A Computational Psycholinguistic Model of Natural Language Understanding

Jerry T. Ball, PhD

www.DoubleRTheory.com
Jerry@DoubleRTheory.com

Abstract

Double R Model is a computational psycholinguistic model of natural language understanding founded on the linguistic principles of Cognitive Linguistics and implemented using the Atomic Components of Thought – Rational (ACT-R) cognitive architecture and modeling environment. Double R Grammar is the Cognitive Linguistic theory underlying Double R Model. In Double R Grammar, the focus is on the representation and integration of referential and relational meaning—two key dimensions of meaning that get grammatically encoded. Double R Process is the psycholinguistic theory of language processing underlying Double R Model. Double R Process is a highly interactive theory of language processing which eschews a separate syntactic analysis feeding a semantic interpretation component in favor of a direct interpretation of the referential and relational meaning of input texts. Double R Model is intended to validate the representation and processing commitments of Double R Grammar and Double R Process and to form the basis for the development of large-scale, functional natural language understanding systems.

Introduction

Double R Model (i.e. Referential and Relational Model) is a computational psycholinguistic model of natural language understanding founded on the linguistic principles of Cognitive Linguistics (Langacker, 1987, 1991; Lakoff, 1988; Talmy, 2003) and implemented using the Atomic Components of Thought – Rational (ACT-R) cognitive architecture and modeling environment (Anderson & Lebiere, 1998). Double R Grammar is the Cognitive Linguistic theory underlying Double R Model. In Double R Grammar, the focus is on the representation and integration of referential and relational meaning—two key dimensions of meaning that get grammatically encoded. Double R Process is the psycholinguistic theory of language processing underlying Double R Model. Double R Process is a highly interactive theory of language processing which eschews a separate syntactic analysis feeding a semantic interpretation component in favor of a direct interpretation of the referential and relational meaning of input texts. Double R Model is intended to validate the representation and processing commitments of Double R Grammar and Double R Process, together called Double R Theory, and to form the basis for the development of large-scale, functional natural language understanding systems.

After introducing the basic theoretical commitments of Double R Model, this paper discusses some of the representational and processing commitments in more detail. It concludes with a processing example that demonstrates a subset of these commitments.

Double R Grammar

Double R Grammar is the Cognitive Linguistic theory underlying Double R Model. In Cognitive Linguistics, all grammatical elements have a semantic basis, including parts of speech, grammatical markers, phrases and clauses. Our understanding of language is embodied and based on experience in the world (Lakoff & Johnson, 1980). Categorization is a key element of linguistic knowledge, and categories are seldom absolute—exhibiting, instead, effects of prototypicality, base level categories (Rosch, 1978), family resemblance (Wittgenstein, 1953), fuzzy boundaries, radial structure and the like (Lakoff, 1987). Our linguistic capabilities derive from basic cognitive capabilities—there is no autonomous syntactic component (Chomsky, 1957, 1965) separate from the rest of cognition. Knowledge of language is for the most part learned and not innate. Abstract linguistic categories (e.g. noun, verb, nominal, clause) are learned on the basis of experience with multiple instances of words and expressions which are members of these categories, with the categories being abstracted and generalized from experience. Also learned are schemas which abstract away from the relationships between linguistic categories. Over the course of a lifetime, humans acquire a large stock of schemas at multiple levels of abstraction and generalization, representing knowledge of language and supporting language comprehension. These schemas constitute what might be called **grammatical semantics** in contrast to the **lexical semantics** of individual lexical items, although the schemas are, for the most part, associated with specific lexical items.

Two key dimensions of meaning that get grammatically encoded are referential meaning and relational meaning. Double R Grammar is focused on the representation and

integration of these two dimensions of meaning within the wider scope of Cognitive Linguistics. Consider the expressions

1. The book on the table
2. The book is on the table

These two expressions have essentially the same relational meaning. They both express the relation "on" existing between "a book" and "a table". However, their referential meaning is significantly different. The first expression, as a whole, refers to an object and is called an **object referring expression** in Double R Grammar. In referring to an object, the first expression uses the determiner "the" to **specify** that the object is salient in the context of use of the expression (and may have previously been referred to). The first expression also uses the word "book" to indicate the type of object being referred to, with "book" functioning as the **head** of the expression. Further, the phrase "on the table" refers to a location with respect to which the object can be identified and functions as a **modifier** in the expression. In referring to a location, the expression "on the table" refers to a second object "the table" and indicates the location of the first object with respect to the second object. Within the modifying expression, the relation "on" functions as the **relational head** with the object referring expression "the table" functioning as a **complement**. In the first expression the relational meaning of "on" is subordinated to referential meaning with the modifying function of "on the table" dominating the relational meaning of "on". That is, although "on" is the relational head of the prepositional phrase "on the table", it is not the head of the overall expression and does not determine the semantic type of that expression.

