American Journal of Computational Linguistics *Microfiche 33 : 18*

INCREMENTAL SENTENCE PROCESSING

RODGER KNAUS

Systems Software Division
Social and Economic Statistics Administration
Bureau of the Census
Washington, D. C. 20233

A human who learns a language can both parse and generate sentences in the language. In contrast most artificial language processors operate in one direction only or require separate grammars for parsing and generation. This paper describes a model for human language processing which uses a single language description for parsing and generation.

1. Choice of Parsing Strategy

A number of constraints limit the processors suitable as models of human language processing. Because short term memory is limited, the listener must absorb incoming words into larger chunks as the sentence is heard. Also because he is expected to reply within a couple seconds after the speaker finishes, regardless of length of the speaker's utterance, the listener must do much of the semantic processing of a sentence as he hears it.

Bever and Watt point out that the difficulty in understanding a sentence S is not predicted by the number of transformations used to generate S. Furthermore the process of detransformation appears too time-consuming (Petrick) for the approximately two seconds before a listener is expected to reply.

A depth first transition network parser (Woods, Kaplan), in which parsing difficulty is measured by the number of arcs traversed, correctly predicts the relative difficulty of active and passive sentences progressive and adjectival present participle sentences and the extreme difficulty of multiple center embeddings. However a syntactically directed depth first parser does not explain why syntactically similar sentences such as

(5A) The horse sold at the fair escaped.

(5B) The horse raced past the barn fell.

vary in difficulty, nor does it explain experiments on the completion and verification of ambiguous sentences (MacKay, Olsen and MacKay) which suggest that a pruned breadth first strategy is used to parse sentences. Sentences with two equally plausible alternatives took longer to process than sentences with only one likely interpretation. This extra processing time may be attributed to the construction of two alternate interpretations over a longer portion of the sentence when more than one interpretation is plausible.

In addition subjects sometimes become confused by the two interpretations of an ambiguous sentence. Finally in experiments in which subjects hear an ambiguous sentence in one ear and a disimbiguating sentence simultaneously in the other ear (Garrett) the interpretation of the ambiguity actually perceived by the subject may be switched between the possibilities by changing the disambiguating sentences.

```
Step 3 (a):  (S (NP (N mail) (N Boxes))
                (V like) (NP) (PP*))
       (b):  (S (NP (NP (N mail) (N Boxes))
                    (PP (PREP like) NP) (PP*))
                V(NP) (PP*))
       (c):  (S (NP (N mail)) (V Boxes)
                (PP (PREP like) NP) (PP*))
       (d):  (S (V mail) (NP (N Boxes))
                (PP (PREP like) NP) (PP*))
       (e):  (S (V mail)
                (NP (NP (N Boxes))
                    (PP (PREP like) NP) (PP*))
                (PP*))
```

After completing the sentence after Step 4, the parser produces phrase markers from a, c, d and e by adding the last word and deleting unfilled optional nodes. The phrase marker obtained from 4B is rejected because it contains an unfilled obligatory V node.

The incremental parser adds each successive sentence word to the partially completed phrase markers built from the earlier part of the sentence. The new word is added at the leftmost oblig unfilled node of each partial phrase marker and at all optional nodes to the left of this node.

Three different operations are used to add a new word to a partial parse. The word may be directly added to an unexpanded node, as in Step 3a above. Alternatively, a new word may be <u>attached</u> to an unfilled node with a left branching acyclic tree built from the grammar such as (PP PREP NP) or (S (NP N (N*)) V (NP) (PP*)). Attaching occurs in steps 1 and 3c.

Finally a subtree of an existing partial phrase marker may be left <u>embedded</u> in a larger structure of the same grammatical category, as in steps 3b and 3e above. The embedding operation uses at most two left branching trees built from the grammar: a tree T1 with a single cycle on the left branch is used to replace the existing subtree E being embedded. In step 3e, for example, the structure (S (V mail) (NP NP (PP*)) (PP*)) would be obtained. The E is used to expand the left-most unexpanded node of T1; for 3 b this results in:

3e. (S (V mail) (NP (NP (N Boxes) (N*)) PP*) (PP*)).

Finally to the resulting structure the new sentence word is added through direct node expansion or attaching with an acyclic left branching tree; in the example above this produces 3e from 3e.'

Using direct expansion attaching and embedding, the incremental parser finds all the phrase markers of sentences in context free or regular expression language; a formal definition of the parser and a proof of its correctness appear in [10].

Sometimes, as at steps 3b and 3e, the same structure (a prepositional phrase in step 2) is used in more than one partial parse. Following Earley's Algorithm, the incremental parser builds a single copy of the shared substructure SØ and maintains pointers linking SØ to nodes in larger structures which SØ expands.

