Advantages of ACT-R over Prolog for Natural Language Analysis

Jerry T. Ball

Air Force Research Laboratory, Wright-Patterson AFB, OH 45433
jerry.ball@wpafb.af.mil

Keywords:
Prolog, ACT-R, Language Analysis, NLP

ABSTRACT: *This paper discusses the advantages of using the ACT-R cognitive architecture over the Prolog programming language for the research and development of a large-scale, functional, cognitively motivated model of natural language analysis. Although Prolog was developed for Natural Language Processing (NLP), it lacks any probabilistic mechanisms for dealing with ambiguity and relies on failure detection and algorithmic backtracking to explore alternative analyses. These mechanisms are problematic for handling ill-formed or unexpected inputs, often resulting in an exploration of the entire search space, which becomes intractable as the complexity and variability of the allowed inputs and corresponding grammar grow. By comparison, ACT-R provides context dependent and probabilistic mechanisms which allow the model to incrementally pursue the best analysis. When combined with a non-monotonic context accommodation mechanism that supports modest adjustment of the evolving analysis to handle cases where the locally best analysis is not globally preferred, the result is an efficient pseudo-deterministic mechanism that obviates the need for failure detection and backtracking, aligns with our basic understanding of Human Language Processing (HLP) and is scalable to broad coverage. The successful transition of the natural language analysis model from Prolog to ACT-R suggests that a cognitively motivated approach to natural language analysis may also be suitable for achieving a functional capability.*

1. Introduction

This paper discusses the advantages of using the ACT-R cognitive architecture (Anderson, 2007) over the Prolog programming language (Clocksin & Mellish, 1984; Ball, 1985) for development of a large-scale, functional, cognitively motivated natural language analysis model (Ball, 2011b). The paper follows two papers which discuss the advantages and challenges of using ACT-R to model natural language analysis (Ball, 2011a, Ball, 2012). This paper provides the historical background for those papers, motivating the transition from Prolog to ACT-R. Over six years in the mid to late 1980's, Prolog was used to develop a natural language analysis system that became the English analysis component of an English-Japanese Machine Translation (MT) system (Ball, 1992). Although the MT system failed to achieve commercial success, the English analysis component was capable of processing an interesting range of inputs with a vocabulary of several hundred words. During development, it became clear that Prolog lacked several features needed to support creation of a fully functional language analysis system capable of processing unrestricted text. In particular, Prolog's inherent non-determinism and its weak, file order based mechanism for selecting between competing logic clauses, combined with failure detection and algorithmic backtracking as the mechanisms for trying alternatives when the selected alternative fails, proved inadequate to handle the rampant ambiguity of natural language.

A non-deterministic process is one in which there are multiple options at some processing step, necessitating a mechanism for choosing between the options. A deterministic process is one in which there is only one option at each step in processing. A monotonically evolving representation is one which can be added to (e.g. the value of a variable in the representation can be instantiated), but does not otherwise change (i.e. the value of a variable which has already been determined cannot be changed).

In 2002, the Prolog based natural language analysis system was ported to ACT-R (Ball, 2003). ACT-R replaces Prolog's file order based selection, serial execution, failure detection and backtracking mechanisms with probabilistic and context dependent mechanisms that support choosing the best alternative at each step in the serial analysis via a parallel conflict resolution process. This combination of a probabilistic and context dependent parallel conflict resolution mechanism followed by a serial execution mechanism allows the natural language analysis model to pursue the best analysis given the preceding context and current input. When combined with a non-monotonic *context accommodation* mechanism that allows the model to make modest adjustments to the evolving analysis without backtracking when the locally best analysis turns out not to be globally preferred (e.g. in incrementally processing the expression "a few books", the initial processing of "a" suggests a singular expression while the subsequent processing of "few" and "books" suggests a plural expression which non-monotonically overrides the initial singular analysis), the result is a *pseudo-deterministic* language analysis capability which presents the appearance and efficiency of deterministic processing, despite the rampant ambiguity which makes truly deterministic processing impossible (Ball, 2011a).

The ACT-R based language analysis model has been under development since 2003. Currently, the model comprises ~1100 productions and ~58,000 declarative memory (DM) elements (primarily part of speech and form specific lexical items) and is capable of processing a much broader range of English language constructions than its Prolog predecessor (www.doublertheory.com/comp-grammer/comp-grammar.htm; Ball, Heiberg & Silber, 2007). The ACT-R based model accepts text input from single words to entire documents, and processes the input incrementally one word or multi-word unit at a time. On a 64-bit quad-core Windows machine with 8 Gig RAM, the model incrementally processes ~200 words per minute without slowing down with the length of the input.

