Generating Logic from ProGram Parse Trees.

W.R. Keller

Cognitive Studies Research Paper

Serial no: CSRP 39

June 1984

ABSTRACT

A program for translating phrase structure trees into expressions of logic is described. The program is designed for use with the Sussex "ProGram" grammar development system, which embodies a context-free parser designed for use with Generalised Phrase Structure Grammars.

The translation strategy is based on that of Montague Grammar. A formulation of intensional logic (IL) in the logic programming language Prolog is used as a meaning representation language. Expressions of IL are associated with individual words in a Semantic Dictionary. Rules of semantic composition combine simple constituent expressions to obtain translations for larger phrases. An additional storage mechanism permits a natural, non-syntactic treatment of quantifier scope ambiguities.

For practical reasons it was decided that the translations produced by the system should be first order equivalent. For this reason the system incorporates constraints which ensure that first order reducibility is maintained.

Generating Logic from ProGram Parse Trees.

W.R. Keller

Cognitive Studies Research Paper

Serial no: CSRP 39

June 1984

1. Introduction

Many computational linguists now believe that natural language understanding is best regarded as a many stage process. On this view, the problem of designing an understanding system is seen as decomposable into various self contained sub-problems. Two factors have helped persuade computational linguists of the need to adopt such an approach. Firstly, experience with complex computational systems has shown that "divide and conquer" strategies are generally preferable to *ad hoc* design techniques. Secondly, theoretical linguistics has lately tended towards a highly modular view of syntactic, semantic and pragmatic specification. It has therefore seemed reasonable to exploit this modularity in the organisation of language understanding systems.

The work presented in this paper is in the spirit of recent research in computational linguistics (Gawron *et al* 1982, Schubert and Pelletier 1982). The goal has been to examine one sub-problem of the entire understanding process:

the translation of sentences into logical form. It has sometimes been argued that there is little point in such an enterprise, that language simply isn't logical (for the classical argument see Strawson 1950). However, implicit in the work described here, is the contention that a description of truth-conditional content may be of use in the semantic interpretation of sentences. It is not being claimed that logical form and sentence meaning are identical, but rather that logical form may usefully serve as input to subsequent stages of analysis. Support for this contention can be found in much recent work on natural language processing. For example, Takao Gunji (1981) has shown the utility of identifying logical form before dealing with apparently extra-logical phenomena such as presupposition and implicature.

More specifically, the objective has been to investigate the use of Montague's semantics as a basis for translating English into logical form. To this end a program has been written which maps phrase structure trees into expressions of intensional logic (IL). At present the translator is constrained such that all intensions and higher-order expressions eventually "drop out". In consequence, the resulting translations are equivalent to expressions of first order logic (FOL). The decision to maintain first order reducibility is motivated by computational rather than linguistic considerations. It is well known that writing proof procedures for higher-order logics presents many

difficulties (Boolos and Jeffrey 1974). Having said this, there remain good reasons for continuing to use higher-order logic as a medium for the representation of meaning. In particular, sentence fragments such as "every man" or "the king of France" defy satisfactory representation within FOL - on this point see Warren (1983).

A consequence of enforcing first order reducibility is the present system's inability to capture certain kinds of semantic ambiguity. In particular, it cannot account for the distinction between non-referential (*de dicto*) and referential (*de re*) readings of sentences. In fact only the referential readings can be obtained. For example, the sentence "john seeks a unicorn" would receive a single, *de re* analysis, paraphrasable as "there exists some particular unicorn for which john is seeking". The alternative *de dicto* interpretation, on which john has no particular unicorn in mind, is unobtainable. Clearly, the relative merits of semantic coverage and computational tractability must be carefully weighed up, and a compromise found to suit a given purpose. In many practical applications (e.g. a natural language front-end to a database) *de dicto* readings may be superfluous. For the semanticist, on the other hand, the logical translation can be an end in itself, in which case faithful reproduction of *de dicto/de re* ambiguities may be required. A feature of the system to be described is its ability to manipulate logical expressions involving intensional operators. This admits the possibility of producing

both *de dicto* and *de re* readings if needed, with relatively minor alterations to the translator.