For all its tree building operations the incremental parser uses a finite set of trees. i.e., the trees with only left sub-nodes expanded and at most one cycle on the leftmost branch. These trees may be computed directly from the grammar and ref-erenced by root and leftmost unexpanded node during the parse.

Using these preconstructed trees, the incremental parser requires only a fixed number of operations to add a new word to a partial parse: a retrieval on a doubly indexed set, copying the left branching tree, and at most four structure changing operations to paste words and trees together.

Like Earley's Algorithm, IP processes each word proportionally to sentence length. However on sentences satisfying a <u>depth difference</u> <u>bound</u>, the parsing time per word is constant. Because humans can't remember large numbers of sentence words but must process speech at an approximately constant rate, a constant parsing time per word is a necessary property of any algorithm modeling human language processing.

Let the depth of constituent C in phrase marker P be defined as the length of the path from the root of C to the root of P. If T1 and T2 are two adjacent terminals with T1 preceding T2, the depth difference from T1 to T2 is defined as the difference in depth between T1 and the root of the smallest tree containing T1 and T2. For example in the phrase marker

```
    (9)  (S (NP (NP (DET the) (N telephone))
                (PP (PREP IN) (NP (DET the) (N room)))
                (V rang) (ADV loudly))
```

the depth difference between "the" and "telephone" is 1 and between "room" and "rang" is 3.

The depth difference between T1 and T2 is the number of nodes from T1 to the node expanded when adding T2 on a postorder traversal from T1 in the partial phrase marker containing T1 but not T2. The depth difference between T1 and T2 also represents the number of constituents of which T1 is the rightmost word.

A proof (requiring a formal definition of the incremental parse) that parsing time per word is constant in depth difference bounded sentences appears in [10]. Informally the depth difference bound places a bound both on the number of next nodes to expand which may follow a given terminal and on the amount of tree traversal which the parser must perform to find each next unexpanded node. Since each modification requires only a fixed number of operations, each of which is bounded on the finite set of at most once cyclic left branching trees, the computation adding a new word to existing partial parses is bounded independently of sentence length.

Natural language sentences tend to have small depth differences. Both right branching sentences and left branching sentences (found in Japanese for example) have an average depth difference over each three or four word segment of two or less. On the other hand sentences are difficult to understand when they have two consecutive large depth differences, such as the multiple center embedding

(10) The rat the cat the dog bit chased died.

or the complex noun phrase in

The pad on a clarinet in the last row which I
fixed earlier for Eb fell out.

Furthermore in ambiguous sentences such as

(11) Joe figured that it was time to take the cat out.

Kimball observes that subjects prefer the reading with the smaller depth difference. Finally, Blumenthal found that subjects tended to understand a multiple center embedded sentence as a conjunctive sentence. The conjunctive sentence contains a rearrangement with lower depth differences of the constituents of the center embedded sentence.

3. Sentence Generation

The syntactic form given to a sentence depends on the information being communicated in a sentence and on the cultural context in which the sentence appears. Clark and Haviland show that a speaker uses various syntactic devices sentences to place the "given" information known to the listener before the information "new" to the listener. Particular syntactic structures are also used to emphasize or suppress particular kinds of information; for example newspaper traffic accident reports usually begin with a passive sentence such as

(12) An elderly Lakewood man was injured when...,
presumably to emphasize the result of the
accident.

To capture the dependence of syntax on semantic content and social context, the sentence generator uses function-like grammar rules of the form

(Rulename Cat Variables Predicate Forms).

Rulename is the name of the rule and cat is the grammatical category of the constituent generated by the rule.

Variables is a list of formal parameters. Usually the variable list contains a variable bound during rule execution to a node in a semantic network and another variable bound to a <u>control association list</u> containing information about the context in which the generated constituent will appear and possibly the syntactic form the constituent should have.

Predicate is a Boolean-valued form on the parameters in Variables. A rule is used only when Predicate is true.

Forms is a list of forms depending on Variables which generate terminals or calls to the grammar for subconstituents of CAT.

An example of a generation rule is

```
(SPI S.(X Y) (Equal (Voice Y) (Quote Passive))
             (NP (Object X) Y)
             (Beverb X)
             (Pap (Action X))
             (M* X Y))
```

which generates simple passive sentences. The variable X is bound to a node in a semantic network and Y to a control association list. The rule is applied only if the control alist contains a passive flag and if the semantic node has an object and action; in general a rule is applied only if the semantic subnodes called in the rule body appear in the semantic net. The form (NP (Obj X) Y) generates a form (NP XØ YØ), where XØ is the semantic node on the object indicator from X, and YØ is the value of Y. Beverb and Pap are procedures which generate respectively a form of the verb "to be" and a past participle form of the verb Action(X). M* is a procedure which generates a list depending on X and Y such as (PP<Value of Time(X)> <Value of Y>) for generating optional prepositional phrases or relative clauses.