2. Prolog

Prolog (i.e. programming in logic) is a programming language developed in the 1970's (cf. Kowalski, 1982; Clocksin & Mellish, 1984; Colmerauer & Roussel, 1993). It is a computational implementation of a logic theorem prover which combines the resolution theorem proving algorithm (Robinson, 1965) and unification based pattern matching. (The unification mechanism is monotonic.) Prolog contrasts with Lisp, which is grounded in the mathematical notion of function application, and procedural languages like Fortran and C which are based on the execution of a sequence of instructions supplemented with jumps, branches and loops. At the time of its introduction, Prolog was viewed primarily as a competitor to Lisp for the development of Artificial Intelligence (AI) programs.

In 1985, I began working on the development of a natural language analysis system using Prolog. At the time, Prolog was considered by many to be the best available programming language for building Natural Language Processing (NLP) systems (cf. Gazdar & Mellish, 1989; Gal et al., 1991). In fact, Prolog was specifically designed for this purpose (Colmerauer & Roussel, 1993) and it was the language of choice in our NLP lab (Wilks & Gomez, 1988). Prolog even comes with a built-in capability for building natural language parsers and generators using the Definite Clause Grammar (DCG) formalism (Pereira & Warren, 1980). DCG supports the specification of grammar rules in a grammar like notation within Prolog. For example, consider the following DCG based rules:

```
s → np, vp.
np → det, n.
vp → v, np.
det → [the].
n → [dog].
n → [cat].
v → [chased].
```

These rules look very much like the rules of a typical context-free grammar:

```
S → NP VP
NP → Det N
VP → V NP
N → "dog"
```

with the addition of a comma to indicate the separation between elements, a period to indicate the end of a rule, the use of a list notation (e.g. `[the]`, `[dog]`) to indicate lexical items, and the use of lowercase letters for rule elements (uppercase letters indicate variables in Prolog). Besides these syntactic differences with typical context free grammar notation, the DCG notation hides two variables that correspond to the list of lexical items to be parsed or generated, split into two difference lists. For example, the list `[the, dog, chased, the, cat]` and the empty list `[]` correspond to difference lists for `[the, dog, chased, the, cat]` as do the lists `[the, dog]` and `[chased, the, cat]`. The simple DCG described above is capable of processing inputs like "the dog chased the cat", "the dog chased the dog", "the cat chased the dog", and "the cat chased the cat". With additional lexical items, many more inputs can be handled. As the input is processed during parsing, words are consumed from the first list until all the words have been consumed. For example, `[the, dog]` is consumed by the `np` rule leaving `[chased, the, cat]` and `[chased, the, cat]` is consumed by the `vp` rule, leaving `[]`. To start the parsing process a rule is called and the difference lists provided. For example,

```
?- s([the, dog, chased, the, cat],[]).
```

calls the `s` rule. Prolog will try to prove that `[the, dog, chased, the, cat]` is consistent with the rules. In this case, the input is consistent and Prolog reports success. If the input were

```
?- s([the, dog, chased],[]).
```

Prolog will determine that the input is inconsistent with the rules since the `vp` rule fails to match. In this case, Prolog reports failure.

Of course, parsers typically do more than report success or failure, and it is possible to add extra arguments to Prolog rules to create a structural representation of the input, called a parse tree. If we revise the rules as follows, Prolog will return a parse tree in addition to success when the parse succeeds:

```
s(s(NP,VP))→ np(NP), vp(VP).
np(np(D,N))→ det(D), n(N).
det(det(the))  → [the].
n(n(dog)) → [dog].
n(n(cat)) → [cat].
v(v(chased)) → [chased].
```

For the input `[the, dog, chased, the, cat]`, the parse tree will look like:

```
s(
  np(
    det(the), n(dog)),
  vp(
    v(chased),
    np(
      det(the), n(cat))))
```

Using the DCG formalism, it is very easy to specify a simple grammar for a fragment of English. It is also possible to specify a grammar that can then be used to either parse inputs or generate outputs, depending on how the variables constituting the difference lists are instantiated. For example, invocation of the `s` rule with variables for the parse tree and first list will cause Prolog to generate sequences of lexical items that are compatible with the grammar:

```
?- s(Tree,Text,[]).
```

This bi-directional character of Prolog is often cited as a major strength of the language.

