other hand, relaxes just one constraint; the number of productions does not change. A completely free-order language (still obligating SUBJ and VERB) requires 22 transitions in diagram form. The equivalent RVG just relaxes another constraint, changing OBJ's *cond* from **−V** to **?V**.

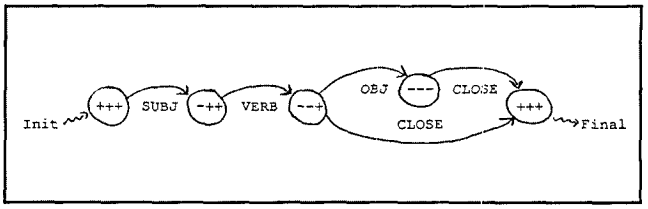

FIGURE 2. Simple FSA for SVO Language

The RVG format is thus more compact than that of simple FSA. In this case, it is better at expressing syntactic obligations, regardless of order. In all of the grammars above, SUBJ and VERB must occur before CLOSE can fire. In both Figures 1 and 3, the cond of CLOSE requires −−?. Only the change vector of SUBJ and VERB can turn off the first two features. We call CLOSE a *terminator* category; it acts as a gate through which only acceptable utterances may pass.

In general, RVG is able to represent a potentially huge number of states and transitions with a remarkably compact transition table (Synindex). RVG constrains the number of potentially reachable states by means of a particular configuration of features in the table. So a rigid-order grammar (Figure 1) has fewer reachable states than a free-order one (Figure 3); but grammar size is the same. For each category there are many FSA transitions but just one RVG production.

Simple FSA are unable to pass information from state to state, except via simple linear precedence. Individual transitions represent local constraints; such as that SUBJ must precede VERB. It is difficult to express discontinuous, non-local constraints. The only recourse is to plot separate paths through the transition network for each non-local constraint. Interaction of constraints soon leads to an explosion in the size of the grammar.

In RVG the state, a vector, keeps a record of some non-local context. Note that this does not add any computational power to FSA. Only the match and change operators are different. Instead of insisting on exact identity, RVG allows partial matching with "?" values. Instead of wholesale substitution, RVG allows partial change. This modification allows feature values to "pass through" productions to subsequent states—a form of constraint propagation. There is thus no need to multiply transitions through intervening states, as in simple FSA. Technically, any RVG is equivalent to some sim-

Synindex

cat	cond	change
SUBJ	+??	-??
VERB	-+?	?-?
OBJ	?-+	??-
CLOSE	--?	+++ InitFinal

Feature Key

Position		Label
1	-	S
2	-	V
3	-	O

Lexicon

word	catlist
George	SUBJ OBJ
Martha	SUBJ OBJ
loves	VERB
	CLOSE

FIGURE 1. RVG for SVO Language

cat	cond	change	Position		Label
SUBJ	+??	-??	1	-	S
VERB	?+?	?-?	2	-	V
OBJ	?-+	??-	3	-	O
CLOSE	--?	+++ InitFinal			

FIGURE 3. RVG for Partially Free-Order Language

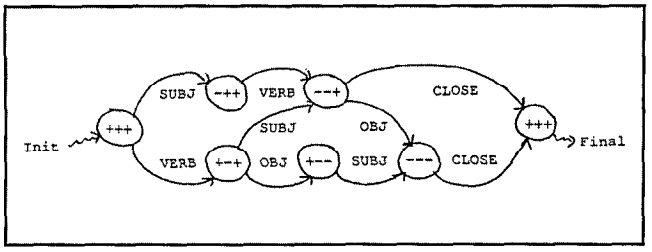

FIGURE 4. FSA for Partially Free-Order Language

ple FSA. But if that FSA must allow for optionality and non-local constraints, it will be much larger.

RVG throws out sentence diagrams. States are no longer simple nodes, and rules are not longer rote patterns. State vectors abstract over state nodes. To be sure, the vectors of a Synindex are equivalent to some vast diagram, but for a natural language, the diagram is enormous and tangled with redundancies. Other formalisms introduce computationally expensive devices to supplement and untangle the diagrams; RVG simply abandons diagrams altogether. The only complexity it adds is the functional complexity of its match and change operators.

Think of grammars as telephone switchboards or computer chips or neural networks. An RVG is able to schedule a great variety of events with a compact, dense matrix of productions and features.

Notation for an RVG Assembler

The feature keys shown in Figures 1 and 3 are not used by the RVG processor itself. Feature vectors are actually in binary form (two bits per ternary feature), and ternary vector operations are composed of fast bitwise logic operators. Since match and change operate on all features at once, we can exploit the bit vector parallelism that already exists on conventional computers.

Feature keys do make it easier to describe vectors, though. The current RVG development environment therefore allows one to create, edit and debug RVGs using a symbolic notation, which is translated into efficient RVG "machine" code.[2] In the following figures, the **ordering_features** section is a feature key, associating labels with feature key. In Figure 5, the *cond* vector of production SUBJ is **+S − HEAD**, which translates to +???− ("?" is the default feature value). A range notation allows one to set all the features in a vector segment to a value: **+S..O** translates to the binary equivalent of +++.

The assembler also provides for macro substitution. The macros section associates a symbol prefixed by "#" with a vector (in symbolic form). For example in Figure 5 macro **#NPOn** expands to the vector **+DET..HEAD**. Macros may then appear in lieu of feature specifications in the grammar. This is a simple way to provide for generalizations, such as the features required to open or close a phrase. The assembler is left-to-right, so it is possible to override values in ranges or macros.

Non-lexical Productions and Phrase Embedding

A lexicon is a collection of entries, each of which hold syntactic, morphological and (eventually) semantic information. The syntactic information is a set of categories, corresponding to productions in the Synindex.

A left-to-right RVG recognizer accepts a word when it finds a production whose *cat* corresponds to one of the word's categories and whose *cond* matches the SynState register; it then advances its state by updating SynState with that production's *change* and consuming the word from the input string. An RVG generator is the same with respect to matching and updating SynState, instead buffering words to an output string.

It is useful to allow non-lexical productions, which advance SynState but do not consume words from input (or buffer words to output). Each production's *lexflag* notes this distinction. "L" for lexical productions and "N" for non-lexical productions.[3]

Figure 5 demonstrates the use of a lexicon and non-lexical categories to process simple noun phrases (NPs).

RVG treats NP embedding as a special case of discontinuity. Non-lexical productions SUBJ and OBJ turn on features **DET** and **HEAD**. Feature **HEAD** acts as a switch, distinguishing clausal from phrasal productions. All of the causal productions (SUBJ through CLOSE) must wait until HEAD is off again—a discontinuous constraint. Meanwhile, the phrasal productions (NAME, DET and NOUN) may occur at various positions in clause structure, because they *ignore* the features pertaining to clauses. E.g., NAME has **?S ?V ?O** in both its *cond* and *change*. The *change* of NAME or NOUN turns **HEAD** off again, re-enabling the clausal productions.

This effect is not so readily achieved by simple FSA. It is tempting to allow the equivalent of SUBJ and OBJ to branch to the same set of transitions for NPs. The

```
ordering_features  S V O DET HEAD
macros  #NPOn +DET..HEAD  #NPOff -DET..HEAD
productions
p SUBJ  N  cond +S -HEAD       change -S #NPOn
p VERB  L  cond -S +V -HEAD    change -V
p OBJ   N  cond -V +O -HEAD    change -O #NPOn
p CLOSE I  cond -S -V -HEAD    change +S..O #NPOff
{Phrasal productions follow}
p DET   L  cond #NPOn          change -DET
p NOUN  L  cond +HEAD          change #NPOff
p NAME  L  cond #NPOn          change #NPOff

entries  {Lexicon}
e George cat NAME  e Martha cat NAME   e loves  cat VERB
e the    cat DET   e quiche cat NOUN   e .      cat CLOSE
```

FIGURE 5. RVG for Embedding Noun Phrases

[2] Our translator is analogous to an assembler, inasmuch as it preserves a close correspondence to the "machine" code. Reed [26] describes a compiler that translates rules in an extended context-free notation into a combination of RVG vectors and context-free rules. While this notation may appear more perspicuous to those trained in the phrase structure paradigm, it is incompatible with the fixed finite resources hypothesis (and its full benefits). Nor does it encourage users to understand and effectively exploit the nature of RVG's expressiveness.