As each rule is applied, the list of terminals and calls to grammar rules generated by the rule is added to a phrase marker representing the structure of the sentence being generated.

Grammar calls in the phrase marker are expanded top down and left to right, in a preorder traversal of the growing phrase marker. As terminals are generated they are printed out.

As an example, illustrating the effect of semantic and social contest on sentence generation, an initial sentence of a traffic accident report,

(13). A man was killed when a car hit him in Irvine.

was generated from the semantic nodes

```
A1:  Agent    AØ        A2:  Agent          AØ:  Class man
     Object   VØ             Action hit
     Action Kill             Object VØ
     Place Irvine            Instrument Car
     Cause AZ
```

and the control alist.

Purpose: Introduction; cases: object, cause, place

using a grammar built for generating traffic accident report sentences. To summarize a trace of the generation, a call to the sentence rule with purpose = introduction generates a sentence call with voice = passive. The passive rule applies and a noun phrase on AØ is called for. Because Purpose = Introduction a NP rule applies which calls for a NP to be generated on the semantic class to which AØ belongs. Because CASES contains TIME and CAUSE, the passive rule generated calls for modifying structures of these CASEs. Because the cause semantic node A2 has an action, the modifier rule M => Relative conjunction S generates the cause while the time is described by a prepositional phrase. The pronoun "him" is generated by a noun phrase rule NP-1 which generates a pronoun when the first semantic argument to the left of the NP-1 call in the generation phrase marker which is described by the same pronoun as the semantic argument A of NP-1 is in fact equal to A.

4. Finding Semantic Preimages

While the generator described in section 3 produces sentences from semantic and contextual information, the incremental parser described in section 2 recovers merely the syntactic structure of a sentence. To obtain the semantic arguments from which a sentence might have been generated a procedure to invert the generation rule forms must be added to the incremented parser.

While the incremental parser begins the construction of constituents top down, it completes them syntactically in a bottom up direction. In fact IP executes postorder traversals on all the syntactic parse trees it builds; of course if a particular partial phrase marker can not be finished, the traversal is not completed. However each node not a <u>tree</u> terminal of a syntactic phrase marker visited by the incremental parser is a syntactically complete constituent.

When the parser visits a syntactically complete constituent C, it applies a function INVERT to find the semantic preimages of C. In finding the semantic structure of C, INVERT has available not only the syntactic structure of C, but also the semantic preimages which it found for subconstituents of C. INVERT finds the set of generation rules which might produce a constituent having the same syntactic form as C. For each such rule R., INVERT constructs all the possible parings between each output-generating form F of R and the constituents of C which F might produce. For example if C is

```
    (S (NP Man) (Beverb is) (PAP Injured))
```

the pairing established for the passive sentence rule would be

```
    (NP (Object X) Y)        (NP the man)
    (Beverb X)               (Beverb is)
    (Pap (Action X))         (Pap Injured)
    (M* X Y)                 NIL
```

The pair ((Equal (Voice Y) PASSIVE) T) is also created, since the rule predicate is true whenever a rule applies.

Each indicidual pair P in such a pairing of a rule form and rule form outputs is processed by a function FIND which returns an association list containing possible values of the rule parameters (X and Y in the example above) which would produce the output appearing in P. For the example above FIND would produce

```
    (( X ((Object Man))) (Y NIL))
    (( X ((Time Past))) (Y NIL))
    (( X NIL) (Y (( Cases Nil)))).
    (( X NIL) (Y (( Voice Passive)))
```

Using an extension to association lists of the computational logic Unification Algorithm, these association lists are unified into a single association list, which for the example is

```
    (( X ((Agent man) (Time Past) (Action Injure))
    (( Y ((Cases Nil) (Voice Passive))))
```

Finally INVERT creates a grammar rule call,

```
    (S ((Agent man)(Time Past)(Action Injure))
       ((Cases Nil)(Voice Passive))))
```

from the association list and stores the result in the inverse image of C.

In finding a semantic preimage, the INVERT function must know which grammar rules might produce a particular grammatical constituent. This information is computed by symbolically evaluating the grammar rules to produce the strings of regular expression grammar nonterminals (as opposed to grammar calls) representing the possible output of each rule. The resulting relation from rules to strings is inverted into a table giving possible rules generating each string.