The reason it is easy to build simple grammars in Prolog is because of the built-in inferencing mechanisms. One need only specify the grammar rules declaratively, and the built-in inferencing mechanisms provide the capability to parse inputs or generate outputs. The details of this processing are largely hidden from the grammar developer. The developer need only specify the starting rule and provide the input list (for parsing) or a variable (for generation). Prolog matches the left hand side of the starting rule and expands the right hand side of the rule. For parsing, the first difference list is unified with a variable in the matching rule; for generation, the variable is pushed down until a rule that provides a list is unified with the variable at some point. Overall, Prolog's inferencing mechanisms are based on non-deterministic selection of a matching rule, an attempt to prove the right hand side of the rule, and failure detection as the mechanism for backtracking and trying alternative rules. Prolog's inferencing mechanisms work well as long as there are few matching choices at each choice point, as is typical of simple grammars. This parsing model is based on techniques used to develop computer programming languages which are specifically designed to limit the number of choices at each choice point during parsing, often using limited lookahead (which Prolog does not provide) to reduce the number of choices to just one. Systems which resolve to a single choice at each "choice point" are referred to as deterministic.

Unfortunately, natural languages are not like computer programming languages in this respect. Natural languages exhibit rampant ambiguity which means that there will be multiple choices at each choice point. Across choice points, the number of alternatives multiplies and the result is an explosion of alternatives (i.e. if choice point A has 4 alternatives and choice point B has 3 alternatives, there are 12 alternatives across both choice points). A non-deterministic processor like that in Prolog is not capable of dealing efficiently with this explosion of alternatives. Anyone who has tried to build more than a simple grammar in Prolog has run into this problem. Non-deterministic systems operating over ambiguous languages are difficult to scale.

As a concrete example, consider the processing of verbs. Verbs can be subcategorized as intransitive, transitive or ditransitive based on the number of arguments they combine with.

1. He_1 ran (intransitive)
2. He_1 kicked (transitive) it_2
3. He_1 gave (ditransitive) me_2 it_3

The simple grammar above only handled the case of transitive verbs like "chased". We can add additional rules to handle the other cases:

```
vp → v. (intransitive)
vp → v, np. (transitive)
vp → v, np, np. (ditransitive)
```

We now have 3 `vp` rules. How does Prolog decide which rule to apply? Most variants of Prolog rely on the order of rules in the program file, selecting matching rules from top to bottom in the file. Given the order above, Prolog will first try the intransitive rule, followed by the transitive rule and the ditransitive rule. With 3 rules, there is only a 33% chance of Prolog picking the correct rule on the first attempt (unless rules are frequency ordered such that more frequently correct rules are attempted first). Ignoring frequency, 67% of the time Prolog will select the wrong rule and have to backtrack on failure until the correct rule is selected. As our grammar gets more complex and additional `vp` rules are added (e.g. to handle verbs like "think" which take a full clause as an argument as in "I think *he likes you*"), and `np` rules get more complex to handle relative clauses and other modifiers (e.g. "the dog *that chased the cat with the red and white striped hat* likes you"), the performance of the grammar will degrade.

Worse, it is not always possible to know when a rule should fail. In "what did he eat", "eat" is a transitive verb with an object argument (i.e. "what") that does not occur in normal position. If we treat "eat" as an intransitive verb based on the absence of the object in the normal position following "eat" (e.g. "I eat *it*"), then we cannot represent that fact that "what" is really the object of "eat". In English, wh-words are moved to the front of sentences to indicate a question, creating what is called a *long-distance dependency* between the wh-word and the normal object position. Long-distance dependencies create a serious challenge for grammar development.

As another concrete example, sentences in English come in many different forms, including declarative, imperative, wh-question, yes-no-question, exclamative:

4. He gave me the ball. (declarative)
5. Give me the ball! (imperative)
6. Who gave me the ball? (wh-question)
7. Did he give me the ball? (yes-no-question)
8. Him give me the ball, no way! (exclamative)

If we generalize sentences to non-finite as well as finite clauses, even more forms are possible:

9. To give me the ball (infinitive clause)
10. For you to give me the ball ("for" infinitive)
11. Giving me the ball (v-ing clause)
12. Given the ball (v-en clause)
13. Who he gave the ball (wh-clause)

A single sentence rule like `s → np, vp.` is insufficient to handle these alternatives. If we add multiple rules with the same left hand element `s`, Prolog's fixed order mechanism will pick alternatives based on program file order during parsing and generation. As the grammar grows to encompass more English language alternatives, the likelihood of choosing the right alternative decreases and the performance of the Prolog grammar degrades. Since the alternatives get multiplied across choice points, as the length of the input (or generated output) increases, the performance of the Prolog grammar program will deteriorate as it has to consider more and more alternatives at each choice point.