[3] The default is "L". Also, the InitFinal production, flagged "I", is always lexical.

problem is, where should the network branch at the end of the phrase—to the node past SUBJ or the one past OBJ? A simple FSA loses track of where it left off. The only recourse is to multiply transitions for NPs at each possible position where they may occur! Another alternative is to introduce *recursive subnetworks*—push the state of the clause level network onto a stack, then traverse a separate phrase level subnetwork. But RVG is able to model phrase embedding without increasing computational complexity. One segment of the state vector shows the identical topology for all NPs.

Note that this topology is part of the same flat vector. It is thus straightforward to model free order languages that allow scrambling of phrasal elements. For example, in Warlpiri the head and modifier of a phrase may be widely separated in clause structure [10]. Such languages use case-marking inflections to identify the role of loose modifiers. In languages like English or Latin, phrase level features screen out clause level productions, as in Figure 5. A phrase-terminating production updates the SynState to close the noun phrase for a particular role. In radically non-configurational languages such as Warlpiri, phrasal modifiers may recur by selectively ignoring constraints. A phrase-terminating production just signals that it has seen some role. Loose modifiers are non-local options.

The advantage of non-lexical productions is that they reduce redundancy of lexical categories, and thereby make grammars more compact. A grammar with non-lexical productions is equivalent to a larger one without them. To eliminate non-lexicals, merge each one with each of the lexical productions that can follow it. For example, the following lexical productions could replace non-lexical production SUBJ in Figure 5:

```
p SUBJ_DET    L
  cond +S #NPOff   change −S − DET
p SUBJ_NOUN L
  cond +S #NPOff   change −S
p SUBJ_NAME L
  cond +S #NPOff   change −S
```

Each production takes the *cond* of SUBJ and composes a *change* vector from SUBJ and the production that could follow it—DET, NOUN and NAME. Since RVG allows multiple non-lexical productions between lexical productions, the equivalent grammar would have to combine all allowable sequences.[4] Non-lexicals thus eliminate a great deal of redundancy in the grammar.

Note that simple FSA could not use empty categories to mimic the compact RVG treatment of NP embedding. Empty categories do not add any capacity to record or propagate non-local constraints.

WH-QUESTIONS

A notorious example of discontinuous constraints is *wh-questions*. These begin with **wh**-words (or phrases) such as **who** or **what**, and require that somewhere in the sentence there be one missing noun phrase (a gap or trace):

(1) Who loves Pamela?
    (gap for subject)
(2) Who does Pamela love?
    (gap for object)
(3) Who do the men think that Pamela loves?
    (gap in complement clause)
(4) Who does George love Pamela?
    (ungrammatical: no gap)

RVG models this dependency easily, as shown in Figure 6.

Production WH turns on feature **+GAP**. Production CLOSE's *cond*, which determines possible final states, requires **−GAP**. So, somewhere between WH and CLOSE, some production must turn this feature off: a discontinuous constraint. The only production which can do so is NGAP. Its *cond* recognizes **+GAP**, and its *change* turns it off. Most other productions have **?GAP** in both *cond* and *change* vectors, so this constraint simply passes through any intervening productions. Thus ternary vector functionality makes constraint propagation very efficient. Note also that vectors allow for multiple constraints: the *cond* of NGAP also requires **#NPOn**, the segment of features that open a noun phrase, so that this production can fire only when a noun phrase is possible *and* a gap is possible. The *change* of NGAP disables both feature **GAP** and all the features for a noun phrase.

Here is how the grammar of Figure 6 recognizes sentences (1) through (3):

(1) WH:who; SUBJ:NGAP:VERB:love; OBJ:NAME:pamela; CLOSE:?;
(2) WH:who; QUES:do; SUBJ:NAME:pamela; VERB:love; OBJ:NGAP:CLOSE:?;
(3) WH:who; QUES:do; SUBJ:DET:the; NOUN:men; VERB:think; CTHAT:that; SUBJ:Name: pamela; VERB:love; OBJ:NGAP:CLOSE:?;

```
ordering_features   S V O AUX GAP DET HEAD
macros  #NPOn +DET..HEAD  #NPOff -DET..HEAD
productions
p SUBJ  N cond +S #NPOff          change -S #NPOn
p VERB  L cond -S +V #NPOff       change -V -AUX
p OBJ   N cond -V +O #NPOff       change -O #NPOn
p CTHAT N cond -V +O #NPOff       change +S..AUX #NPOff
p CLOSE I cond -S -V -GAP #NPOff change +S..AUX -GAP..HEAD
{Productions for Wh-questions follow}
p WH    L cond +S -GAP #NPOff     change -GAP
p QUES  L cond +S +AUX #NPOff     change -AUX
p NGAP  N cond +GAP #NPOn         change #NPOff -GAP
p DET   L cond +DET               change -DET
p NOUN  L cond +HEAD              change #NPOff
p NAME  L cond #NPOn              change #NPOff

{A simple lexicon for wh-questions--morphology not shown}
entries
e Pamela cat NAME   e man    cat NOUN   e quiche cat NOUN
e love   cat VERB   e think  cat VERB   e eat    cat VERB
e .      cat CLOSE  e ?      cat CLOSE
e the    cat DET    e that   cat DET CTHAT REL {ambiguous}
e do     cat QUES   e who    cat WH REL
```

FIGURE 6. RVG for WH-Questions

[4] The processor guarantees a finite number of states by requiring that a non-lexical fire at most once between lexical productions. See the section on Subcategorization.

The last category before each word is a lexical production; e.g., WH before *who*, VERB before *love*, etc. Any categories preceding the lexical category are non-lexical; e.g., SUBJ and NGAP before VERB.

Implementation of unbounded dependencies is particularly elegant. The complement clause, introduced by production CTHAT, right-embeds—reusing the same state register. The change vector of CTHAT prepares State for a new clause: **+S..AUX#NPOff**. It defaults to **?GAP**. Therefore the value of feature **GAP** passes from the matrix clause right on through to the right-embedded complement clause. In sentence (3), production NGAP fires after OBJ in the complement. As usual, NGAP must still fire before CLOSE can. No complicated gap-passing mechanism is necessary.

Adding more possibilities for noun phrase gaps is just a matter of introducing the corresponding possibilities for noun phrases. Allowing for prepositional wh-questions (e.g., *In which box did the robot put the hammer*?) is a matter of an additional constraint, which will interact with different productions.

Embedding

Clause embedding was Chomsky's [7] primary evidence against finite automata for processing natural languages. Human performance for center-embedding is, however, severely limited. For example:

(4) The mouse the cat chased squeaked.
(5) The mouse the cat the dog bit chased squeaked.

Embedding object relatives once is not unusual, but twice is boggling. Miller and Chomsky [22] acknowledge that such limitations indicate that the human sentence processing mechanism must indeed be finite. (Grammatical competence, which abstracts away from such non-linguistic considerations as stammering or memory limitations, is non-finite.)

RVG allows shallow center-embedding with a tri-leveled state register:

```
                    SynState
ClauseLevel ⟶ Main clause vector
              1st embedded vector
              2nd embedded vector
```

The SynState register remains finite. Only now it has ordered levels as well as features. RVG shifts from level to level by changing the value of the index, *ClauseLevel*. A pair of actions manage clause-shifting: *ShiftDown* to increment ClauseLevel and *ReturnUp* to decrement it.

The tri-leveled SynState register is similar to a processing model of Cowper [9]. As in Cowper's model, RVG prefers to iterate at the same level whenever possible. Note that there is no limit on left- or right-embedding:

(6) My mother's girl friend's husband's car broke down.
(7) I saw a dog that chased a cat that caught a mouse that ate some cheese.

Neither left-embedding of genitive phrases, as in (6), nor right-embedding of complement clauses as in (7), need invoke clause-shifting. Once the *obligatory* constituents of a clause—subject, predicate and complementizer in (4)—have appeared, the processor can right-embed. RVG, like a simple FSA, puts no limit on edge embeddings (iteration).

The tri-leveled SynState does not increase complexity in any significant way. There is an equivalent single-level SynState model. Recall that a segment of the SynState vector manages NPs. Similarly, three segments of one long, flat vector could manage three clause levels. Shifting from clause to clause, by opening and closing windows on relevant segments, would then no longer require any special actions. The drawback of a flat vector technique is that, for each level, the grammar must replicate most productions, with different *cond* and *change* corresponding to each clause segment. E.g., VERB0 would match and change feature **V0** in the main clause segment. VERB1 would match and change **V1** in the first embedded segment, etc. The clause-shifting mechanism simply eliminates this redundancy.

Figure 7 demonstrates center- and right-embedding of relative clauses. It is a modification of the grammar of Figure 6 (using the same lexicon).

