CONSTRAINING MONTAGUE GRAMMAR FOR COMPUTATIONAL APPLICATIONS

Harold Einar Jowsey

Ph.D. in Artificial Intelligence

University of Edinburgh

1990

I hereby declare
(a) that this thesis has been composed by myself and
(b) that the work is my own.

H. E. Jowsey.

Abstract

This work develops efficient methods for the implementation of Montague Grammar on a computer. It covers both the syntactic and the semantic aspects of that task. Using a simplified but adequate version of Montague Grammar it is shown how to translate from an English fragment to a purely extensional first-order language which can then be made amenable to standard automatic theorem-proving techniques.

Translating a sentence of Montague English into the first-order predicate calculus usually proceeds via an intermediate translation in the typed lambda calculus which is then simplified by lambda-reduction to obtain a first-order equivalent. If sufficient sortal structure underlies the type theory for the reduced translation to always be a first-order one then perhaps it should be directly constructed during the syntactic analysis of the sentence so that the lambda-expressions never come into existence and no further processing is necessary. A method is proposed to achieve this involving the unification of meta-logical expressions which flesh out the type symbols of Montague's type theory with first-order schemas.

It is then shown how to implement Montague Semantics without using a theorem prover for type theory. Nothing more than a theorem prover for the first-order predicate calculus is required. The first-order system can be used directly without encoding the whole of type theory. It is only necessary to encode a part of second-order logic and this can be done in an efficient, succinct, and readable manner. Furthermore the pseudo-second-order terms need never appear in any translations provided by the parser. They are vital just when higher-order reasoning must be simulated.

The foundation of this approach is its five-sorted theory of Montague Semantics. The objects in this theory are entities, indices, propositions, properties, and quantities. It is a theory which can be expressed in the language of first-order logic by means of axiom schemas and there is a finite second-order axiomatisation which is the basis for the theorem-proving arrangement. It can be viewed as a very constrained set theory.

Contents

1. Introduction

In "Universal Grammar" (UG; Montague 1970) and "The Proper Treatment of Quantification in Ordinary English" (PTQ; Montague 1973) Richard Montague showed how to specify a denotational semantics for a fragment of English by means of translation into a model-theoretically interpreted intensional logic (IL) based on the type theory of (Church 1940). Then in (Gallin 1975) his student, Daniel Gallin, provided a proof theory for Montague's IL which is complete with respect to a generalisation of Church's type theory suggested by (Henkin 1950). Gallin also showed how to replace IL by the extensional two-sorted type theory (Ty2).

In this thesis I define a simplified version of Montague Grammar (SMG) which is no less adequate than UG or PTQ, suitable for use as a front-end to a general reasoning system. The potential applications of Montague Grammar envisaged here are not restricted to database query, in that there is an illocutionary symmetry whereby for each interrogative query the corresponding indicative statement can be asserted into the database as a fact.

Despite the work of Gallin there is still an unfortunate problem which arises for Montague's translations. In type theory a unification algorithm must seek substitutions which equalise terms modulo lambda-conversion. This is a source of great inefficiency (although not necessarily incompleteness) in theorem provers for type theory which is an omega-order logic. The problem is that unification is undecidable even in the case of second-order logic (Goldfarb 1981); thus there is no effective procedure for determining whether or not two terms of the language of second-order logic have a common instance. Because in the general case it is not possible to tell whether a unification call will terminate, it is necessary to unify in parallel with the search process associated with first-order logic thereby adding considerable computational overheads.

By contrast, if a semantics for SMG can be rendered into first-order logic then standard theorem proving techniques such as resolution (Robinson 1965), where the unification algorithm is part of a single inference step, can be used. It is clear that new constraints are needed in order to achieve this. These constraints are implicit in the fragment itself which does not exercise much of omega-order logic. Even following Montague and Gallin there is many a sentence whose translation has first-order equivalents. Therefore in this work I replace type theory with a Theory of Montague Semantics (TMS) which provides a five-sorted extensional logic in which every sentence of the fragment has a first-order translation.

In fact, TMS can be axiomatised entirely within first-order logic which suggests abandoning omega-order logic even during the process of translation. The early approach of translating into IL and then doing simplifications towards a first-order form requires a very complicated simplifier as well as the parser. It will be shown that by going directly into first-order logic this computational overhead can be avoided. By using semantic templates, which carry enough type information to translate using only first-order unification, the translation can be built up incrementally during parsing. Thus a remarkably short parser and translator for Montague Grammar is obtained, which can form a self-contained and useful aid in the study of semantics. A systematic correspondence between type theory and semantic templates will be described so that readers may see for themselves how to implement extensions of Montague Grammar.

The whole front-end is presented in this thesis as a parser and translator for SMG written in Prolog (Clocksin and Mellish 1981). Intended as an aid in the further study of the SMG-TMS framework, it constructs analysis trees of the traditional kind and simultaneously builds up a translation in five-sorted first-order predicate logic. Since the program outputs first-order logic it could be used as a front-end for any of the systems developed in artificial intelligence for manipulating first-order well-formed formulas. Although it is easier than type theory for which there currently exists no practical theorem prover, TMS is not trivial and I will be concerned to demonstrate that a finite axiomatisation suffices to cover TMS when the system is properly organised. No particular theorem prover is presented but the system is adapted to the popular resolution method.

1.1. The Range of Influence

What is on offer here is not the solution to a single big problem but the hope of managing the complexity of Montague Grammar, which has entangled everyone else. For this, harmonious solutions to a host of smallish problems had to be found so that one way forward could be presented.

Natural language understanding has been broken down into a syntactic parsing problem and a semantic theorem-proving problem but before now there has been no simultaneous solution to both problems for Montague Grammar. In particular, the common use of the typed lambda calculus to build up the semantics during parsing overburdens the theorem prover as we have seen. So semantic processing has always been restricted to lambda-reduction or evaluation with respect to finite domains. Conversely restricting to the first-order predicate calculus for the sake of the theorem prover raises difficult problems for semantic representation during parsing which have apparently *never* been solved, let alone *uniformly* solved, for a *significant* Montagovian fragment prior to this work. The solution in my early attempt (Jowsey 1986) was not uniform.

The nest of problems is so complex in its interrelations that having dealt with the parser as above, new difficulties concerning the lack of abstraction facilities in first-order languages are reflected back to the theorem prover: the ghost of lambda still haunts us. These difficulties have now been overcome despite the doubts expressed in (Jowsey 1986). A finite axiom system which has not previously been published completes this thesis. More research work remains to be done because the considerable control issues in both the parser and the theorem prover have not been tackled at all.

The early approach to computational Montague Grammar is summarised in (Friedman 1978) which contains an extensive bibliography. The tendency was to organise such systems as a long composition of separate functions, typically: phrase structural analysis, quantifier scoping, translation into IL, simplification of the IL expression, and semantic interpretation of the IL expression with respect to a finite model (which can be viewed as an extension of simplification). This is reminiscent of multi-pass compilers for programming languages, in contrast to which the present system follows one-pass compilers with their greater elegance and efficiency. That approach probably reached its zenith in the PHLIQA 1 question-answering system (Bronnenberg &al. 1980). Whilst its syntax was clumsy but adequate, its semantics was very restricted. Simplification is something that a theorem prover for type theory would do but it is just not very much. Perhaps it might be possible to add facilities to such a system to raise it to the present standard. However it would not be clear what was being aimed at. Completeness would be too much to expect whereas with first-order logic, completeness is the obvious target. Work continues in the earlier tradition. In (Bainbridge 1984) the Prolog interpretation of Definite Clause Grammars is adapted to Montague Grammar using a well-formed substring table to handle left-recursion. Because the database is side-effected to record the table, the declarative interpretation of Prolog is lost. Analysis trees are output, ready for translation into unsimplified IL by a post-order tree walk. It appears that the system would be slow. In this thesis a quick bottom-up parser, coded in declarative Prolog, will simultaneously yield both analysis trees and 'simplified' first-order translations.

This thesis shares its aims but not its methods with those of (Schubert and Pelletier 1982). Their arguments against generalised quantifiers are powerful but cannot be accepted at present because to do so would be fundamentally non-Montagovian. However, from the point of view taken here, the main problem with their work is that while they claim to compute a 'conventional' logical translation their target language is actually rather unconventional. It is difficult to find specific objections to their system as it is not precisely formulated. For example, they do not commit themselves as to whether to use a first-order or a second-order language. However their language is clearly not the same as the standard language presented in introductory logic textbooks. By contrast, the one employed here differs only by the use of sorted quantifiers (which can be viewed as a syntactic sugar). We agree in stressing that it is semantics, not syntax, which is important. Yet Schubert and Pelletier give no formal semantics for their language, which could therefore with enough ingenuity be interpreted as anything less powerful than third-order logic. It has been a central concern that this work should be sufficiently rigorous to be open to constructive criticism.

Let's consider the Chat-80 question-answering system described in (Pereira 1983) because it uses a logic superficially resembling the present logic. Pereira's parser is an extension of Definite Clause Grammar called Extraposition Grammar about which the present work is unconcerned. His logical translations are again constructed by walking the analysis trees. Furthermore Pereira's semantic formalism of Definite Closed-World Logic lacks illocutionary symmetry. His theorem prover inherits the closed-world assumption from Prolog through negation by failure, and adds a groundness assumption whereby any atomic sentence in or deduced from the database must be ground. Standard logic has none of these deficiencies and they are not considered acceptable in a theorem prover for Montague Grammar, although they may be acceptable in the geography domain of Chat-80.

The Prolog textbook of (Clocksin and Mellish 1981) gives an often cited example from (Pereira, Pereira, and Warren 1979) of a small Definite Clause Grammar which constructs a first-order translation incrementally by unification, as is done here. The difference is that it uses numerous spread parameters which this implementation wraps in a single structure called a semantic template, systematically derived from the appropriate type symbol.

The lecture notes of (Pereira and Shieber 1987) contain a chapter in which they develop a fragment with similar coverage to the extensional fragment of Chapter 2. Moreover the way in which they build up first-order formulas by unification uses a notational variant of the semantic templates which I invented independently and had already published in (Jowsey 1986). Pereira and Shieber explain templates as representing lambda-expressions (although they admit that the analogy breaks down). The present explanation of templates as types appears to be an improvement. However their approach is allied to mine and their notes are sufficiently clear that a skillful programmer could probably construct all the purely syntactic parts of my system under their guidance, *provided* that the programmer knew TMS beforehand.