The second expression refers to a situation and is called a **situation referring expression** in Double R Grammar. The second expression uses the auxiliary "is" to provide a temporal specification for the situation, fulfilling a referential function similar to that of the determiner "the" in "the book" and "the table". The relational meaning of the second expression is about "being on" and not just "being", with "on" functioning as the relational head of the situation referring expression. The relational head of a situation referring expression is called a **predicate** in Double R Grammar—reflecting the assertional function of the relational head. Note that "on" in the first expression is not functioning as a predicate, since it is presupposed and not asserted. That is, relational heads of modifying expressions are not predicates in Double R Grammar, they are (modifying) **functions**. In the second expression, the object referring expression "the book" functions as the subject (argument) of "being on" with "the table" functioning as the object (argument). Referentially, there is also a reference to a location "on the table", which competes with the expression of the relational meaning of "on" as reflected in the difference between:

3. What is the book on?
4. Where is the book?

where 3 highlights the relation "on" in asking about the object of that relation and 4 highlights the reference to a location using "where" to do so.

The terms **specifier**, **head**, **modifier** and **complement** are borrowed from X-Bar Theory (Chomsky, 1970). It is acknowledged that X-Bar Theory captures an important grammatical generalization, but X-Bar theory is in need of semantic motivation (Ball, 2003a). In Double R Grammar, these terms are used to express referential functions that combine to form object, location and situation referring expressions (among others). Referring expressions in turn function as arguments in relational structures (and complements in corresponding referential structures). The joint encoding of referential and relational meaning leads to representations that simultaneously reflect both these important dimensions of meaning, with trade-offs occurring where the encoding of referential and relational meaning compete for expression.

The specifier determines the referential type of a referring expression whereas the head determines the semantic type of the expression. Consider the referring expression

5. The kick

in which the specifier "the" determines the expression to be an object referring expression, whereas, the word "kick" determines the expression to be a type of action (called a **Type Specification** in Langacker, 1991). In this expression, the specifier has the effect of objectifying the action expressed by "kick" and allowing it to be referred to as though it were an object. Note that since the inherent meaning of "kick" is not affected (only its function), there is no need to assume that the part of speech of "kick" is a noun instead of a verb in this expression. And if we allow verbs (especially action verbs) to function as heads of object referring expressions (i.e. noun phrases), then one of the primary syntactic arguments against the meaning based definition for parts of speech is nullified (Ball, 2003b).

Double R Process

Double R Process is the psycholinguistic theory underlying Double R Model. It is a highly interactive theory of language processing. Representations of referential and relational meaning are constructed directly from input texts. There is no separate syntactic analysis that feeds a semantic interpretation component. The processing mechanism is driven by the input text in a largely bottom-up, lexically driven manner. There is no top-down assumption that a privileged linguistic constituent like the sentence will occur (vice Townsend & Bever, 2001). There is no phrase structure grammar and no top-down control mechanism. How then are representations of input text constructed? Operating on the text from left to right, schemas corresponding to lexical items are activated. For those lexical items which are relational or referential, these schemas establish expectations which both determine the possible structures and drive the processing mechanism. A short-term working memory (Kintsch, 1998) is available for storing arguments which have yet to be integrated into a relational or referential structure, partially instantiated relational and referential structures, and completed structures. If a relational or referential entity is encountered which expects to find an argument to its left in the input text then that argument is assumed to be available in short-term working memory. If the relational or referential entity expects to find an argument to its right in the input text, then the entity is stored in short-term working memory as a partially completed structure and waits for the occurrence of the appropriate argument. When that

argument is encountered it is instantiated into the stored relational or referential structure. Instantiated arguments are not separately available in short-term working memory. This keeps the number of separate linguistic units which must be maintained in short-term working memory to a minimum.

ACT-R

ACT-R is a cognitive architecture and modeling environment for the development of computational cognitive models. It is a psychologically validated cognitive architecture which has been used extensively in the modeling of higher-level cognitive processes (see the ACT-R web site for an extensive list of models and publications). ACT-R includes symbolic **production** and **declarative memory** systems integrated with subsymbolic **production selection** and **spreading activation** and **decay** mechanisms. Production selection involves the parallel matching of the left-hand side of all productions against a collection of **buffers** (e.g. goal buffer, retrieval buffer, visual buffer, auditory buffer) which contain the active contents of memory and perception. Production execution is a serial process—only one production is executed at a time. The parallel spreading activation and decay mechanism determines which declarative memory chunk is put into the retrieval buffer for comparison against productions. The combination of symbolic and subsymbolic mechanisms makes ACT-R a hybrid system of cognition. The **noise** parameter used by these computational mechanisms adds stochasticity to the system. ACT-R supports **single inheritance** of declarative memory chunks and limited, variable-based **pattern matching** (including a **partial-matching** capability). ACT-R incorporates **learning** mechanisms for learning both declarative and procedural knowledge. Version 5 of ACT-R (Anderson et al., 2002) adds a **perceptual-motor component** supporting the development of embodied cognitive models. With the addition of the perceptual-motor component, and the use of buffers as the interface between various cognitive modules (e.g. vision module, auditory module, production system, declarative memory), ACT-R is referred to as an "integrated theory of the mind".