Besides the problem of fixed order selection, Prolog relies on failure detection to determine when to backtrack and try an alternative rule. Unfortunately, English grammar is too extensible and variable for failure detection to be used as a viable processing mechanism. Consider the following examples:

14. a_{sing} few_{plur} $books_{plur}$
15. a *Bin Laden* confident
16. the paperboy *porched* the newspaper
17. the *airspped* restriction
18. www.*thefreedictionary*.com

In 14, there is an incompatibility in number between "a" and "few books". A grammar that insisted on compatibility in number would fail on this input. In 15, the proper noun "Bin Laden" is being used as a modifier, an atypical use for a proper noun. A grammar which didn't allow proper nouns to function as modifiers would fail on this input. In 16, the word "porched" functions as a verb, although it is derived from the noun "porch". A grammar which only allowed "porch" to be a noun would fail. In 17, "airspeed" is misspelled and in 18, "the", "free" and "dictionary" are concatenated together to create a URL. A grammar which didn't handle such variability would fail to handle these examples. Failure of the best (if somewhat faulty) alternative in a Prolog based grammar is very problematic. Once the grammar fails on the best alternative, it will explore all possible alternatives before terminating with failure (or returning a less desirable, but grammatically acceptable alternative). This is effectively worst case behavior which in Prolog is c^n (exponential!) where c is a constant that depends on the number of logic clauses and n is the number of words in the input (cf. Allen, 1995, p. 73). In a complex grammar, exploring the entire space of alternatives can consume extensive time and memory even for relatively small n.

Some of these problems can be addressed by tying rule selection more closely to the actual input, effectively converting Prolog's rules from being context free to being context dependent. To accomplish this, it is necessary to precede rule selection with a rule that first extracts the next input and then ties rule selection to the extracted input. This approach creates a more efficient parser, but gives up on the bidirectional capabilities of Prolog in the process. It is easier to work with the new approach outside the DCG formalism. The `parse` rule shown below (the model is now specific to parsing or language analysis), ties the rule selection (`process_rule`) to the actual input via the preceding `lookup` rule. The variables `Text_in` and `Text_out` are the difference lists for the text input and the variables `Tree_in` and `Tree_out` are the difference lists for the parse tree. The variable `Entry` in the `lookup` rule holds the result of the lookup. Note that selection of `process_rule` is now dependent on the result of the `lookup` rule variable `Entry` and this variable is dependent on the text input. In the sample `lookup` rule, the word "cat" is extracted from `[cat|R]` (in Prolog, `[H|T]` is the notation for the head (`H`) and tail (`T`) of a list, where tail is everything but the head). The variable `Entry` is set to `n(cat)`(i.e. "cat" the noun) when the `lookup` rule is matched using Prolog's powerful unification mechanism, and `Rest` is set to the remainder of the input via unification.

```
parse(Text_in,Text_out,
      Tree_in,Tree_out):-
 lookup(Entry,Text_in,Text_out),
 process_rule(Entry,
      Tree_in,Tree_out).

lookup(n(cat),[cat|Rest],Rest).

process_rule(n(N),
 [np(det(D),_)|R],
 [np(det(D),n(N))|R]).
```

The `lookup` rule works well as long as there is no ambiguity – i.e. if "cat" is always a noun. But many words can be used in different parts of speech – even "cat" in "he likes to cat about on the weekend". Once we add multiple `lookup` rules for each possibility,

```
lookup(n(cat),[cat|Rest],Rest).
lookup(v(cat),[cat|Rest],Rest).
```

we are back in the position of relying on Prolog's fixed order selection mechanism to select a `lookup` rule. In this case, we can put the noun use of "cat" before the verb

use, since the noun use is much more frequent. However, we would like to be able to condition the preference on the context. Following "to", "cat" is more likely to be a verb ("to cat about"), then following "the". We need some way to adjust the preferences based on the context. Prolog does not support this (short of the draconian use of retracts followed by asserta/assertz to change the order of the `lookup` clauses).

Summarizing, a Prolog based parser could be significantly improved by incorporating context and probabilities into the rule selection process, and eliminating the use of failure detection and backtracking. To achieve this, rule selection must be conditioned on the current lexical item and its part of speech, and the wider context, as well. For example, handling the occurrence of a wh-word at the beginning of a sentence that corresponds to the object of the main verb (e.g. "$what_i$ did he eat obj_i") necessitates consideration of the wider context. The more context that is brought to bear, the more specialized the rules can be, and the less likely they are to be misapplied. The basic value of using context to guide parsing when there is ambiguity is overlooked in the predominant focus on context free grammar formalisms with their efficient processing algorithms relative to context sensitive grammars. The addition of probabilities to context free rules (probabilistic context free grammars or PCFGs) is a means of encoding global context. Further conditioning rule selection on specific lexical items (lexicalized probabilistic context free grammars or LPCFGs)—the current state of the art in computational linguistics (cf. Collins, 2003)—brings additional context to bear. Combined with algorithms which only retain a subset of the most likely outputs across choice points, parsing can be relatively efficient, but with some risk that the ultimately best choice may be pruned away.