The translator itself carries out no syntactic analysis, but is intended as an "add on" to the Sussex "Pro-Gram" grammar development system: (Evans and Gazdar 1984). ProGram is a context-free parser employing a Generalised Phrase Structure Grammar or GPSG: (Gazdar and Pullum 1982, Gazdar, Klein, Pullum and Sag 1982). An interesting aspect of the present work has been to investigate the problems of using a Montague-style semantics in this context.

An outline of Montague Semantics and its relevance to Computational Linguistics is provided in section 2. Section 3 presents a brief overview of the translator and gives details of input and output. In section 4 the translation process is described, and a detailed example showing how a tree is mapped into a corresponding expression of IL is given in section 5.

2. Montague Grammar and Computational Linguists

The system of formal semantics on which this work is based was first developed by Richard Montague in the early 1970's. (Montague 1970). For the computational linguist, Montague's theory is attractive for several reasons:

i. It is model theoretic. Although "model" in this sense is an abstract mathematical concept, it has a familiar computational analogue: the database. In a natural

language understanding system, the database may take the place of the model of formal theory, characterising objects in the universe of discourse and the relationships between them. Manipulation of the database contents might then reflect changes in states of affairs. Hierarchical organisation of many databases can be used to record changes over time, events, etc. (for example see Gunji 1981).

ii. It is truth conditional. The meaning of a (declarative) sentence may be regarded as the conditions that would have to obtain in the world in order for the sentence to be true. This makes sense computationally if the database is considered to be a partial (i.e. finite) world model. Sentence meaning can then be evaluated with respect to the state of affairs existing in the current database (or even some previously current database).

iii. It is a mathematically precise theory of semantics resting on the contention that natural languages are by no means dissimilar to the sorts of formal logical systems employed by philosophers and logicians.

iv. Particularly attractive is Montague's rule-to-rule hypothesis. Roughly speaking the hypothesis states that the categories and rules of the semantics, must stand in a one to one relation with the categories and rules of the syntax. Another important idea is that the

semantics should be compositional * The meaning of some syntactic expression E, is a function of the meanings of the expressions F1,..,FN (and only these) just in case there is some syntactic rule combining F1,»«,FN to give E.

The remainder of this section is intended as a brief introduction to Montague Srammar. Only those aspects of Montague's strategy which *are* directly relevant to subsequent sections of the paperare dealt with. For a more detailed exposition the reader is referred to Dowty ejt 'aL (1981). Readers already familiar with Montague's semantic programme may wish skip the rest of this section.

To begin with it is important to understand certain unconventional features of the intensional logic (ID used by Montague, particularly the meaning of the terms 'intension' and 'extension'.

Suppose that m is a non-logical constant of a category e denoting the individual Mary *(e* is the category of individuals)- Then the! ijn^jn^on. of m. (written..^m) belongs to the category Cs,,e)$_f$ and may be thought of as the concept $_o$f the individual Mary. Likewise, suppose that the translation into IL of the intransitive verb run i® the one place predicate tyn/ ®f category<*e,t*> (the category of functions from individuals to truth values). The concept of run' - i.e. ^{A}Cjun' - is then of category <*s,<e,t*>> which is also known as IV (for Intransitive Verb phrase) and is the semantic category

appropriate to the traditional VP of syntax. Such concepts of predicates may be referred to as properties, thus ^run' denotes a property of individuals.

For particular reasons Montague chose to denote proper names by sets of properties of individuals. The translation of Mary (i.e. Mary') is then given by the lambda expression

$$\lambda P[{}^{\vee}P(m)]$$

where ${}^{\vee}P$ denotes the extension of P. Since P is a property of individuals - category $\langle s,\langle e,t\rangle\rangle$ - then ${}^{\vee}P$ is a predicate of category $\langle e,t\rangle$. Montague introduced a convention - the the so-called "brace convention" - to permit such expressions to be written more perspicuously. Thus Mary' may be written

$$\lambda P[P\{m\}]$$

Either way, Mary' is of category $\langle\langle s,\langle e,t\rangle\rangle,t\rangle$, which is also known as T (for Term phrase) and is the semantic category corresponding to the syntactic category of NP.

