ACL Lifetime Achievement Award

The Right Tools: Reflections on Computation and Language

William A. Woods*
ITA Software, Inc.

1. Introduction

Good morning. I want to thank the ACL for awarding me the 2010 Lifetime Achievement Award. I'm honored to be included in the ranks of my respected colleagues who have received this award previously. I want to talk to you this morning about the evolution of some ideas that I think are important, with a little bit of historical and biographical context thrown in. I hope you'll find in what I say not only an appreciation for some of the ideas and where they came from, but also a trajectory that continues forward and suggests some solutions to problems not yet solved.

1.1 Space: The First Frontier

Figure 1 is a picture of a moon rock. It has been argued that the modern era of computers, and specifically the creation of DARPA and the subsequent invention of the Internet, were all stimulated and driven by the race to conquer space. Coincidentally, the beginnings of my own career in computing and computational linguistics are also tied to space.

On October 4, 1957, the Soviet Union astounded the world by launching earth's first artificial satellite. Soon after the Sputnik launch, the Smithsonian Astrophysical Observatory contacted an astronomer, Joseph Brady, at the University of California's Lawrence Radiation Laboratory in Livermore, California. They wanted to know if the lab would track Sputnik and predict what was going to happen to it. Joe Brady had the right tools for the job. He and his colleagues had been computing planetary orbits in the solar system, and they could directly apply their programs to satellite orbits. Joe was able to successfully predict the exact day that Sputnik was last seen in the night skies over Washington, DC (December 1, 1957), 58 days after it was launched. Joe Brady and his fellow astronomers were meticulous about accuracy. (Incidentally, the Vanguard I satellite that the U.S. launched five months later is still up there and was still transmitting signals in 1964. It was the first satellite to use solar cells, clearly the right tool for that job!) Three years later, Joe Brady and his colleague Nevin Sherman would initiate me into the small group of people who could program computers.

When Sputnik went up, I was a freshman in high school and just learning algebra, which I thought was the greatest thing since sliced bread. When I graduated from high

* ITA Software, Inc., 141 Portland Street, Cambridge, MA 02139, USA. E-mail: wwoods@itasoftware.com.
This article is an extended version of the talk given on receipt of the ACL's Lifetime Achievement Award at the Association's annual meeting in Uppsala, Sweden, on July 13, 2010.

Figure 1
A photograph of lunar rock, sample 10046.

school, I went to work for Joe Brady at the Livermore Laboratory as a programming intern, before starting college at Ohio Wesleyan University. I began my professional computing career by proofreading 300 years of Mars observations. I worked at the Livermore Labs for four summers and learned a lot of things from that job, not the least of which were patience, diligence, and thoughtful programming. I first learned to program in numerical machine language on an IBM 650, the machine that Donald Knuth credits as the origin of linked lists. Because the memory on the IBM 650 was a rotating drum, latency could be optimized by distributing instructions through memory and linking them together with next-instruction pointers. If you located your instructions carefully, the next instruction would be rolling up to the read heads just as it was needed.

Later I programmed the IBM 709 and the Univac LARC. The LARC (which stood for Livermore Advanced Research Computer) was one of the first supercomputers, and was designed specifically for the Livermore Laboratory to perform scientific computations. One of the programs I wrote for the LARC was a sine and cosine routine that was accurate to half a decimal digit in the 12th decimal place and was subsequently incorporated into the Labs' Fortran library.

Twelve years later, on July 20, 1969, the United States astounded the world by visiting the moon and subsequently returning with 47.5 pounds of lunar rocks. The following summer, Jeffrey Warner, from the NASA Manned Spacecraft Center Lunar Receiving Laboratory, visited me at Harvard University, where I was a recently minted Assistant Professor. He wanted to know if I could build a question-answering system that would allow lunar scientists to directly access the data that he had collected. It turned out that I had the right tools for the job. I had just built a system for answering natural English questions, using a methodology that could be applied to arbitrary domains. I had already applied that system to airline flight schedules and to questions about states, arcs, and paths in an ATN grammar.

At the invitation of Danny Bobrow, then at Bolt Beranek and Newman, Inc. (BBN), I moved to BBN in 1970 and pursued this endeavor as my first project there. (However, I continued to teach my courses at Harvard for several years thereafter and returned

to Harvard later as a Gordon McKay Professor of the Practice of Computer Science.) With the help of Ron Kaplan, Bonnie Webber, and others at BBN, I was able to apply my system to the lunar rocks domain and demonstrate the Lunar Sciences Natural Language Information System, answering live questions from lunar scientists, at the Second Annual Lunar Science Conference in January 1971. This demonstration, six months after the project began, proved the validity of my claim that this methodology could be applied easily to new domains. The basic elements, a general-purpose ATN grammar for English and a procedural-semantics framework for semantic interpretation, were powerful tools. I kept track of the questions that were asked at the conference and the answers that the system produced, and I analyzed the things it got wrong. I cataloged the successes and the failures (most of which were due to simple errors or missing lexical entries or rules), and presented the results at the AFIPS Fall Joint Computer Conference in 1973 (Woods 1973).

So this talk is going to be about tools: a bit about tools in general, but mostly about the tools of our trade. First, however, I want to tell you a little bit about myself that may help illustrate where I'm coming from and how I approach problems.

1.2 Beginnings

I consider myself fortunate to have grown up in West Virginia at a place and time when dogs could run free, and so could little boys (see Figure 2). When I was in the second grade, I showed my first interest in linguistics when I told my teacher that the English pronoun system would work better if the pronoun *you* were confined to the singular and *you all* was used for the plural. When I was about the same age, my father taught

Figure 2
A photograph of me sitting on a rock in West Virginia.

me to use basic hand tools. I have been building things ever since. In high school, I built my own computer. My algebra teacher had given me a handout that showed how a mechanical relay could represent the binary numbers 0 and 1, and based on that and a book called *The Basics of Digital Computers*, I built a binary adder and complementer and entered it in a local science fair. I made my own relays for that computer, using metal that I salvaged from used dog-food cans and wire that I unwound from cast-off steering yokes from TV picture tubes. Figure 3 is a picture of me with my computer at the science fair. I won first prize in the mathematics division and second place overall in the fair. I have no doubt that this adventure was instrumental in my getting the job at the Livermore Laboratory.

In college, I majored in Math and Physics and fell in love with the machine shop in the Physics lab. Some of my physics experiments involved building the equipment I needed to do the experiment. I learned to use a drill press and a metal lathe. I liked the metal lathe so much that when I graduated, I bought one of my own. The thing about lathes is that they are more of a platform than a tool. They hold a piece of material and rotate it against a cutting tool that cuts away material until you're left with the shape you want. You can make almost any shape that is rotationally symmetric. On a wood lathe, you select various gouges and cutting tools and hold them against the work to make the cuts. On a metal lathe, the tool is mounted on a carriage that is moved automatically along the work, while an operator (or a gear box, or a computer program) makes precision adjustments. For many jobs, you make a custom cutting tool by taking a bar of tool steel and grinding away metal to produce exactly the tool you need for the job. Figure 4 shows some of the cutting tools I made and one of the resulting products. You can see that I enjoyed the work.

What I learned from the lathe is that you don't have to confine yourself to an existing set of tools. You can make custom tools that are tailored to exactly the job you

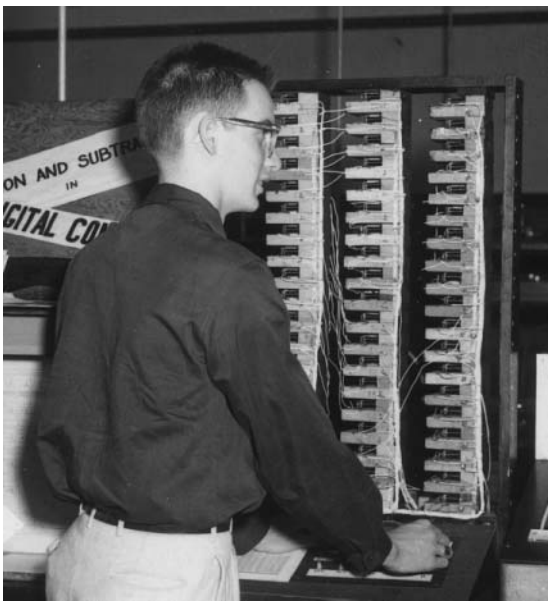


Figure 3
A photograph of me in front of my science fair computer.

Figure 4
A photograph of some lathe tools I made and a miniature canon I made with them.

want to do. This is a perspective that I've carried into my research, and one that I want to share with you today.

2. Language and Computation

I want to start with the observation that language is fundamentally computational. Computational linguistics has a more intimate relationship with computers than many other disciplines that use computers as a tool. When a computational biologist simulates population dynamics, no animals die. When a meteorologist simulates the weather, nothing gets wet. But computers really do parse sentences. Natural language question-answering systems really do answer questions. The actions of language are in the same space as computational activities, or alternatively, these particular computational activities are in the same space as language and communication.

2.1 The Tools of Our Trade

So when I talk about tools of our trade you might think first of the computer, but that's not exactly what I have in mind. I'm thinking more of abstract things like theories, algorithms, formalisms, and methodologies. In my graduate training, I learned many such tools, including finite-state machines, context-free grammars, linear bounded automata, Turing machines, circuit theory, lattice theory, first-order logic, and algorithms for parsing, search, optimization, and theorem-proving. I was fortunate to take Sheila Greibach's course in Formal Language Theory the year before she discovered abstract families of languages. When I took it, she was teaching constructive proofs for things like converting context-free grammars to Greibach normal form, constructing deterministic minimal state finite-state machines, and intersecting context-free grammars with regular sets. These algorithms have been staples of my subsequent research. Tools of our trade also include principles, notations, architectures, and resources such as dictionaries and corpora.

3. Formalisms

The first tool I want to talk about is the ATN grammar formalism, but first some background.