Figure 7 distinguishes two ways to introduce noun phrase post-modifiers. Productions MODC and MODR both set up a new clause—with #ClauseOn in their *change* vectors. MODC center-embeds, by invoking action Shiftdown, so that its change applies to the next clause level. MODR, on the other hand, right-embeds—that is, it simply reuses the current clause level. Another difference is that MODC turns on feature **MTERM**, which forces MODEND to fire eventually, and invoke action ReturnUp. Here is a sample parse:

(8) Men who hate men that eat quiche love pizza.
SUBJ:NOUN:men; MODC:REL:who;
SUBJ:NGAP:VERB:hate; OBJ:NOUN:men;
MODR:REL:that; SUBJ:NGAP:VERB:eat;
OBJ:NOUN:quiche; NPEND:MODEND:VERB:love;
OBJ:NOUN:pizza; CLOSE:.;

The first relative clause center-embeds (MODC), since

```
ordering_features  S V O AUX GAP DET HEAD NTERM REL MTERM
macros  #NPOn +DET..NTERM  #NPOff -DET..NTERM
    #ClauseOn +S..AUX -GAP..MTERM  #ClauseOff -S..MTERM ?O
p SUBJ   N cond +S #NPOff          change -S #NPOn
p VERB   L cond -S +V #NPOff       change -V -AUX
p OBJ    N cond -V +O #NPOff       change -O #NPOn
p CTHAT  L cond -V +O #NPOff       change +S..AUX #NPOff
p CLOSE  I cond #ClauseOff         change #ClauseOn
p WH     L cond +S -GAP -REL #NPOff change +GAP
p QUES   L cond +S +AUX #NPOff     change -AUX -REL
p NGAP   N cond +GAP #NPOn         change -GAP #NPOff
p DET    L cond #NPOn              change -DET
p NOUN   L cond +HEAD..NTERM       change -DET..HEAD
p NAME   L cond #NPOn              change #NPOff
{NP post-modifiers start a new clause, with result vector.}
p MODC   N cond +NTERM +V -REL     change #ClauseOn +REL +MTERM
         action ShiftDown {center-embed}
p MODR   N cond +NTERM -V          change #ClauseOn +REL
p REL    L cond +S -GAP +REL #NPOff change +GAP  {Similar to WH}
{New terminators: NPEND for phrases, MODEND for post-modifiers}
p NPEND  N cond +NTERM             change -NTERM
p MODEND N cond #ClauseOff +MTERM  change -REL -NTERM
         action ReturnUp
```

FIGURE 7. Adding Relative Clauses

it occurs to the left of the predicate, whereas the second one right-embeds (MODR), since it occurs to the right of the predicate. When MODC fires, it invokes action ShiftDown and turns feature MTERM on. This feature value passes right through MODR, and eventually triggers MODEND, which invokes action ReturnUp.

Global constraints on movement in other formalisms have direct and simple implementation in RVG. For example, consider Ross's Complex NP Constraint [28], which rules out sentences like this:

(9) *What have you met the man who invented?

Though there are two *wh*-words in this sentence, one cannot fill both gaps in the relative clause—the missing subject and object of **invented**. The grammar of Figure 7 models this constraint in terms of feature **GAP**. MODR's *cond* requires that **GAP** be off. In other words, **GAP** is obligatory at each clause level. Since MODC invokes ShiftDown, its *change* applies to a different feature **GAP**, at a lower clause level.

In many dialects of English, sentence (10) is grammatical but (11) is not:

(10) Who does George believe saw Martha?
(11) *Who does George believe that saw Martha?

That is, subject gaps may not appear after the complementizer **that**. One way to model this constraint, variously called the Empty Subject Filter or Sentential Subject Constraint, is with an additional feature, **NOSUBJGAP**. Production CTHAT's *change* turns on **NOSUBJGAP**. NGAP's *cond* requires **−NOSUBJGAP**, so that it cannot fire after SUBJ, in this context. Finally, VERB's *change* arbitrarily turns **NOSUBJGAP** off again, thus enabling NGAP after OBJ (or any other phrase position).

RVG represents non-local constraints (alias "movement" or "extraposition") the same way as local ones: in terms of features whose values propagate through state vectors. For example, one may *extrapose* the object to the front of a sentence, as in **Quiche George loves!** Simply add one production to Figure 7:

p TOPIC N cond +S #NPOff change −O #NPOn

The *cond* of TOPIC is the same as that of SUBJ, allowing it occur at the beginning of the sentence. The *change* is the same as that of OBJ, enabling a noun phrase and—here is the crucial part—disabling a subsequent OBJ. The latter is a discontinuous constraint.[5]

Comparison with other Syntactic Formalisms

Context-free phrase structure rules are almost as poor as FSA at handling non-local constraints. The proliferation of state-symbols that riddles simple FSA also plagues phrase structure (PS) rules. The patterns on the right-hand sides of PS rules inherit an inability to pass information from state-symbol to state-symbol, other than via convoluted linear precedence. Transition networks and rule patterns are good at linear constraints but poor at non-linear ones. Chomsky [7] noted the inadequacy of PS rules to describe the versatility of natural language syntax without loss of generality. He therefore introduced Transformational Grammar, relegating PS rules to a base that generates only kernel sentences. RVG eliminates PS rules altogether.

Recognizing the problems with PS rules, proponents of Generalized Phrase Structure Grammar (GPSG) [13] have developed a more abstract format. By separating immediate dominance (ID) from linear precedence (LP) information, they get a more succinct representation of syntax, especially of free order languages. ID rules specify the obligatory constituents in a *local tree*; this is roughly analogous to how RVG models obligatory requirements in terminator categories like CLOSE. Though ID/LP itself is succinct, it does little to alleviate the non-local constraint propagation problem. The GPSG scheme still generates an "object grammar" with large local sets of PS rules. The more varied the linear sequences of a language, the larger the object grammar. Estimates of the number of rules in an object grammar for a natural language range as high as trillions or more [1, 30]. As Shieber [30] points out, grammar size does affect processing complexity; efficient traditional context-free algorithms are $O(n^3 \mid G \mid^2)$, where G is the size of the grammar. He outlines an algorithm that can parse using the much smaller number of ID/LP rules, directly. However, Barton et al. [1, pp. 191ff] show that Shieber's algorithm is exponential in grammar size, due to expansion of ID rules. RVG avoids this problem by eschewing expansion of rules (and recursive expansion of rules) altogether.

Many modern syntactic formalisms have in common the paradigm of unification. Basically, unification is an algorithm for matching graph structures (usually directed acyclic graphs). Computational linguists use this technique to write grammars that involve propagating constraints through syntactic trees in the state space. Thus unification can be a way to tackle the non-local constraint propagation problem. However, this adds complexity on top of context free power. Features "percolate" up and then "drip" down trees; moreover, most formalisms add additional global filters or principles to govern this activity.[6] Whereas unification matches complex graph structures, RVG just matches flat vectors. Whereas unification is monotonic (once a variable is bound it remains bound), RVG allows changes. Unification ensures that category-symbols (and features under them) are recoverable. Since recoverability is not an issue with state-symbols, the full power of the change operator can be exploited. RVG has an unabashedly left-to-right bias. It passes constraints through a changing state vector, and is willing to reuse a state register rather than expand the fixed state space.

[5] A more efficient scheme for topicalization would delay the decision about whether the first noun phrase is SUBJ or TOPIC until the end of the phrase. In other words, let productions SUBJ, TOPIC (and for that matter OBJ) fire after noun phrases, in order to avoid reanalysis.

[6] For example, the Head-Feature Principle of GPSG. Barton et al. [1] show how this succinct but powerful mechanism can lead to computational intractability.

ATNs use transition diagrams chiefly to represent *local* ordering. Discontinuous constraints typically involve setting, passing and testing myriad separate registers. This is what gives ATNs Turing machine power. Without restrictions on the use of registers, guaranteeing even polynomial recognition is impossible. The ATN treatment of wh-questions is especially difficult, typically passing a "hold" list from network to network. Reining in such a powerful mechanism, in accordance with "movement" constraints, is even harder.

It is interesting that Woods [34] actually anticipated something very much like RVG:

> In the absurd extreme, it is possible to reduce any transition network to a one-state network by using a flag for each arc and placing conditions on the arcs which forbid them to be followed unless one of the flags for a possible immediately preceding arc has been set. The obvious inefficiency here is that at every step it would be necessary to consider each arc of the network and apply a complicated test to determine whether the arc can be followed.