Double R Model

Double R Model is the computational implementation of Double R Theory in ACT-R. Double R Model is currently capable of processing an interesting range of grammatical constructions including: 1) intransitive, transitive and ditransitive verbs; 2) verbs taking clausal complements; 3) predicate nominals, predicate adjectives and predicate prepositions; 4) conjunctions of numerous grammatical types; 5) modification by attributive adjectives, prepositional phrases and adverbs, etc. Double R Model accepts as input as little as a single word or as much as an entire chunk of discourse—using the perceptual component of ACT-R to read words from a text window. Unrecognized words are simply ignored. Unrecognized grammatical forms result in partially analyzed text, not failure. The output of the model is a collection of declarative memory chunks that represent the referential and relational meaning of the input text. Although Double R Model is essentially a computational psycholinguistic model, it is intended to be used as the basis for development of large-scale, functional language comprehension systems and

the current coverage of the model will need to be extended significantly to support that objective.

Inheritance vs. Unification

Unification allows for the unbounded, recursive matching of two logical representations. Unification is an extremely powerful pattern matching technique used in many language processing systems based on Prolog. Unfortunately, it is psychologically too powerful. For example, the following two logical expressions can be unified:

p(a,B,c(d,e,f(g,h(i,j),K),l))
p(X,b,c(Y,e,f(Z,T,U),l))

where capitalized letters are variables and lowercase letters are constants. Humans are unlikely to be capable of performing such unifications consciously or otherwise without significant effort and an external scratch pad (i.e. short-term working memory does not have the capacity to retain more than a few variable bindings simultaneously).

On the other hand, although extremely powerful, unification does not support the matching of types to subtypes. Thus, if we have a verb type with intransitive and transitive verb subtypes, unification cannot unify a chunk of type verb with a chunk of type intransitive verb or transitive verb. Unification's inability to match types to subtypes often results in a proliferation of rules (or conditions on rules) to handle the various combinations. For example, the verb type can be variableized and a test for the valid types can be used to constrain the variable (e.g. Verb-Type equal verb or Verb-Type equal intrans-verb or Verb-Type equal trans-verb). With inheritance, a production that checks for a verb type will also match a transitive verb and an intransitive verb type (assuming an appropriate inheritance hierarchy). Humans appear to be able to use types and subtypes in appropriate contexts with little awareness of the transitions. For example, when processing a verb, all verbs (used predicatively) expect to be proceeded by a subject, but only transitive verbs expect to be followed by an object. Thus, humans presumably have available a general production that applies to all verbs (or even all predicates) which will look for a subject preceding the verb, but only a more specialized production for transitive verbs (or transitive predicates) which will look for an object following the verb.

Inheritance supports the matching of two representations without requiring the recursive matching of their subparts (unlike unification) so long as the types of the two representations are compatible. Types are essentially an abstraction mechanism which makes it possible to ignore the detailed internal structure of representations when comparing them. For example, once the model has identified an expression as an object referring expression, the model can match the object referring expression against productions without consideration of the internal structure of the object referring expression. Of course, there may be productions that do consider the internal structure, but types are useful here as well. Instead of having to fully elaborate the internal structure, types can be used to partially elaborate that structure. For example, if a production is specifically concerned with object referring expressions headed by a quantifier (e.g. "some" in "some of the books"), the production can check to see that the

head is of the appropriate type, providing a (limited) unification like capability where needed.

In sum, inheritance and limited pattern matching provide a psychologically plausible alternative to a full unification capability.

To take advantage of inheritance, Double R Model incorporates a type hierarchy (a tangled hierarchy or lattice, with multiple inheritance, is preferred, but ACT-R currently only supports single inheritance). Representative elements of the top levels of the current hierarchy of types (below `top-type`) are shown below:

```
Lexical-type

    Pronoun
    Proper-noun
    Noun
    Adjective
    Verb
    Preposition
    Adverb
    Determiner
    Quantifier
    Auxiliary
    Negative

Referential-type

    Head
    Specifier
          Object Specifier
          Predicate Specifier
    Modifier
          Object Modifier
          Relation Modifier
    Complement

Referring-expression-type

    What-referring-expression
          Object-referring-expression
          Situation-referring-expression
          Predicate-referring-expression
    Where-referring-expression
          Location-referring-expression
          Direction-referring-expression
    When-referring-expression
    Why-referring-expression
    How-referring-expression
    How-much-referring-expression

Relation-type

    Relation
          Predicate
          Function
    Argument
    Term
```

The more specialized a production is, the more specialized the types of the chunks in the goal and retrieval buffers to which the production matches will need to be. The most general productions match a goal chunk whose type is `top-type` and ignore the retrieval buffer chunk.