The year that Sputnik went up was the same year that Noam Chomsky published *Syntactic Structures*. When I started graduate school at Harvard, Transformational Grammar was all the rage, and I attended lectures and seminars by linguists such as Barbara Hall, George Lakoff, and Haj Ross. Two things impressed me about George Lakoff. One of them was the meticulous examples that he would muster to support his theories. The other was the fact that he would completely abandon a theory and look for a new one as soon as he encountered a key example that his previous theory couldn't handle. Initially, these linguists were trying to write specific sequences of transformational rules in an attempt to capture the regularities of English. Even as I watched, however, this proved too difficult, and the effort shifted to proposing constraints on such rule sets, without attempting to specify complete sequences of rules.

At the time, there were people trying to write parsers for transformational grammars, but the transformational grammar formalism didn't lend itself to efficient parsing algorithms. Stanley Petrick at IBM worked for many years on a parsing system for transformational grammars, and it was never very fast.

In contrast, the Harvard Predictive Analyzer developed by Kuno, Oettinger, and Greibach for machine translation used a form of context-free grammar that became known as Greibach normal form. With this formalism, they could parse real sentences from real documents in reasonable times. In Greibach normal form, the right-hand side of every rule started with a terminal symbol, so the operation of the parser was to simply take the next word from the input, pair it with the symbol on the top of the stack, and look for matching rules to replace the stack symbol with a sequence of new symbols. Despite this efficiency, this was an exponential parsing algorithm, unlike the CKY algorithm which was an n-cubed algorithm.

The problem with these context-free grammars, however, was that expressing the detailed knowledge of real English syntax in a context-free grammar formalism could require thousands of rules and generate hundreds of parses for real sentences. This was because each time you wanted to account for a feature, such as number agreement between a subject and object, or constraints between a category of verb and the kinds of complements it could take, it would require copying a portion of the grammar to produce specialized versions for each different case. This generally involved doubling the size of some portion of the grammar for each new feature.

For example, consider the following expansions:

(1) S → NP V (NP)

(2) S → NP Vi
S → NP Vt NP

(3) S → NPs Vis
S → NPs Vts NP
S → NPp Vip
S → NPp Vtp NP

In order to capture the constraint between transitive verbs and direct objects, we have to make two copies of the original rule. In order to capture number agreement between singular and plural subjects and singular and plural verbs, we have to make two copies

of each of these. After a number of such doublings, the size of the grammar can become quite large and unmanageable.

3.1 ATN Grammars

In my doctoral thesis, which was devoted to semantic interpretation of natural language questions, I presented a transformational grammar for a subset of English as an existence proof that English sentences could be mapped into the kinds of syntactic structures that I wanted for input to my semantic interpreter. Later, when I wanted to implement a parser that would actually do it, neither context-free grammars nor transformational grammars were good tools for the job. I wanted a tool that could do everything that a transformational grammar could do, but do it efficiently.

I developed ATN grammars to meet this need. ATN grammars are represented by state transition diagrams similar to finite-state transition diagrams, except that they allow transitions to be labeled with phrase categories as well as individual word categories, and each phrase category has a corresponding transition diagram that specifies how to recognize (or generate) phrases of that kind. In addition, the transitions can be augmented with conditions and actions that can record information about constituents parsed and can be tested to determine whether the transition is to be allowed or not. Final states in the diagram are represented by POP arcs that can also have conditions and can specify the structure to be returned for the constituent recognized. Transitions are ordered, so that the grammar can express preferences about the relative order of possible parses. Actions can include setting a weight that can also express preferences among alternative parsings. These ATN grammars were able to do the kind of deep-structure analyses of English sentences that transformational grammars could do, but do them efficiently. I published my ATN paper in the *Communications of the ACM* in 1970 (Woods 1970).

I found that ATN diagrams were good tools for evolving a grammar as I uncovered more and more of the patterns and regularities of natural English. Unlike context-free grammars, I could set features for number agreement and test them later. I could test whether a verb could be transitive before permitting a transition to pick up a direct object. I found that after my ATN grammar reached a certain maturity, the changes needed to handle new phenomena correctly were generally small and focused. When I encountered a sentence that should have parsed, but didn't, or one that didn't parse correctly, usually it was only necessary to tighten or relax a condition, or add a new transition to pick up something and then join back into the main flow at a later state. This was in marked contrast to the context-free grammar formalism, where changes that were conceptually small might involve copying large portions of the grammar and introducing a host of name variations to express a new feature. Figure 5 shows a portion of the ATN grammar for the LUNAR system (Woods, Kaplan, and Nash-Webber 1972). I found that something like 50 states and 200 transitions could account for quite a diverse range of natural English.

3.1.1 Making Sense of Shorthands. ATNs developed from my observation that common notations that linguists used to write compact rules with optional, repeatable, and alternative constituent sequences, like the following, were the same operations that defined regular expressions:

(4) $S \rightarrow NP\ V\ (NP)\ PP^*$

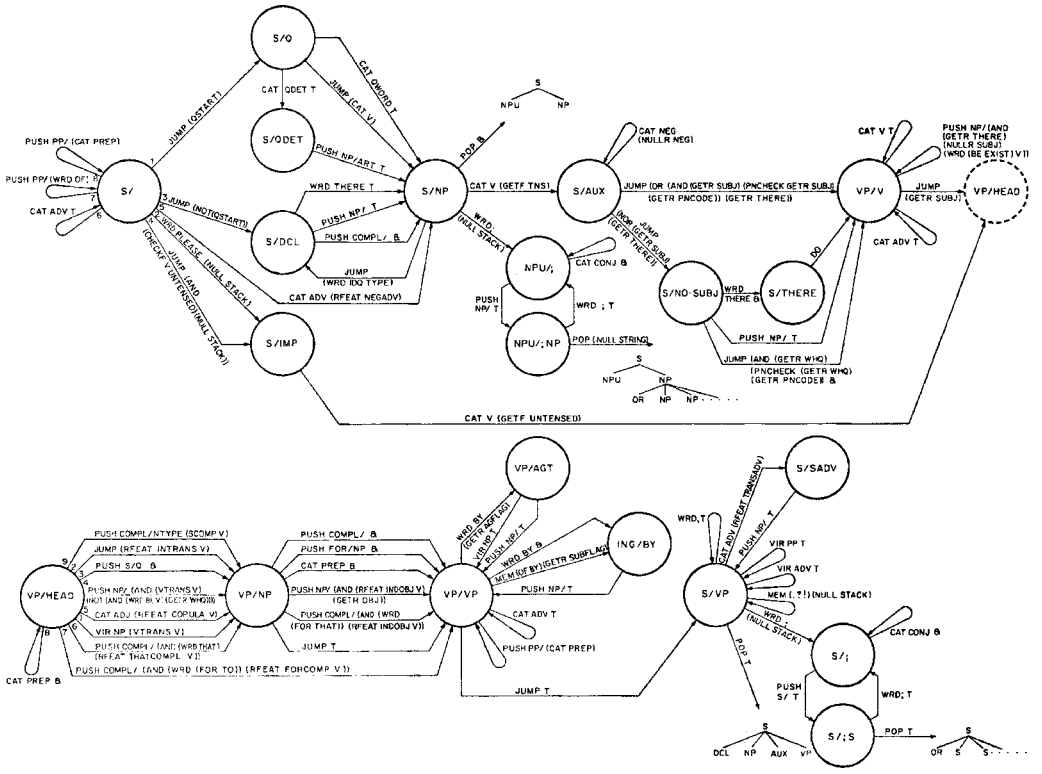


Figure 5
A portion of the ATN grammar for the LUNAR system.

(5) $S \rightarrow NP \{Vi \mid Vt\ NP\}\ PP^*$

(6) $S \rightarrow NP \{AUX\ Vtu\ NP \mid AUX\ Viu \mid Vt\ NP \mid Vi\}\ (PP)^*$

Regular expressions, I knew, were equivalent to finite-state machines, which were typically represented by finite-state transition diagrams. So what if we replaced the right-hand sides of these rules with transition diagrams? I realized that one could consider such transition diagrams to be a form of pushdown-store automaton. Thus, we could generalize finite-state transition diagrams to pushdown-store automata by allowing nonterminal symbols on transitions. This provided a natural parsing algorithm for grammars that used these notations, and would provide a better semantics for the use of such notations than just thinking of them as a shorthand for a possibly infinite set of ordinary context-free grammar rules.

This basic generalization of finite-state machines is now known as a recursive transition network or RTN. RTNs are essentially ATNs without the augmentations. RTNs are functionally equivalent to context-free grammars, but they have a capability that ordinary context-free grammars do not. With an RTN (or an ATN) it is possible to factor together common parts of different context-free rules while maintaining the constituent structure of the original. In an RTN (and in ATNs), one can distinguish a finite-state portion of the machinery and a constituent-structure portion of the machinery and keep the two distinct. One can apply factoring transformations to the finite-state part without changing the constituent structure. Figure 6 illustrates a simple factoring of

common parts of rules in the transformation of an ordinary context-free grammar into an equivalent RTN.

3.1.2 Augmenting the Transition Network. ATNs add augmentations to the basic recursive transition network of an RTN. Associated with the states of the grammar are a set of registers that can be carried forward through an analysis of a sentence. Conditions and actions on the transitions (or "arcs") of the grammar can set and test these registers. Transitions are blocked if the conditions fail, and the information carried in the registers can be used to build the structure returned by the transition network when it completes a phrase. The completion of a phrase is indicated by a POP arc, which also can have a condition, and which specifies how to build the structure to be returned for the phrase. PUSH arcs indicate transitions that can be made if a phrase of the specified type is parsed. This can be done by pushing the current configuration onto a stack and invoking the transition diagram for a phrase. When the phrase that was pushed for is completed, the result is returned as the value of the suspended transition. Parsing then continues from the configuration that is popped from the stack. This is the most direct algorithm for parsing with an ATN, but, as we shall see, it is not the only one.