The primary means of combining expressions of Montague's IL to give new expressions is functional application. Since Mary' is a function from properties of individuals to truth values, and ^run' is a property of individuals, Mary' may be applied to ^run'. The result, after lambda conversion, and taking into account the brace convention, is run'(m) which is of semantic category t - the category appropriate to (declarative) sentences. The expres-

sion is the translation into IL of the English sentence "Mary runs".

Similarly by treating love' as a function from Term phrases (T) to Intransitive Verb phrases (IV), the meaning of "John loves Mary" may be obtained by first applying love' to Mary' and then applying John' to the result.

Finally, note that certain sentences such as "everyone loves someone" are ambiguous for reasons having to do with the relative scope of quantifiers. In order to deal with such quantifier scope ambiguities Montague specified various rules for 'quantifying-in' term phrases. Using these rules, term phrases may be introduced into IL expressions containing free variables. The order in which term phrases involving quantifiers (e.g. 'everyone' or 'a woman') are introduced, dictates the relative scope of the quantifiers. In general the quantifier associated with the last term phrase to be introduced, receives wider scope than any introduced previously. It should be evident how this may lead to the different readings available to sentences like "everyone loves someone".

In the present system, the use of a phrase structure grammar makes it impossible to adopt Montague's own analysis of scope ambiguities. Instead the translator incorporates a system loosely based on the "storage" mechanism originally proposed by Cooper (1975).

3. Overview of the Translator.

The program for translating phrase structure trees into expressions of intensional logic is written in the programming language Prolog. Detailed knowledge of Prolog is not assumed in what follows, but the reader may wish to consult a standard textbook such as "Programming in Prolog" (Clocksin and Mellish 1981). The translator may be conveniently divided into four components.

1. A module responsible for performing the mapping from trees to expressions of IL. This module accepts as input phrase structure trees of the same form as those produced by the ProGram parser (Evans and Gazdar 1984) and associates with each, one or more expressions of IL. (The precise details of input and output will be given shortly).

2. A module containing rules of semantic composition, one rule for each kind of tree node which may occur in the input. The semantic rules thus match tree fragments characterised by a mother category, and the categories of the immediate daughters. To this extent the form of the rules depends on the grammar used for parsing sentences in the first place. Each rule shows how a complex expression for a node is to be built from simpler expressions associated with the daughter nodes. The module also contains rules for quantifying-in term phrases. That is, introducing translations of NP's into

IL expressions containing free variables.

3. A semantic dictionary associating expressions of IL with a small number of English words.

4. A module for reducing complex expressions of IL to simpler, equivalent expressions.

Input to the translator takes the form of phrase structure trees having the following characteristics. Each tree is an instance of 'node', a term in the language Prolog, and of the form

node(N, F, D).

Here N is taken to be the category name for the mother category of the node (e.g. s for sentence, vp for verb phrase, and so forth). A feature tree F is associated with the mother category, and D is a list of the daughter sub-trees. Each daughter sub-tree is also an instance of node, except where N is a lexical category (i.e. the category corresponding to some word; e.g. v for verb, or n for noun, etc.). In this case D is simply a word.

For example, (1) is the phrase structure tree for "kim sees bill" (feature information has been omitted for clarity).

(1)

```
node(s, _, C
        node(np, _, t
                 node(pn, _, kirn)
                      3),
        node(vp, _, C
                 node(v, _, sees),
                 node(np, _, C
                          node(pn, _, bill)
                               3)
                      3)
           3)
```

The translator produces as output expressions of intensional logic, conveniently encoded in Prolog. The Prolog •formulation of IL will henceforth be called the 'logical representation language' (LRL for short)- Nothing hangs on this though, at an abstract level IL and LRL are one and the same language.