The common assumption that a context sensitive grammar would necessarily be less efficient than a context free grammar is based on theoretical notions having to do with the power of grammar formalisms. Context sensitive grammars have been shown to be more powerful than context free grammars (Chomsky, 1956). A subset of context sensitive grammars called mildly context sensitive grammars, require on the order of n^6 computations (where n is the length of the input) in the worst case where all alternatives have to be explored in comparison to n^3 for context free grammars (Joshi, 1985). Note the implication that the parser gets slower and slower with the length of the input (in the worst case) since the function, n^6, is polynomial. But a context sensitive grammar has the important advantage of not needing to explore all alternatives. Combining a context sensitive grammar with probabilistic mechanisms for rule selection makes it possible for the parser to explore a single, or very few number of alternatives, resulting in a highly efficient best-first parse (cf. Allen, 1995, p. 216) requiring on the order of n serial computations—i.e. linear with the length of the input when only a single alternative is pursued and assuming constant time for the context sensitive pattern matching computations needed for rule selection. Human language processing (HLP) appears to use a combination of serial and parallel mechanisms which support linear processing overall (the human language processor does not slow down with the length of the input). However, on serial hardware, the parallel pattern matching computations must be computed serially which affects the linear slope.

3. ACT-R

The ACT-R computational cognitive architecture is the culmination of more than 40 years of empirical and computational research (Anderson, 2007). ACT-R integrates a procedural memory system (skill knowledge) implemented as a production system (on top of a discrete event simulation) with a declarative memory (DM) system (knowledge of facts) implemented in frame-like *chunks* (named and typed lists of slot/value pairs) organized into an inheritance hierarchy. ACT-R includes several peripheral modules including visual, aural, vocal, and manual which provide the perceptual-motor capabilities of ACT-R. There is also a goal module which determines the current goal and an imaginal module to support problem solving. Procedural memory is the central component of ACT-R. Each peripheral module contains at least one buffer for storing the current output from the module. Processing within modules to determine buffer outputs occurs in parallel. The outputs in the buffers are accessible to the central procedural memory system (see Figure 1).

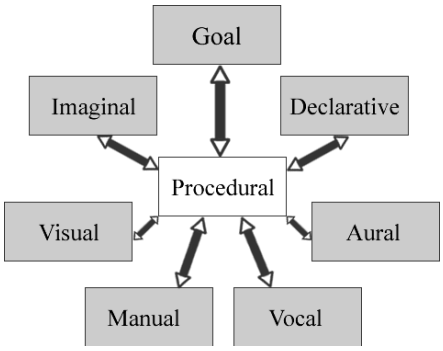

Figure 1. ACT-R cognitive architecture (Anderson, 2007)

Processing in ACT-R's procedural memory involves the parallel selection and serial execution of a sequence of productions. Production execution can result in a perceptual-motor action (e.g. visual attention shift, mouse movement), a modification to the contents of a buffer, or a DM retrieval.

Productions in ACT-R correspond to the logic rules of Prolog and DM chunks correspond roughly to Prolog facts (a type of logic rule, like the `lookup` rule, that only has a left hand-side). The parallel production selection

process in ACT-R is inherently context dependent. Productions match against the contents of buffers which provide the context for production selection. In addition, ACT-R productions are assigned utilities that determine which matching production is selected for execution. The process of matching productions and selecting the production with highest utility if referred to as *conflict resolution*. The closest equivalent in Prolog relies on the fixed top to bottom order for serial selection of logic rules which match the current context.