Indeed, RVG might look like such a "one-state" network, since it throws out the diagrams. Nevertheless, an RVG processor does pass through many states, as feature values change. Woods simply did not see the possibility of decomposing the state(-symbol) itself. Instead of both arcs and flags, RVG uses just features to forbid or permit transitions. Instead of an indefinite number of registers, RVG collects all constraints on order in just one current state register. Instead of the "complicated test" that worries Woods, RVG applies an efficient ternary vector match operation.

Since human syntactic performance does not call for unbounded embedding, it is possible to avoid even context-free power. Instead of the computational complexity of a push-down automaton, RVG adds the functional complexity of ternary vector match and change. Instead of abstract, graph-structured categories, RVG has abstract, vectorized states. Its features are features of ordering, abstracted over the state-nodes in transition diagrams, or the states *between* category symbols in rule patterns. RVG's states are thus able to be sensitive to context—non-local constraints—without being "context-sensitive" in the automata-theoretic sense.[7]

BOUNDARY BACKTRACKING

Structural ambiguity is a crucial problem for natural language processing. It is typically treated as a nondeterministic search problem, testing various structural alternatives until accepting one (or more) interpretation. Common methods of dealing with ambiguity backtracking, pseudo-parallelism or chart parsing—put no bound on the search space. Unbounded search is, however, computationally wasteful and cognitively unrealistic.

[7] *Petri nets* anticipate the possibility of automata that straddle the traditional Chomsky hierarchy. RVGs are strongly equivalent to safe Petri nets, which are in turn weakly equivalent to FSA (see [16] for details).

The fact that RVG is equivalent to FSA does *not* necessarily solve this problem. A nondeterministic FSA still involves a search through a search space that grows, in the worst case, exponentially. There are well-known methods to convert any nondeterministic FSA into a deterministic one. This conversion may, however, explode grammar (machine) size exponentially. Because RVG already eliminates combinatorial redundancies by explicitly allowing for optionality and non-local constraints, conversion to simple deterministic FSA is neither trivial nor desirable.

Instead, RVG tolerates nondeterminism, by putting strict limits on it. The technique is *boundary backtracking.*

RVG watches each major syntactic boundary (such as opening or closing a clause or phrase). At each boundary-crossing, it stores its system state in an associated register. If the processor ever gets stuck, it can backtrack to a previous system state—but only to one actually held in a register. There is an array of boundary registers. The fixed size of this array puts a cap on the growth of the state-space for nondeterministic search.

Each register associates with a major boundary in left to right syntax. Here is a set (not necessarily definitive) of boundaries:

```
Curr        (the current state)
Word        (when a word is processed)
OpenClause  (when a new clause opens)
MidClause   (when about to process main predicate)
CloseClause (when clause right-embedding is possible)
Phrase      (when a NP opens, or closes left of
               predicate)
```

Since RVG allows center-embedding only to a finite depth, it maintains **OpenClause** through **Phrase** by ClauseLevel. The processor automatically adds the clause level number to the boundary name at run time: i.e., **OpenClause0**, **OpenClause1**, **OpenClause2**, **MidClause0**, etc.

The Algorithm

If RVG were completely nondeterministic, its algorithm would be almost the same as that of a nondeterministic FSA. The only difference is ternary vector match and change. Boundary backtracking is a control schema for nondeterministic search within a fixed state space. It has three basic aspects:

1) *Local parallelism.* From word to word, the processor tries all syntactic interpretations. It searches depth-first, looking for all possible sequences of non-lexical productions leading to a lexical production for a given word.

2) *Saving states.* The grammar must explicitly specify when to save states in boundary registers. Associated with a few productions are *save* actions. For example, production OPENC invokes action **save OpenClause**. A save action puts the current system

state with the boundary name or a temporary SaveList. The processor copies these states into the named boundary registers when it reaches a lexical category. In preparation for the next word, it also automatically updates the **Curr** and **Word** registers.

3) *Backtracking.* Access to boundary registers is Last In First Out, mediated by a separate array of boundary subscripts, *Resume.* The processor tries to continue from states in boundary registers until no more are left. Initially, and after each word, it finds the **Curr** register. If this state fails (no production matches), or if a production successfully completes a sentence, the processor gets the next available state via *Resume.* Thus it tries to report all possible interpretations. The limit on this behavior is that *save* actions reuse state registers.[8]

Examples

The phenomenon of *garden-path* sentences suggests that there are severe limits in human performance for keeping track of ambiguities. For example:

(12) The horse raced past the barn fell.

People do not readily understand this sentence, even though there is a perfectly grammatical interpretation. The problem is the category ambiguity of **raced**. It could be either intransitive (serving as main predicate) or transitive (serving as a passive post-modifer of horse). Most people prefer the intransitive reading; they are thus "led down the garden path," and cannot recognize the correct interpretation.

When the RVG processor crosses the same boundary twice, it reuses the associated register. It therefore loses any state already held in that register. This allows the state space to remain bounded; it can also lead to garden-path effects. Suppose that **race** has three syntactic categories, NOUN, VINTRANS and VTRANS. The processor chooses the first one that fits the current context. In this case, because of the verbal inflection of **raced**, it skips NOUN in favor of VINTRANS.

Between **horse** and **raced** the processor goes through two boundary productions. The first production, closing a noun phrase to the left of the predicate, triggers action **save Phrase**. Before accepting the main predicate, another production triggers action **save MidClause**. After these, lexical production VITRANS fires. Before processing to the next word, the processor automatically saves states from the current SaveList into boundary registers. In this case there are three: **Phrase0**, **MidClause0** and **Word**.

After the processor accepts the preposition **past**, and starts another noun phrase, it again saves in register **Phrase0**. It therefore loses the state previously stored in this register—the state closing the noun phrase between **horse** and **raced**. So, when the processor gets to fell, and backtracks, it will not find the passive post-modifier interpretation of **raced** anywhere. Note that the post-modifier interpretation is not available in **MidClause0**, since by then the processor had already closed off the noun phrase.

Contrast the processor's behavior for this sentence:

(13) The horse found by the barn fell.

Here, **found** is also ambiguous, between active or passive transitive. Choosing the active interpretation first, the processor closes off the NP, and saves states in **Phrase0** and **MidClause0**. This path gets stuck when it reaches **by**, where it expects a direct object. So the processor backtracks. In this case, there has been no intervening NP. The processor finds the passive interpretation of **found** in register **Phrase0**, and successfully analyzes it as a post-modifier.

Sentence length does not necessarily limit boundary backtracking. (Thus BB contrasts with the Sausage Machine of [8].) Hence it has no problems with sentences like these:

(14) Have the boys take the exam.
(15) Have the boys taken the exam?
(16) Have the boys who have a reputation for playing hookey taken the exam?

To be sure, there may be backtracking. Suppose **have** has categories QUES and IMP. The processor will recognize sentence (14) directly. For sentences (15) and (16), it will backtrack to the **OpenClause0** boundary, reanalyze **taken** as IMP, then accept the rest of the sentence directly.

Local parallelism

The boundary backtracking algorithm, notwithstanding the name, actually combines aspects of backtracking and parallel search. Both use bounded resources. Local processing (of words) is parallel and exhaustive; nonlocal processing (of constituents) is serial and preferential. In accordance with recent psycholinguistic research [31], search for lexical entries is parallel and exhaustive. In addition, search for all local syntactic interpretations—any non-lexical productions leading to a lexical production for a given word—is also parallel and exhaustive. This search for non-lexical interpretations is called *local parallelism.* The bound on local parallelism is related to the number of non-lexical productions in the grammar; in practice, it is quite small. Nonlocal processing is serial. When necessary, it backtracks to states held in boundary registers. The bound on nonlocal processing is the number of registers. (It is nonlocal processing that would otherwise lead to a combinatorial explosion in state space.)

The following examples illustrate the role of local parallelism:

(17) Is the block sitting on the table?
(18) Is the block sitting on the table red?

At first glance sentence (18) looks very similar to (12), the garden-path sentence. Again, the processor must choose between either a main predicate or a post-modifier interpretation of a verb. The difference is,

[8] In order to guarantee that *Resume* never exceeds its fixed limit, the processor throws out any duplicate subscripts before storing a new one.

raced is ambiguous between two distinct categories (intransitive vs. transitive), whereas **sitting** involves just one category. The ambiguity of (18) is just a matter of non-lexical interpretation. The processor must determine if **sitting** is the main verb, as in (17), or a post-modifier, as it turns out in (18). The processor looks for all possible local non-lexical interpretations for a given lexical category, and saves all boundaries traversed along the way. Between **block** and **sitting**, the main predicate reading of **sitting** saves states in **Phrase0** and **MidClause0**. In parallel, the post-modifier reading of **sitting** saves a state in **MidClause1**. When the processor reaches **red**, backtracking is able to resume from register **MidClause1**.