An admirable alternative attempt to implement Montague Grammar is that of (Miller and Nadathur 1986) whose aims and methods, though not solutions, are close to those of this thesis. Their approach is based on a higher-order extension of Prolog called λProlog. In essence, they have derived a programming language from Ty2 by analogy with the way in which Prolog is derived from full first-order logic. A program in λProlog consists of higher-order definite clauses and is executed by a depth-first search through the database, as in Prolog. However each clause is itself executed by an inner depth-first search through the many unifiers which arise from the undecidable unification algorithm of (Huet 1975). Obviously their system is far from complete but enough simple cases such as lambda-reduction work properly for it to be useful as a programming language. The major advantage of using such a system is that Montagovian translations can be encoded very transparently as data structures. In particular, bound variables pose no problems. To continue within their paradigm would still involve writing a full theorem prover for Ty2 in λProlog. Although this would be easier than writing such a theorem prover in Prolog, I am not convinced that it could be made sufficiently efficient. Of course, having done the lambda-reductions very pleasantly, it would still be possible to give up and switch to a first-order theory but then, having abandoned theoretical higher-order characterisation, their proposal would reduce to the advocation of λProlog as a favourite programming language. Nevertheless I acknowledge the force of the argument for the uncompromising use of higher-order logic without following that line of investigation myself.

Throughout the writing of this thesis, (Pereira 1983) has been a guiding example of thesis style.

1.2. The Plan of this Thesis

The rest of Chapter 1 reminds the reader about certain logics pertinent to the following original work; Chapter 2 is a scaled-down version of Chapter 3 which may be more comprehensible to the uninitiated; Chapters 3 and 4 contain the substance of the thesis; and Chapter 5 concludes it. A colloquial description of the thesis topic is to make Montague Grammar computational by getting rid of lambdas. Chapter 3 is about using unification instead of lambda-reduction in an analyser for a Montague fragment, and Chapter 4 is about a first-order theory of Montague Semantics. The primary device of Chapter 3 is mapping English categories into logical types and the implementation of types by semantic templates, and that of Chapter 4 is mapping (irreducible) sentences, predicates, and terms into propositions, properties, and quantities respectively and the arrangement of these by abstract predicates.

Prolog procedures are used freely in Chapters 2 and 3 but no tutorial material on Prolog is included as such tutorials are now widely available. The reader is reassured that the grammars are coded in a pure "vanilla" Prolog (without '!' or assert, but with '\+') which admits of a declarative interpretation. Likewise the reader of Chapter 4 is assumed to be familiar with the fundamental ideas of resolution theorem proving as taught, for example, in first-year artificial intelligence courses. Again, no unusual methods are used. The colloquial message of this thesis is that Montague Grammar can be bent to fit standard techniques without breaking it. Appendices contain the extra uninteresting procedures required to make a usable system, and the entire normalised axiomatisation.

1.3. Logical Targets

For a given fragment of English the complexity of the translation technique should be inversely proportional to the complexity of the target language in any reasonable system. I have assumed that the inherent complexity of the system is constant but can be divided between translation and target to one's taste. This thesis will only illustrate the extremes. In the specification a simple translation method will go into a difficult logic but in its implementation trouble will be taken to translate into an easier logic.

1.3.1. The Omega-Order Language

This is an extremely general and powerful logic which offers far more than is required for the SMG fragment but which allows straightforward translation.

Gallin's Two-Sorted Type Theory (Ty2):

t	-- the type of truth values
e	-- the type of entities
s	-- the type of indices
$\langle a,b\rangle$	-- the type of functions from type a to type b
$c_{n,a}$	-- the nth constant of type a
$v_{n,a}$	-- the nth variable of type a
$(\alpha ' \beta)$	-- apply α to β
$\lambda\upsilon\ \alpha$	-- the function which yields α when applied to an argument υ
$(\alpha = \beta)$	-- α is equal to β

These logical expressions are theoretically sufficient to provide translations from any language in Montagovian universal grammar. Constants, variables, applications, and lambda abstractions yield a formalism rather similar to a functional programming language. However equality can only be provided as an effective primitive for programming if it is restricted to entities, indices, and truth values (Booleans). It is the introduction of equality for functions which makes this a logic. The familiar logical connectives and quantifiers may be defined at some length in terms of the expressions already given but it is rather a tour de force to do so. They will be notated as follows:

F	-- falsity
T	-- truth
$\neg\ \phi$	-- it is not the case that ϕ
$[\phi \wedge \psi]$	-- both ϕ and ψ
$[\phi \vee \psi]$	-- either ϕ or ψ or both
$[\phi \rightarrow \psi]$	-- if ϕ then ψ
$[\phi \leftrightarrow \psi]$	-- ϕ if and only if ψ
$\forall \upsilon\ \phi$	-- for all υ, ϕ
$\exists \upsilon\ \phi$	-- there exists at least one υ such that ϕ

Alternatively all *sentences* (logical expressions of type 't') may be expressed using only constants, variables, applications, implications, and universal quantifications.

1.3.2. Modal and Intensional Languages

Although Ty2 has types 't', 'e', 's', and <a,b>, the SMG fragment will be formulated so that 't' and 's' always occur paired as <s,t> in types mapped from categories of English. From this point of view it would be more appropriate to use a *modal* omega-order logic rather than Ty2. That is, a logic with types 'o', 'e', and <a,b> where 'o' stands in for <s,t>. Montague used an *intensional* omega-order logic (IL) with types 't', 'e', <a,b>, and <s,a>. The extensional types of Ty2 contain all the intensional types of IL and the intensional types contain all the modal types. IL seems awkward because it is neither broad enough to be maximally general, nor narrow enough to be minimally sufficient.

IL is characterised by the relativisation of constants to indices which, lacking their own type, must be manipulated by special operators. If i is a particular index variable (say $v_{0,s}$) then every constant $c_{n,a}$ of IL really stands for ($c_{n,<s,a>}$ i) and the expressions ${}^{\vee}\alpha$, ${}^{\wedge}\alpha$, $\Box\phi$, and $<>\phi$ stand for (α i), $\lambda i \alpha$, $\forall i \phi$, and $\exists i \phi$ respectively. These facilities fit SMG even less well than they fit PTQ.

1.3.3. The First-Order Language

This is the most constrained logic which is still expressive enough to cover the SMG fragment. It will be shown how to translate into it and how to do theorem proving with it. The notation is designed to be compatible with the syntax of Prolog terms. In the logic itself terms are either constants or variables. There are no functions.

Five-Sorted First-Order Predicate Logic:

e	-- the sort of entities
s	-- the sort of indices
o	-- the sort of propositions
p	-- the sort of properties
q	-- the sort of quantities
rel1(α)	-- predicate *rel1* of α
rel2(α, β)	-- predicate *rel2* of $<\alpha,\beta>$
rel3(α, β, γ)	-- predicate *rel3* of $<\alpha,\beta,\gamma>$
rel4(α, β, γ, δ)	-- predicate *rel4* of $<\alpha,\beta,\gamma,\delta>$
0	-- falsity
1	-- truth
~ ϕ	-- it is not the case that ϕ
(ϕ & ψ)	-- both ϕ and ψ
(ϕ v ψ)	-- either ϕ or ψ or both
(ϕ => ψ)	-- if ϕ then ψ
(ϕ <=> ψ)	-- ϕ if and only if ψ
all(υ:a, ϕ)	-- for all υ of sort a, ϕ
exists(υ:a, ϕ)	-- there exists at least one υ of sort a such that ϕ

Here is a sketch of the semantics of the first-order language. Let S = {e, s, o, p, q}. A model structure M is a sextuple $<D_e, D_s, D_o, D_p, D_q, F>$ with domain $D = \cup_{a\in S}(D_a)$ such that $D_a \neq \{\}$ for any $a \in S$, $D_o \subseteq \mathbf{P}(D_s)$, $D_p \subseteq \mathbf{P}(D_e \times D_s)$, $D_q \subseteq \mathbf{P}(D_p \times D_s)$, $F(\alpha) \in D$ when α is a constant, and $F(\textit{reln}) \subseteq \times_n(D)$ when *reln* is an n-ary relation. Let an assignment over M be a function G such that $G(\alpha) = F(\alpha)$ when α is a constant and $G(\upsilon) \in D$ when υ is a variable. Let satisfaction be a ternary relation '|=' such that for all models M and assignments G over M, it is not the case that M,G|=0; if ϕ and ψ are sentences then M,G|=(ϕ=>ψ) iff M,G|=ϕ materially implies that M,G|=ψ; the other connectives go similarly; if υ is a variable, $a \in S$, and ϕ is a sentence then M,G|=all(υ:a,ϕ) iff for all $x \in D_a$, M,G'|=ϕ where G' is the assignment such that $G'(\upsilon) = x$ and for all terms $\alpha \neq \upsilon$, $G'(\alpha) = G(\alpha)$; the existential quantifier goes similarly; and if *reln* is an n-ary relation and α_1 through α_n are terms then M,G|=*reln*(α_1,...,α_n) iff $<G(\alpha_1),...,G(\alpha_n)> \in F(\textit{reln})$.

P(A) is the powerset of A.

Furthermore ten symbols are singled out by definitions to denote entities, an index, and certain special relations:

$$F(j),\ F(m),\ F(b),\ F(k) \in D_e, \qquad F(!) \in D_s,$$

$$F(=) \equiv \{ \langle x,y\rangle \in D \times D \mid x = y \},$$

$$F(<) \equiv \{ \langle x,y\rangle \in D_s \times D_s \mid x < y \}$$
where '<' is a left linear partial ordering on D_s,

$$F(true) \equiv \{ \langle x,y\rangle \in D_o \times D_s \mid y \in x \},$$

$$F(ppty) \equiv \{ \langle x,y,z\rangle \in D_p \times D_e \times D_s \mid \langle y,z\rangle \in x \},$$

$$F(qtty) \equiv \{ \langle x,y,z\rangle \in D_q \times D_p \times D_s \mid \langle y,z\rangle \in x \}.$$

Appropriate axioms will be provided for these non-logical (Montague Semantics theoretic) symbols in due course.

A sentence ϕ is true in a model M (M is a model of ϕ) iff for all assignments G over M, $M,G \models \phi$. A sentence is valid, satisfiable, falsifiable, absurd iff it is true in all models, some models, not all models, no models respectively. A set of sentences entails ϕ iff every model of all its members is also a model of ϕ. It is contradictory iff there is no model of all its members.

This qualifies as a first-order language partly because it has a first-order syntax in which relations are not varied, quantified, or related but another contributing factor is that in its semantics there is no quantification over arbitrary subsets of the domain. In particular, it is not demanded that $D_o = \mathbf{P}(D_s)$ but merely that D_o be fixed by the model, be non-empty, and only contain subsets of D_s. It goes similarly for D_p and D_q. However models in which D_o, D_p, and D_q are big enough to satisfy certain comprehension principles will be of special interest because such models are intended interpretations of the language.

It may not be obvious that to interpret a fragment through a first-order language can still qualify as Montague Grammar. Clearly, SMG as defined below preserves the semantic analyses of UG or PTQ. Indeed faithfulness to Montague was an important discipline that made the present work feasible. Suppose the first-order implementations given below are correct. If the sorts 'e', 's', 'o', 'p', and 'q' of the first-order language are interpreted respectively like the types 'e', 's', <s,t>, <e,<s,t>>, and <<e,<s,t>>,<s,t>> of the omega-order language then we can see that the same semantics for SMG is retained in its implementation. To argue rigorously all the way from Montague's UG to the culminating TMS would involve some care over IL versus Ty2, characteristic functions versus sets, and standard versus general models but I believe that the spirit of Montague Grammar is being preserved.