LRL may be defined in the usual recursive fashion. Throughout the following definition E, Ei and E2 are intended to represent arbitrary expressions in LRL, and V is some Prolog variable. (For present purposes, a Prolog atom can be any sequence of lower-case letters, e.g. "love", "everyone", "bill", and a Prolog yaxlabXe is any sequence of letters and numbers that begins with a capital; e.g. "He", "Xi", "Bill").

i. Any atom or variable may be an expression of LRL.

ii. Any term of the form not(E> is an expression of LRL.

iii. Any term of the form C(E1,E2) where C is one of and, or, egual, Implies, i£f (if and only if) is an expres-

sion of LRL.

iv. Any term of the form *Q*(V,E) where *Q* is one of forall, exists, is an expression of LRL.

v. Any term of the form lambda(V,E) is an expression of LRL.

vi. Any term of the form: (a) app(E1,E2), (b) app(E1,{E2}) is an expression of LRL.

vii. The terms ^E and ~E are expressions of LRL.

The intended interpretation of LRL expressions involving the logical operators (ii. and iii.) should be clear to those familiar with the predicate or propositional calculus. For expressions involving quantifiers (iv.) V is the variable quantified over and E is an arbitrary expression of LRL within the scope of the named quantifier (usually E will contain free occurrences of V). Definitions v. and vi. are meant to capture lambda abstraction and functional application respectively. In v. V is the variable bound by the lambda operator. In vi.(a) it is intended that E1 be applied to E2. Definition vi.(b) introduces Montague's brace convention and indicates that the extension of E1 be applied to E2. The operators '^' and '~' introduced in vii. correspond to Montague's intension and extension operators respectively.

The semantic dictionary associates expressions of LRL

with English words. For example the translation of bill into LRL is:

lambda(P, app(P, <bill>>>

As an example of an expression involving quantifiers, everyone translates intos

lambda(P, forall(X, implies(app(person, X), app(P, -CXJ))))

4. Ijhj? Translation Process.

Siven a phrase structure tree as input, the translator sets about interpreting each node of the tree, working from the bottom up- In other words, the translation rules are applied in a recursive fashion to the tree such that*

trans(tree) « combine(trans(sub-tree1),-.,trans(sub-treeN))

Since leaf nodes associate words with syntactic categories, interpretation begins by looking up the translation of a word in the semantic dictionary. This is then assigned as the translation of the mother category in accordance with the appropriate semantic rule.

In general, once every daughter sub-tree has been assigned a translation, then the mother category may also have a translation associated with it. An expression of LRL is built from the translations of the daughter nodes, as prescribed by the semantic rule for a node of the given type. Note that the system effectively adheres to the rule

to rule hypothesis, in that each type of node has only one corresponding semantic rule. Once an expression for the mother category has been assembled, it is systematically reduced to its simplest equivalent form. A simple semantic rule for noun phrases is shown in (2).

(2).

```
combine( node(np,NP,
         [node(det,DET,_),node(n,N,_)] ), [] ) :-
         reduce( app(DET,^N), NP ).
```

Rule (2) matches any np tree node which has det and n daughter nodes. A translation for the noun phrase is formed by applying the translation of the determiner to the intension of the noun translation, and then reducing the result.

Sometimes, a semantic rule will specify that the translation of a particular daughter node must be stored for later use. In such cases, a dummy expression having a free variable - the reference variable - is substituted in place of that daughter's translation. The latter is added to a list of ''free terms', with a note of the newly introduced reference variable. Once the root node of the phrase structure tree has been interpreted the list of free terms is emptied by quantifying-in each expression, taking account of its associated reference variable. The storage operation is thus very similar to the mechanism proposed by Robin Cooper (1975). A successful mapping has been achieved once the root node has a reduced expression of LRL assigned to it, and the store of free terms is empty.