The difference between sentences (12) and (18) is subtle. The garden-path sentence involves a tensed intransitive verb (the first lexical category of **raced**), which cannot function as a post-modifier. It crosses boundaries only at the main clause level—**Phrase0** and **MidClause0**. Sentence (18), on the other hand, involves a progressive verb (**sitting**), which can function as either a main verb or a post-modifier. It crosses boundaries at both the main and first embedded clause levels, in parallel. Thus it is possible to resume sentence (18), but not garden-path sentences like (12).

This model affects the way one categorizes verbs. Consider another example:

(19) Was the book read to the children interesting?

Many dictionaries list **read** as either transitive or intransitive. Why, then, doesn't sentence (19) lead to a garden path? The answer is that **read** is a transitive verb with an optional truncation of its object (like **eat**, **cook**, etc.). On this analysis, **read** does not actually have an intransitive category; instead it has just one major category, plus a subcategory that options a missing, implied object. (Subcategorization is discussed in an upcoming section.)

An advantage of local parallelism is that it could without much difficulty be modified to accommodate non-syntactic criteria. The syntactic component makes predictions about preferences by the order of productions—analogous to arc-ordering in ATNs [33]. We have seen that **race** prefers intransitive before transitive; a grammar of English models the effect of minimal attachment [11, 15] by ordering a phrase-closing production before productions that open post-modifiers. Category preferences are not the whole story, of course; semantics or intonation can bias the attachment preferences as well. In the right context, even garden-path sentences become comprehensible. For example:

(20) There were two horses in the field. The horse raced past the barn fell.

RVG provides for actions associated with productions, already used to shift clause level and save states. Other actions may perform tests on semantic structure to provide on-line guidance to a parser or generator. Though syntax is autonomous in the sense that it has its own data structures, there can be interaction between syntax and semantics during processing, especially near phrase and clause boundaries (see [2, 6, 12]). In an interactive processor, while trying local attachments in parallel, syntax proposes and semantics disposes. Suppose we hook a referential semantics module up with RVG. Let production DEFEND, which fires just before closing a definite noun phrase, invoke referential analysis for the phrase. If there is an unambiguous referent for the description so far, reference succeeds. (In the *null* context, reference succeeds, finding unambiguously nothing.) For the context of (20), reference for **the horse** fails, because there are two equally plausible referents. DEFEND fails, so the processor looks for the postmodifier interpretation instead.

Boundary Backtracking vs. Bounded Lookahead

Boundary backtracking is an alternative to the *bounded lookahead* scheme of Marcus [16]. His processor also posits limited resources for resolving structural ambiguity. It builds up partial constituents in a lookahead buffer having a small, fixed number of cells. When the buffer gets full, the processor must commit itself to an interpretation; it allows no backtracking. By this account, garden-path sentences occur when the processor is forced to make such a commitment, incorrectly.

As Church [8, p. 57] notes, "in some sense, [bounded] backup, lookahead and parallelism are all very similar." Whereas unbounded backtracking saves *all* choice alternatives at every choice point, bounded backtracking would presumably fix the size of the backtracking stack. It is not clear, however, that there is any arbitrary limit that would both account for garden-path sentences and still process the temporarily ambiguous sentences that people *do* understand.

Boundary backtracking, on the other hand, does not rely on arbitrary bounds on artificial data structures (such as a stack or a buffer). Instead, it is motivated by perceptual boundaries in syntax. There is much psycholinguistic evidence for such boundaries in human sentence processing. Click studies [12] and eye-fixation studies [6] indicate that reading comprehension activity increases at phrase and clause boundaries. Structurally ambiguous units retard processing time up to clause-closing boundaries, beyond which they do not [2].

Marcus' model makes no provision for noticing legitimate structural ambiguities, e.g., **They are flying planes**. In a footnote, Marcus does suggest that his processor could flag any output that is potentially ambiguous, but "some external mechanism will then be needed to force the interpreter to reparse the input, taking a different analysis path" [20, p. 13]. Instead of ruling out nondeterminism, boundary backtracking reins it in. It remains sensitive to some (but not all) ambiguity. So it discovers the second interpretation of **They are flying planes** after it returns to the **MidClause0** register. Boundary backtracking does forget many ambiguities, as in garden-path sentences like (12). Whether humans retain just those ambiguities that the boundary backtracking model does is an interesting

empirical question. The computational import is that there will be far fewer possibilities for a processor to consider.

Some readers may object that a model that simply rejects garden-path sentences is too stringent. Indeed, Marcus notes that native speakers can understand garden-path sentences with conscious effort. "A higher level problem solver uses a set of grammatical heuristics . . . to discover where the processor went astray" [20, p. 205]. As with globally ambiguous sentences, Marcus' solution smacks of a *homunculus*. The "external mechanism" or "higher level problem solver" is far more powerful than the sentence processor itself, since it is able to "force" the processor to behave differently. Church [8] suggests adding an ad hoc horse-racing rule to the grammar.

The boundary backtracking model suggests a simple and general heuristic. Suppose we allocate one more register, **Extra**. The processor can then accept a garden-path sentence by saving, in **Extra**, a state from another boundary register—just as it is about to be reused. For example, when the processor fails to recognize sentence (12), it can start over, only this time copying the contents from **Phrase0** to **Extra** before it gets reused. It can then backtrack to **Extra** as it would to a state in any other boundary register. This strategy adds no significant complexity to the algorithm. It has the virtue of allowing the processor's coverage to degrade gradually, in a manner similar to human performance [8].

Finally, the boundary backtracking algorithm is reversible. Boundaries support re-starts for natural language generation as well as parsing. When speakers stammer or rephrase their speech, they appear to resume at boundaries [29]. Boundary registers can help a computational generator by providing a small number of definite states at which failed attempts may resume.

Adjunction

RVG's approach to right-embedding, discussed the previous section on wh-questions, predicts the awkwardness of sentences like these:

(21) Mary sang a song that she had learned in Europe before the war to her children.

(22) I called the guy who smashed my brand new car a rotten driver.

Sentence (21) *right-embeds* upon reaching the complementizer **that**. By then, all obligatory constituents in the matrix clause have appeared. Later, the processor cannot adjoin **to her children** to the matrix clause. This policy is similar to Kimball's principle of early closure [15], or Cowper's "Poker Principle" [9]—once the obligatory cards are on the table, one cannot pick them up again.

Though sentences (21) and (22) are awkward, they are certainly comprehensible. Cowper notes that the lost clause "must be retrieved from some kind of less immediate memory storage in order for the last constituent to be added [9, p. 46]. She does not explain the nature of this auxiliary storage. It is, I suggest, a boundary register. When the processor right-embeds, it also saves its state in register **CloseClause0**. If necessary, it can retrieve this state later. This behavior is similar to *right association* [8, 11]. The RVG processor prefers to attach material to the current clause, but if necessary can *explicitly resume* a clause from a **CloseClause** register, in order to add a late adjunct or optional argument. Explicit resuming is a failure-driven strategy. Unlike backtracking, it does not actually return to a prior position in the input stream. Instead, it processes the current word with an earlier syntactic state—at the end of an earlier matrix clause.

PLANNED IMPROVEMENTS

Thus far this article has presented a basic architecture for syntax as a scheduling system, ordering sequences of events in time. (Indeed, it is a general-purpose automata, with applicability for scheduling any comparably complex sequence of events [3].) There are of course many other issues for syntax, which is only part of natural language as a whole.

Subcategorization

The current scheme only provides for major categories. Subcategories *could* be represented by multiplying major categories, but it is useful to avoid such redundancies. For example, the verb **put** requires a locative phrase—one cannot say ***I put it**. We would rather not have to introduce distinctions between all the various kinds of verbs and their complements at the major category level.

A better approach is to factor out subcategories, as non-lexical productions that precede the major lexical productions. In other words, subcategory productions will be semi-lexical. Like non-lexical productions, subcategories do not consume (or produce) words; nevertheless, lexical entries must be able to specify their subcategories. Each lexical entry will have a category set (implemented as a bit vector): the set of productions that may fire up to this word. For example, the lexical entry for **put** will include in its category set (among other things) a non-lexical subcategory LOCREQ. This production fires before the lexical production for the verb. The change vector of LOCREQ turns on a feature in the state register, +LOC. The clause terminator production, CLOSE, requires that this feature be turned off, which only a production recognizing a locative phrase can do.