1.4. On the Correctness of the Program

The Prolog program can be judged correct just in case for every sentence ϕ in the SMG fragment and ψ in the first-order language, if it translates ϕ to ψ then there is a ϕ' in Ty2 such that $\phi \Rightarrow \phi'$ and $[\phi' \leftrightarrow \tau\psi]$ is valid in Ty2, where $\Rightarrow$ is the translation relation of the specification and τ is the translation function defined below. There are no free variables in ψ.

A Mapping from TMS into Ty2:

$$\begin{array}{ll}
\tau \mathrm{e} & \equiv \mathrm{e} \\
\tau \mathrm{s} & \equiv \mathrm{s} \\
\tau \mathrm{o} & \equiv \langle \mathrm{s}, \mathrm{t} \rangle \\
\tau \mathrm{p} & \equiv \langle \mathrm{e}, \langle \mathrm{s}, \mathrm{t} \rangle \rangle \\
\tau \mathrm{q} & \equiv \langle \langle \mathrm{e}, \langle \mathrm{s}, \mathrm{t} \rangle \rangle, \langle \mathrm{s}, \mathrm{t} \rangle \rangle
\end{array}$$

$$\begin{array}{ll}
\tau \mathit{rel1}(\ \alpha\ ) & \equiv (\tau \mathit{rel1}\ \ \tau\alpha) \\
\tau \mathit{rel2}(\ \alpha,\ \beta\ ) & \equiv (\tau \mathit{rel2}\ \ \tau\alpha\ \ \tau\beta) \\
\tau \mathit{rel3}(\ \alpha,\ \beta,\ \gamma\ ) & \equiv (\tau \mathit{rel3}\ \ \tau\beta\ \ \tau\alpha\ \ \tau\gamma) \\
\tau \mathit{rel4}(\ \alpha,\ \beta,\ \gamma,\ \delta\ ) & \equiv (\tau \mathit{rel4}\ \ \tau\gamma\ \ \tau\beta\ \ \tau\alpha\ \ \tau\delta)
\end{array}$$

$$\begin{array}{ll}
\tau 0 & \equiv \mathrm{F} \\
\tau 1 & \equiv \mathrm{T} \\
\tau\ \tilde{}\phi & \equiv \neg\ \tau\phi \\
\tau(\phi\ \&\ \psi) & \equiv [\tau\phi \wedge \tau\psi] \\
\tau(\phi\ \mathrm{v}\ \psi) & \equiv [\tau\phi \vee \tau\psi] \\
\tau(\phi \Rightarrow \psi) & \equiv [\tau\phi \rightarrow \tau\psi] \\
\tau(\phi \Leftarrow\Rightarrow \psi) & \equiv [\tau\phi \leftrightarrow \tau\psi]
\end{array}$$

$$\begin{array}{ll}
\tau \mathrm{all}(\ \upsilon{:}\mathrm{a},\ \phi\ ) & \equiv \forall\xi\ \tau^{\xi}_{\upsilon}\phi \quad \text{where } \upsilon = \mathrm{Tn} \text{ and } \xi = \mathrm{v}_{\mathrm{n},\tau \mathrm{a}} \\
\tau \mathrm{exists}(\ \upsilon{:}\mathrm{a},\ \phi\ ) & \equiv \exists\xi\ \tau^{\xi}_{\upsilon}\phi \quad \text{where } \upsilon = \mathrm{Tn} \text{ and } \xi = \mathrm{v}_{\mathrm{n},\tau \mathrm{a}}
\end{array}$$

$$\begin{array}{lll}
\tau^{\xi}_{\upsilon}\upsilon & \equiv \xi & \text{if } \upsilon \text{ is a variable} \\
\tau\alpha & \equiv \mathit{CAPITALISE}\ \alpha & \text{if } \alpha \text{ is a constant}
\end{array}$$

$$\begin{array}{lll}
\tau \mathit{reln} & \equiv \mathit{CAPITALISE}\ \mathit{reln} & \text{if } \mathit{reln} \text{ is a relation other than } =,\ \mathrm{true},\ \mathrm{ppty},\ \mathrm{qtty} \\
\tau = & \equiv \lambda x\ \lambda y\ (x = y) & \\
\tau \mathrm{true} & \equiv \lambda p\ \lambda i\ (p\ i) & \\
\tau \mathrm{ppty} & \equiv \lambda x\ \lambda P\ \lambda i\ (P\ x\ i) & \\
\tau \mathrm{qtty} & \equiv \lambda \mathrm{P}\ \lambda \mathcal{P}\ \lambda i\ (\mathcal{P}\ \mathrm{P}\ i) &
\end{array}$$

The notation τ^{ξ}_{υ} stands for translation in the context where the first-order variable υ is being translated to the Ty2 variable ξ. The function *CAPITALISE* could be defined by numerous cases like: *CAPITALISE* run $\equiv$ RUN. The choice of argument order for relations is arbitrary.

It would be an enormous task to prove the correctness of any program so non-trivial as a parser for Montague Grammar. Evidence of its correctness will be provided by displaying both the TMS and equivalent Ty2 translations.

2. An Extensional Fragment

Let's begin with a tiny extensional fragment which will illustrate the fundamentals without cluttering them with the apparently unmotivated extras which are ultimately required. The regular mode of presentation during this chapter and the next will be to alternately specify and then program portions of the grammar.

2.1. The Syntactic Categories and their Semantic Types

This fragment comprises only two rules. A noun phrase (T) followed by a verb phrase (IV) makes a sentence (S) and a determiner (DET) followed by a common noun (CN) makes a noun phrase. Therefore five categorial symbols will be required. Formal definitions of Montagovian fragments will always be kept together between the heading 'CATEGORY(S)' or 'LEXICON' or 'RULE(S)' and the ending '□'.

```
CATEGORIES

Cat = {S, CN, IV, T, DET}

S    -- Sentence
CN   -- Common Noun
IV   -- Intransitive Verb (verb phrase)
T    -- Term (noun phrase)
DET  -- DETerminer

type(S)   = t
type(CN)  = type(IV)  = <e,t>
type(T)   = <<e,t>,t>
type(DET) = <<e,t>,<<e,t>,t>>
□
```

These are all atomic symbols. I have followed UG in the manner in which the semantic types of the categories have been individually specified. A recursive type map as in PTQ is inappropriate because the categories have no structure to recurse on.

2.1.1. Logical Abbreviations

For the extensional fragment it will not be necessary to use Montague's intensional logic or Gallin's two-sorted type theory. Church's one-sorted type theory is expressive enough. To avoid labouring this point the sub-theory of Ty2 without 's' will actually be used.

$\mathrm{Con}_a = \{ c_{0,a}, c_{1,a}, c_{2,a}, \ldots \}$

let $x \equiv v_{0,e}$ $\quad y \equiv v_{1,e}$

let $P \equiv v_{0,<e,t>}$ $\quad Q \equiv v_{1,<e,t>}$

Beware that the logical abbreviations in this chapter will have to be redefined for the SMG fragment.

2.2. The Syntactic Rules and the Translation Rules

The syntactic rules recursively specify P_A, the set of phrases of each category A. The rule S1† is the base case, for i>1 they have the form

Si. If $\alpha \in P_A$ and $\beta \in P_B$ and ... then $F_n(\alpha, \beta, \ldots) \in P_C$,

where F_n is some previously defined syntactic combination function.

The translation rules simultaneously specify $\Rightarrow$, a translation relation such that for all $\alpha \in P_A$ there are logical expressions α' of type type(A) such that $\alpha \Rightarrow \alpha'$. If the antecedent of Si holds then

Ti. if $\alpha \Rightarrow \alpha'$ and $\beta \Rightarrow \beta'$ and ... then $F_n(\alpha, \beta, \ldots) \Rightarrow (\gamma\ \alpha'\ \beta'\ \ldots)$.

2.2.1. The Lexicon

This provides information about B_A, the set of basic phrases of each category A. The rule S1 is the syntactic rule of lexical insertion. The rules T1 provide translations which assign lexical items appropriate functions in the semantics. Montague called them basic rules.

The (translations of) common nouns and intransitive verbs denote (the characteristic functions of) sets of entities.* In particular, the word **entity** denotes the universal set of entities which contains everything of sort 'e'. Proper nouns denote type-lifted functions in the usual manner of Montague Grammar and determiners denote binary relations between sets of entities.

* The pedantic parenthesised phrases will usually be omitted in statements such as this.

† See page 20

LEXICON

$B_S = \{\}$
$B_{CN} = \{$**man**, **horse**, **entity**$\}$
$B_{IV} = \{$**runs**, **lives**$\}$
$B_T = \{$**John**, **Kate**$\}$
$B_{DET} = \{$**every**, **the**, **a**, **no**$\}$

S1. If $A \in Cat$ and $\alpha \in B_A$ then $\alpha \in P_A$.

T1a. **man** $\Rightarrow M$
horse $\Rightarrow H$
runs $\Rightarrow R$
lives $\Rightarrow L$

where $M, H, R, L \in Con_{<e,t>}$

T1h. **entity** $\Rightarrow \lambda x\ T$

T1d. **John** $\Rightarrow \lambda P\ (P\ j)$
Kate $\Rightarrow \lambda P\ (P\ k)$

where $j, k \in Con_e$

T1f. **every** $\Rightarrow \lambda P\ \lambda Q\ \forall x\ [(P\ x) \rightarrow (Q\ x)]$
the $\Rightarrow \lambda P\ \lambda Q\ \exists x\ [\forall y\ [(P\ y) \leftrightarrow (x=y)]\ \wedge\ (Q\ x)]$
a $\Rightarrow \lambda P\ \lambda Q\ \exists x\ [(P\ x)\ \wedge\ (Q\ x)]$
no $\Rightarrow \lambda P\ \lambda Q\ \neg\exists x\ [(P\ x)\ \wedge\ (Q\ x)]$

□

Corresponding syntactic and translation rules will always be kept side by side. The S-numbers and T-numbers follow those used in PTQ (where they exist) in order to assist the comparison of my fragments with PTQ. This accounts for their apparent disorder because I have rearranged the presentation according to the complexity of the semantic types involved at each stage.

I will often give the Prolog clauses for a rule widely separated from its formal statement so as not to delay commentary best placed between them. The S-numbers and T-numbers serve to strengthen their connection.

2.2.2. Programming the Lexicon

Every lexical entry has the form

word := *syntax* : *semantics*.

The semantic field often refers to a constant of some sort or a relation of some arity ($1 \leq n \leq 4$) with sorted parameters and these are declared thus:

```
constant( constant:sort ).

relation( reln, υ1:sort1, ..., υn:sortn, reln(υ1,...,υn) ).
```

For this extensional fragment the only sort is 'e' and all the relations are unary except for equality.

The syntactic field contains the category of the word. The Prolog atoms whose names are the lower case versions of the members of Cat will be used. Care should be taken to distinguish between 't' the Prolog of T and 't' the type of S. The programming language and the specification language unfortunately have to share the same alphabet but no ambiguity is thereby introduced.