Default Rules

ACT-R's inheritance mechanism can be combined with the subsymbolic production utility parameter—which influences production selection—to establish default rules. Since all types extend a base type (i.e. `top-type`), using the base type as the value of the goal chunk in a production will cause the production to match any goal chunk. If the production is assigned a production utility value that is lower than competing productions, it will only be selected if no other production matches (ignoring stochasticity). A sample default production is shown below:

```
(p process-default--retrieve-prev-chunk
   =goal>
   ISA top-type
   =context>
   ISA context
   state process
   chunk-stack =chunk-stack
   =chunk-stack>
   ISA chunk-stack-chunk
   this-chunk =chunk
   prev-chunk =prev-chunk
   ==>
   =context>
   state retrieve-prev-chunk
   chunk-stack =prev-chunk
   +retrieval> =chunk)
(spp process-default--retrieve-prev-chunk :p 0.75)
```

where the parentheses reflect the underlying lisp implementation, `p` identifies a production, `process-default--retrieve-prev-chunk` is the name of the production, `=goal>` identifies the goal chunk, `=context>` identifies a context chunk, `ISA context` is a chunk type, `state` is a chunk slot, `process` is a slot value, `==>` separates the left-hand side from the right-hand side and variables are preceded by `=` as in `=chunk`. This default production causes the previous chunk to be retrieved from declarative memory (using the `+retrieval>` form) if no other production is selected. To make this production a default production, the production utility parameter is set using the `spp` (set production parameter) command to a value of 0.75 (the default value is 1.0).

The Context Chunk and Chunk Stack

The current ACT-R environment provides only the goal and retrieval buffers (and perhaps the visual and aural buffers) to store the partial products of language comprehension. The lack of a stack is particularly constraining, since a stack is the

primary data structure for managing the kind of (limited) recursion that occurs in language. There needs to be some mechanism for retrieving previously processed words from short-term working memory in last-in/first-out order during processing (subject to various kinds of error that can occur in the retrieval process). A stack provides this (essentially error free) capability. It is expected that a capacity to maintain about 5 separate linguistic chunks in short-term working memory is needed to handle most input—supporting at least one level of recursion (and perhaps two for the more gifted). The goal chunk could be adapted for this purpose, except that it is also the basis for creation of new declarative memory chunks and activation spread and these architectural needs would conflict. Further, it would be difficult to get the kind of stack like behavior needed out of the slots in the goal chunk.

To overcome these problems, Double R Model introduces a context chunk containing a bounded, circular stack of links to declarative memory. As chunks are stacked in the circular stack, if the number of chunks exceeds the limit of the stack, then new chunks replace the least recently stacked chunks (supporting at least one type of short-term working memory error). The actual number of chunks allowed in the stack is specified by a global parameter. This parameter is settable to reflect individual differences in short-term working memory capacity. Chunks cannot be directly used from the stack. Rather, the stack is used to provide a template for retrieving the chunk from declarative memory. Essentially, the chunk on the stack provides a link to the corresponding declarative memory chunk. Since the chunk must be retrieved from declarative memory before use, the spreading activation and partial matching mechanisms of ACT-R are not circumvented and retrieval errors are possible—unlike the goal stack of earlier versions of ACT-R. Thus, the bounded, circular stack of links to declarative memory avoids the arguments against the goal stack of earlier versions of ACT-R, adds the insight of activated pathways to declarative memory, and retains the insights that motivated the inclusion of a goal stack in earlier versions of ACT-R.

Besides storing the chunk stack, the context chunk is also used to separate out state information from the goal chunk. Since the goal chunk is the basis for creating new declarative memory chunks, storing the chunk stack in it would result in the chunk stack being stored with each new declarative memory chunk. While this might be used to support a kind of episodic memory where the context in which a word occurs is stored with the declarative memory chunk created during the processing of the word, ACT-R 5.0 does not currently provide a mechanism for transitioning episodic memory into semantic memory (i.e. abstracting from the context of use), and storing the context with a chunk has undesirable side-effects within the ACT-R environment (e.g. it interferes with the spreading activation mechanism). To avoid such problems a separate context chunk is maintained and made available to all productions. Although, the existence of a separate context chunk that productions match to violates the ACT-R 5.0 architecture where only the buffers are supposed to be used for this purpose, earlier versions of ACT-R allowed multiple chunks to be matched on the left-hand side of productions and this functionality is still available in ACT-R 5.0 environment.