Beside the use of utilities for production selection during conflict resolution, ACT-R provides an activation mechanism for retrieval of chunks from DM. This retrieval mechanism functions like the `lookup` rule above which determines the part of speech of the current input—using a fixed top to bottom selection order if there are multiple alternatives. However, ACT-R's activation based retrieval mechanism is probabilistic. If a production which attempts a retrieval is selected and executed, the production provides a retrieval template that determines what kind of chunk is eligible to be retrieved. The chunk matching the retrieval template with the highest activation based on the prior history of use of the chunk (base level activation) and current context (context activation) is selected. The retrieval mechanism integrates hard constraints based on the retrieval template (e.g. retrieve a chunk that is a part of speech) with soft constraints based on spreading activation (e.g. activate chunks in memory which match the letters and trigrams of the input, and the preceding context) and base level activation. To see how the retrieval mechanism works, consider the processing of the word "cat". The retrieval template will specify retrieval of a part of speech. Only chunks which are parts of speech are eligible to be retrieved. In addition, the letters "c", "a", and "t" and the trigrams "wbca", "cat" and "atwb" ("wb" stands for word boundary) will spread activation from specialized `letter` and `trigram` buffers. Further, if "the" occurs before "cat", a bias for a noun part of speech will be spread from a specialized `context` buffer. If "to" occurs before "cat", a bias for a verb part of speech will be spread instead. These specialized buffers represent an extension of the ACT-R architecture that is specific to language analysis (Ball, 2011b). Note that if the actual input were "catt", "cat" the noun might still be retrieved if it is the most highly activated part of speech in DM (if there is no chunk corresponding to "catt" that is a part of speech). The spreading activation mechanism supports a form of *partial matching* based on *soft constraints* or *preferences* (cf. Wilks, 1975). By comparison, Prolog's unification mechanism does not allow for partial matching. Once a part of speech is retrieved and placed in the `retrieval` buffer, it becomes part of the context for subsequent production selection and execution.

ACT-R integrates the probabilistic, context dependent production selection and DM retrieval mechanisms described above, with support for a hierarchy of chunk types and a single inheritance mechanism. The language analysis model makes extensive use of ACT-R's inheritance mechanism, both in the definition of the grammatical ontology and in the production matching and selection process. As noted above, productions are matched against buffers during production selection. Productions may selectively match against any number of buffers, from 0 (in which case the production may always match) to all the buffers. Productions which match against a chunk in a buffer must specify the type of the chunk being matched. The match to the type succeeds if the chunk in the buffer is of the specified type (`isa verb` in the production matches `isa verb` in the buffer) or the specified type is a super-type of the chunk in the buffer (`isa part-of-speech` in the production matches `isa verb` in the buffer). We use the capability to match to a type or super-type extensively. Specialized productions match to chunks in buffers of a very specific type (`isa verb` in the production), whereas a more general production may match the same chunk as a high level super-type (`isa part-of-speech` in the production). Specialized productions have higher utility than competing general productions since they are more likely to be useful in a matching context than the more general production.

As a simple example of the use of the inheritance hierarchy, consider the process of retrieving a part of speech followed by the processing of the retrieved part of speech. To retrieve a part of speech, a production executes that provides a retrieval template specifying the `part-of-speech` super-type. Any chunk which is a subtype of `part-of-speech` is eligible to be retrieved. Once a chunk is retrieved, productions which are specific to the retrieved part of speech (e.g. `noun, verb`) can match the retrieved chunk. For the input "cat", if a `noun` part of speech is retrieved, this `noun` chunk will provide part of the context in which subsequent productions execute. If the context also includes a `noun-phrase` chunk whose `head` is yet to be integrated, a production which matches the `noun` and `noun-phrase` chunks can be selected. This production can then integrate the `noun` chunk as the `head` of the `noun-phrase` chunk. If there is no `noun-phrase` chunk with an empty `head`, a lower utility production which creates a new `noun-phrase` chunk and integrates the `noun` as the `head` can be selected and executed. This lower utility production will only be selected if the higher utility production does not match the context and is not in the conflict set. This makes it possible for the model to handle the case where a determiner like "the" projects a `noun-phrase` chunk with a missing `head` (e.g. "the…") as well as the case where a noun occurs without a determiner (e.g. "*rice* is good for you").

4. Probabilistic Prolog

Although there is little interest in adopting cognitively plausible mechanisms like those provided in ACT-R into Prolog, there have been attempts to add probabilistic mechanisms to Prolog and DCG to bring them in alignment with prevailing PCFG formalisms. According to Nivre (downloaded from http://w3.msi.vxu.se/~nivre/teaching/statnlp/pdcg.html):

> A very simple way of parsing with a probabilistic context-free grammar (PCFG) is to use the built-in DCG available in most implementations of Prolog. All that is required is that every category symbol (term) is extended with an extra argument for the probability of that constituent, and that every rule is extended with a Prolog call in order to multiply the probabilities of the daughters with the probability of the rule in order to obtain the probability of the entire constituent. Thus a PCFG rule of the form:
>
> x_0 --> $x_1 ... x_n$: p
>
> where p is the rule probability, will be translated into the following DCG rule:
>
> x_0(P0) --> x_1(P1), ..., x_n(Pn),
> { P0 is p*P1*...*Pn }.