The semantic field contains a semantic template. Let a well-formed formula (WFF) of first-order logic be either a sentence or a term (of any sort). Then a semantic template is a pattern of WFFs. Instead of using higher-order beta-reduction with lambda-expressions, the essential idea of this program is to use first-order template unification which consists of matching patterns of WFFs.

It must first be clearly seen that a semantic template in the program does not share the form of the lambda-expression in the specification. The correct analogy is with the *type* of the expression. Indeed the template shows the type directly though it has to carry more information than that. The following table displays the correspondence where α and β are terms of sort 'e' and ϕ, ψ, and χ are sentences of first-order logic. Which terms and sentences they are depends upon the specific semantics to be programmed.

Category	*Type*	*Template*
S	t	ϕ
CN, IV	<e, t>	$[\alpha \mid \phi]$
T	<<e, t>, t>	$[[\alpha \mid \phi] \mid \psi]$
DET	<<e, t>,<<e, t>, t>>	$[[\alpha \mid \phi] \mid [[\beta \mid \psi] \mid \chi]]$

The template comes from the type by replacing diamond brackets by square brackets, commas by vertical bars, sort symbols by like sorted term schemas, and 't's by sentence schemas. These conventions remain throughout. They should be understood as dogma before undertaking the study of why or whether they work. The reasons lie very deep. Under this regime it is the semantic component of the combination rules which specifies how the templates are to be unified.

Prolog itself is not a typed language and therefore one opportunity for declarative explicitness has been lost. Although attractive variable names have been chosen, no convention is enforced by Prolog and so no information is actually conveyed by those choices. In practice, it does not matter that there is no type checking because type violations are avoided like other bugs by careful programming. This includes following an implicit type regime which in the case of semantic templates is naturally the Montagovian type. For example, variables A and X in the program may correspond respectively to types 't' and 'e' in the specification. However there is no general convention on variable names. Ideally a language supporting such a convention would be preferable.

Throughout this thesis lines beginning with '| ' hold horn clauses (or '%' comments) in the first-order metalanguage of the implementation which can be interpreted as Prolog code. Although the entire program will in due course be so given, the clauses may be disordered and occasionally spurious. Nevertheless Prolog programmers should have no difficulty in constructing a runnable system. In (Jowsey 1986) I quoted a program exactly but it led to presentational problems which are avoided here.

```
| % S1, T1a
| man    := cn  : [X|man(X)].
| horse  := cn  : [X|horse(X)].
| runs   := iv  : [X|run(X)].
| lives  := iv  : [X|live(X)].
|
| relation( man,   X:e, man(X) ).
| relation( horse, X:e, horse(X) ).
| relation( run,   X:e, run(X) ).
| relation( live,  X:e, live(X) ).
```

Let's explore the difference between the programmed translation and a possible Ty2 translation. The template [X|man(X)] is *quite unlike* the expression λx(M x). For example:

- λx(M x) = M by eta-reduction or extensionality but [X|man(X)] ≠ man because a pair and a constant do not unify;

- x is bound in λx(M x) but X is free in [X|man(X)];

- (λx(M x) j) = (M j) by beta-reduction but [X|man(X)](j) is ill-formed;

- λj(M j) is ill-formed because j is a constant but [j|man(j)] is an instance of [X|man(X)].

The fundamental difference is that whereas λυα denotes a function in Ty2, [β|α] denotes an ordered pair in Prolog.

```
| % S1, T1h
| entity := cn  : [X|1].
```

The expression λxT denotes the constant function of truth. The template [X|1] which programs it has a different role, as explained above.

```
| % S1, T1d
| john    := t    : [[j|A]|A].
| kate    := t    : [[k|A]|A].
|
| constant( j:e ).
| constant( k:e ).
```

Many illustrative examples will be included throughout this text. They will all be genuine examples of interaction with the programs which I am presenting. The symbol '<' is the "awaiting input" prompt. The output will comprise the analysis tree and, following the symbol '>', the logical translation. An equivalent translation in Ty2 will also be given for comparison. The symbol 'f0' denotes concatenation.

EXAMPLE - 1

```
<John runs.

 f0| john
   | runs

>run(j)

 (R j)
```

Examples such as this may employ rules which have yet to be defined because they often exist to illuminate preceding lexical entries, from which they should not be separated by rule definitions.

```
| % S1, T1f
| every   := det : [[X|A]|[[X|B]|all(X:e,A=>B)]].
| the     := det : [[Y|A]|[[X|B]|exists(X:e,all(Y:e,A<=>X=Y)&B)]].
| a/an    := det : [[X|A]|[[X|B]|exists(X:e,A&B)]].
| no      := det : [[X|A]|[[X|B]|~exists(X:e,A&B)]].
|
| relation( =, X:e, Y:e, X=Y ).
```

The technique of this program is generally to shift function applications in the lambda-body of the Ty2 translation over to argument unification in the semantic template. For example λP ... (P x) ... corresponds to [[X|A] ... A ...] with [X|A] doing P and A doing (P x).

The lexicon is total over the whole vocabulary of the fragment. There must be no hidden words. Consequently both **a** and **an** have lexical entries but the '/' abbreviation allows them to be merged into a single clause.

Equality is declared to be a binary relation between entities. At present, it is only being used in definite descriptions as follows:

EXAMPLE -2

```
<The horse lives.

 f0| f0| the
   |   | horse
   | lives

>exists(X1:e,all(X2:e,horse(X2)<=>X1=X2)&live(X1))
```

$\exists x\ [\forall y\ [(H\ y) \leftrightarrow (x=y)]\ \wedge\ (L\ x)]$

2.2.3. The Combination Rules

The syntactic combination functions F_n usually follow those used in PTQ but F_0 is always used for simple binary concatenation in my fragments. Indeed that is the only mode of combination needed here. Note that the **a/an** business is handled as part of concatenation. It is outside of phrase structure and syntactic categories are irrelevant to it.

RULES of functional application

$F_0(\alpha, \beta) = \alpha\ \beta$ unless α = **a** and β begins with a vowel sound;
$F_0($**a**$, \beta) =$ **an** β if β begins with a vowel sound.

S2. If $\delta \in P_{DET}$ and $\alpha \in P_{CN}$ then $F_0(\delta, \alpha) \in P_T$.

T2. if $\delta \Rightarrow \delta'$ and $\alpha \Rightarrow \alpha'$ then $F_0(\delta, \alpha) \Rightarrow (\delta'\ \alpha')$.

S4. If $\alpha \in P_T$ and $\beta \in P_{IV}$ then $F_0(\alpha, \beta) \in P_S$.

T4. if $\alpha \Rightarrow \alpha'$ and $\beta \Rightarrow \beta'$ then $F_0(\alpha, \beta) \Rightarrow (\alpha'\ \beta')$.
□

The rule S4 only involves simple concatenation because each IV has already been entered in the lexicon in its third person singular present morphology.

Here is a sample derivation of a sentence in the extensional fragment showing how each syntactic rule and translation rule is used.

```
every ∈ B_DET                                  man ∈ B_CN
------------S1,T1f                             ---------S1,T1a
every ∈ P_DET                                  man ∈ P_CN
every ⇒ λP λQ ∀x [(P x) → (Q x)]               man ⇒ M          runs ∈ B_IV
----------------------------------------------------S2,T2  ----------S1,T1a
       every man ∈ P_T                                       runs ∈ P_IV
       every man ⇒ (λP λQ ∀x [(P x) → (Q x)] M)              runs ⇒ R
-------------------------------------------------------------------------S4,T4
             every man runs ∈ P_S
             every man runs ⇒ ((λP λQ ∀x [(P x) → (Q x)] M) R)
                             = (λQ ∀x [(M x) → (Q x)] R)
                             = ∀x [(M x) → (R x)]
```

The final translation is produced in an unreduced form which has to be simplified by repeated lambda-reductions introducing a considerable computational overhead in more complicated cases. Although it is theoretically possible to interleave lambda-reduction with parsing nevertheless the same steps must be done and they involve creating a variable scope for each lambda-expression and quantification. The program is going to avoid this by producing a first-order translation immediately using semantic templates, which only have one universal scope.

2.2.4. Programming the Combination Rules

The program works by eliminating higher-order constructs entirely. For example, the translation rule **runs** ⇒ R must be eliminated because R (an unapplied predicate constant) is second-order. This has been achieved by the redistribution of the set-theoretic information in it. Taking functions to be sets of ordered pairs, R is {<j,(R j)>, <k,(R k)>, ...}. In this rather loose notation S1 and T1a imply that

runs $\in P_{IV}$,
runs ⇒ {<j,(R j)>, <k,(R k)>, ...}.

The information about whether **John runs, Kate runs**, etc. which is in the meaning of **runs** is displayed horizontally. Now the programmed rule is

```
| runs := iv : [X|run(X)].
```

but the free variable is implicitly universally quantified so this amounts to

```
| runs := iv : [j|run(j)].
| runs := iv : [k|run(k)].
          .     .
          .     .
          .     .
```

in which the same information is displayed vertically.

Almost all the information in the fragment about particular semantic functions is contained in the lexicon in vertical displays. The combination rules then have the task of assembling this information so as to emulate function application. They use first-order unification to do it.

Every rule has the form

$$fn \;>>\; syntax_1 : semantics_1 \;+\; \ldots \;+\; syntax_n : semantics_n \;=>\; syntax_0 : semantics_0 .$$

The two rules below illustrate the method by which function application is programmed. In both cases it happens that the applied function is on the left and its argument is on the right of the concatenation.

```
| % S2, T2
| f0 >> det:[P|PP] + cn:P => t:PP.
|
| % S4, T4
| f0 >> t:[P|A] + iv:P => s:A.
```

The following example trace of the translation mechanism illustrates the unification process and how it implements function application.

```
[1] every              := det  :  [[X|A]|[[X|B]|all(X:e,A=>B)]].
[2] man                := cn   :  [Y|man(Y)].
[3] every man          := t    :  [[X|B]|all(X:e,man(X)=>B)].
[4] runs               := iv   :  [Z|run(Z)].
[5] every man runs     := s    :  all(X:e,man(X)=>run(X)).
```

Lines [1], [2], and [4] are simply reminders of previously seen lexical entries with distinct variables renamed apart. Line [3] results from applying rule S2, T2 to [1] and [2]. Line [5] results from applying rule S4, T4 to [3] and [4]. In [3] and [5] the ':=' operator is being misused in an obvious way because the left-hand side is a phrase here whereas only words are proper in that position. This trace contains no more information than is implicit in the usual analysis tree.

```
EXAMPLE -3

<Every man runs.

 f0| f0| every
   |   | man
   | runs

>all(X1:e,man(X1)=>run(X1))

 ∀x [(M x) → (R x)]
```

It has been shown by (Pereira and Shieber 1987) that it is possible to delay template unification until after the parse but this needlessly complicates the program. Interestingly, that is the reverse of the situation with lambda-reduction.