The context chunk maintains several pieces of information in addition to the chunk stack. Its definition (as specified by a `chunk-type`) in the model is shown below:

```
(chunk-type context state rel-context sit-context text-context word
prev-word-1 prev-word-2 repeat chunk chunk-stack)
```

In this chunk-type definition, context is the name of the chunk, state is a slot that provides state information to guide production selection, rel-context is a slot that identifies the current relational context (typically determined by a specifier), sit-context is a slot that contains information about the current situation context, text-context is a slot that contains information about the larger discourse context, word contains the lexical item being processed, word-prev-1 and word-prev-2 contains the previous two words processed, repeat is yes if the word has been attended to previously and no-more if there are no more words in the input, chunk contains the most recently processed chunk, and chunk-stack contains the entire chunk stack.

Lexical and Functional Entries

The lexical entries in the model provide a limited amount of information which is stored in the word and word-info chunks. The definition of the word and word-info chunk types are provided below:

```
(chunk-type word word-form word-marker)
(chunk-type word-info word-marker word-root word-type word-subtype word-morph-type)
```

The word-form slot of the word chunk contains the physical form of the word (represented as a string in ACT-R); the word-marker slot contains an abstraction of the physical form. The word-root slot contains the value of the root form of the word. The word-type slot contains the lexical type of the word and is used to convert a word-info chunk into a lexical-type chunk for subsequent processing. A word-subtype slot is provided as a workaround for the lack of multiple inheritance in ACT-R 5.0. The word-morph-type slot supports the encoding of additional grammatical information (although that information is not currently being used).

Sample lexical entries for a noun and verb are provided below:

```
(cow-wf isa word
    word-form "cow"
    word-marker cow)
(cow isa word-info
    word-marker cow
    word-root cow
    word-type noun
    word-morph-type third-per-sing)

(running-wf isa word
    word-form "running"
    word-marker running)
(running isa word-info
    word-marker running
    word-type verb
    word-root run
    word-subtype intrans-verb
    word-morph-type pres-part)
```

Note that there is no indication of the functional roles (e.g. head, modifier, specifier, predicate, argument) that particular lexical items may fulfill. Following conversion of `word-info` chunks into `lexical-type` chunks (e.g., `verb`, `adjective`), functional roles are dynamically assigned by the productions that are executed during the processing of a piece of text. Since functional role chunks are dynamically created, only `chunk-type` definitions exist for functional categories prior to that processing. As an example of a `chunk-type` definition for a functional category, consider the category `pred-trans-verb` (i.e. transitive verb functioning as a predicate) whose definition involves several hierarchically related `chunk-types` as shown below:

```
(chunk-type top-type head)
(chunk-type (rel-type
    (:include top-type)))
(chunk-type (pred-type
    (:include rel-type))
       subj spec mod post-mod)
(chunk-type (pred-trans-verb
    (:include pred-type)) obj)
```

The `top-type` `chunk-type` contains the single slot `head`. All types are subtypes of `top-type` and inherit the `head` slot. `Rel-type` is a subtype of `top-type` that doesn't add any additional slots. `Pred-type` is a subtype of `rel-type` that adds the slots `subj`, `spec`, `mod`, and `post-mod`. It is when a relation is functioning as a predicate that these slots become relevant. `Pred-trans-verb` is a subtype of `pred-type` that adds the slot `obj`. Summarizing, `pred-trans-verb` contains the slots `head`, `subj`, `spec`, `mod` (i.e. pre-head), `post-mod` (i.e. post-head), and `obj`, all of which are inherited from parent types except for the `obj` slot.

The following production creates an instance of a `pred-trans-verb` and provides initial values for the slots:

```
(p process-verb--convert-to-pred-trans-verb
 =goal>
   ISA verb
   head =verb
   subtype trans-verb
   =context>
   ISA context
   state convert-verb-to-pred-verb
   ==>
   +goal>
   ISA pred-trans-verb
   subj none
   spec none
   mod none
   head =goal
   post-mod none
   obj none
   =context>
   state retrieve-prev-chunk)
```

In this production, a verb (subtype of lexical-type) whose subtype slot has the value trans-verb is converted into a pred-trans-verb for subsequent processing. The only slot of pred-trans-verb that is given a value other than none is the head slot whose value is set to be the goal chunk (i.e. head =goal). This production has the effect of assigning a transitive verb the functional role of predicate (specialized as a transitive verb predicate). Its selection and execution is based on the previous context which set the value of the state slot of the context chunk to convert-verb-to-pred-verb and on having a goal chunk of type verb whose subtype slot has the value trans-verb.