Semantic rule (3) applies to any vp node having v and np daughter nodes. The noun phrase translation is not combined directly with that of the verb, but instead is stored along with a reference variable. This latter is taken from the translation of the dummy element he(N). A partial interpretation for the verb phrase is obtained by applying the verb translation to the intension of the dummy expression He, and then simplifying the result.

```
(3).
    combine( node(vp, VP,
                [node(v,V,_),node(np,NP,_)]), [N:NP] ) :-
                once( trans(he(N), He) ),
                reduce( app(V, ^He), VP ).
```

Rule (4) below is for quantifying-in noun phrase translations. It takes a partial sentence interpretation S1 and introduces a term phrase TP. This is achieved by applying TP to the expression

```
^lambda(V, S1)
```

where V is the reference variable associated with TP in the list of free terms. The resultant lambda expression is then reduced.

```
(4).
        do_quant_in( s, S1, V:TP, S ) :-
                var(V),
                reduce( app(TP, ^lambda(V, S1)), S ).
```

Varying the order in which free terms are picked from the storage list and quantified-in may lead to several different expressions being produced for a given phrase struc-

ture tree. This is in fact exactly what is required, since it allows quantifier scope ambiguities to be handled in an elegant, non-syntactic fashion. The system is designed to ensure that if the quantifying-in stage is ever re-done, then the list of free terms will be permuted. Asa consequence, all interpretations potentially arising from quantifier scope ambiguities may be produced.

5. An Example.

The translation process may be illustrated in detail with reference to a step by step example. A phrase structure tree for the sentence "every man loves a woman" is shown in (5). The tree is of the same form as those produced by the ProGram parser, but all feature information has been omitted. Details of syntactic features are not required by the translator and are thus irrelevant to our interests. Also, it is assumed that words which appear at terminal tree nodes are in their base forms. This permits the translator to look up words directly in the semantic dictionary without having to perform prior morphological analysis, an unnecessary complication. It should be pointed out that this minor simplification is merely a convenience, and does not otherwise affect the translation process.

The translation process starts with the node:

```
node(det, _, every)
```

(5).

```
node(s, _, [
        node(np, _, [
                node(det, _, every),
                node(n, _, man)
                    ]),
        node(vp, _, [
                node(v, _, love),
                node(np, _, [
                        node(det, _, a),
                        node(n, _, woman)
                            ])
                    ])
            ])
```

At this stage the determiner every is looked up in the semantic dictionary. This yields the expression

```
lambda(P,
    lambda(Q,
        forall(X, implies(app(P, {X}), app(Q, {X})))))
```

which becomes the translation of the det node, in accordance with the corresponding semantic rule. The rule builds a new instance of node, paralleling that of the phrase structure tree, but having the translation of the mother category in place of feature information. In this case the 'semantic node' is

```
node(det, Lexpr, every)
```

where Lexpr is the lambda expression for every. Since this expression cannot be reduced, interpretation continues with the 'noun' node:

```
node(n, _, man)
```

Here exactly the same sequence of operations are performed,

resulting in the semantic node:

node(n, man, man)

In this case the word man., has no special translation into LRL associated within the semantic dictionary, and is therefore left as it stands. The translation of man is to be thought of as a one place predicate on the domain of individuals (or equivalently, a set of individuals)•

Again no reduction of the translation is necessary, and the current $node_{fc}$ becomes the noun phrase node immediately dominating every and man. A semantic rule for nodes of type np (i.e. rule(2) of section 4) shows how a translation for this node may be obtained. It is achieved by constructing (and then reducing) an expression of the form:

app(DET, ^N)

Here DET is the translation of the determiner node (i.e. of fiyjSEY > and "N is the intension of the translation associated with the noun node (i.e. Aman). This results in the expression

```
app (lambda(P,
       lambda(Q,
          foralKX,
             implies(app(P, <X>), app(Q, CX>))))), --man)
```

which after lambda conversion becomes

lambda(Q, forall(X, implies(app("'man, {X>>, app(Q, <X>)>))

This may be further reduced by taking account of the brace convention to give the final expression

lambda(Q, forall(X, implies(app(man, X), app(Q, CXJ))))

The present stage of the process is then completed by building a semantic node -for the noun phrase having this translation.