It cannot be determined from the orthography whether to use **a** or **an**. Furthermore the decision is determined by lexical adjacency rather than by phrase structure. Contrast **a unicorn** with **an uninvited unicorn**. Therefore the program will do the test upon lexical insertion using the information declared in the 'an' procedure*. All words which could be preceded by **a** or **an** and which require the latter will be declared.

```
| an( uninvited ).
| an( entity ).
```

```
EXAMPLE -4

<An entity lives.

 f0| f0| a
   |   | entity
   | lives

>exists(X1:e,1&live(X1))

=exists(X1:e,live(X1))

 ∃x [T ∧ (L x)]
```

Development of the tiny extensional fragment now ceases because it has fulfilled its role in illustrating the fundamental ideas. In the next chapter the SMG fragment will take those ideas a lot further.

* Prolog predicates will always be called "procedures" just to distinguish them from other kinds of predicate. No bias towards procedural interpretation is implied.

3. Simplified Montague Grammar (SMG)

I am concerned to show how to translate sentences in a fragment of English which is derived from Montague's PTQ and UG into first-order logic. The coverage of this fragment will be just as extensive as that of Montague's fragments. It will contain phrases of category sentence, sentential adverb, auxiliary verb, noun phrase, determiner, adjective, common noun, verb phrase, noun phrase complement verb, sentence complement verb, verb phrase complement verb, adverb phrase, and preposition. However its semantics will not be stated using IL but, following (Groenendijk and Stokhof 1982), using Ty2.

The SMG fragment was first presented in (Jowsey 1985) where it had an IL semantics. The first parser and translator for it was in (Jowsey 1986) where the program was explained by means of IL. This experience has convinced me that Ty2 provides a better vehicle for explanation than IL does. Therefore Ty2 replaces IL throughout this thesis. This allows a uniform and exceptionless correspondence between type theory and my system for translating into first-order logic which was previously lacking.

3.1. The Type Map

It is well known that Montague's fragments of English included treatments of modal and intensional matters and so does SMG. However SMG follows UG and almost all later Montague Grammarians in excluding the individual concepts of PTQ, which were the only objects that had non-modal intensionality. Therefore the indexical aspects of the semantics of SMG are most economically encompassed by using a modal logic but for the sake of easy explanation and implementation I am not allowing myself anything other than sorted extensional logic in this thesis. So the standard alternative to modal logic must be used and that is to include explicit indices having the sort 's'.

Montague's original formulations of type theory have been modified by several authors. Bennett makes certain moves to simplify the original mapping from English categories to semantic types. The most well-known system is that given by Dowty, Wall, and Peters who adopt Bennett's type map. Therefore we shall confine our comments to that system and ours.

The correct placement of indices in the type assigned to each category of English is an entirely mechanical matter. In the scheme which I have adopted sentences denote sets of indices and these systematically replace the truth values of the extensional fragment. The semantic templates in the program follow these types as described previously.

CATEGORIES

Cat is the smallest set such that

(1) S, IV, CN $\in$ Cat
(2) if A, B $\in$ Cat then A/B, A//B $\in$ Cat.

type $\in$ $\mathrm{Typ}^{\mathrm{Cat}}$ such that

```
type(S)    = <s,t>
type(IV)   = type(CN)    = <e,<s,t>>
type(A/B)  = type(A//B)  = <type(B),type(A)>      whenever A, B ∈ Cat.
```
□

In contrast, the type map of (Bennett 1974) and (Dowty, Wall, and Peters 1981) went as follows:

```
type(S)    = t
type(IV)   = type(CN)    = <e,t>
type(A/B)  = type(A//B)  = <<s,type(B)>,type(A)>      whenever A, B ∈ Cat.
```

The benefit of treating the types of primitive categories intensionally is that rules of functional application always have the same simple form even when their arguments must be intensional objects.

The coverage of SMG cannot be guessed from the set 'Cat' which contains symbols for an infinity of categories whereas only thirteen of them are categories of phrases. This enables a recursive type map to be given instead of thirteen arbitrary cases but there is no other reason for it. The use of single-slash and double-slash is like that of IV and CN distinguishing verbal and non-verbal categories of the same type. Abbreviatory symbols are assigned to the ten primary slash categories as follows.

AS	=	S/S	TV	=	IV/(S/IV)
AUX	=	S//S	SV	=	IV/S
T	=	S/IV	IVV	=	IV//IV
DET	=	(S/IV)/CN	IAV	=	IV/IV
ACN	=	CN/CN	TAV	=	(IV/IV)/(S/IV)

Together with the three primitive categories S, IV, and CN, these comprise the thirteen primary categories. The symbols have been chosen from Montague's papers or for coherence with those in his papers.

The primary categories are those which are categories of phrases in the fragment. Montague actually had a primitive category (let's call it E with type(E) = <s,e>) which failed to be primary. IV and CN were defined as S/E and S//E respectively. The idea of taking IV and CN as primitives comes from Bennett and goes with the exclusion of individual concepts. Montague was also content to leave a few primary slash categories unabbreviated. In my program, all slash expressions will be eschewed in favour of their abbreviations considered there to be atomic symbols but the corresponding semantic templates will still be justified according to the above type map.

My type map shares with Montague's the property of assigning intensionality generously so as to cater for the most general case. For example, the objects of transitive verbs and prepositions are always taken intensionally although most objects are essentially extensional, and intensional determiners are theoretically possible although none seem to occur in practice. This provides a uniformity of approach at the expense of an unexploited increase in typical complexity.

3.1.1. Logical Abbreviations

$\mathrm{Con}_a = \{ c_{0,a}, c_{1,a}, c_{2,a}, \ldots \}$

let $i \equiv v_{0,s}$ $j \equiv v_{1,s}$ $k \equiv v_{2,s}$

let $x \equiv v_{1,e}$ $y \equiv v_{3,e}$

let $P \equiv v_{0,<e,<s,t>>}$ $Q \equiv v_{1,<e,<s,t>>}$

3.2. Predicates, Terms, and Determiners

In this section I again cover the same ground as the extensional fragment but this time in a way that can be extended to the full SMG fragment. The opportunity arises to introduce in an undemanding context terminology and notation which will be required later. Throughout this chapter, I will regularly alternate within sections between specification and implementation, slowly building up the complexity of the grammar.

```
CATEGORIES

S            -- Sentence
CN           -- Common Noun
IV           -- Intransitive Verb (verb phrase)
T   = S/IV   -- Term (noun phrase)
DET = T/CN   -- DETerminer

type(S)   = <s,t>
type(CN)  = type(IV)  = <e,<s,t>>
type(T)   = <<e,<s,t>>,<s,t>>
type(DET) = <<e,<s,t>>,<<e,<s,t>>,<s,t>>>
□
```

Common nouns and intransitive verbs will be collectively referred to as *predicates*. This is just a generalisation over syntactic categories and not semantic terminology.

In SMG, sentences denote functions from indices to truth values which characterise sets of indices and are referred to as *propositions*. Predicates denote functions from entities to propositions which characterise properties of entities but will be referred to simply as *properties* since without qualification entities will be understood. Terms denote functions from properties to propositions which characterise properties of properties (of entities) and will be referred to as *quantities* (of entities). This terminology is derived from the usual practice of identifying sets of sets of entities with generalised quantifiers.* Determiners denote functions from properties to quantities.

* If a theory of mass terms uses the word "quantity" then its meaning there is not necessarily the same.

LEXICON

B_A = {} if $A \in$ Cat-{CN, IV, T, DET, AS, AUX, SV, ACN, IAV, IVV, TV, TAV}
B_{CN} = {**man**, **woman**, **child**, **horse**, **fish**, **park**, **pen**, **entity**, **unicorn**, **mermaid**}
B_{IV} = {**run**, **walk**, **talk**, **think**, **live**}
B_T = {**John**, **Mary**, **Bill**, **Kate**}*
B_{DET} = {**every**, **the**, **a**, **no**}

S1. If $A \in$ Cat and $\alpha \in B_A$ then $\alpha \in P_A$.

T1a. if $\alpha \in B_{CN}$-{**entity**} U B_{IV} U B_{SV} U B_{ACN}-{**red**} U B_{IAV} U B_{IVV} U

{**conceive**, **worship**} U {**about**} then

$\alpha \Rightarrow c_{n,\,type(A)}$ for some natural number n.

T1h. **entity** $\Rightarrow \lambda x\ \lambda i\ T$

T1d. **John** $\Rightarrow \lambda P\ (P\ J)$
Mary $\Rightarrow \lambda P\ (P\ M)$
Bill $\Rightarrow \lambda P\ (P\ B)$
Kate $\Rightarrow \lambda P\ (P\ K)$

where J, M, B, K $\in Con_e$

T1f. **every** $\Rightarrow \lambda P\ \lambda Q\ \lambda i\ \forall x\ [(P\ x\ i) \rightarrow (Q\ x\ i)]$
the $\Rightarrow \lambda P\ \lambda Q\ \lambda i\ \exists x\ [\forall y\ [(P\ y\ i) \leftrightarrow (x{=}y)] \wedge (Q\ x\ i)]$
a $\Rightarrow \lambda P\ \lambda Q\ \lambda i\ \exists x\ [(P\ x\ i) \wedge (Q\ x\ i)]$
no $\Rightarrow \lambda P\ \lambda Q\ \lambda i\ \neg\exists x\ [(P\ x\ i) \wedge (Q\ x\ i)]$

□

The most important consequence of the above definition of B_A is that B_S = {} and so there are no primitive sentences in the fragment. The vocabulary of SMG is based upon that of PTQ. Although **horse** is not in the PTQ fragment it has been reinstated from the text. Another source is UG from which **entity** and **no** have been taken. The determiners are exactly those of UG. The only other change from PTQ is that **price**, **temperature**, **rise**, **change**, and **ninety** have been replaced by **child**, **mermaid**, **think**, **live**, and **Kate** respectively. The former vocabulary furnished examples illustrating individual concepts which are no longer treated. I am responsible for the latter vocabulary.

The definitions of B_A and T1a are very explicit and exhibit foresight about what categories and basic phrases will occur later. The difficulty could have been hidden by saying "unless mentioned elsewhere". Constants introduced by virtue of T1a can be abbreviated in upper case. For

* In fact B_T also includes the subscripted pronouns $\mathbf{he}_n$ used for quantification.
No more lies about SMG.

example, **man** ⇒ MAN where MAN ∈ $Con_{<e,<s,t>>}$.

Because I have already implemented the extensional fragment more simply, it is clear that here I use heavier machinery than needed. However it is necessary to understand trivial applications of the new methods before their substantial applications can be understood.