Adding category sets to the lexicon constrains non-lexicals generally. For example, phrase-opening productions require words that could legitimately open phrases—determiners, adjectives, nouns, etc., but not tensed verbs, prepositions, etc. Specifying constraints in this form can fine-tune the description and performance of grammars. Category sets also provide a way to guarantee that a non-lexical production will fire at most once between words. This guarantee eliminates the possibility of infinite cycles between words, and makes local parallelism tractable.

Agreement

RVG is unusual in that features control precedence relations. In most other formalisms, features enforce agreement, e.g., determiner and head noun must agree in number. RVG *could* include agreement features in its state vectors. DET_PL turns on a feature **+PL** which rules out NOUN_SG, etc. This would lead to some redundancy among productions, though, multiplying every combination of productions by every possible way they might agree. A better approach is to manage agreement features separately. In addition to ordering vectors, each state register configuration includes morphosyntactic agreement vectors. Generalized actions combine agreement values from inflections and lexical entries with values in the agreement vector. Such actions cut across syntactic categories and thus avoid duplication. So long as the agreement register is finite and reusable, there is no significant increase of complexity. (The current architecture already provides for an efficient treatment of inflectional morphology and idioms [17]).

Conjunction

Boundary backtracking suggests a systematic way to handle conjunction that avoids a great deal of redundancy. Conjuncts typically attach at boundaries:

(23) Joe loves Sue and Bob.
Word

(24) Joe loves Sue and her husband.
Phrase0

(25) Joe loves Sue but dislikes Bob.
MidClause0

(26) Joe loves Sue but he dislikes Bob.
Clause0

(27) Joe loves Sue and Bob Martha.
Clause0

As with adjunctions, processing conjunctions may involve trying to explicitly resume from states available in boundary registers. Sentence (25), for example, resumes from the State stored in the MidClause0 boundary, just before *loves*. To be sure, conjunction may involve more than resuming a state. As sentence (27) suggests, a conjunction production should also option possibilities for ellipsis—in terms of ordering features.

Structures

The current system produces as output a linear trace of the productions fired for each possible recognition of a sentence. Building diagnostic traces does not add any significant complexity to the recognition algorithm. It simply concatenates a record for each production fired to the trace. Boundary registers maintain a copy of a current trace as part of a state configuration. The fixed finite resources hypothesis obviously restricts the kinds of diagnostic structures RVG can possibly build. It cannot build indefinitely center-embedded tree structures, nor can it keep track of all conceivable parses for ambiguous sentences. These restrictions are not onerous, though.

While helpful for grammar development, linear traces are not grammar structures, nor are they intended as such. Rather, actions associated with productions can do the work necessary to support interpretation. The current model already provides for actions to handle center-embedding and saving states in boundaries. To these we are adding a repertoire of actions for compositional interpretation.

The notion of using actions associated with productions to support interpretation is a common one, found for example in ATNs. The problem here is to limit the complexity of these actions. The *modus operandi* with respect to actions is that they, too, abide by the fixed finite resources hypothesis. The syntactic categories and ordering positions of natural languages are closed classes. That is why RVG can schedule productions with a small grammar and a fixed finite register configuration. Similarly, affixes, pronouns and grammatical roles are also closed classes. This suggests that agreement, anaphora and predicate-argument calculus are also susceptible to processing by a small repertoire of generalized actions and a fixed finite register configuration. While long-term memory is presumably quite large and not necessarily real time in response (that's why people need external memory aids), short-term memory is quite limited and thus real time. The key to real-time sentence processing is determining the structure of this short-term register configuration.

Our approach draws on ideas from other researchers who have sought to limit computational performance capability to finite state. Church [8, p. 66] dealt with the problem of keeping track of ambiguous prepositional phrases. Rather than maintaining separate interpretations for each possible attachment (which could lead to an exponential growth in state space), Church advocated a pseudo-attachment strategy. To quote his thesis: "YAP has a marked rule to pseudo-attach (attach both ways) when it sees both alternatives and it cannot decide which is correct." Martin et al. [21, p. 279] apply this idea to several notoriously ambiguous constructs, including reduced relatives, conjunction and noun-noun modification. "The approach taken here is to flatten the syntactic structure of these phrases. . . . In this way, the parser will not waste time trying all possible bracketings; it will be content with a canonical one that represents them all." Basically, this amounts to a strategy of least commitment: if there is a locally unresolvable ambiguity, then allow an alternative category that explicitly leaves things undetermined. This attitude is crucial in a processor with restricted resources. To take a simple example, the words **the** and **sheep** are ambiguous or undetermined, depending on how one looks at them, with respect to number. If they are ambiguous, the processor must consider singular and plural possibilities separately. If they are undetermined, on the other hand, it does not have to decide. The processor generates just one canonical representation, which it may refine subsequently.

ANALYSIS OF COMPLEXITY

What is the time complexity of RVG? There are two considerations: 1) the size of the processor and 2) the size of the grammar. The RVG performance model makes important improvements in both of these dimensions: processor size is fixed as a small number of boundary registers, and grammar size is held down by ternary vector functionality.

Processor size is the number of boundary registers. Grammar size is the number of categories, c, which must be considered at each state. Factoring out c, it can be shown that boundary backtracking is linear with respect to input.

Consider the simple case where just a single boundary register allows backtracking to grammatical boundary B. In a sentence of n words (call them $w_1, w_2, \ldots, w_n$), B can be crossed from 0 to n times. Suppose B is crossed exactly once, at arbitrary word w_i:

$$w_1 w_2 \ldots w_{i-1} w_i^B \ldots w_n$$

Words w_1 through w_{i-1} will be considered just once (since no states are saved for backtracking prior to word w_i). Words w_i through w_n can be considered, in the worst case, c times, where c is the number of alternatives stored in the boundary register associated with B. The worst case occurs when, for each of the first $c - 1$ alternatives in B, the recognizer goes all the way to the last word of the sentence (w_n) before failing and backtracking to word w_i:

$$T = t(w_1 \ldots w_{i-1}) + c^*t(w_i \ldots w_n) \leq c^*t(w_1 \ldots w_n)$$

T is $O(n)$, or proportional to the length of the input string, whatever c might be.

Now consider the general case, where B is crossed x times, $0 \leq x \leq n$. Here is where the policy of reuse comes in, every time B is crossed. Let $w_{i1}, w_{i2} \ldots, w_{ix}$, $(1 \leq i1 < i2 < \ldots < ix < n)$ be the words at which B is crossed:

$$w_1 \ldots w_{i1-1} w_{i1}^B \ldots w_{i2-1} w_{i2}^B \ldots w_{ix}^B \ldots w_n$$

Again, words w_1 through w_{i1-1} are considered just once, since no state is saved prior to the first crossing of B. Words w_{i1} through w_{i2-1} may be considered, in the worst case, $c1$ times, where $c1$ is the number of alternatives saved in B. The worst case occurs when, for each of the $c1 - 1$ interpretations, w_{i1} through w_{i2-1} are considered before a failure occurs (at $i2 - 1$). The processor must then backtrack to word w_{i1} in order to get the next interpretation. Note that words w_{i1} through w_{i2-1} can be considered no more than $c1$ times. As soon as w_{i2} is accepted, backtracking to any word prior to w_{i2} will be impossible, since a new State will then be saved in B, "forgetting" any state from before w_{i2}. Similarly, w_{i2} through w_{i3-1} are considered, in the worst case, $c2$ times (where $c2$ is the number of alternatives saved at word w_{i2}). And so on through words w_{ix} to w_n, which can be considered no more than cx times. Thus, the maximum recognition time is:

$$T = t(w_1 \ldots w_{i1-1}) + c1^*t(w_{i1} \ldots w_{i2-1}) + c2^*t(w_{i2} \ldots w_{i3-1}) + \ldots + cx^*t(w_{ix} \ldots w_n) \leq c'^*t(w_1 \ldots w_n), \quad \text{where } c' = \max\{c1, c2, \ldots, cx\}$$

When only a single boundary register is used, recognition time is $O(n)$.

The case of **b** boundary registers is a further generalization of this result. Registers BR_1 through BR_b (ordered as some permutation of BR_1, BR_2, BR_3, ..., BR_{b-1}, depending on the order in which the associated boundaries are crossed) each appear in the Resume array at most once. For each of $c1$ alternatives saved in BR_1 (the first register pushed into the Resume array), the recognizer may consider at most $c2$ interpretations saved in BR_2 (the second register pushed into Resume). Similarly, for each interpretation in BR_2, the recognizer can consider at most $c3$ interpretations saved in BR_3, and so forth. Thus, the maximum number of times any word w_i may be considered when using **b** boundary registers is $c1^*c2^*\mathbf{c}3^* \ldots {}^*\mathbf{cb}$. For a sentence of n words recognized with **b** boundary registers:

$$T \leq cn = O(n), \quad \text{where } c = c1c2^* \ldots {}^* \mathbf{cb}$$

Therefore, with **b** boundary registers, recognition time is still $O(n)$.