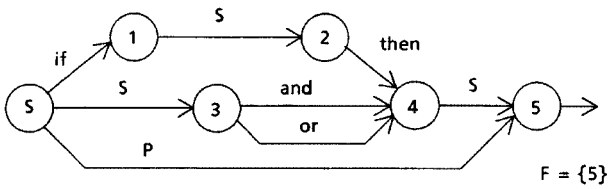
ATN actions can also pass parameters into and out of constituents, and a special HOLD list can carry information for long-distance dependencies equivalent to the traces of transformational grammar theory. A special SYSCONJ meta operator was implemented to handle conjunctions, including reduced conjunction constructions, without requiring special grammar rules for conjunction. This machinery could handle constructions like:

(7) The man drove his car through and completely demolished a plate glass window

Here, common parts of an underlying conjunction are factored out and appear only once in the surface structure, so that we appear to be conjoining the fragments *drove his car through* and *completely demolished*. The SYSCONJ machinery would automatically handle this kind of construction and build an appropriate deep structure. This kind of

1. S → if S then S
2. S → S and S
3. S → S or S
4. S → P

(a) sample context free grammar



(b) an equivalent transition network

Figure 6
Factoring the common parts of rules.

conjunction would otherwise seem to require something like a combinatory categorical grammar.

My ATN grammar parser could also use a special kind of POP arc called an SPOP that would assign a movable modifier to a preferred scope by searching up the stack for alternative locations for that modifier and then picking the scope that made the most semantic sense. This feature, called "selective modifier placement" would parse a sentence like:

(8) Does American have a flight from an East coast city to Chicago?

with *to Chicago* attached as a modifier to *flight* instead of the immediately preceding *city,* because there was a semantic rule that would allow *flight* to use *to Chicago*, although there was no such rule for *city*.

Unlike context-free grammars, ATNs lend themselves to representing regularities in so-called "free word-order languages." Whenever the regularities in the language are not well expressed by the order of constituents, these regularities can be moved out of the state transitions and into the registers. Suppose, for example, you can have subject, verb, and object in any order, except for the sequence ⟨object, subject, verb⟩, which is not permitted. To model this, you can simply have three self-looping transitions on a single state, with a condition on the subject arc that blocks it if the object is already set. All three transitions will be blocked if they have already been followed once. If the language requires at least a verb, but possibly no subject or object, then the POP arc will have a condition that requires a verb.

3.1.3 ATN Transformations. As I said, RTNs and ATNs permit transformations that share common parts of rules without changing the constituent structure. This sharing can reduce the number of configurations that have to be enumerated and explored by a parsing algorithm. If you are willing to change the constituent structure assigned by the grammar and accept a weakly equivalent grammar (one that accepts the same strings, but doesn't necessarily assign them the same structures), then you can obtain even more efficiency. For example, you can replace any nonterminal symbol with the transition network that defines it, as long as the symbol doesn't itself occur in its own transition diagram. If you do this until you can't do it anymore, you get a grammar whose only non-terminals are recursive symbols. This gets more of the grammar into the state transition diagram, where it can be optimized by standard finite-state optimization procedures.

Further, you can replace any left and right recursive symbols with jump transitions and turn these recursions into iterations. This is illustrated in Figure 7. First you eliminate left recursion by removing the transition that pushes for S from state S and adding instead a jump transition from each final state (in this case, state 5) to the state that the left-recursive transition went to (in this case, state 3). These two operations are marked with (1) in the figure. Then you eliminate right recursion by removing any transition that pushes for an S to reach a final state and replacing it with a jump transition that goes back to the start state. These two operations are marked (2) in the figure.

This gets even more of the grammar into the transition diagram where you can optimize it. In fact, one can argue that this gets as much as possible of the grammar into finite-state machinery. If you have done all of these transformations, then you have a grammar whose only nonterminal symbols are self-embedding recursive symbols. There is a theorem, called the Chomsky–Schützenberger Theorem, that shows self-embedding recursion to be the essential element that distinguishes context-free

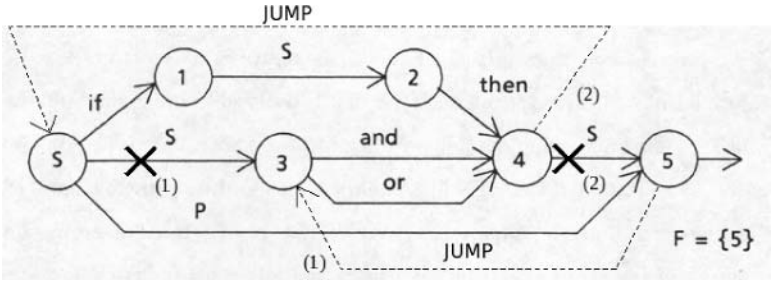


Figure 7
Turning recursions into iterations.

languages from finite-state languages. Specifically, it shows that a language is properly context-free, as opposed to finite-state, only if every context-free grammar for it contains at least one self-embedding recursive nonterminal.

So by these transformations, we can essentially move as much of the grammar as possible into finite-state machinery, where it can be optimized with finite-state optimization algorithms. Figure 8 illustrates the result of applying these transformations followed by this optimization.

3.1.4 Earley's Algorithm. The same year that my ATN paper was published in *CACM*, Jay Earley's algorithm was also published in *CACM* (Earley 1970). When Earley's algorithm came out, I was very excited by it. Originally designed for extensible programming languages, Earley's algorithm had the remarkable property that not only did it parse any context-free grammar in at most n-cubed time, but it automatically achieved the best known computation bounds on all of the subclasses of context-free grammar that were known to have time bounds less than n-cubed. By indexing things carefully, and doing just what was needed, Earley's algorithm automatically achieved n-squared bounds on linear grammars and unambiguous grammars and linear bounds on deterministic grammars, without having to be told what kind of grammar it was being given. Prior to Earley's algorithm, special parsing algorithms were used to obtain those tighter bounds.

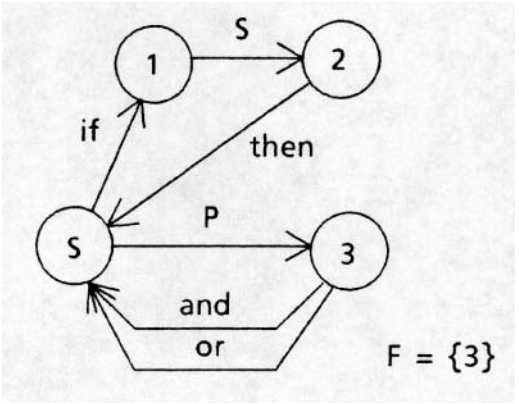


Figure 8
A highly optimized RTN equivalent to Figure 6.

Earley's algorithm achieved its basic *n*-cubed result by getting rid of the stack used by pushdown-store automata, and instead leaving the equivalent of the stack partially represented and threaded through a network of "triples" stored in columns corresponding to positions in the input string. It achieved the tighter *n*-squared and linear bounds by indexing the triples in such a way that no unnecessary searching was required at any point. Earley kept track of progress through a rule with what he called "dotted rules." These were copies of the original context-free grammar rule with a dot somewhere in the right-hand side to mark the position in the rule that the parser had reached so far. A dotted rule with the dot at the beginning of the right-hand side represented the beginning point for processing a rule, and one at the end represented a rule that had been completely matched. These dots marked the progress of the algorithm through the rule. The algorithm operated on triples consisting of the number of the rule, the position of the dot, and the column where the current constituent was begun.

3.1.5 Earley's Algorithm and ATN Grammars. It turned out that Earley's algorithm was a natural for ATN grammars. All you had to do was use the states from the ATN in place of the dotted rule (i.e., the rule number and the position of the dot). Looked at this way, the dotted rules are simply states in an ATN. Earley's three operations of prediction, scanning, and completion are simply the PUSH, CAT, and POP arcs of the ATN. Earley's algorithm can be applied directly to RTNs and to certain restricted classes of ATNs. Generalizatons of Earley's algorithm can be applied to more general ATNs.

By transforming a context-free grammar into an ATN, you can combine the common parts of different rules and achieve a reduction in the number of states in the parsing computation. Further, you can apply the transformations described earlier and produce a very compact, minimally branching, minimal state diagram, such as the one in Figure 8, and apply Earley's algorithm to that. Figure 9 illustrates a normal Earley algorithm parse of the context-free grammar in Figure 6, and Figure 10 illustrates the equivalent parse for the optimized equivalent RTN grammar of Figure 8. The result of the optimizations is a nearly deterministic parse that needs to consider only 12 state configurations compared to 50 state configurations for the original.

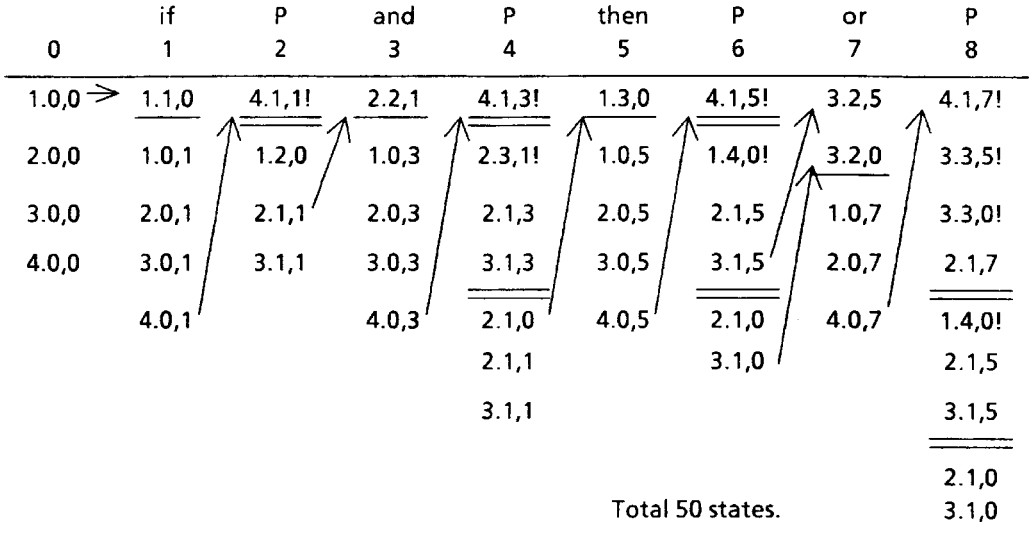


Figure 9
An Earley parse of the grammar in Figure 6.