In the program, common nouns will now feature gender for agreement purposes. Their semantic fields will each take an additional indexical parameter. Henceforth templates of the form [α1|...[αn|ϕ]...] will be abbreviated to [α1,...,αn|ϕ] as Prolog list notation permits.

```
| % S1, T1a
| man          := cn(masc)     : [X,I|man(X,I)].
| woman        := cn(fem)      : [X,I|woman(X,I)].
| child        := cn(Gend)     : [X,I|child(X,I)].
| horse        := cn(neut)     : [X,I|horse(X,I)].
| fish         := cn(neut)     : [X,I|fish(X,I)].
| park         := cn(neut)     : [X,I|park(X,I)].
| pen          := cn(neut)     : [X,I|pen(X,I)].
| unicorn      := cn(neut)     : [X,I|unicorn(X,I)].
| mermaid      := cn(fem)      : [X,I|mermaid(X,I)].
|
| relation( man,        X:e, I:s, man(X,I) ).
| relation( woman,      X:e, I:s, woman(X,I) ).
| relation( child,      X:e, I:s, child(X,I) ).
| relation( horse,      X:e, I:s, horse(X,I) ).
| relation( fish,       X:e, I:s, fish(X,I) ).
| relation( park,       X:e, I:s, park(X,I) ).
| relation( pen,        X:e, I:s, pen(X,I) ).
| relation( unicorn,    X:e, I:s, unicorn(X,I) ).
| relation( mermaid,    X:e, I:s, mermaid(X,I) ).
```

Only the unavoidable aspects of these changes are present in the new lexical entry for the word **entity** because truth as a logical constant must not be made index dependent.

```
| % S1, T1h
| entity       := cn(neut)     : [X,I|1].
```

Various abbreviations are employed with the ':=' procedure. The 'v' functor abbreviation as it occurs in a lexical entry such as

```
| v(run,runs,ran,Tense) := iv(Tense) : [X,I|run(X,I)].
```

is designed to get the effect of having the lexical entries

```
| run     := iv(base) : [X,I|run(X,I)].
| runs    := iv(pres) : [X,I|run(X,I)].
| ran     := iv(past) : [X,I|run(X,I)].
```

so that the lexicon can be maintained as a total function over the vocabulary without redundancy among the assignments to related verbs. The 'v' functor, defined* below, allows the horizontal abbreviation of vertical redundancy in the verbal paradigm.

```
v(Base,Present,Past,base) = Base
v(Base,Present,Past,pres) = Present
v(Base,Present,Past,past) = Past
```

It is provided that Tense equals either 'base', 'pres', or 'past'.

```
| % S1, T1a
| v(run,      runs,     ran,       Tense) := iv(Tense)   : [X,I|run(X,I)].
| v(walk,     walks,    walked,    Tense) := iv(Tense)   : [X,I|walk(X,I)].
| v(talk,     talks,    talked,    Tense) := iv(Tense)   : [X,I|talk(X,I)].
| v(think,    thinks,   thought,   Tense) := iv(Tense)   : [X,I|think(X,I)].
| v(live,     lives,    lived,     Tense) := iv(Tense)   : [X,I|live(X,I)].
|
| relation( run,       X:e, I:s, run(X,I) ).
| relation( walk,      X:e, I:s, walk(X,I) ).
| relation( talk,      X:e, I:s, talk(X,I) ).
| relation( think,     X:e, I:s, think(X,I) ).
| relation( live,      X:e, I:s, live(X,I) ).
```

Montague used ι* as an abbreviation for λP (P ι). Although SMG will not require such abbreviations its implementation in Prolog will if massive redundancy is to be avoided later. The following definitions illustrate how they will be implemented. A definition such as X *= Sem stands for X* = Sem. A relation is defined rather than a function because Prolog only allows executable code to be attached to relations.

Terms will now feature both gender and case.

```
| % S1, T1d
| john        := t(masc,Case) : j* .
| mary        := t(fem, Case) : m* .
| bill        := t(masc,Case) : b* .
| kate        := t(fem, Case) : k* .
|
| X *= [[X|F]|F] :-
|          constant( X:e ).
|
| constant( j:e ).
| constant( m:e ).
| constant( b:e ).
| constant( k:e ).
```

The '/' operator abbreviation allows two words to share the same lexical assignment without

* The original idea. However the base information must also be available to supply the root of the verb, whatever its tense.

writing it twice. It is expanded when 'v' terms are expanded. The notation 'a(n)', which is a popular alternative to 'a/an', could be implemented easily but with less generality.

```
| % S1, T1f
| every      := det      : [[X,I|A],[X,I|B],I|all(X:e,A=>B)].
| the        := det      : [[Y,I|A],[X,I|B],I|exists(X:e,all(Y:e,A<=>X=Y)&B)].
| a/an       := det      : [[X,I|A],[X,I|B],I|exists(X:e,A&B)].
| no         := det      : [[X,I|A],[X,I|B],I|~exists(X:e,A&B)].
|
| relation( =,      T:S, H:S,      T=H ).
```

RULES of functional application

$F_0(\alpha, \beta) = \alpha\ \beta$ unless α = **a** and β begins with a vowel sound;
$F_0($**a**$, \beta) =$ **an** β if β begins with a vowel sound.

S2. If $\delta \in P_{DET}$ and $\alpha \in P_{CN}$ then $F_0(\delta, \alpha) \in P_T$,

T2. if $\delta \Rightarrow \delta'$ and $\alpha \Rightarrow \alpha'$ then $F_0(\delta, \alpha) \Rightarrow (\delta'\ \alpha')$.

$F_4(\alpha, \beta) = \alpha\ \beta 1$ where $\beta 1$ is the result of replacing the first verb (i.e. member of B_{IV}, B_{TV}, B_{SV}, or B_{IVV}) in β by its third person singular present.

S4. If $\alpha \in P_T$ and $\beta \in P_{IV}$ then $F_4(\alpha, \beta) \in P_S$,

T4. if $\alpha \Rightarrow \alpha'$ and $\beta \Rightarrow \beta'$ then $F_4(\alpha, \beta) \Rightarrow (\alpha'\ \beta')$.

let $! \in Con_s$
□

The implemented rules are very like those given previously but they now take proper account of the features I have introduced.

```
| % S2, T2
| f0 >> det:[P|PP] + cn(Gend):P => t(Gend,Case):PP.
|
| an( alleged ).
| an( entity ).
|
| % S4, T4
| f4 >> t(Gend,subj):[P|F] + iv(pres):P => s:F.
|
| constant( !:s ).
```

When the final sentential semantic template F which is the goal of the parse is produced it will be unified with [!|A] where ! denotes the current index (here and now) and A is the translation of the analysed sentence in first-order logic.

EXAMPLE 1

```
<John runs.

 f4| john
   | run

>run(j,!)

 (RUN J !)
```

EXAMPLE 2

```
<No unicorn lives.

 f4| f0| no
   |   | unicorn
   | live

>~exists(X1:e,unicorn(X1,!)&live(X1,!))
```

$\neg\exists x\ [(\text{UNICORN}\ x\ !) \wedge (\text{LIVE}\ x\ !)]$

3.3. Propositional Modifiers

I will now extend the fragment with two further categories which have the same semantic type in that the translations of phrases in those categories both denote functions from propositions to propositions.

CATEGORIES

```
AS  = S/S     -- Ad-Sentence (sentential adverb)
AUX = S//S    -- AUXiliary verb

type(AS)  = type(AUX) = <<s,t>,<s,t>>
```
□

It is possible to incorporate these new categories into SMG without disturbing the previous definitions because all the semantics of the extensional fragment was rewritten to include explicit indices which now give the leverage required to do the work which in Montague's intensional logic was done by modal and tense operators.

LEXICON

B_{AS} = {**necessarily**, **possibly**}
B_{AUX} = {**does**, **doesn't**, **did**, **didn't**, **will**, **won't**}

T1c. **necessarily** $\Rightarrow \lambda p\ \lambda i\ \forall j\ (p\ j)$
possibly $\Rightarrow \lambda p\ \lambda i\ \exists j\ (p\ j)$
does $\Rightarrow \lambda p\ p$
doesn't $\Rightarrow \lambda p\ \lambda i\ \neg(p\ i)$
did $\Rightarrow \lambda p\ \lambda i\ \exists j\ [(j<i) \wedge (p\ j)]$
didn't $\Rightarrow \lambda p\ \lambda i\ \neg\exists j\ [(j<i) \wedge (p\ j)]$
will $\Rightarrow \lambda p\ \lambda i\ \exists j\ [(i<j) \wedge (p\ j)]$
won't $\Rightarrow \lambda p\ \lambda i\ \neg\exists j\ [(i<j) \wedge (p\ j)]$

where $p \equiv v_{0,<s,t>}$

and $< \in Con_{<s,<s,t>>}$ $(\alpha < \beta) \equiv ((< \alpha)\ \beta)$

$\neg(i<i)$ -- irreflexive

$[(i<j) \wedge (j<k)] \rightarrow (i<k)$ -- transitive

$[(i<k) \wedge (j<k)] \rightarrow [(i<j) \vee (i=j) \vee (j<i)]$ -- left linear

□

The above vocabulary is rather innovative. It comes from PTQ by adding **possibly** and **does**, and by replacing **has** and **hasn't** by **did** and **didn't** respectively. Montague claimed to translate the present perfect but the formula he gave is better as a translation of the simple past. I maintain that denotationally **John does run** equals **John runs** and **John did run** equals **John ran**. I have nothing to say about **John has run** except that it does not equal **John ran**. Here SMG is further from PTQ than anywhere else. It is morphologically simpler as well as allowing the semantics to give a better rendering.

This extensional analysis directly states the indexical manipulations which would be buried in the interpretation of intensional logic. Translations of sentences containing tensed verbs employ a relation of temporal anteriority between indices which can be formalised in a number of ways. Montague had indices as world-time pairs with a total linear ordering on the times. The simplest approach is to leave indices unanalysed and impose temporal anteriority directly on them as a partial ordering. The relation '<' denotes temporal anteriority. Three meaning postulates in SMG define it to be a left linear partial ordering on the indices. It could also have been right linear to follow PTQ more closely but I want models in which past and future are unalike.

```
| % S1, T1c
| necessarily := as          : [[I|A],J|all(I:s,A)].
| possibly    := as          : [[I|A],J|exists(I:s,A)].
| does        := aux         : [F|F].
| doesnt      := aux         : [[I|A],I|~A].
| did         := aux         : [[I|A],J|exists(I:s,I<J&A)].
| didnt       := aux         : [[I|A],J|~exists(I:s,I<J&A)].
| will        := aux         : [[I|A],J|exists(I:s,J<I&A)].
| wont        := aux         : [[I|A],J|~exists(I:s,J<I&A)].
|
| relation( <,        I:s, J:s,        I<J ).
```

It is a benefit of treating English sentences as denoting propositions that rules of functional application such as S9, T9 always have the same simple form even when their arguments must be intensional objects.

RULE of functional application

S9. If $\alpha \in P_{AS}$ and $\phi \in P_S$ then $F_0(\alpha, \phi) \in P_S$.

T9. if $\alpha \Rightarrow \alpha'$ and $\phi \Rightarrow \phi'$ then $F_0(\alpha, \phi) \Rightarrow (\alpha' \ \phi')$.

□

RULE of tense and sign

$F_1(\alpha, \beta, \gamma) = \alpha \ \beta \ \gamma$

$F_2(\alpha, \beta) = \alpha \ \beta 1$ where $\beta 1$ is the result of replacing the first verb in β by its third person singular past.

S17. If $\alpha \in P_T$ and $\gamma \in P_{AUX}$ and $\beta \in P_{IV}$ then $F_1(\alpha, \gamma, \beta)$, $F_2(\alpha, \beta) \in P_S$.