Attention is now switched to the verb phrase sub-tree. First a semantic tree node is built for the verb node:

node(v, love, love)

The noun phrase sub-tree is then interpreted in an identical fashion to that described for 'every man'. The translation for the determiner a is

```
lambda(P,
    lambda(Q,
        exists(Y, and(app(P, CY>), app(Q, CY>)))))
```

The following semantic node is built for the common noun wjsman:

node(n, woman, woman)

These translations *are* combined exactly as before to give (after reduction)

lambda(G!, exists(Y, and(app(woman, Y), app(Q, <Y>))))

The verb phrase node itself may now be processed. For the first time a semantic rule which stores a daughter node's translation is encountered (see rule (3) of section 4). To begin with, the rule specifies that the translation of a special dummy element he(N) should be looked *up* in the semantic dictionary. The translation is

lambda(P, app(P, {X1}))

where X1 is the reference variable and is unique to this expression. (Each subsequent translation of he(N) will contain a new reference variable). Next, the noun-phrase translation is associated with the new reference variable and stored for later use. The term X1 : NP - where NP is the noun phrase translation - is then the first free term to be added to the storage list. The use of the reference variable X1 will become apparent later on, when the noun-phrase translation is quantified in. In the meantime, the translation of he(N) is combined with the translation of the verb node to yield:

app(love, ^lambda(P, app(P, {X1})))

This expression cannot be reduced further by the usual operations of functional application and so forth. However, since we are aiming at first order reducibility for the expressions produced by the translator, it is expedient at this stage to apply a special 'extensioning' reduction. The extensioning process yields the new expression

app(love, X1)

The idea is that for any predicate P which applies to expressions of the form

$$\hat{}\lambda P[\check{}P(X)]$$

(known as the individual sublimation concept of X) there is a corresponding predicate P* which applies only to indivi-

duals. Further, Fv will be true of an individual X just in case P is true of X's sublimation concept.

The slightly confusing thing about the 'extensioning' operation used by the translator is that the applied predicate is not renamed- That is, love does not become *loyM** or some such. In practice this isn't really worrying. All the hard work of keeping track of changes in the interpretation of predicates is done automatically "behind the scenes[1]' by the module responsible for reducing LRL expressions.

Having assembled a semantic node for the verb phrase, the root node is now interpreted. Once again this involves storing a noun phrase translation (i.e. that for the noun phrase 'every man') and combining a dummy element with the translation of the other daughter node. Following reduction the expression obtained is

app(app(love, XI), X2)

Interpretation of the phrase structure tree is now essentially complete. All that remains is to empty the storage list by quantifying-in each free term. A quantification rule for the sentence node indicates how this may be achieved (rule (4) section 4). First the expression

lambda(X2, S)

is formed, where S is the translation of the sentence node. The variable X2 is the reference variable associated with the free term to be quantified in (in this case the

translation of 'every man'). The free term is then applied to the intension of this new expression giving

```
app( lambda(Q,
        forall(X,
           implies(app(man, X), app(Q, {X}))),
             ^lambda(X2, app(app(love, X1), X2))    )
```

This rather complex expression reduces to the more familiar

```
forall(X, implies(app(man, X), app(app(love, X1), X)))
```

which becomes the new translation associated with the root node.

The second free term is now dealt with in exactly the same way. Following reduction, interpretation is complete (modulo re-doing the quantifying-in stage) with the result

```
exists(Y, and(app(woman, Y),
              forall(X, implies(app(man, X),
                                app(app(love, X), Y)))))
```

As previously noted, attempting to re-do the quantifying-in stage results in the storage list being reordered. In this case, the free terms will be chosen in the opposite order, with the consequence that the translator will produce the new expression

```
forall(X, implies(app(man, X),
                  exists(Y, and(app(woman, Y),
                                app(app(love, X), Y)))))
```

In this way both readings of "every man loves a woman" may be obtained.