Productions

Sample productions were shown above in the discussion of default rules and in the creation of functional roles. This section provides some additional examples. The read-next-word production initiates the find-attend-encode sequence for reading the next word from the computer screen (using ACT-R's perceptual component).

```
(p read-next-word
   =goal>
   ISA word
   =context>
   ISA context
   state start
   - repeat no-more ;; no more words
   ==>
   =context>
   state find)
```

Note that the goal is represented by the declarative word chunk as opposed to a more procedurally oriented goal chunk like read-word. This is fallout from the fact that the goal chunk is the basis for creating declarative memory chunks. The start value of the state slot in the context chunk is the primary basis for the selection of this production (along with the type of the goal chunk). Again, putting the state slot in the context chunk avoids the need to encode procedural information in the goal chunk. The "-repeat no-more" entry in the production indicates that this production only applies if the value of the repeat slot is not (negation is indicated by the "−") no-more where no-more indicates that the last word in the text has already been read.

The next production uses the word-marker slot of the word chunk to retrieve the word-info chunk.

```
(p retrieve-word-info
   =goal>
   ISA word
   word-marker =word-marker
   =context>
   ISA context
   state retrieve
   ==>
   +retrieval>
   ISA word-info
   word-marker =word-marker
```

```
   =context>
   state retrieve-word-info)
```

The `word-info` chunk is then used to create a `lexical-type` chunk (e.g. `verb`) which becomes the goal. The productions that convert `word-info` chunks into `lexical-type` chunks are special in that the `:effort` parameter is set to 0.0. The `:effort` parameter determines how long it takes a production to execute (default is 0.05 sec or 50 msec). Setting the value to 0.0 means that the production takes no time to execute. The presumption is that the procedure that effects this conversion is substituting for an automated type conversion mechanism that the ACT-R 5.0 modeling environment does not currently provide.

```
(p convert-word-to-verb
   =goal>
   ISA word-info
   word-type verb
   word-subtype =word-subtype
   =context>
   ISA context
   state convert
   ==>
   +goal>
   ISA verb
   head =goal
   subtype =word-subtype
   =context>
   state process)
(spp convert-word-to-verb :effort 0.0)
```

There are a few other kinds of "housekeeping" productions which are accorded zero effort in the model (e.g. the stack chunking procedures). In general, "housekeeping" productions are used to effect various data manipulations that are external to the basic processing mechanism. The `process-verb--convert-to-pred-trans-verb` production discussed above is another example of a "housekeeping" production.

The next production matches a `verb` goal chunk and in the context of an `obj` (i.e. object referring expression) converts the `verb` type into a `rel-head` type

```
(p process-verb--obj-context--convert-to-rel-head
   =goal>
   ISA verb
   head =verb
   =context>
   ISA context
   state retrieve-prev-chunk
   rel-context obj
   ==>
   +goal>
   ISA rel-head
   mod none
   head =goal
   post-mod none)
```

`Rel-head` (i.e. `relational-head`) is a subtype of `head`. The next production matches a `head` goal chunk (which could be a `rel-head`) and an `obj-spec` (i.e. `object-specifier`) retrieval chunk and creates a new `obj-refer-expr` (i.e. `object-referring-expression`) which becomes the goal. Together, these two productions support to use of verbs as (relational) heads of object referring expressions following an object specifier (e.g. "kick" in "the kick").

```
(p process-head--prev-chunk-is-obj-spec
   =goal>
   ISA head
   =context>
   ISA context
   state retrieve-prev-chunk
   =retrieval>
   ISA obj-spec
   ==>
   +goal>
   ISA obj-refer-expr
   spec =retrieval
   mod none
   head =goal
   post-mod none
   referent none-for-now
   =context>
   state process
   rel-context none)
```

The creation of an object referring expression causes the value of the `rel-context` slot to be set to `none` indicating the end of the object referring expression context.

Context Accommodation vs. Backtracking

Context accommodation is a mechanism for changing the function of an expression based on the context without backtracking. For example, when an auxiliary verb like "did" occurs it is likely functioning as a predicate specifier as in "he did not run" where the predicate is "run" and "did not" provides the specification for that predicate. However, auxiliary verbs may also function as predicates when they are followed by a noun phrase as in "he did it". Determining the ultimate function of an auxiliary verb can only be made when the expression following the auxiliary is processed. In a backtracking system, if the auxiliary verb is initially determined to be functioning as a predicate specifier, then when the noun phrase "it" occurs, the system will backtrack and reanalyze the auxiliary verb, perhaps selecting the predicate function on backtracking. However, note that backtracking mechanisms typically lose the context that forced the backtracking. Thus, on backtracking to the auxiliary verb, the system has no knowledge of the subsequent occurrence of a noun phrase to indicate the use of the auxiliary verb as a predicate. Thus, the system can only randomly select a new function for the auxiliary verb which may or may not be that of a predicate.