The significance of this result is that it avoids the potentially exponential blowup associated with simple unbounded backtracking, which never "forgets" any states. Note that local parallelism is quite tractable (so long as grammar size is), but global parallelism is not, at least not for arbitrarily long, complicated sentences.

The second dimension is grammar size, which also has a significant impact on processing time [1]. RVG is efficient with respect to grammar size in two respects. First, it eliminates a great deal of redundancy found in formalisms that overly commit to strict linear precedence. Thus the number of productions is comparatively small. As it should be: the syntactic categories of a language are a closed class. With p productions (each with two vectors) and f features, an RVG of size p^*2^*f can represent a state space with at least f^f FSA transitions. (The number of FSA transitions may be even greater if there are iterative productions, with no corresponding increase in size for the equivalent RVG.) Second, constraint propagation through state vectors sharply reduces the size of the reachable state space.[9]

Here is how Reed [25, p. 38] describes the efficiency of RVG:

> The vector of one of the RVG states may absorb, representationally, many additional productions. Thus, the RVG-based parser offers the efficiency of being less sensitive to the branching factors of grammars.

Holding down branching factors is a direct consequence of holding down grammar size.

[9] Similarly, Barton et al. [1] show how constraint propagation could improve the performance of the Kimmo morphological analyzer.

What about the number of features? The number of ordering features f should also be small: the left-to-right ordering positions of a language are another closed class. We may estimate f as the sum of $L + N$, where L is the number of local constituent positions and N is the number of non-local constraints. L has to do with the linear precedence of categories; e.g. in English, subject appears before predicate, quantifiers precede adjectives, etc. Note that L is somewhat smaller than the number of categories, since many categories share the same temporal position. For example, many categories may appear in the predicate position—verbs, prepositions, adjectives, etc. Instead of many state-nodes, there is just one feature representing this abstract position. N has to do the non-local constraints between constituents, such as gapping and constraints on gapping. Note that it is possible to eliminate redundancy here as well. We have seen, for example, that the Complex NP Constraint requires no extra features. Both L and N are thus reasonably small, so f will be as well. Crucially, there is no combinatorial redundancy here.

EMPIRICAL RESULTS

I have conducted computational experiments to quantify the efficiency of the RVG algorithm. The design of these experiments is similar to those of Tomita [32, chap. 6] and Reed [25].[10]

The experiments investigate the relationship of parsing times to sentence length, sentence ambiguity and grammar size. Parsing time, for Tomita, is CPU time minus time for garbage collection. Measuring time in this manner, though of interest, is problematic, since much depends on CPU capabilities and details of implementation. Moreover, garbage collection is a significant factor. The RVG architecture generates no garbage, whereas the Earley, Tomita, and Reed algorithms have considerable memory overhead. (To get polynomial time they keep track of all partial constituents in memory.) Therefore, following Reed [25], I also measured *abstract machine time*—the number of state changes. Grammar size and memory requirements are also instructive in evaluating efficiency as well as learnability of architectures.

Two experiments compared th RVG formalism with efficiency context-free parsers. The first one involves a grammar and sentnece set ($1 \leq n$) obtained by the following schema:

```
      det  noun verb det  noun
E.g., The robot saw  a    cat  in the park

      (prep det noun)^(n-1)
      with  a  telescope ...
```

Tomita showed that parsing time for this algorithm grows *polynomially* with respect to sentence length. I predicted linear growth for RVG with boundary backtracking. The first experiment confirms this prediction, both in terms of CPU and abstract machine time (see Figures 8 and 9). The second experiment involved implementing an RVG grammar with the same coverage as Tomita's Grammar III, for which he provides a testbed of 40 sentences (Appendix G). These sentences, taken from computer science textbooks, average 11.2 words in length, and involve a variety of syntactic structures: relative and reduced relative clauses, gerunds, infnitive and **that** complements, sentential, conjuncts, noun-noun modification, etc. This experiment further corroborates RVG's efficiency (see Figures 10 and 11). In terms of CPU time, RVG is performing about an order of magnitude faster than Tomita's algorithm—in compiled C on an IBM AT versus interpreted Lisp on a DEC-20. In terms of abstract machine time, RVG is performing one to two orders of magnitude faster (based on statistics from [25].

RVG gains efficiency by a) ruling out spurious ambiguities by constraint propagation and b) losing potential interpretations to reuse. The RVG model weeds out many structurally plausible but uninteresting interpretations. Indeed, of the forty sentences in Tomita's testbed, only one is reported as ambiguous: **Time flies like an arrow**. The parser gets the proverb in which **time** is a noun as well as an analysis in which **time** is an imperative verb (by backtracking to the **OpenClause** boundary). It does not report interpretations in which **time** is a denominal or adjective modifying **flies**. These possibilities *are* available locally, while in the noun phrase, but are lost by the time the parser reaches the end of the sentence. Getting the *right* interpretation will, to be sure, require integration of syntax and semantics. Semantics must choose the preferred interpretation while it is available, locally.

Perhaps as significant as its efficiency in time is that the RVG algorithm pre-allocates all memory resources, statically. This eliminates the overhead of dynamic memory allocation and garbage collection. Polynomial algorithms such as Earley's and Tomita's have a high memory overhead, maintaining a chart or graph for all possible interpretations. Hence the RVG algorithm looks especially promising for real-time applications running in relatively small machines.

RVG eschews transition diagrams and phrase structure rules, as well as the trees which become great forests in Tomita's algorithm. It is significant that Tomita's grammar badly overgenerates, ignoring issues of agreement and gapping. These are of course difficult to express in terms of simple context-free rules, but straightforward in RVG. Tomita's grammar has about 220 rules, while the RVG version has 60 rules, each with two vectors of 30 features. An important prediction is that whereas context-free grammars will get enormous as coverage increases, the size of an RVG will grow very slowly. Other formalisms trade more computational complexity for less grammar size.

Both the features and productions of RVG are susceptible to parallelism. The current RVG processor, running on conventional computers, already exploits feature parallelism. It sees all the features in a vector at

[10] Reed's work is an amalgamation of RVG's constraint propagation technique [4] and Earley's algorithm [10]. Like Earley's, Reed's algorithm is $O(n^3)$; it does not exploit short-term memory limitations of human performance.