6. Conclusions.

Compositional, model-theoretic semantics offers an approach to natural language processing which combines theoretical rigour with a disciplined methodology. If the computational linguist initially finds the strategy imposed by formal semantics rather unpromising, further investigation is likely to be more rewarding. Perhaps surprisingly, it turns out that many key ideas characteristic of model theoretic semantics, may be related quite naturally to familiar programming concepts.

What has been described in this paper is a system which employs Montague's semantics as a basis for the translation of English into logic. A formulation of IL in Prolog permits a precise and transparent semantic specification. Semantic rules may be correlated closely with the syntactic structures to be interpreted, making semantic generalities relatively easy to capture. The alignment of rules and structures in this way amounts to a version of Montague's rule to rule hypothesis. It is possible, however, that a single semantic rule may correspond to a whole set of syntactic rules, each of which admits the same sort of tree fragment.

The current translator is equipped to deal with a fairly small sample of English declarative sentences. Nevertheless it has shown itself to be capable of analysing complex semantic phenomena such as quantifier scope ambiguities. Providing semantic rules for a somewhat wider range

of (declarative) constructs should not pose any significant problems.

A number pf improvements to the translator are envisaged:

i. Currently semantic rules must be written as Prolog clauses. It should be possible to generate each clause automatically from a description of the expression to be built and the corresponding grammatical rule.

ii. The translator presently has a rather rigid storaqe/quantifying-in strategy. All noun phrase translations are stored and then quantified-in at the last moment* This has the advantage of treating noun phrase translations in a uniform fashion, but can lead to unwanted results. For example "kirn sees bill[11] will receive two (identical) interpretations since there are two ways of quantifying-in the translatipns of kjun and bill.

iii. It would be convenient for the user to have some degree of control aver the translation process. A choice the user might wish to make which has already been mentioned, is between FQL equivalence and IL representations including *de dicto* analyses.

BIBLIOGRAPHY

Boolos, George and Richard Jeffrey (1974) Computability and Logic. Cambridge University Press, London.

Clocksin, William and Christopher Mellish (1981) Programming in Prolog. Berlin: Springer-Verlag.

Cooper Robin. (1975) Montague's Semantic Theory and Transformational Syntax. Doctoral dissertation, University of Massachusetts, Amherst.

Dowty, David R. Robert E. Wall & Stanley Peters. (1981). Introduction to Montague Semantics, Synthese language library; v. 11. D. Reidel, Dordrecht Holland.

Evans, Roger and Gerald Gazdar (1984) The ProGram Manual. University of Sussex Cognitive Science Research Paper 35 (CSRP 035)

Gawron, Jean Mark, Jonathan King, John Lamping, Egon Loebner, Anne Paulson,Geoffrey Pullum, Ivan Sag & Thomas Wasow. (1982) Processing Enlish with a Generalised Phrase Structure Grammar. Proceedings of the 20'th Annual Meeting of the Association for Computational Linguistics, University of Toronto, June.

Gazdar, Gerald, and Geoffrey Pullum (1982) Generalised Phrase Structure Grammar: A Theoretical Synopsis. Bloomington: Indiana University Linguistics Club mimeo. Also available as University of Sussex Cognitive Science Research Paper 7 (CSRP 007).

Gazdar, Gerald, Ewan Klein, Geoffrey Pullum, and Ivan Sag (1982) Coordinate structure and unbounded dependancies. In M. Barlow, D. Flickinger & I.A. Sag (eds.) Developments in Generalised Phrase Structure Grammar: Stanford Working Papers in Grammatical Theory, Volume 2. Bloomington: Indiana University Linguistics Club, 38 - 68. Also available as University of Sussex Cognitive Science Research Paper 6 (CSRP 006).

Gunji, Takao (1981) Toward a Computational Theory of Pragmatics - Discourse, Pressuposition, and Implicature. Unpublished doctoral thesis, Ohio State University.