A better alternative is to accommodate the function of the auxiliary verb in the context which forces that accommodation. In this approach, when the noun phrase "it" is processed and the auxiliary verb functioning as a predicate specifier is retrieved, the

function of the auxiliary verb can be accommodated in the context of a subsequent noun phrase to be a predicate. Context accommodation avoids the need to backtrack and allows the context to adjust the function of an expression just where that accommodation is supported by the context. Of course, there may cases where the context accommodation mechanism breaks down and some form of backtracking is needed (e.g. garden-path sentences), but in such cases backtracking is likely to involve a jump back to the beginning of a major constituent (e.g. clause) and some contextual information will be carried back with the jump. In any case, a reverse-depth-first, context-unraveling backtracking mechanism like that provided in Prolog is psychologically implausible.

Processing Example

As an example of the processing of a piece of text and the creation of declarative memory chunks to represent the meaning of the text, consider the processing of the following text:

> The old dog lover is asleep.

The processing of the word "the" results in the creation of the following declarative memory chunks:

```
Goal25
    isa DETERMINER
    head The
Goal26
    isa OBJ-SPEC
    head Goal25
    mod None
```

The first chunk, `goal25,` is a `determiner` whose `head` slot has the value `The`. This chunk represents the inherent part of speech of the word "the". The second chunk, `goal26`, is an `obj-spec` (i.e. `object-specifier`) whose `head` slot has the value `goal25` and whose `mod` slot has the value `none`. This second chunk represents the function of "the" in this particular text. Note that if "the" were the only word in the input text, the creation of these two chunks would still occur since the processing mechanism works bottom-up from the lexical items and makes no assumptions about what will occur independently of the lexical items.

The processing of the second word "old" leads to the creation of the following declarative memory chunks:

```
Goal32
    isa ADJECTIVE
    head Old
Goal33
    isa OBJ-MOD
    head Goal32
    mod None
```

where `goal32` represents the inherent part of speech of "old" and `goal33` represents the function of "old" (i.e. `object-modifier`) in the current context.

The processing of the third word “dog” creates the following declarative memory chunks:

```
Goal39
    isa NOUN
    head Dog
Goal40
    isa HEAD
    head Goal39
    mod Goal33
    post-mod None
Goal41
    isa OBJ-REFER-EXPR
    head Goal40
    referent None-For-Now
    spec Goal26
    mod None
    post-mod None
```

`Goal41` is a full object referring expression and contains a `referent` slot to support a link to an object in the situation model corresponding to this piece of text. The model does not currently establish the value of the `referent` slot since this capability is not yet implemented.

The processing of the fourth word “lover” creates or modifies the following declarative memory chunks:

```
Goal47
    isa NOUN
    head Lover
Goal48
    isa HEAD
    head Goal47
    mod Goal40
    post-mod None
Goal41
    isa OBJ-REFER-EXPR
    head Goal48
    referent None-For-Now
    spec Goal26
    mod None
    post-mod None
```

Note that `goal47` (i.e. “lover”) is now the head of `goal48` which is the head of the `object-referring-expression` (i.e. `goal41`) with `goal39` (i.e. “dog”) functioning as the head of `goal40` which is functioning as a modifier of `goal48`. Also, note that `goal33` (i.e. “old”) modifies `goal39` (i.e. “dog”) and not `goal48` (i.e. “dog lover”). This is equivalent to the expression “the lover of old dogs” rather than “the old lover of dogs”, both of which are possible interpretations. Having “old” modify “dog” rather than “dog lover” is probably not the preferred interpretation of this piece of text, but it does point out the advantage of having an implemented model to make such decision points apparent. The current model can be modified to support the alternative interpretation by addition of a production that has the intended effect, however, there is currently no

mechanism for preferring this new production over the existing production. Such a mechanism would need to be able to distinguish between collocations like "old dog lover" and collocations like "old house renovator" where the modification works the other way.

Continuing with the next word "is" leads to the creation of the following chunks:

```
Goal54
    isa REG-AUX
    head Is-Aux
Goal55
    isa PRED-SPEC
    head Aux-1
    mod None
    modal-aux None
    neg None
    aux-1 Goal54
    aux-2 None
    aux-3 None
```

Note that the `pred-spec` (i.e. `predicate-specifier`) chunk type has a `modal-aux`, `neg`, and three auxiliary slots (`aux-1`, `aux-2`, and `aux-3`) to handle the range of predicate specifiers that can occur in English. For this instance of a `pred-spec` (i.e. `goal55`), `goal54` fills the `aux-1` slot and functions as the head of `goal55`.

The processing of the final word "asleep" creates the following declarative memory chunks:

```
Goal61
    isa ADJECTIVE
    head Asleep
Goal62
    isa PRED-ADJ
    head Goal61
    subj Goal41
    spec Goal55
    mod None
    post-mod None
```

In the context of the predicate specifier "is" (i.e. `goal55`), the adjective "asleep" functions as a predicate adjective filling the `head` slot of `goal62`. `Goal41` (an object referring expression) fills the `subj` slot (i.e. subject) of `goal62`.