T17. if $\alpha \Rightarrow \alpha'$ and $\gamma \Rightarrow \gamma'$ and $\beta \Rightarrow \beta'$ then $F_1(\alpha, \gamma, \beta) \Rightarrow (\gamma' \ (\alpha' \ \beta'))$,
$F_2(\alpha, \beta) \Rightarrow \lambda i \ \exists j \ [(j<i) \wedge (\alpha' \ \beta' \ j)]$.

□

```
| % S9, T9
| f0 >> as:[F|G] + s:F => s:G.
|
| % S17, T17
| f1 >> t(Gend,subj):[P|F] + aux:[F|G] + iv(base):P => s:G.
|
| % S17, T17
| f2 >> t(Gend,subj):[P,I|A] + iv(past):P => s:[J|exists(I:s,I<J&A)].
```

EXAMPLE 3

```
<Necessarily Mary walks.

 f0| necessarily
   | f4| mary
   |   | walk

>all(I1:s,walk(m,I1))
```

∀j (WALK M j)

EXAMPLE 4

```
<Mary didn't walk.

 f1| mary
   | didnt
   | walk

>~exists(I1:s,I1<!&walk(m,I1))
```

$\neg\exists$j [(j<!) $\wedge$ (WALK M j)]

EXAMPLE 5

```
<Mary walked.

 f2| mary
   | walk

>exists(I1:s,I1<!&walk(m,I1))
```

$\exists$j [(j<!) $\wedge$ (WALK M j)]

3.4. Propositional Attitude Verbs

The distinguishing characteristic of the propositional modifiers was that they all involved the indexical manipulation of sentences but that there was no manipulation of propositional terms in the program. Now we want propositions to be arguments of verbs such as **believe**. In the Montagovian system this is easy to do since propositions are available as functions in Ty2. However in my first-order implementation propositions are represented as index-sentence pairs, which are *not* terms in the logic of translation but templates.

CATEGORY

```
SV  = IV/S     -- Sentence complement Verb

type(SV)  = <<s,t>,<e,<s,t>>>
```
□

Sentence complement verbs are the propositional attitude verbs and denote functions from propositions to properties.

LEXICON

B_{SV} = {**believe**, **assert**, **deny**, **know**, **prove**}
□

This is the vocabulary of UG. PTQ only has **believe** and **assert**.

```
| % S1, T1a
| v(believe, believes, believed, Tense) := sv(Tense)  : believe* .
| v(assert,  asserts,  asserted, Tense) := sv(Tense)  : assert* .
| v(deny,    denies,   denied,   Tense) := sv(Tense)  : deny* .
| v(know,    knows,    knew,     Tense) := sv(Tense)  : know* .
| v(prove,   proves,   proved,   Tense) := sv(Tense)  : prove* .
|
| R *= [[J|A],X,I|exists(U:o,Phi&Psi)] :-
|         relation( R, X:e, U:o, I:s, Phi ),
|         Psi = all(J:s,true(U,J)<=>A).
|
| relation( believe,  X:e, U:o, I:s, believe(X,U,I) ).
| relation( assert,   X:e, U:o, I:s, assert(X,U,I) ).
| relation( deny,     X:e, U:o, I:s, deny(X,U,I) ).
| relation( know,     X:e, U:o, I:s, know(X,U,I) ).
| relation( prove,    X:e, U:o, I:s, prove(X,U,I) ).
|
| relation( true,     U:o, I:s,      true(U,I) ).
```

The solution which I have adopted is to introduce a new first-order sort 'o' of propositions and to relate objects of this sort to indices via a relation 'true' such that true(U,I) holds iff the proposition U is true at the index I. Other authors such as (Thomason 1980) have also advocated taking propositions as a sort rather than as a functional type.

The Montagovian propositional abstracts $\lambda i \phi$ are then eliminated by saying that there is a proposition which is true at just those indices where ϕ holds. For example

John believes that ϕ
$\Rightarrow$ (BELIEVE $\lambda i \phi$ J),
(BELIEVE $\lambda i \phi$ J !)
= $\exists p$ [(BELIEVE p J !) $\wedge$ (p=$\lambda i \phi$)]
= $\exists p$ [(BELIEVE p J !) $\wedge$ $\forall i$ [(p i) $\leftrightarrow$ ϕ]]
= exists(U:o,believe(j,U,!)&all(I:s,true(U,I)<=>ϕ)).

The hard work of setting up this analysis has been done in the lexicon. The actual rule required is simply one of functional application.

RULE of functional application

$F_6(\alpha, \beta) = \alpha$ **that** β

S7. If $\delta \in P_{SV}$ and $\phi \in P_S$ then $F_6(\delta, \phi) \in P_{IV}$,

T7. if $\delta \Rightarrow \delta'$ and $\phi \Rightarrow \phi'$ then $F_6(\delta, \phi) \Rightarrow (\delta' \ \phi')$.
□

```
| that          := part(that)    : [].
|
| % S7, T7
| f6 >> sv(Tense):[F|P] + part(that):[] + s:F => iv(Tense):P.
```

Note that the requirement that the lexicon be total has been fulfilled by including an entry for the particle **that**. Words which are syncategorematic in SMG must be categorised in the implementation but get a null semantics.

EXAMPLE 6

```
<John believes that Kate thinks.

 f4| john
   | f6| believe
   |   | f4| kate
   |   |   | think

>exists(U1:o,believe(j,U1,!)&all(I2:s,true(U1,I2)<=>think(k,I2)))

 (BELIEVE (THINK K) J !)
```

This holds iff there is a proposition U1 such that John believes U1 and U1 is that Kate thinks.

This example illustrates how intensionality is expressed in TMS. It can be seen that the TMS and the Ty2 are equivalent because (THINK K) equals λi(THINK K i) which has the form $\lambda i\phi$ studied above. Suppose that everything which thinks talks and vice versa:

```
all(X:e,think(X,!)<=>talk(X,!))
```

Then it does *not* follow from the example that John believes that Kate talks. The bad inference is blocked in TMS, Ty2, and in Montague's original IL by modal closure. Only the method of closure varies between logics. In TMS it is a universal quantifier, in Ty2 it is lambda, and in IL it was the cap operator.

It is reassuring to follow the trace of this example and see how the translation mechanism handles opaque contexts such as the sentential complements of **believe**.

```
[1] kate := t(fem,Case) : [[k|F]|F].
[2] thinks := iv(pres) : [X,I|think(X,I)].
[3] kate thinks := s : [I|think(k,I)].
[4] believes := sv(pres) : [[J|A],Y,K|
                    exists(U:o,believe(Y,U,K)&all(J:s,true(U,J)<=>A))].
[5] that := part(that) : [].
[6] believes that kate thinks := iv(pres) : [Y,K|
                    exists(U:o,believe(Y,U,K)&all(J:s,true(U,J)<=>think(k,J)))].
[7] john := t(masc,Case) : [[j|G]|G].
[8] john believes that kate thinks := s : [K|
                    exists(U:o,believe(j,U,K)&all(J:s,true(U,J)<=>think(k,J)))].
```

In the program, sentences are translated to templates of the form [I|ϕ] as in lines [3] and [8]. The lexical entry for **believes** is computed to be [4] and goes with [3] into rule S7, T7 to result in [6]. The index variable I of [3] is unified with J in [4] and passes inside [6] and [8]. A different one K is introduced by [4] and becomes the index variable of [6] and [8]. K will be bound to '!' upon output.

Simple applications of S7, T7 such as this always effect a complete change of context and therefore only de dicto readings can be produced so far. A different kind of rule will be required for de re readings.

3.5. Predicate Modifiers

The next step up in the type hierarchy involves the introduction of three categories of ad-word which are semantically functions from properties to properties. To get this into first-order logic properties will be introduced as a new sort 'p' and a relation 'ppty' will be employed so that ppty(V,X,I) holds iff the entity X has the property V at the index I.

CATEGORIES

```
ACN = CN/CN    -- Ad-Common Noun (adjective)
IAV = IV/IV    -- Intransitive AdVerb (adverb phrase)
IVV = IV//IV   -- Intransitive Verb complement Verb

type(ACN) = type(IAV) = type(IVV) = <<e,<s,t>>,<e,<s,t>>>
```
□

I have included a simple treatment of the intersective adjective **red** in SMG but have made no attempt to distinguish the subsective **tall** because it is like the adverbs **rapidly**, **slowly**, and **voluntarily** which Montague was content to leave.

LEXICON

B_{ACN} = {**red**, **tall**, **alleged**}
B_{IAV} = {**rapidly**, **slowly**, **voluntarily**, **allegedly**}
B_{IVV} = {**try**, **wish**}

T1g. **red** $\Rightarrow \lambda P\ \lambda x\ \lambda i\ [(RED\ x\ i) \wedge (P\ x\ i)]$

where RED $\in Con_{<e,<s,t>>}$
□

Because PTQ has no adjectives, **tall** and **alleged** are from UG while **red** is mine. The rest is the vocabulary of PTQ.

3.5.1. Extensional Adjective

```
| % S1, T1g
| red          := acn          : [[X,I|A],X,I|red(X,I)&A].
|
| relation( red,       X:e, I:s, red(X,I) ).
```

Only syntactically a predicate modifier, **red** reduces semantically to a conjoined predication and so properties are not yet required.

EXAMPLE 7

```
<A red horse lives.

 f4| f0| a
   |   | f0| red
   |   |   | horse
   | live

>exists(X1:e,(red(X1,!)&horse(X1,!))&live(X1,!))

 ∃x [[(RED x !) ∧ (HORSE x !)] ∧ (LIVE x !)]
```

3.5.2. Intensional Predicate Modifiers

We know that **tall**, **rapidly**, **slowly**, and **voluntarily** are subsective: every tall man is a man, if John ran rapidly then John ran, etc. Moreover it seems that **tall** is extensional and that exceptionally it does not create an opaque context (Keenan 1984). However SMG follows Montague in not recognising these facts.

```
| % S1, T1a
| tall        := acn          : tall* .
| alleged     := acn          : alleged* .
|
| rapidly     := iav          : rapid* .
| slowly      := iav          : slow* .
| voluntarily := iav          : voluntary* .
| allegedly   := iav          : alleged* .
|
| v(try,    tries,    tried,    Tense) := ivv(Tense) : try* .
| v(wish,   wishes,   wished,   Tense) := ivv(Tense) : wish* .
|
| R *= [[Y,J|A],X,I|exists(V:p,Phi&Psi)] :-
|         relation( R, X:e, V:p, I:s, Phi ),
|         Psi = all(Y:e,all(J:s,ppty(V,Y,J)<=>A)).
|
| relation( tall,      X:e, V:p, I:s, tall(X,V,I) ).
| relation( alleged,   X:e, V:p, I:s, alleged(X,V,I) ).
| relation( rapid,     X:e, V:p, I:s, rapid(X,V,I) ).
| relation( slow,      X:e, V:p, I:s, slow(X,V,I) ).
| relation( voluntary, X:e, V:p, I:s, voluntary(X,V,I) ).
| relation( try,       X:e, V:p, I:s, try(X,V,I) ).
| relation( wish,      X:e, V:p, I:s, wish(X,V,I) ).
|
| relation( ppty,      V:p, X:e, I:s, ppty(V,X,I) ).
```

This implementation is really a straightforward application of the principles already described apart from the novelty of the templates for predicate modifiers. Again I have provided for the hard work to be done in the lexicon so that only functional application rules are required.