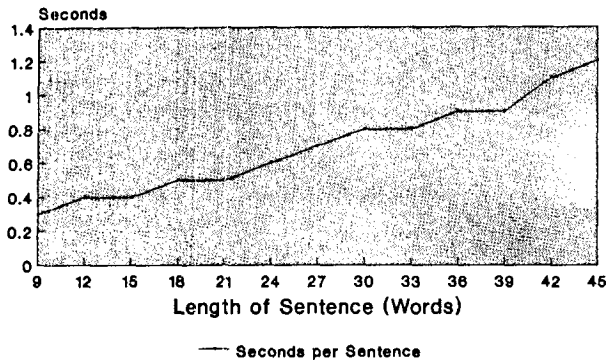

FIGURE 8. PP Attachment—CPU Time

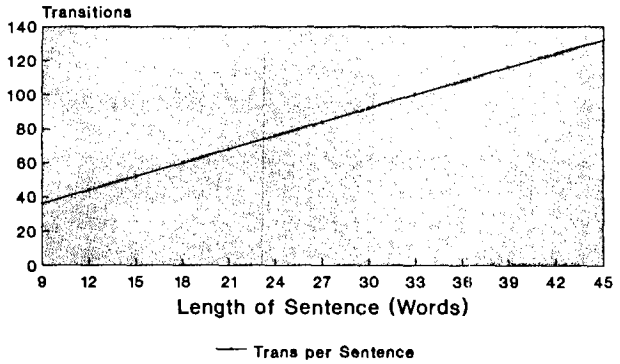

FIGURE 9. PP Attachment—Abstract Machine Time

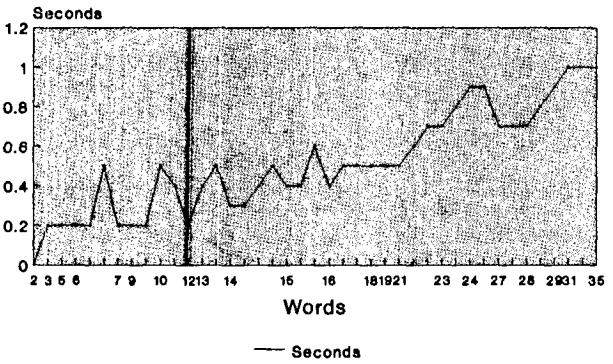

FIGURE 10. Tomita's Testbed—CPU Time

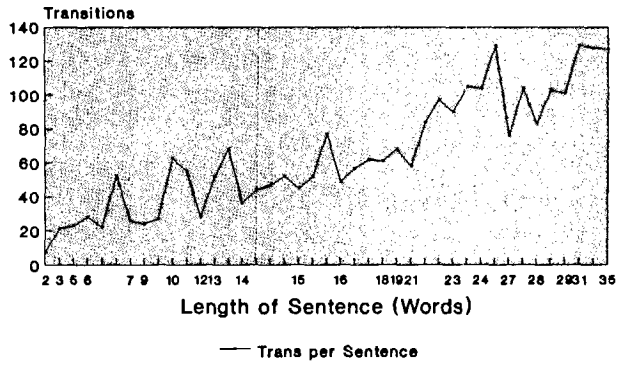

FIGURE 11. Tomita's Testbed—Abstract Machine Time

once, as a whole, just as a simple FSA sees all the bits in a state-symbol, whole. The only difference is that RVG looks at the bits with a ternary rather than a binary logic. The operators of the ternary vector logic look at all features in a vector in parallel. On binary computers, ternary vectors are implemented as a pair of bit vectors, and ternary operators exploit the low-level parallelism of bit-wise operators, e.g., on a 32-bit machine, 32 features at a time. (Such micro-parallelism is of course plausible in neural machines as well.)

Because RVG represents more information about syntactic state in the state itself, it is able to be quite compact in its representation of natural language grammars, and hence quite efficient in processing them.

Acknowledgments. RVG originates in unpublished work by A.E. Kunst, to whom I am also grateful for help with this article. Brad James helped develop the analysis of complexity. Thanks also to Edwin J. Kay for help with subsequent drafts.

REFERENCES

1. Barton, G., Berwick, R., and Ristad, E. *Computational Complexity and Natural Language.* MIT Press, Cambridge, Mass., 1987.
2. Bever, T., Garrett, M., and Hartig. R. The interaction of perceptual processes and ambiguous sentences. *Memory and Cognition* 1, 3 (1973), 277–286.
3. Blank, G. Responsive system control using register vector grammar. In *Proceedings of the IEEE International Symposium on Intelligent Control* (Philadelphia, Penn., Jan. 19–20, 1987), pp. 461–466.
4. Blank, G. Register vector grammar: A new kind of finite state automaton. In *Proceedings of the Ninth International Conference on Artificial Intelligence* (UCLA, Aug. 18–23, 1985), pp. 749–756.
5. Bresnan, J., Kaplan, R., Peters, S., and Zaenan, A. Cross-serial dependencies in Dutch. *Linguistic Inquiry 13,* 4 (1982), 613–635.
6. Carrithers, C., and Bever, T. Eye-fixation patterns during reading confirm theories of language comprehension. *Cog. Sci. 8, 2* (1984), 157–172.
7. Chomsky, N. *Syntactic structures.* Mouton, The Hague, Netherlands, 1957.
8. Church, K. *On memory limitations in natural language processing.* IU Linguistics Club, Bloomington, Ind., 1982.
9. Cowper, E. *Constraints on sentence complexity: A model for syntactic processing.* Ph.D. dissertation, Brown University, 1976.
10. Earley, J. An efficient context-free algorithm. *Commun. ACM 6,* 8 (Aug. 1970), 94–102.
11. Frazier, L., and Fodor, J. The sausage machine: A new two-stage parsing model. *Cognition 6,* 4 (1978), 291–295.
12. Garrett, M., and Bever, T. The perceptual segmentation of sentences. In *The Structure and Psychology of Language,* T. Bever and W. Weksel, Eds. Holt, Rinehart & Winston, New York, 1970.
13. Gazdar, G., Klein, E., Pullum, G., and Sag, I. *Generalized Phrase Structure Grammar.* Harvard University Press, Cambridge, Mass., 1985.
14. Hale, K. On the position of Walpiri in a typology of the base. *Indiana University Linguistics Club,* Bloomington, Ind., 1981.
15. Kimball, J. Seven principles of surface structure parsing in natural language. *Cognition 2,* 1 (1973), 15–47.
16. Kunst, A. Petri net automata and the representation of natural languages. Unpublished MS.
17. Kunst, A., and Blank, G. Processing morphology: Words and cliches. In *Computing in the Humanities,* R.W. Bailey, Ed. North-Holland, The Hague (1982), 123–131.
18. Langendoen, D. Finite-state parsing of phrase-structure languages and the status of readjustment rules in grammar. *Linguistic Inquiry 6,* 4 (1975), 533–554.
19. Langendoen, D., and Langsam, Y. The representation of constituent structures for finite-state parsing. In *Proceedings of 22nd Annual Meeting of the Association for Computational Linguistics.* (Stanford, Calif., 1984), pp. 24–27.
20. Marcus, M. *A theory of syntactic recognition for natural language.* MIT Press, Cambridge, Mass., 1980.

21. Martin, W., Church, K., and Patil, R. Preliminary analysis of a breadth-first parsing algorithm: Theoretical and experimental results. In *Natural Language Parsing Systems*. L. Bolc, Ed. Springer Verlag, Berlin, 1987.
22. Miller, G., and Chomsky, N. Finitary models of language users. In *Handbook of Mathematical Psychology*, R. D. Luce et al., Eds. Wiley, New York, 1963.
23. Milne, R. Resolving lexical ambiguity in a deterministic parser. *Comp. Ling.* 12, 1 (1986), 1–12.
24. Pullum, G. Syntactic and semantic parsability. In *Proceedings of COLING84* (Stanford University, July 1984), pp. 112–122.
25. Reed, J. An efficient context-free parsing algorithm based on Register Vector Grammars. In *Proceedings of the Third Annual IEEE Conference on Expert Systems in Government*. (1987), pp. 34–40.
26. Reed, J. Compiling phrase structure rules into Register Vector Grammar. In *Proceedings of the Fifth Annual IEEE Conference on AI Systems in Government*. (1989).
27. Reich, P. The finiteness of natural language. *Language 45*, 4 (1969), 831–43.
28. Ross, J. Constraints on Variables in Syntax. Ph.D. dissertation, MIT, Cambridge, Mass., 1967.
29. Schegloff, E. The relevance of repair to syntax-for-conversation. In *Syntax and Semantics, 12: Discourse and Syntax*, Academic Press, New York, 1979, pp. 261–288.
30. Shieber, S. Direct parsing of ID/LP grammars. *Ling. Phil.* 7 (1984), 135–54.
31. Seidenberg, M., Tanenhaus, M. Leiman, Jr., and Bienkowski, M. Automatic access of the meaning of ambiguous words in context: Some limitations of knowledge-based processing. *Cog. Psy. 14*, 4 (1982), 489–537.
32. Tomita, M. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, Norwell, Mass., 1987.
33. Wanner, E. The ATN and the sausage machine: Which one is baloney? *Cog. 8* (1980), 209–225.
34. Woods, W. Transition network grammars for natural language analysis. *Commun. ACM* 13, 10 (Oct. 1970), 591–606.
35. Yngve, V. A model and an hypothesis for language structure. In *Proceedings of the American Philosophical Society 104* (1960), 444–466.

CR Categories and Subject Descriptors: I.2.7 [**Artificial Intelligence**]: Natural Language Processing—*language parsing and understanding*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods and Search—*backtracking*

General Terms: Design, Experimentation

Additional Key Words and Phrases: Computational complexity, computational linguistics, finite automata, generation, grammars, nondeterminism, parsing, search, syntax

ABOUT THE AUTHOR:

GLENN DAVID BLANK is an assistant professor in the Computer Science and Electrical Engineering Department at Lehigh University. His research interests include computational linguistics, cognitive science, knowledge-based systems and intelligent real-time systems. He received a Ph.D. in Cognitive Science from the University of Wisconsin–Madison in 1984. Author's Present Address: Lehigh University, Computer Science and Electrical Engineering Department, Packard Laboratory 19, Bethlehem, PA 18015-3084. gdb0@lehigh.edu or glennb@scarecrow.csee.lehigh.edu.