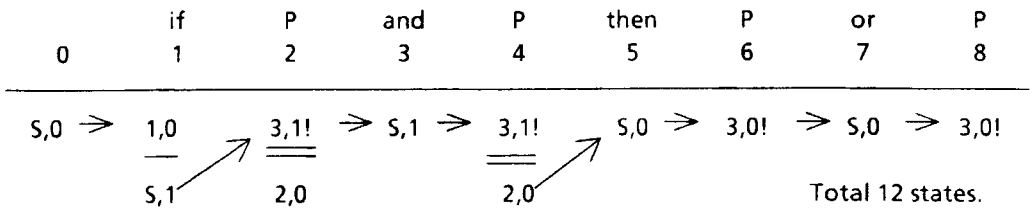


Figure 10
The Earley parse for the grammar in Figure 8.

Adapting Earley's algorithm to more general ATN grammars is more complicated than for simple RTNs, and beyond the scope of this talk, but I'd be happy to discuss it with anyone who is interested.

3.1.6 Theoretical Properties of ATNs. Viewed as automata, ATNs have a number of interesting theoretical properties. For example, without some kind of restriction on the conditions and actions on the arcs, ATN grammars would be Turing complete. In particular, any single action on an arc could be Turing complete. This has the advantage of assuring us that whatever linguistic phenomena we encounter can be captured in this formalism, but it could also make the outcome of an ATN computation semi-decidable. Clearly, as a linguist, you'd like your grammar formalism to be at least decidable.

To address this issue, I was able to prove that, with certain restrictions, parsing a sentence with an ATN grammar would be not only decidable, but primitive recursive. The only necessary restrictions were that the conditions and actions on the arcs be primitive recursive, and there must be no arbitrarily repeatable chains of singleton self-embeddings or jump transitions that could be followed in loops without consuming any input. Because any reasonable grammar would automatically satisfy these restrictions, these restrictions are effectively no restriction at all. So, parsing with any reasonable ATN grammar will be at worst primitive recursive.

As automata that are capable of assigning deep structures to natural English sentences with a primitive recursive algorithm, ATNs answered a theoretical question about the computational complexity of natural language parsing that was concerning some transformational grammarians at the time. Noam Chomsky wanted the transformational grammar formalism to express exactly the computational power required by the human language facility, which he assumed and wanted to be weaker than a Turing machine. But the transformational grammar formalism as he invented it was shown to be Turing complete, and even when constraints were added to try to avoid this, Joyce Friedman was able to prove that the result was still Turing complete.

So showing that the kinds of transformational accounts of English syntax that the transformational grammarians were developing could be expressed in an ATN formalism that was less powerful than a Turing machine and whose computations were decidable provided an answer to this question about the necessary computational power of the human language facility.

Of course you'd like your parsing algorithm to be much more efficient than an arbitrary primitive recursive computation, and in fact you'd like it to be better than exponential in the length of the input string. Applying Earley's algorithm to ATNs shows that there are at least subclasses of ATN that are *n*-cubed, *n*-squared, and even linear.

3.1.7 Applications. ATNs have been applied to a variety of problems. I originally developed them in the context of a question-answering system for airline flight schedules and then applied that framework to answering questions about states, edges, and paths in ATN grammars. The same grammar and interpretation framework could easily handle constructions from either domain, and in fact could handle the union of the two domains. You could ask this system things like the following:

(9) Does American have a flight from Boston to Chicago?

(10) Is there a Jump arc from state S/ to S/NP?

The second of these was actually asking about the very grammar used to parse both of them.

Moon Rocks. The first major test of the ATN grammar formalism was the LUNAR system that my colleagues and I developed at BBN for the NASA Manned Spacecraft Center to answer questions about the Apollo 11 moon rocks (Woods, Kaplan, and Nash-Webber 1972). Bonnie Webber and Ron Kaplan were part of that team, among others. The LUNAR domain was interesting in that it involved fluent natural English with a technical vocabulary. The lunar scientists even had some linguistic constructions that one doesn't encounter in ordinary speech. They thought of samples as partitioned into "phases" and would ask for things like:

(11) nickel concentrations in the olivine phase in sample S10046

Here, the noun *olivine* is actually a parameter to the operator *phase*, occurring as a prenominal modifier. Parameters to such operators are more commonly presented as postnominal prepositional phrases.

Other examples of questions LUNAR could handle are:

(12) In how many samples has apatite been identified?

(13) Give me analyses of the olivine phase of each breccia.

(14) What is the average Be concentration in breccias?

(15) What are the Be concentrations in each breccia?

(16) What samples contain silicon and do not contain sodium?

One of the interesting things about this domain was that many of the common English function words were also the names of chemical elements (As, At, Be, I, In, and He) as were the common names Al and Mo. This put a burden on the ATN to deal with the ambiguity of these terms. (LUNAR was implemented at a time when computer input was typically all upper case, so capitalization was not available as a clue.) In this domain, these words could not be used as reliable syntactic anchors as they were in many other grammars of the time.

In addition to parsing sentences, the ATN parser in LUNAR could be run in a mode in which it would keep track of every step of the parse and could display a browsable tree structure of the entire parse search space. At any point in this tree, you could reenter the parsing process and follow the computation step-by-step from that point. This facility was invaluable for debugging grammars and developing an ATN grammar

with large coverage. Tools such as this, which can help a developer understand what a system is doing, can be extremely important.

Continuous Speech Understanding. After the LUNAR project, BBN became involved in the first DARPA speech understanding project (1971–1976). In 1971, no one had any idea how to do speech understanding. Pierre Vicens and Raj Reddy had done a tiny pilot using Terry Winograd's blocks-world domain to show that high level constraints from syntax and semantics could make up for ambiguity at the acoustic level. To do that, however, they restricted themselves to sentences containing the word *block*. The algorithm knew that and searched for that word as an anchor. The word *block* was a good anchor because it begins and ends with plosives, which show up as "notches" in a spectrogram and are thus easy to locate. Clearly that algorithm wouldn't generalize.

I organized a workshop at Harvard that pulled together experts in acoustics, signal processing, speech recognition, linguistic phonology, computational linguistics, and artificial intelligence, and we shared what we knew. The outcome of this conference and other discussions led to the "Newell Report" (Newell et al. 1973), on the basis of which DARPA decided to fund a program in Continuous Speech Understanding. BBN was one of the contractors, John Makhoul was our speech and signal processing expert, and I was the principal investigator and language expert. Other participants were Lynn Bates, Geoff Brown, Chip Bruce, Craig Cook, Jack Klovstad, Bonnie Webber, Richard Schwartz, Jerry Wolf, and Victor Zue. Dennis Klatt consulted on the project. Ultimately, we produced a system called HWIM, for Hear What I Mean (Woods et al. 1976). HWIM understood sentences in the context of an interactive trip-planning and travel-budget-management system we called TRIPSYS.

One of the things we discovered about speech was that people generally don't pronounce all of the words in a sentence with equally clear articulation. In unstressed parts of a sentence, words are typically pronounced with little effort and their acoustic recognition is less reliable. In stressed portions of the utterance, words are more clearly pronounced and can be more reliably recognized. This led to an algorithm I developed, called the shortfall density algorithm. This algorithm could work outward from these islands of reliability until they collided with each other and could be combined into larger islands. This posed an interesting challenge for the syntactic component of the system, which was required to judge whether an arbitrary sequence of words could be extended to a complete well-formed sentence and, if possible, to predict the classes of words that would be compatible with such extensions at each end of the island.

I developed a parsing algorithm that could take an ATN grammar, index it by the constituents on the arcs of the grammar, and for any word that was hypothesized in an utterance, find all of the arcs in the grammar that could use that word. I could then test, for any two adjacent words, which of the arcs compatible with the first word could be connected to one or more of the arcs of the second by some combination of pushes, pops, and jumps. I could then combine sequences of words into islands and record a compact, factored record of all of the possible paths that the grammar could follow through that sequence of words. The arcs at the ends of these islands could be used to predict the classes of words that the lexical recognizer should look for at those ends. When a word was added to one end, the algorithm could propagate constraints through the island to the other end to possibly narrow the predicted possibilities at the other end.

For the internal structure of this "island" parse, I used a trick similar to Earley's to leave the details of a chain of pushes unspecified until needed. However, I did it for both pushes and pops and for jumps as well. A resulting path through the island had the structure of a "stile" (a set of steps used to cross a fence by going up one side and down

the other). At the top was a word or word sequence, which in general could be popped to by something on the left and could push for something on the right. The details of whether that pop or push was direct or indirect were left unspecified until more context was available to determine the answer. Figure 11 gives an example of such a stile path covering a fragment. The wavy lines in the figure represent the indefiniteness of the details of a pop or push transition. Although I developed this algorithm in the context of speech, I also considered it useful for processing ill-formed utterances. If someone makes an error in an utterance such as doubling or dropping a word or inserting the wrong word at some point, an island-driven parser can parse the rest of the utterance into partially overlapping islands and look for places where a small change could make a connection in or around the overlap area. The shortfall density algorithm could even be used to rank the likelihoods of alternative repairs and find the most likely correction.

One of the interesting experiments that we never got to complete with the HWIM system was to annotate the transition network grammar with information about prosodic signatures that should be present if certain syntactic transitions are followed. For example, the sentence *Have any people done chemical analyses on this rock* was misheard by one version of our system as *Give any people done chemical analyses on this rock*, which astoundingly parsed and passed all of the semantic and pragmatic filters. One parse was equivalent to *Give [me] any people-done chemical analyses on this rock*. However, if that was the correct parse, then there should have been a very salient prosodic signature on the phrase *people-done*, and the main verb *give* should have been stressed. Because neither was the case for the given utterance, this false interpretation would have been rejected, and the correct interpretation would have been found. We got as far as making the annotations in the grammar, but had not yet been able to test them when the project ended.