Using this approach with the simple grammar in section 2, with a few extra np, vp and lexical rules added, gives

```
s(P0)→ np(P1), vp(P2),
       {PO is 1.0*P1*P2}.
np(P0)→ det(P1), n(P2),
       {PO is 0.7*P1*P2}.
np(P0)→ pn(P1),
       {PO is 0.3*P1}.
vp(P0)→ v(P1), np(P2),
       {PO is 0.5*P1*P2}.
;; transitive
vp(P0)→ v(P1),{PO is 0.3*P1}.
;; intransitive
vp(P0)→ v(P1), np(P2), np(P3),
       {PO is 0.2*P1*P2*P3}.
;; ditrans
det(1)→ [the].
n(0.5)→ [dog].
n(0.5)→ [cat].
v(0.6)→ [ate].
v(0.4)→ [chased].
```

While this approach adds probabilities, it provides no mechanism for selecting the most probable parse when there is ambiguity, other than using file order. Prolog does provide a `findall` relation that exhaustively finds all solutions. `Findall` could be used to collect and then order the solutions in terms of probabilities, but this requires an exhaustive search which results in exponential worst case behavior. Just adding probabilities to Prolog is a non-solution. By comparison, ACT-R's conflict resolution mechanism leads to selection and execution of the production with the highest utility at each step in processing—implementing a breadth-first search at each step, and pursuing only the best path. Although it is possible to implement breadth-first search in Prolog, breadth-first search alone is insufficient if it is exhaustive. Some mechanism for pursuing the best, or at least a bounded number of alternatives, at each step in processing, is also needed. A version of Prolog with this combination of mechanisms would be quite far removed from standard Prolog. ACT-R provides just this combination out of the box.

5. Conclusions

Although Prolog provides mechanisms that support the rapid development of simple grammars, Prolog lacks several features that are needed for development of a large-scale, functional language analysis capability. ACT-R's combination of probabilistic and context dependent mechanisms for production selection and DM retrieval with inheritance based pattern matching, combined with best-first search, overcome the main limitations of Prolog. The transition from Prolog to ACT-R has made it possible to expand the capabilities of the language analysis system well beyond those of the Prolog predecessor and suggests that a cognitively motivated approach to natural language analysis may also be suitable for achieving a functional capability.

Although Prolog and ACT-R are superficially very different (logic theorem prover vs. production system), and the motivations behind their development also differ (NLP vs. computational cognitive modeling), ACT-R's productions share much in common with Prolog's logic clauses. It is not too big a stretch to view ACT-R as a probabilistic and context dependent variant of Prolog which implements best first search via the combination of parallel conflict resolution at each step with serial execution of the best alternative. The introduction of probabilities and context dependence into a logic theorem proving language like Prolog has important theoretical and computational efficiency implications. Logical deduction, which is grounded in the philosophical tradition of valid inference and correct reasoning, is often considered to be incompatible with notions of probability and uncertainty which can lead to logical contradictions. It is only recently that probabilistic or Markov logics have gained some acceptance and are becoming more common (Richardson & Domingos, 2006). Beyond the introduction of probabilities and context dependence, a best first search mechanism which pursues the best, or limited number of preferred alternatives, is needed to support efficient processing.

ACT-R provides probabilistic functionality similar to a Markov logic within the framework of a psychologically motivated production system architecture. In addition to probabilities, the production system architecture puts

context dependence center stage. The combination of probabilities and context dependence in a production system architecture that implements a best first search strategy—supplemented with a context accommodation capability for recovering when the locally best solution is not globally preferred—results in an efficient *pseudo-deterministic* language analysis system. Computational challenges remain. With a mental lexicon approaching 60,000, a serial implementation of ACT-R's parallel retrieval mechanism leads to a steep linear slope that is computationally expensive if all lexical items are candidates for retrieval. We have had to constrain this mechanism to reduce the slope constant and make it less expensive (Freiman & Ball, 2010). We have also run into process space limitations on a 32-bit computer system, necessitating upgrade to a 64-bit system. We remain optimistic that the upgrade to a 64-bit system will be sufficient to meet our research and development requirements for a large-scale, functional, general purpose language analysis capability that executes at near human reading rates of 200-300 words per minute.

6. Acknowledgements

My thanks to Jeffrey Jackson for providing helpful comments on an earlier draft of this paper.

7. References

Allen, J. (1995). *Natural Language Understanding, 2nd Ed.* Redwood City, CA: Benjamin/Cummings.

Anderson, J. (2007). *How Can the Human Mind occur in the Physical Universe?* NY: Oxford.