Montague Richard (1970) The proper treatment of quantification in ordinary English. In Thomason 1974, 247 - 270.

Schubert, L & F.J. Pelletier (1982) From English to Logic: Context-Free Computation of "Conventional" Logical Translation, Computational Linguistics 8. 27 - 44.

Strawson, Peter (1950) On Refering. Mind 59 320 - 344.

Thomason, Richard (1974) Formal Philosophy: Selected Papers of Richard Montague. Yale Univ. Press, New Haven, CT.

Warren, David (1983) Using Lambda-Calculus to Represent Meanings in Logic Grammars, Proceedings of the 21'st Annual Meeting of the Association for Computational Linguistics, Toronto, June.

Appendix.

Sample Translations.

Listed below are a small number of analyses produced by the translator program. To begin with, a very simple example:

"kim runs"

```
app(run, kim)
```

A rather more complex example involving a definite description:

"the man runs"

```
exists(_27, forall(_28, and(iff(equal(_27, _28), app(man, _28)),
      app(run, _27))))
```

Transitive verbs are treated as functions from individuals to intransitive verb denotations.

"kim sees bill"

```
app(app(see, bill), kim)
```

"a woman sees john"

```
exists(_9, and(app(woman, _9), app(app(see, john), _9)))
```

The next sentence exhibits quantifier scope ambiguity, and has two translations associated with it. Note the relative ordering of exists and forall in each case.

"every man loves some woman[11]

forall(.19, iipliesfappdan^ J9), exists! J!B, and(app(«oaan, JB),
app(app(love$_f$ *Jit), J9)))))*

Backtrack!ng produces the alternative interpretations

existsi_2B, 4nd(app(*oaan$_f$ _28i, forall(_i9, iipliesiappiian, J9>,
appiappdove, J1), *J9))))f*

The next sentence actually has six identical interpretations, since there are six ways of quantifying-~in jkijn, fjjdlo «/\nd a_bone« Note that ditransitive verbs such as give translate into -functions from individuals to transitive verb denotations.

"kirn gives -fido a bone[11]

exists(.16, and(app(bone, J6!, app(app(app(give, fido), _16>$_f$ kit)))

In the "following example, the six translations are distinct syntactically, but there are only three different semantic interpretations. (Only the orderings of the existential quantifiers with respect to the universal quantifier are truth-conditionally significant).

'•everyone gives the dog a bone"

foraiHJi, iiplies(app(per5on, J1), exists(J2, andUppCbone, _12),
exists(j3, forall.(J4, and (iff (equal *iJZ*$_\tau$ J4), appldog, J4)),
applapp(app(give, J3), -J2J, JDHJI))) ? ?

exists(^i2, and(app(bone, ^12), foralli_li, i§plies(app(person, ^ii),
exists! 13, foralli_ *u]* and«i«(equal(J3,J4), apptdog, _141),
applapp(app<give, J3), J2), .ill))))))) ? j •

forall(J1, iiplies(app(person, ^iil, enstsl^iS, forall(J4, and(iif(
equal(J3$_f$ J4)$_t$ app(dog, ,14))$_f$ exists(_12, andfappibone, ^.12),
app(appUpp(give, _13), J2), _11)))))'))) ? ;

existsJ.13, faralK.14, and(iff(equal(.13, *J$),* appCdog, .14)),
faraiK.li, isp1ies(app(person, .11), exist5(_12, and(app(bane, .12),
app(app(app(give, J3), *J2U Jl*)))))))) ? ,'

exists (^12, and (app (bone, .12), exists(^i3, forall(J4, and(iff(
equal(.13, .14), appidog, .14)), forall't.il, iipliestapp'tperson,'.!!),
app(appUpp(giv6, .13)^ .12), .11))))))) ? ;

exist5<J3, foralli.M, and(i«<equal(.13, .14), app(dog, .14)),
• exist5(.12, andiapptbone, J2), foralH.ll, iipiie£(app(person, .11),
app (app (app (give, .13)," .12), *Jl*)))))))) ?