Conversational Discourse Structure. One of my students, Rachel Reichman, used an interesting kind of ATN to express models of discourse structure she called context spaces (Reichman 1981). Reichman studied real conversations and found consistent, regular

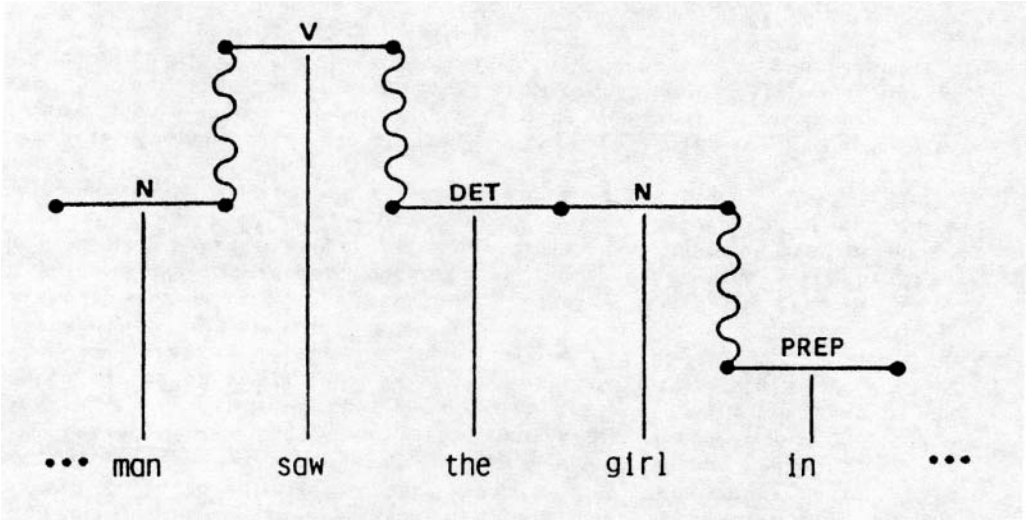


Figure 11
A stile path covering a fragment.

structure above the sentence level across a variety of kinds of discourse and various topics. She was able to use the conditions and actions on ATN arcs to characterize how discourse moves depended on context and could track focus, topic, and role through the discourse. She could even detect when a speaker switched sides in a debate.

Unlike other discourse grammars, which tend to think of discourse structure in terms of sequences of categories, Reichman could use the conditions on the ATN transitions to capture discourse moves such as "Speaker 2 utters a declarative sentence that is truth-functionally inconsistent with the previous statement by speaker 1". She called this transition a "challenge." The content in these discourse ATNs was all in the conditions. The category labels on the transitions played a very minor role, if any.

One of Reichman's contentions was that there is a structure of possible moves in discourse that is orthogonal to the intentions of the speaker, and from which the speaker has to choose, just as a chess player chooses possible moves from among the legal moves of chess. In this way, her work complemented that of Perrault, Cohen, Allen, and Bruce, dealing with the beliefs, desires, and intentions of speakers and hearers. It seemed to me that her work also provided a missing link for David McDonald's low-level tactics of sentence generation. Reichman's discourse ATNs could provide the tracking mechanism for the parameters that McDonald's system required to decide things like whether to passivize a sentence or not. Reichman's discourse ATNs could also track the "reference times" required for Reichenbach's account of perfect and progressive tenses. It had previously been a mystery to me where these parameters were expected to come from.

Other Applications. ATNs and related automata have been applied to a variety of other problems:

- ATN grammars, without a computer, have been used for field linguistics to record the evolving understanding of grammars of native languages (Joseph Grimes, personal communication).
- ATNs were used by Joyce Friedman and David Warren to represent the grammar of Montague's Proper Treatment of Quantification in English (PTQ) and thus provide a parsing algorithm for Montague Grammar (Friedman and Warren 1978).
- ATNs were used for other kinds of perception. For example, ATNs were used to recognize patterns of chromosomes (Chou and Fu 1975).

3.2 Generalized Transition Networks

A Generalized Transition Network (GTN) is a generalization of an ATN that can be used for generalized perception. In GTNs, the transitions are augmented with actions of sensory perception that can be directed to any point in a perceptual space, replacing the ATN's implicit parsing of sequences of constituents in left-to-right-order. State transitions in GTNs represent moving to more complete states of knowledge as a result of additional measurements, without any assumption that those measurements are being applied to a provided sequence of inputs. In a GTN, the results of previous measurements can determine where the next measurement is to be taken. Students in my classes at Harvard used GTNs to recognize basic strokes in images of cuneiform

tablets and gliders and other basic patterns in simulations of cellular automata running Conway's game of Life.

3.3 Cascaded ATN Grammars

A Cascaded ATN Grammars (CATN) (Woods 1980) is a cooperating sequence of ATN transducers, each feeding its output to the next stage. One example of a cascade of ATNs arose in the speech understanding project. The lexical analyzer in HWIM knew about coarticulation effects between phonemes that could reduce the pronunciation of a phrase like *hand label* by suppressing the *d* in the context between *n* and *l*. This was captured by an algorithm that could be viewed as an ATN transducer that nondeterministically transcribed a sequence of input phonemes into a sequence of words. In this transducer, the transmission of the words and the consumption of the input phonemes could be somewhat asynchronous. The ATN transducer needed to consume the *l* of the next word *label*, before it could confirm the appropriate context for the missing *d* in *hand*. So by the time it transmitted the word *hand*, it was already working on the input for the next word *label*. Registers in the ATN could remember the hypothesized current word while the ATN entered a state that dealt with the coarticulation effects, which could be shared by all words ending in *nd*. Then when the confirming *l* was consumed, the remembered word was transmitted.

Another example of an ATN cascade would consist of an ATN transducer to recognize syntactic structure, followed by an ATN that performs a semantic interpretation. Subsequent stages could do pragmatic interpretation, discourse structure, and plan recognition. Cascading provides benefits similar to sequential decomposition of finite-state machines: It reduces the overall combinatorics and produces a simpler model. By factoring these stages apart, we can reduce the extent to which phenomena have to be learned and recorded multiple times in different contexts. For example, we don't have to learn the basic structure of a noun phrase, including the English system of quantifiers and determiners, repeatedly, for all of the different semantic classes of head noun that we might want to distinguish. If we're running a machine learning algorithm, this means that we don't have to encounter examples of all of these combinations in order to learn the correct model. Also, evidence for syntactic structure will be pooled over different semantic variations and vice versa. Finally, what we learn for the syntactic stage can carry over to other domains and other applications.

Separating syntactic and semantic processing into two stages of a cascade can also optimize the flow of information between the two. For example, when parsing a noun phrase, the syntactic stage can parse the determiner and any adjectives up to the head noun before transmitting anything to the next stage. Then it can transmit the head noun and any premodifiers. This allows the semantic interpretation stage, which is largely head-driven, to start its processing with the most important information. Then the syntactic stage can parse any subsequent PPs or other postnominal modifiers and transmit them, followed finally by transmitting the determiner information. After the semantic stage has interpreted the meanings of the head noun and its modifiers, it can share the portion of the semantic stage that analyzes determiners.

4. Factoring

This discussion of cascading illustrates a principle I call factoring: the sharing of common parts of grammars and hypotheses. Factoring is a key element of what makes ATNs efficient, both for grammar development and for syntactic processing. I distinguish two

kinds of factoring: conceptual factoring and hypothesis factoring. Conceptual factoring is the sharing of common parts of grammars or other formal models to help a linguist or a grammar-learning algorithm to capture generalities. Hypothesis factoring is the sharing of common parts of alternative hypotheses that arise in parsing, perception, or learning. Hypothesis factoring helps a parsing algorithm or learning algorithm operate more efficiently by generating fewer cases that need to be considered independently. Earley's trick to avoid enumerating different alternative stack contexts while processing the transitions in a constituent is an example of factoring, as is my generalization of this trick to handle middle-out island parsing for speech. It's also the purpose of the RTN optimization algorithm I described earlier, and one of the principal benefits of cascading.

Usually factoring transformations that merge the common parts of grammar rules provide both kinds of factoring benefits, although occasionally these two principles could compete with each other. In general, one should look for conceptual factoring issues at the level of how a linguist might interact with a formalism to understand models and theories. Hypothesis factoring optimizations can then be applied behind the scene by compilers that transform models into efficient structures for processing. For example, if a linguist prefers to formulate grammars as sets of rules, these could be transformed to ATNs behind the scene and optimized to obtain efficient parsing algorithms. I seem to recall that Martin Kay compiled some of his chart-parsing algorithms into something very similar to ATNs before parsing.

5. Procedural Semantics

I want to turn now to the topic of Procedural Semantics.

In a term paper that led to my thesis on "Semantics for a Question-Answering System" (Woods 1967, 1979), I proposed procedural semantics to resolve the dilemma I had about how to get a computer to answer questions about a database. According to my dictionary, "semantics" was the relationship between signs and symbols and the things they denote or mean. So what, I wanted to know, is meaning? For my application, semantics had to be more than just assigning features and calling them semantic, or drawing diagrams and calling them semantic networks. I wanted to know what we could store in our presumably finite brains that could conceivably play the roles we attribute to meanings. The best I could find in the philosophy literature was a quote from Carnap: "To know the truth conditions of a sentence is to know what is asserted by it—in usual terms, its 'meaning'." The only thing I knew that could finitely represent the truth conditions of a sentence (which is an infinite set of assignments of truth values to propositions in all possible worlds) was some form of procedure: a Turing machine, a Post production system, or a computer program.

The idea of procedural semantics is that the semantics of natural language sentences can be characterized in a formalism whose meanings are defined by abstract procedures that a computer (or a person) can either execute or reason about. In this theory:

- The meaning of a noun is a procedure for recognizing or generating instances.
- The meaning of a proposition is a procedure for determining if it's true.
- The meaning of an action is the ability to do it or to tell if it has been done.

This theory can be thought of either as an alternative to the standard Tarskian semantics for formal systems or as an extension of them. The idea is that the computational

primitives consisting of represented symbols, the ordered pair, the assignment of values to variables, conditional branching and iteration, addition and subtraction, and the subroutine call, together with sensorimotor operators that interact with a real world, constitute a stronger (and more well-understood) foundation on which to build a theory of meaning than does set theory and the logical operations of universal and existential quantification over an all-inclusive infinite universe. These latter theories have their own paradoxes and incompleteness issues, which we usually ignore, due to familiarity. For more about the limitations of classical logic alone as a basis for meaning, see Woods (1987).