Ball, J. (2012). Explorations in ACT-R Based Language Analysis – Memory Chunk Activation, Retrieval and Verification without Inhibition. In N. Russwinkel, U. Drewitz & H. van Rijn (eds), *Proceedings of the 11th International Conference on Cognitive Modeling*, 131-136. Berlin: Universitaets der TU Berlin.

Ball, J. (2011a). Explorations in ACT-R Based Cognitive Modeling – Chunks, Inheritance, Production Matching and Memory in Language Analysis. *Proceedings of the AAAI Fall Symposium: Advances in Cognitive Systems*.

Ball, J. (2011b). A Pseudo-Deterministic Model of Human Language Processing. In L. Carlson, C. Hölscher, & T. Shipley (Eds.), *Proceedings of the 33rd Annual Conference of the Cognitive Science Society* (pp. 495-500). Austin, TX: Cognitive Science Society.

Ball, J. (2003). Beginnings of a Language Comprehension Module in ACT-R 5.0. *Proceedings of the Fifth International Conference on Cognitive Modeling*. Edited by F. Detje, D. Doerner and H. Schaub. Universitaets-Verlag Bamberg. ISBN 3-933463-15-7.

Ball, J. (1992). PM, Propositional Model, a Computational Psycholinguistic Model of Language Comprehension Based on a Relational Analysis of Written English. Ann Arbor, MI: UMI Dissertation Information Service.

Ball, J. (1985). A Consideration of Prolog. Report No. MCCS-85-171. Memoranda in Computer and Cognitive Science, Computing Research Laboratory, New Mexico State University, Las Cruces, NM 88003.

Ball, J., Heiberg, A. & Silber, R. (2007). Toward a Large-Scale Model of Language Comprehension in ACT-R 6. In R. Lewis, T. Polk & J. Laird (Eds.) *Proceedings of the 8th International Conference on Cognitive Modeling*. 173-179. NY: Psychology Press.

Ball, J., Myers, C., Heiberg, A., Cooke, N., Matessa, M., Freiman, M. and Rodgers, S. (2010). The synthetic teammate project. *Computational and Mathematical Organization Theory, 16*(3), 271-299.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory* (2): 113–124.

Clocksin, W. & Mellish, C. (1984). *Programming in Prolog, 2nd Ed*. NY: Springer-Verlag.

Collins, M. (2003). Head-Driven Statistical Models for Natural Language Parsing. *Journal of the Association for Computational Linguistics, 29*, 589-637.

Colmerauer, A. & Roussel, A. (1993). The birth of Prolog. *ACM SIGPLAN Notices* 28: 37.

Freiman, M. & Ball, J. (2010). Improving the Reading Rate of Double-R-Language. In D. D. Salvucci & G. Gunzelmann (Eds.), *Proceedings of the 10th International Conference on Cognitive Modeling* (pp. 1-6). Philadelphia, PA: Drexel Univ.

Gal, A., Lapalme, G., Saint-Dizier, P. & Somers, H. (1991). *Prolog for Natural Language Processing*. Chichester, UK: Wiley.

Gazdar, G., Klein, E., Pullum, G. & Sag, I. (1985). *Generalized Phrase Structure Grammar*. Oxford: Basil Blackwell.

Gazdar, G. & Mellish, C. (1989). *Natural Language Processing in Prolog*. Addison-Wesley.

Joshi, A. (1985). How much context-sensitivity is necessary for characterizing structural descriptions? In D. Dowty, L. Karttunen & A. Zwicky (eds.) *Natural Language Parsing: Theoretical, Computational and Psychological Perspectives*. NY: Cambridge University Press, 206-250.

Kowalski, R. (1982). Logic as a computer language. *Logic Programming.* Edited by K. Clark and S. Tarnlund. NY: Academic Press.

Pereira, F. & Warren, D. (1980). Definite clause grammars for language analysis – A Survey of the Formalism and a Comparison with Augmented Transition networks. *Artificial Intelligence*, 13:231-2787.

Richardson, M & Domingos, P. (2006). Markov logic networks. *Machine Learning*, 62:107-136.

Robinson, J.A. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1): 23-41.

Wilks, Y. (1975). A Preferential Pattern-Seeking Semantics for Natural Language Inference. *Artificial Intelligence* 6: 53-74.

Wilks, Y. & Gomez, R. (1988). New Mexico State University's Computing Research Laboratory. *AI Magazine*, 9 (1), 79-94.

Author Biography

JERRY BALL is a senior research psychologist in the Human Effectiveness Directorate, 711th Human Performance Wing, Air Force Research Laboratory. He has a Masters Degree in Computer Science from the University of Florida and a PhD in Cognitive Psychology from New Mexico State University.