Following the processing of "asleep" the model attempts to read the next word. The failure to read a word signals the end of processing and a wrap-up production is executed. This production converts `goal62` into a situation referring expression resulting in the creation of `goal65` with `goal62` filling the `head` slot.

```
Goal65
    isa SIT-REFER-EXPR
    head Goal62
    referent None-For-Now
    mod None
```

At the end of processing a single chunk of type `situation-referring-expression` is available in the `chunk-stack` to support subsequent processing.

Summary and Future Research

Double R Model may be the first attempt at the development of a Natural Language Understanding system founded on the principles of Cognitive Linguistics and implemented in the ACT-R cognitive modeling environment. Much work remains to be done. Double R Model has not yet reached a scale at which it can handle more than a token set of English. To expand the symbolic capabilities of Double R Model we are evaluating the integration of the CYC knowledge base (Lenat et al., 2003), WordNet (Miller et al., 2003), and FrameNet (Fillmore et al., 2003). CYC could provide the basis for creation of a situation model to ground the referring expressions in a text, thereby, supporting a fuller representation of referential meaning. WordNet will support the expansion of the lexicon to a full complement of lexical items. FrameNet, with some mapping to Double R Grammar, could provide constructional schemas for relational and referential lexical items. To expand the subsymbolic capabilities of Double R Model (e.g. in support of lexical disambiguation), we are evaluating the use of Latent Semantic Analysis (LSA) (Landauer et al.), and considering improvements to ACT-R's single-level spreading activation mechanism. In this regard, LSA might provide an empirical basis for determining the strength of association of declarative memory chunks and multiple-level spreading activation (like that proposed in earlier ACT* theory, Anderson, 1983) would eliminate the need for direct association of all related declarative memory chunks.

References

Anderson, J. R. (1983). *The Architecture of Cognition*. Cambridge, MA: Harvard University Press.

Anderson, J. & Lebiere, C. (1998). *The Atomic Components of Thought.* Mahway, MJ: LEA.

Anderson, J., Bothell, D., Byrne, M. and LeBiere, C (2002). *An Integrated Theory of the Mind.* http://act-r.psy.cmu.edu/papers/403/IntegratedTheory.pdf

Ball, J (2003a). "Towards a Semantics of X-Bar Theory." http://www.DoubleRTheory.com/papers/other/SemanticsOfXBarTheoryPDF.pdf

Ball, J. (2003b). "Is the Head of a Noun Phrase necessarily a Noun." Presentation at ICLC 2003. http://www.DoubleRTheory.com/presentations/doubler/ICLC Presentation.pps

Chomsky, N. (1965). *Aspects of the Theory of Syntax*. Cambridge, MA: The MIT Press.

Chomsky, N. (1957). *Syntactic Structures*. The Hague: Mouton.

Chomsky, N. (1970). "Remarks on Nominalization." In R. Jacobs & P. Rosembaum, eds., *Readings in English Transformational Grammar*. Ginn, Waltham, MA.

Fillmore et al. (2003). FrameNet. http://www.icsi.berkeley.edu/~framenet/

Kintsch, W. (1998). *Comprehension, a Paradigm for Cognition.* New York, NY: Cambridge University Press.

Lakoff, G. (1988). "Cognitive Semantics." In *Meaning and Mental Representation*. Edited by U. Eco, M. Santambrogio & P. Violi. Indianapolis: Indiana University Press.

Lakoff, G. (1987). *Women, Fire and Dangerous Things.* Chicago: The University of Chicago Press.

Lakoff, G., & M. Johnson (1980). *Metaphors We Live By.* Chicago: The University of Chicago Press.

Landauer et al. (2003). Latent Semantic Analysis (LSA). http://lsa.colorado.edu/

Langacker, R. (1987). *Foundations of Cognitive Grammar, Volume 1, Theoretical Prerequisites.* Stanford, CA: Stanford University Press.

Langacker, R. (1991). *Foundations of Cognitive Grammar, Volume 2, Descriptive Applications.* Stanford, CA: Stanford University Press.

Lenat et al. (2003). CYC. http://www.cyc.com

Miller et al. (2003). WordNet. http://www.cogsci.princeton.edu/~wn/

Rosch, E. (1978). "Principles of Categorization." In *Cognition and Categorization*. Edited by E. Rosch & B. Lloyd. Hillsdale, NJ: LEA.

Talmy, L. (2003). *Toward a Cognitive Semantics, Vols I and II.* Cambridge, MA: The MIT Press

Townsend, D. and T. Bever (2001). *Sentence Comprehension.* Cambridge, MA: The MIT Press.

Wittgenstein, L. (1953). *Philosophical Investigations*. New York: MacMillan.