A procedural semantics foundation allows for a definition of the standard logical operators as well as extensions of them to deal with generalized quantifiers, as well as questions and imperative operations. Building on computational primitives, like the ordered pair and conditional iteration, seemed to me to be at least as well understood, and more grounded, than building on the logical operations of universal and existential quantification over an infinite universe and then having to make some kind of extension to handle the meanings of questions and imperatives. Moreover, because procedural specifications can be installed in a machine, where they can be physically executed, they can interact with sensors like keyboards and cameras and with output devices like printers and manipulators, so that this approach allows meanings that actually interact with a physical world, something that no previous theory of meaning had been able to achieve.

5.1 Semantics for a Question-Answering System

In the case of question answering, procedural semantics allows one to decouple the parsing and semantic interpretation of English sentences from the details of the storage and representational conventions of the information in the database. The procedural semantics approach allows a computer to understand, in a single, uniform way, the meanings of conditions to be tested, questions to be answered, and actions to be carried out. Moreover, it permits a general-purpose system for language understanding to be used with different databases, and even combinations of databases that may have different representational conventions and different data structures, and it allows questions to be answered by results that are computed from the data in the database without being explicitly stored.

In "Semantics for a Question-Answering System" (Woods 1967), I formulated a theory of procedural semantics and a methodology for assigning semantic interpretations to parsed sentences of English. This amounted to compiling the English sentence into an equivalent abstract procedure, expressed in a meaning representation language (MRL) that was an extension of conventional predicate calculus notations. This MRL extended the logical inventory with commands and actions in addition to propositions, and it introduced generalized and typed quantifiers. The range of quantification for a typed quantifier was specified by an abstract procedure for enumerating elements of the class, and the class could be parametrically specified.

This can be illustrated by an example from the LUNAR system discussed earlier. The semantic interpretation of the question *What is the average concentration of Aluminum in each breccia?* was as follows:

(17)
```
(FOR EVERY X5 / (SEQ TYPECS) : T ;
    (PRINTOUT (AVGCOMP X5 (QUOTE OVERALL) (QUOTE AL2O3))))
```

This semantic interpretation can be read as follows:

> For every x5 in the class TYPECS such that the universally true condition T is true, print out the value computed by the averaging function AVGCOMP for the sample x5 for the overall concentration of Al2O3, where TYPECS is the name used in the database for "Type-C rocks" (that's how breccias are encoded in the database) and Al2O3 is the chemical name for Aluminum Oxide (which is how Aluminum is stored in the database).

Note that answering this question involves the computation of an average that was not explicitly stored in the database, and that the grounding for the semantics of terms such as Aluminum and breccia is in a database whose structure and content and encoding conventions were previously and independently defined.

LUNAR was influential not only as an example of a successful natural language question-answering system and an exemplar of ATN parsing and procedural semantics, but as a stimulus for a lot of subsequent intellectual development. For example, Ron Kaplan built on the things he learned from working with the LUNAR ATN grammar to develop his formalism of Lexical Functional Grammars, which in turn influenced the development of Head-driven Phrase Structure Grammars. Danny Bobrow proposed the idea of spaghetti stacks (Bobrow and Wegbreit 1973) in response to my description of a Lisp feature that would make it easier to implement ATN parsers. Spaghetti stacks are now used to implement continuations in modern Lisp dialects such as Scheme. And many commercial systems have followed LUNAR as a model when developing question-answering systems for practical applications.

5.2 Formal Meaning Representation Language

LUNAR's meaning representation language was an extension of the Predicate Calculus with generalized quantifiers and imperative operators. Some examples of the schemata for these quantifiers and operators are:

(18) `(FOR <quant> <vbl> / <class> : <condition> : <command>)`

(19) `(FOR <quant> <vbl> / <class> : <condition> : <condition>)`

(20) `(TEST <condition>)`

(21) `(PRINTOUT <designator>)`

For example, Example (18) above can be read "For `<quant>` `<vbl>` in the class `<class>` such that `<condition>` is true, do `<command>`." This is the schema for a quantified command. Example (19) can be read: "For `<quant>` `<vbl>` in the class `<class>` such that the first `<condition>` is true, the second `<condition>` is also true." This is the schema for a quantified proposition. The quantifiers `<quant>` in these schemata include not only the traditional universal and existential quantifiers `EVERY` and `SOME`, but also nontraditional quantifiers like `THE` and `(MORETHAN <number>)` and a generic quantifier `GEN` which corresponds to the use in English of undetermined noun phrases with a plural noun (e.g., birds have wings). The paradigm can accommodate numerical quantifiers as exotic as *an even number of* or *a prime number of* and all sorts of *typically*, *often*, *rarely*, and "unless-contradicted" probabilistic quantifiers as well.

The schema (TEST <condition>) is a command to test a condition and print out Yes or No according to the result, and (PRINTOUT <designator>) is a command to print out a name or description of the referent of the specified designator.

The fact that LUNAR's meaning representation language uses typed quantifiers and provides for additional restrictions on the range of quantification permits a uniform treatment of different quantifiers and different kinds of head nouns and modifiers in English noun phrases. For example, *Some tall men play basketball* has the following interpretation:

(22) `(FOR SOME X / MAN : (TALL X) ; (PLAY X BASKETBALL))`

The sentence *All long flights to Boston serve meals* has the following interpretation:

(23) `(FOR EVERY X / (FLIGHT-TO BOSTON) : (LONG X) ; (SERVE-MEAL X))`

If we try mapping English directly to classical logic, we need different treatments for noun phrases with universal versus existential quantifiers. For example, for *Some tall men play basketball*, the three predicates (MAN X), (TALL X), and (PLAY X BASKETBALL) are all conjoined under the quantifier (for some x, x is a man and x is tall and x plays basketball), whereas for *All tall men play basketball*, the first two conditions would be conjoined as the antecedent of an implication whose consequent is the third condition (for all x, if x is a man and x is tall, then x plays basketball). For nonstandard quantifiers, such as *the* or *rarely*, mapping to classical logic is different still.

5.3 Reasoning with Meanings

The procedural semantics framework allows procedural interpretations to be treated in two ways. In the simplest way, the semantic interpretation is simply executed as a program to compute an answer to a question. However, in a more general case, the system can take the interpretation as an object to be reasoned about and possibly modified. For example in the following query from my Airline flight schedules application:

(24)
```
(FOR EVERY X / FLIGHT :
          (AND (OPERATOR X AA) (CONNECT X BOSTON CHICAGO)) ;
          (PRINTOUT X))
```

the system can reason that it could get the same result more efficiently by using a special enumeration function that uses a database index to enumerate only the flights that go from Boston to Chicago to specify the class of the quantifier, resulting in:

(25)
```
(FOR EVERY X / (FLIGHT-FROM-TO BOSTON CHICAGO) :
           (OPERATOR X AA) ; (PRINTOUT X))
```

This is an example of what I called "smart quantifiers," quantifiers that apply reasoning to their range of quantification and any specified filters on that range to see if there are more efficient ways to enumerate the same effective range.

There are lots of other reasons why a system might want to reason about the interpretation of a request before acting on it: it may not be appropriate to take literally, or there might be some additional helpful actions to take, or the request might be estimated to be unduly expensive, or there may be a more efficient way to do it, or the user may

not be authorized to do what is being requested, or the system may have reasons not to do the thing that is requested, and so forth. So, we would like semantic representations to be useful both for execution as a procedure and as objects of mechanical reasoning.

5.4 Answering Questions about Paths in an ATN

When I first implemented my semantic interpreter for the airline flight schedules domain, I had no actual database of flight schedules (other than a paper copy of the *Official Airline Guide*). This was before there existed a multiplicity of on-line databases. What I did have, though, was my implemented ATN parser with its ATN grammar, which consisted of a set of states with transitions to other states. So to test out my theory, and to illustrate that I could apply the theory to a database whose structure had been previously determined, I decided to make the ATN grammar itself the object of my requests. I wrote semantic interpretation rules to allow me to ask questions like *Is there a jump arc from S/ to S/NP?* or *Is there a non-looping path from S/NP to S/POP?*. These rules coexisted with my rules for airline schedules and both shared the same grammar for natural language input. One could ask the system *Is there a connection from S/NP to S/V?* or *Is there a connection from Boston to Chicago?* and the semantic rules would disambiguate which domain was intended.

Being able to query paths was especially interesting, because they didn't actually exist as objects in the grammar, but had to be constructively enumerated by the enumeration functions for the path class. Moreover, because there were a potentially infinite number of such paths, the formulation of quantifier classes defined by generators turned out to be essential. I used smart quantifiers to infer when I could use specialized generators for non-looping paths, paths rooted at a given start state, paths between two end points, and so on. I used a resolution theorem prover to prove that the conditions for one of my specialized generators were implied by the filters on the quantifier, and then used instantiated variables from this proof to provide the parameters for the resulting generator. Because I was only using the theorem prover to seek a more efficient generator for the paths, if the theorem prover took longer to come up with an answer than an estimate of how much would be saved by using the result, then it would abandon the effort and use the original quantifier.

5.5 Procedural Semantics in LUNAR

When Jeff Warner approached me from NASA about answering questions about the Apollo 11 moon rocks, I already had the machinery in place for parsing and answering English questions and a rule-based system for adding more domains. All I had to do was build a dictionary for the new vocabulary, write the appropriate semantic interpretation rules, oh, and by the way, implement a database system, import their data into it, write the procedures to define the semantics, and do all this in 256 K words of memory on a PDP-10! The PDP-10 in question was a DEC machine that BBN had modified to support virtual memory, multiple forks, and time-sharing. I was programming in Lisp and my language-processing program took up one 256-K fork all by itself, so Danny Bobrow enlisted Lisp wizard Alice Hartley to add the ability for Lisp to create and control subforks, and I implemented the database and database retrieval programs in a separate fork.