The heart of this symbolic evaluator is a function ETERM on the output generating forms of a rule which returns a list all lists of regular expression nonterminals representing the output of a form. ETERM takes advantage of the similar syntax of most grammar rule forms, and is defined in simplified form (with comments in angle brackets) as

```
Eterm (form) =
    if atom (form) then NIL
    <terminates recursion>
    else if car (form) is a grammatical category
      then list (list (car (form)))
      <these forms generate a single grammar call>
    else if car (form) = FUNCTION or LAMBDA
      then ETERM (cadr (form))
    else if car (form) = LAMBDA
      then ETERM (caddr (form))
    else if car (form) = LIST
      if form is not properly contained in a LIST
      expression
      then Mapcar((Function Concatenate)
                  (Cartesian
                 ((Mapcar (Function ETERM)
                          cdr (form))))
     <outer LISTS are used to create lists of grammar calls>
    else if form is inside a LIST expression
      ETERM (cadr (form))
      <inner lists are used to create grammatically>
    else if car (form) = MAPCONC then make optional
         and repeatable all the nonterminals returned
         in ETERM ([function argument of MAPCONC])
```

```
else if car (form) = COND
    then MAPCONC((LAMBDA(X) ETERM ([last form in X])
                 (cdr form)
    <returns alternatives from each branch of the COND>
else if car (form) is a user-defined function
    then ETERM ([definition of function])
else if there is a stored value for ETERM (form)
    then that value
else ask the the user for help
```

The function FIND which returns possible bindings for rule variables when given a rule form and its output is defined below. The variable ALIST holds the value of the association list being hypothesized by FIND; this variable is NIL when FIND is called from INVERT.

Like ETERM, the definition of FIND is based on the rules for evaluating recursive functions.

```
FIND (Alist form value)=
    if eval(form alist)=value then list (Alist)
    else if recursion depth exceeded, then NIL
    else if atom (form) then list (Merge (list (cons
                                  (form Value)) Alist)
    else if car (form)= COND
        let L = clauses which might be entered by
                evaluating form
        then Mapconc (FM 1)   where
        FM (clause) = list (Merge Find (Alist Car (clause)T)
                                  Find (Alist last (clause)))
    else if car (form) = Quote then if cadr (form) = value
                                        then Alist else NIL
    else if car (form) is a defined function
        then FIND (Alist (Substitute cdr (form) for
                          formal parameters in definition
                          of car (form))
                      Value)
    else if car (form) = MAPCONC (fn lst)
        then Merge (Find (Alist lst value)
                    For each X in lst, Merge (Alist for X))
    <this clause makes the assumption, which works in
      practice, that fn generates either one-element
      or empty lists>
    else NIL
```

With a definition of FIND similar to the one above, the parser found the preimage

```
(S (((place ((class (park))))
    (agent ((class (man))))
    (action (walked]
```

[the extra parentheses denote lists of alternatives] for the sentence

(13) The man walked in the park.

generated by the grammar

```
[SØ S (X) T (NP (Agent X)) (V(Action X))
            (Optional ((PP (Place X) ((Case Place]
[NPØ NP (X) T (Det X) (N (Class X]
[PPØ PP (XY) T (Prep XY) (NPX]
```

and the preposition function

```
Prep (XY) = Selectq (Assoc CASE Y)
                    (Place IN)
                    (Instrument WITH)
                    (Source FROM]
```

5. Implementation

The processors described in this paper have been programmed in University of California, Irvine, LISP and run in about 45K on a PDP-10 computer.

References

1. Bever, Thomas G. 1970. In [7] and [5].
2. Clark, Herbert H. and Haviland, Susan E. 1975 Social Sciences Working Paper, 67. U.C. Irvine.
3. Colby, Benjamin N. 1973. American Anthropologist 75, 645-62.
4. Florres d'Arcaio and Levalt, eds. 1970 Advances in Psycholinguistics, North Holland, Amsterdam.
5. Garrett, Merrill, F. 1970. in [5].
6. Haynes, John R. 1970. Cognition and the Development of Language. John Wiley.
7. Kaplan, Ronald M. 1972. A.I. 3, 77-100
8. Kimball, John 1974. Cognition 2,1,15-47.
9. Knaus, Rodger. 1975. Ph.D Thesis. U.C. Irvine.
10. MacKay, Donald G. 1966. Perception and Psychophysics. 426-36.
11. Olson, James N. and MacKay, Donald G. JVLVB 13, 45770.
12. Petrick, S. R. In [14].
13. Rustin, Randall. 1973. Natural Language Processing.
14. Watt, Wm. 1970. In [7].
15. Woods, Wm. 1973. In [14].