At NASA, Jeff Warner had extracted all of the data from the conference proceedings of the First Annual Lunar Science Conference, normalized the units, and cross-referenced the data with the articles from which it came. He could generate answers

to questions by having his Fortran programmer write programs to query this database. He wanted to know if he could get his Fortran programmer out of the loop. He had collected a set of queries to illustrate the kinds of questions he wanted to be able to answer, and a vocabulary of 3,000 words that included all of the chemical elements, a large vocabulary of mineral names, and every word that appeared in the proceedings of the First Annual Lunar Science Conference. For each word, he had recorded its most common syntactic category in English. I had to take this list and expand it with additional categories for ambiguous words and add in syntactic and semantic features and semantic interpretation rules for the words that needed them.

The procedural semantics approach worked fine, and I learned some interesting things about semantics and quantifiers from this experience (Woods 1978). One of the interesting discoveries was how the "average" operator interacted with quantification and treated the generic quantifier differently from a quantifier like *each.* If the quantifier is *each* you get a separate average for each value, but if the quantifier is generic or *all*, the average is computed over all of the values. LUNAR answered 78% of the queries asked of it at the Second Annual Lunar Science Conference, and 90% of those queries fell within its scope. However, LUNAR was far from being a complete solution. If you asked LUNAR, *What is a breccia?*, it would reply *S10046*. S10046 was indeed a breccia, and LUNAR was programmed to give you what you asked for. If you asked it *What is S10046?*, it would reply *S10046*, since that was a sample that was equal to S10046. LUNAR simply found referents of referring expressions and gave you their names. It had no model of the purpose behind the user's question or of different kinds of answers for different purposes.

6. Knowledge and Language

I want to shift now and talk about knowledge.

It doesn't take much thought to realize that background knowledge and contextual knowledge is essential to language interpretation. For example, knowledge is necessary to resolve the ambiguity (and get the joke) in Groucho Marx's famous *Time flies like an arrow (but fruit flies like a banana)*. Here, both syntactic and semantic ambiguity are resolved by knowing that there are fruit flies, but not time flies, that arrows fly but bananas don't, and that *flies* can be used metaphorically for *moving swiftly*.

Consider a spoken utterance that could be segmented either as *his wheat germ and honey* or *his sweet German honey*. One would need to know something about the context of this utterance in order to venture a prediction as to which interpretation was intended. Here, we need knowledge even to know what words we are hearing.

In the HWIM system, the sentence *Show me Bill's trips to Washington* was misheard as *Show me Bell's trips to Washington* in the context of a travel planning system that knew travel plans for a group of people that included Bill Woods and Alan Bell. There is a minimal difference of one phoneme between these two sentences (one letter in the written orthography), and there is only one feature difference between these two vowels. The acoustic scores of the two hypotheses were virtually identical, and the correct choice happened to come second. However, the system could easily have resolved the choice by using a semantic interpretation to check the trip database to learn that Bill Woods was scheduled to go to Washington, while Alan Bell was not.

I once called home on the telephone and asked my young son, who answered, *Is your mother there?* He said, *Yes*. I said *Can I speak to her?* He said, *Yes*. Finally, I said, *Tell her to come to the phone. OK*, he said. We know that interpreting speech acts depends on beliefs, desires, and intentions, but how do we manage and acquire all of the knowledge

it takes to correctly infer those beliefs and desires and intentions and do so efficiently at the right time?

6.1 Requirements for Knowledge Representation

We need a system that can organize and use large amounts of world knowledge and facilitate the efficient associative access to that knowledge during the analysis of sentences. My experiences in a variety of natural language applications have convinced me that understanding and using knowledge is the bottleneck in both speech and natural language processing (Woods 2007). A key problem is how to find the pieces of knowledge relevant to a problem from among all of the knowledge in a large knowledge base.

We need a representation system that can satisfy two requirements:

1. It should be expressively adequate to represent all of the necessary elements of natural language questions, commands, assertions, conditions, and designators.
2. It should be structured to support semantic interpretation, retrieval, and inference.

6.2 Links and Logic, KL-One et al.

The KL-One project at BBN (∼1977–1983) attempted to develop a representation to meet these conditions. KL-One was a knowledge representation system developed as part of a research project on Knowledge Representation for Natural Language Understanding. A number of people worked on this contract, including: me, Madeline Bates, Rusty Bobrow, Ron Brachman, Bertram Bruce, Eugene Ciccarelli, Phil Cohen, Brad Goodman, Norton Greenfeld, Andrew Haas, Robert Ingria, David Israel, Jack Klovstad, David McAllester, Ray Reiter, James Schmolze, Candace Sidner, Marc Vilain, Bonnie Webber, Martin Yonke, and Frank Zdybel. The project began as an attempt to develop a knowledge representation system suitable to represent and deliver all of the knowledge required for human-level reasoning (Sidner et al. 1981). In it, we sought to combine the best features of two traditions:

1. logical reasoning, which is rigorous and formal, but often counterintuitive, and which has algorithms that match expressions, substitute values for variables, and invoke rules, and
2. associative networks, which are structured and intuitive, but typically informal; however, they support efficient algorithms that follow paths through links to draw conclusions.

We wanted the associativity of link-based representations, in order to exploit efficient path-following algorithms, but we also needed representations with a clean and well-understood semantics. A key element of our approach was based on Ron Brachman's thesis on "Structured Inheritance Networks" (Brachman 1977). Ron was one of my thesis students at Harvard, and his thesis arose from my challenge to figure out how to index material at the sentence level so that one could find where particular things were said. Structured inheritance networks not only related concepts to each other by generality, but also aligned corresponding roles of those concepts.

One of the achievements of the KL-One project was the creation of a knowledge representation system whose semantics were sufficiently well defined that an algorithm could automatically place new concepts at the correct position in a conceptual taxonomy. I wrote the first algorithm to do this, which I called the MSS algorithm (for Most Specific Subsumer). This algorithm would automatically find the most specific concepts in an existing taxonomy that subsumed a new concept (i.e., were more general than or equivalent to the new concept). The new concept could then be added to the taxonomy directly under those concepts. An analogous algorithm, the MGS algorithm (for Most General Subsumee), could find the most general concepts that were subsumed by the new concept. James Schmolze wrote subsequent "classifiers" for KL-One.

KL-One began a wave of research in knowledge representation, inspired a vast number of complexity results, initiated the field now known as Description Logic, and spawned a family of related systems (Woods and Schmolze 1992).

6.3 Understanding Subsumption and Taxonomy

Although the original KL-One was focused on the structure of concepts, most of the subsequent work it inspired adopted a declarative approach, based on first-order logic. In this work, subsumption was identified with logical implication and set inclusion. While most of this flurry of activity was going on, I was involved in a couple of startup companies and watching all this from the sidelines. However, I felt that the declarative semantics approach had thrown out the baby with the bathwater by eliminating all of the intuitions for how the structure of links can support efficient algorithms. I also felt that the extensional subsumption criterion was a mistake. In 1990, during an interim appointment at Harvard University, I started to revisit the original goals of KL-ONE in light of where the field had gotten, under sponsorship from the Kapor Family Foundation. I wanted a representational system that would be an efficient and principled methodology for organizing knowledge, and I came to focus on a different criterion for subsumption that I called "intensional" rather than "extensional." The result was my 1991 paper on "Understanding Subsumption and Taxonomy" (Woods 1991).

The idea of intensional subsumption is that for one concept to subsume another, there must be a direct and recognizable relationship between the meanings of the concepts. It is not sufficient merely to have a set inclusion of their logical extensions. Because it takes a theorem to prove that context-free languages are the same set of languages as the languages accepted by a pushdown-store automaton, these two concepts must have different meanings, even though they have the same extension. Both meanings are essentially procedural. The first says that the language is accepted by a context-free grammar. The second says that the language is accepted by a pushdown-store automaton. The proof involves showing that these two procedures produce the same results. If it takes this much reasoning to determine that one concept implies the other, then that's not intensional subsumption.

My definition of intensional subsumption was that each part of the more general concept subsumes some part of the more specific concept. Thus [a man with a pet] subsumes [a man with a dog], because pet subsumes dog and a man is a man. Both of these subsume [a man with a dog and a cat].

I was able to extend this notion of subsumption to include "gap" predicates, which have sufficient conditions and necessary conditions that are not equivalent, leaving an undefined gap in between. So-called "natural kinds," like [chair], which are supposedly not definable, can often be modeled with such gap predicates. For example, one can

specify some necessary conditions for being a chair (you can sit on it, it was intended to be sat on, it has a back), and some additional conditions that are collectively sufficient for being a chair (four legs, one seat, one back), while leaving odd cases undefined (sitting on a log, in a crook of a tree).

The idea of intensional subsumption proved to be both more expressive than the extensional semantics approach, and also computationally more tractable. I was able to show that under certain assumptions about the structure of a conceptual taxonomy, a new concept could be assimilated by an MSS algorithm in sublinear time (on the order of the log of the size of the taxonomy). The MGS algorithm was less well-behaved, but could be expected to take this same sublinear time on the average.

With this machinery in hand, I was able to return to KL-One's original goals, seeking a knowledge representation structure to organize everything necessary for human-level reasoning. My thesis is that we use a conceptual taxonomy, based on something like intensional subsumption, to organize everything we know. In it, we can record what do do about different situations, efficiently find the most specific applicable rules, record rules for acquiring more information, record alternatives to consider, and record priorities and procedures for doing things. An example of this is the famous Nixon diamond, shown in Figure 12. The classic conundrum is based on the fact that Nixon is a Republican, and Republicans are usually Hawks, but Nixon is also a Quaker, and Quakers are usually Doves. So which is Nixon? Many AI researchers have felt that some inheritance principle should answer such questions. My opinion is that the job of the knowledge representation system is to identify the greatest-lower-bound concept [Republican Quaker] as the locus of the issue, classify it under both Republican and Quaker, classify Nixon under that, and propose that some other component figure out the political leanings of Republican Quakers (via a poll?). Then for any future person, if they are classified under both Republican and Quaker, the MSS algorithm will place them under the more specific concept [Republican Quaker] and they will inherit whatever answer this poll recorded there.

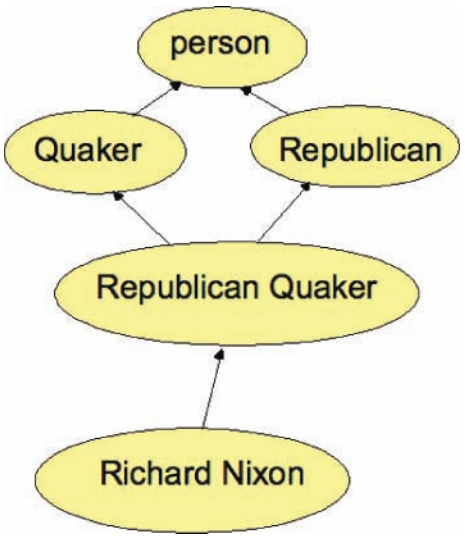


Figure 12
The Nixon Diamond.

6.4 A Practical Application of Conceptual Taxonomy

In 1991, I joined the new Sun Microsystems Laboratories in Burlington, Massachusetts, carrying with me my new theory of conceptual taxonomy. My focus was to be on improving search technology. The rationale was that if we could improve the ability to find specific information in text, it would have wide applicability. The goal was to understand phrases, handle paraphrase variations, find specific passages, and help people find specific information quickly. In addition to these goals, I was also interested in having a laboratory for gaining experience with subsumption technology on large populations of natural concepts. The population in this case was all of the words and phrases extracted from unrestricted text.

My Knowledge Technology Group at Sun Labs developed a search engine we called Nova that truly delivered on these goals. It contained a universal lexicon and morphology for general English that enabled me to apply Nova to any new subject matter with no initial preparation. It had a core lexicon of about 40,000 words, which we expanded by rule to a lexicon of approximately 150,000 word forms. The expansion was based on a known word list and a set of approximately 1,200 morphological rules that could analyze an unknown word and produce a complete lexical entry for future use. These lexical entries included syntactic word categories, semantic features, and preferences among word senses. The lexicon knew semantic subsumption facts for approximately 20,000 words. The indexer contained a scanning ATN grammar that could extract basic phrases, which were then automatically classified into a conceptual taxonomy that was automatically created for each collection we indexed. In addition to finding specific passages, Nova allowed you to browse in the conceptual taxonomy to get ideas about how to generalize a query and to understand what paraphrases exist in the material that have already been covered by what you asked. One experiment showed a five times speedup in human search productivity using Nova, compared to conventional document retrieval technology.

This search technology was incorporated into several Sun products and deployed internally to search Sun's e-mail archives. Among the people who worked on this project that you might know are Phil Resnik, Paul Martin, and Peter Norvig. Steve Green and I, with help from Paul Martin, implemented the final Java version of this project.

6.5 Generalized Perception

Since working on Nova, I've been thinking about generalized perception at a level that subsumes natural language understanding, speech understanding, visual scene recognition, and general situation awareness. This is in some sense the opposite of text search. It's more like you have a huge taxonomy of queries and only one text. When you are presented with a situation, you want all of the queries that it would satisfy to wake up, and you'd like to be alerted to the most specific ones, which will in turn provide you with information about what to do or expect in that situation. The "queries" in this case are the concepts in your taxonomy. In addition, the taxonomy should serve as a kind of "grammar" that can analyze the elements of a situation and characterize how they relate. Such a structure, I believe, is at the core of intelligent behavior, including natural language use.

Imagine such a conceptual taxonomy inserted as a stage in a CATN at the position where semantic interpretation is to occur. The taxonomy would find the most specific subsumers of each partial interpretation as it accrues, and notify the earlier stage if the

pieces don't make sense. This stage would turn phrases into concepts, relate them to other concepts, and provide associated information such as interesting specializations and other elements to expect. I'd like to find some good applications to explore these ideas in a context where the result could actually help people do things.

7. Methodology

Before I close, I'd like to say a few words about a methodology I call Directed Research. This is how I approach problems, and I recommend it for your consideration. The idea is to understand real problems that one would like to solve, and to do it with the standards of the highest quality research. This combines the best features of "applied research" and "basic research." I've always found it productive to look at the details of real problems. Real problems often reveal issues that you wouldn't think of otherwise. It's important to look at the details. Try to understand what would be necessary to solve the whole problem. At this point, don't settle for approximations. If you have a practical job to do, and it's important to get it done quickly as well as possible, and you can only do that by partially solving the problem, then by all means do that. That's practical engineering, and I do that with my Engineer's hat on. But that's not going to advance the science, and with my Scientist's hat on, I'll keep worrying the problem, trying to discover what it takes to really do the job.

It's really useful here to have an arsenal of intellectual tools to try to fit to the problem, but pay attention to the fit. Don't restrict yourself to existing tools, however nice they are. If the fit is not good, look for tools that can really do the job. Modify old ones or invent new ones as necessary. That's my message about "the right tools."

I was asked how I decide to stop working on one problem and work on a new one. For me that is easy: Once I understand a technology well enough to know its strengths and weaknesses and what it can and can't do, I start working on the next problem. I always have a queue of problems I want to work on, and I can usually find a match between one of them and something that someone needs.

8. The Best Is Yet to Come

In closing, I want to observe that there seems to have been an evolution in the things I've worked on that is moving closer and closer to the goal of truly effective person–computer communication. We still haven't begun to try some of the most interesting applications of language and computation. I'd like to make more progress on the following goals:

- genuine interchange of knowledge between a person and a machine;
- cooperative problem-solving and decision making with a machine partner; and
- a machine that can understand what you need and respond appropriately.

I believe that doing this will require integrating all of the tools I've just presented plus some machine learning and some extraction technology. And I fully expect to have to invent a few more tools as well.

You can find links to some of my papers and follow my progress on my personal Web page: `http://parsecraft.com`.

Thank you.

References

Bobrow, D. G. and B. Wegbreit. 1973. A model and stack implementation of multiple environments. *Communications of the ACM*, 16(10):591–603.

Brachman, R. A. 1977. *A Structural Paradigm for Representing Knowledge*. Ph.D. thesis, Harvard University.

Chou, S. M. and K. S. Fu. 1975. Transition networks for pattern recognition. Technical Report TR-EE 75-39, School of Electrical Engineering, Purdue University, Indiana.

Earley, J. 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102.

Friedman, J. and D. S. Warren. 1978. A parsing method for Montague Grammars. *Linguistics and Philosophy*, 2(3):347–372.

Newell, A., J. Barnett, J. Forgie, C. Green, D. Klatt, J. C. R. Licklider, M. Munson, R. Reddy, and W. Woods. 1973. *Speech Understanding Systems: Final Report of a Study Group*. North-Holland, American Elsevier.

Reichman, R. 1981. Modeling informal debates. In *IJCAI'81: Proceedings of the 7th International Joint Conference on Artificial intelligence*, pages 19–24, San Francisco, CA.

Sidner, C. L., M. Bates, R. J. Bobrow, J. Schmolze, R. J. Brachman, P. R. Cohen, D. J. Israel, B. L. Webber, and W. A. Woods. 1981. Research in knowledge representation for natural language understanding: Annual report. BBN Technical Report 4785, Bolt Beranek and Newman, Inc., Cambridge, MA.

Woods, W. A. 1967. *Semantics for a Question-Answering System*. Ph.D. thesis, Division of Engineering and Applied Physics, Harvard University. Also available as Report NSF-19, Computation Laboratory, Harvard University, September 1967. Republished in Garland Publishing's Outstanding Dissertations in Computer Science series, Garland Publishing, 1979.

Woods, W. A. 1970. Transition network grammars for natural language analysis. *Communications of the ACM*, 13(10):591–606. Reprinted in Yoh-Han Pao and George W. Ernest (eds.), *Tutorial: Context-Directed Pattern Recognition and Machine Intelligence Techniques for Information Processing*. IEEE Computer Society Press, Silver Spring, MD, 1982. Also reprinted in Barbara Grosz, Karen Sparck Jones, and Bonnie Webber (eds.), *Readings in Natural Language Processing*, San Mateo, CA, Morgan Kaufmann, 1986, pages 71–87.

Woods, W. A. 1973. Progress in natural language understanding: an application to lunar geology. In *AFIPS '73: Proceedings of the June 4-8, 1973, National Computer Conference and Exposition*, pages 441–450, New York.

Woods, W. A. 1978. Semantics and quantification in natural language question answering. In M. Yovits, editor, *Advances in Computers*. Academic Press. Reprinted in Barbara Grosz, Karen Sparck Jones, and Bonnie Webber (eds.), *Readings in Natural Language Processing*, San Mateo, CA, Morgan Kaufmann, 1986, pages 205–248.

Woods, W. A. 1979. *Semantics for a Question Answering System*. New York, Garland Publishing.

Woods, W. A. 1980. Cascaded ATN grammars. *American Journal of Computational Linguistics*, 6(1):1–12.

Woods, W. A. 1987. Don't blame the tool. *Computational Intelligence*, 3(1):228–237.

Woods, W. A. 1991. Understanding subsumption and taxonomy: A framework for progress. In John Sowa, editor, *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. San Mateo, CA, Morgan Kaufmann, pages 45–94.

Woods, W. A. 2007. Meaning and links: A semantic odyssey. *AI Magazine*, 28(4):71–92.

Woods, W. A., M. Bates, G. Brown, B. Bruce, C. Cook, J. Klovstad, J. Makhoul, B. Nash-Webber, R. Schwartz, J. Wolf, and V. Zue. 1976. Speech understanding systems: Final technical progress report, volumes i–v. BBN Technical Report 3848, Bolt Beranek and Newman Inc., Cambridge, MA.

Woods, W. A., R. M. Kaplan, and B. L. Nash-Webber. 1972. The lunar sciences natural language information system: Final report. BBN Report No. 2378, Bolt Beranek and Newman Inc., Cambridge, MA. Available from NTIS as N72-28984.

Woods, W. A. and J. G. Schmolze. 1992. The KL-ONE family. *Computers & Mathematics with Applications*, 23(2-5):133–177.