RULES of functional application

S3a. If $\delta \in P_{ACN}$ and $\alpha \in P_{CN}$ then $F_0(\delta, \alpha) \in P_{CN}$.

T3a. if $\delta \Rightarrow \delta'$ and $\alpha \Rightarrow \alpha'$ then $F_0(\delta, \alpha) \Rightarrow (\delta' \alpha')$.

S10. If $\delta \in P_{IAV}$ and $\alpha \in P_{IV}$ then $F_0(\alpha, \delta) \in P_{IV}$.

T10. if $\delta \Rightarrow \delta'$ and $\alpha \Rightarrow \alpha'$ then $F_0(\alpha, \delta) \Rightarrow (\delta' \alpha')$.

$F_7(\alpha, \beta) = \alpha$ **to** β

S8. If $\delta \in P_{IVV}$ and $\alpha \in P_{IV}$ then $F_7(\delta, \alpha) \in P_{IV}$.

T8. if $\delta \Rightarrow \delta'$ and $\alpha \Rightarrow \alpha'$ then $F_7(\delta, \alpha) \Rightarrow (\delta' \alpha')$.
□

```
| % S3a, T3a
| f0 >> acn:[P|Q] + cn(Gend):P => cn(Gend):Q.
|
| % S10, T10
| f0 >> iv(Tense):P + iav:[P|Q] => iv(Tense):Q.
|
| to            := part(to)      : [].
|
| % S8, T8
| f7 >> ivv(Tense):[P|Q] + part(to):[] + iv(base):P => iv(Tense):Q.
```

EXAMPLE 8

```
<An alleged horse lives.

 f4| f0| a
   |   | f0| alleged
   |   |   | horse
   | live

>exists(X1:e,exists(V2:p,alleged(X1,V2,!)&
 all(X3:e,all(I4:s,ppty(V2,X3,I4)<=>horse(X3,I4))))&
 live(X1,!))

 ∃x [(ALLEGED HORSE x !) ∧ (LIVE x !)]
```

This holds iff there are an entity X1 and a property V2 such that X1 lives, X1 is alleged V2, and V2 is to be a horse.

EXAMPLE 9

```
<John runs rapidly.

 f4| john
   | f0| run
   |   | rapidly

>exists(V1:p,rapid(j,V1,!)&all(X2:e,all(I3:s,ppty(V1,X2,I3)<=>run(X2,I3))))

 (RAPID RUN J !)
```

This holds iff there is a property V1 such that John does V1 rapidly and V1 is to run.

EXAMPLE 10

```
<John tries to think.

 f4| john
   | f7| try
   |   | think

>exists(V1:p,try(j,V1,!)&all(X2:e,all(I3:s,ppty(V1,X2,I3)<=>think(X2,I3))))

 (TRY THINK J !)
```

This holds iff there is a property V1 such that John tries V1 and V1 is to think.

3.6. Transitive Verbs and Prepositions

We now come to what are semantically the most complex types in SMG. These are for functions from quantities to properties, and functions from quantities to functions from properties to properties. Quantities will be assigned the sort 'q' and qtty(W,V,I) will hold iff the property V has the quantity W at the index I.

All the sorts required to give a first-order account of SMG have now been presented. Terms are of the sort entity 'e', index 's', proposition 'o', property 'p', or quantity 'q'.

```
CATEGORIES

TV  = IV/T    -- Transitive Verb or Term complement Verb
TAV = IAV/T   -- Transitive AdVerb or Term complement AdVerb (preposition)

type(TV)  = <<<e,<s,t>>,<s,t>>,<e,<s,t>>>
type(TAV) = <<<e,<s,t>>,<s,t>>,<<e,<s,t>>,<e,<s,t>>>>
□

LEXICON

B_TV = {find, lose, catch, eat, love, date, be, seek, conceive, worship}
B_TAV= {in, about}

T1b. find      ⇒ λP λx (P λy (FIND y x))
     lose      ⇒ λP λx (P λy (LOSE y x))
     catch     ⇒ λP λx (P λy (CATCH y x))
     eat       ⇒ λP λx (P λy (EAT y x))
     love      ⇒ λP λx (P λy (LOVE y x))
     date      ⇒ λP λx (P λy (DATE y x))
     be        ⇒ λP λx (P λy λi (x=y))
     seek      ⇒ λP (try' (find' P))   where try ⇒ try' and find ⇒ find'.
T1i. in        ⇒ λP λP λx (P λy (IN y P x))
```

where $P \equiv v_{0,<<e,<s,t>>,<s,t>>}$

and FIND, LOSE, CATCH, EAT, LOVE, DATE $\in Con_{<e,<e,<s,t>>>}$

and IN $\in Con_{<e,<<e,<s,t>>,<e,<s,t>>>>}$

□

This is the vocabulary of PTQ. Although **catch** and **worship** are not in the fragment they have been reinstated from the text.

3.6.1. Extensional Transitive Verbs

Like the intersective adjective **red** the verbs **find**, **lose**, **catch**, **eat**, **love**, **date**, and **be** are entirely extensional.

```
| % S1, T1b
| v(find,    finds,    found,    Tense) := tv(Tense)  : find* .
| v(lose,    loses,    lost,     Tense) := tv(Tense)  : lose* .
| v(catch,   catches,  caught,   Tense) := tv(Tense)  : catch* .
| v(eat,     eats,     ate,      Tense) := tv(Tense)  : eat* .
| v(love,    loves,    loved,    Tense) := tv(Tense)  : love* .
| v(date,    dates,    dated,    Tense) := tv(Tense)  : date* .
|
| R *= [[[Y,I|Phi]|F],X|F] :-
|         relation( R, X:e, Y:e, I:s, Phi ).
|
| relation( find,     X:e, Y:e, I:s, find(X,Y,I) ).
| relation( lose,     X:e, Y:e, I:s, lose(X,Y,I) ).
| relation( catch,    X:e, Y:e, I:s, catch(X,Y,I) ).
| relation( eat,      X:e, Y:e, I:s, eat(X,Y,I) ).
| relation( love,     X:e, Y:e, I:s, love(X,Y,I) ).
| relation( date,     X:e, Y:e, I:s, date(X,Y,I) ).
```

There was obviously great redundancy in the statement of the lexical rule T1b which could have been removed by extending Montague's superscript '*' notation used in **John** ⇒ J* to cover **find** ⇒ FIND*. This is essentially the approach which I have used in the implementation. Do not confuse this with Montague's subscript '$_*$'. Montague had **love** ⇒ LOVE and **John loves Mary** ⇒ ((LOVE$_*$ M) J) but I have **love** ⇒ LOVE* and **John loves Mary** ⇒ ((LOVE M) J).

EXAMPLE 11

```
<John loves Mary.

 f4| john
   | f5| love
   |   | mary

>love(j,m,!)

 (LOVE M J !)
```

The reason why I have made these changes is to get rid of Montague's meaning postulates. Many of the meaning postulates of PTQ disappear with individual concepts. That which made individual constants rigid designators disappears because an extensional logic is used with constants of sort 'e' for proper names which ensures rigid designation. The rest go with the change from subscript '$_*$' to superscript '*'.

```
| % S1, T1b
| v(be,        is,        was,        Tense) := tv(Tense)   : [[[Y,I|X=Y]|F],X|F].
```

The translation of **be** is essentially the equality relation.

EXAMPLE 12

```
<John is Bill.

 f4| john
   | f5| be
   |   | bill

>j=b

 (J=B)
```

This treatment of the copula does not easily accommodate predicative adjectives. In **John is red**, it does not appear that **is** corresponds to equality. However in UG, Montague provided the following equivalent.

EXAMPLE 13

```
<John is a red entity.

 f4| john
   | f5| be
   |   | f0| a
   |   |   | f0| red
   |   |   |   | entity

>exists(X1:e,(red(X1,!)&1)&j=X1)

=red(j,!)

 ∃y [[(RED y !) ∧ T] ∧ (J=y)]
```

3.6.2. Intensional Transitive Verbs

```
| % S1, T1b
| v(seek,     seeks,     sought,     Tense) := tv(Tense)   : seek* .
|
| seek *= [[[Z,J|find(Y,Z,J)],J|A],X,I|exists(V:p,try(X,V,I)&Phi)] :-
|           Phi = all(Y:e,all(J:s,ppty(V,Y,J)<=>A)).
```

The translation of **seek** is the composition of the translations of **try** and **find**.

EXAMPLE 14

```
<John seeks a unicorn.

 f4| john
   | f5| seek
   |   | f0| a
   |   |   | unicorn

>exists(V1:p,try(j,V1,!)&
 all(X2:e,all(I3:s,ppty(V1,X2,I3)<=>
 exists(X4:e,unicorn(X4,I3)&find(X2,X4,I3)))))

 (TRY λx λi ∃y [(UNICORN y i) ∧ (FIND y x i)] J !)
```

This holds iff there is a property V1 such that John tries V1 and V1 is to find a unicorn.

```
| % S1, T1a
| v(conceive,conceives,conceived, Tense) := tv(Tense)   : conceive* .
| v(worship, worships, worshipped,Tense) := tv(Tense)   : worship* .
|
| R *= [[[Y,J|ppty(V,Y,J)],J|A],X,I|exists(W:q,Phi&Psi)] :-
|          relation( R, X:e, W:q, I:s, Phi ),
|          Psi = all(V:p,all(J:s,qtty(W,V,J)<=>A)).
|
| relation( conceive,  X:e, W:q, I:s, conceive(X,W,I) ).
| relation( worship,   X:e, W:q, I:s, worship(X,W,I) ).
|
| relation( qtty,      W:q, V:p, I:s, qtty(W,V,I) ).
```

This is the first genuine use of quantities. It is the implementation of Montague's famous treatment of the opaque objects of intensional transitive verbs.

EXAMPLE 15

```
<John conceives a unicorn.

 f4| john
   | f5| conceive
   |   | f0| a
   |   |   | unicorn

>exists(W1:q,conceive(j,W1,!)&
 all(V2:p,all(I3:s,qtty(W1,V2,I3)<=>
 exists(X4:e,unicorn(X4,I3)&ppty(V2,X4,I3)))))

 (CONCEIVE λQ λi ∃x [(UNICORN x i) ∧ (Q x i)] J !)
```

This holds iff there is a quantity W1 such that John conceives (of) W1 and W1 is to be a property of a unicorn.

3.6.3. Extensional Preposition

```
| % S1, T1i
| in            := tav           : in* .
|
| R *= [[[Y,I|exists(V:p,Phi&Psi)]|F],[Z,J|A],X|F] :-
|          relation( R, X:e, V:p, Y:e, I:s, Phi ),
|          Psi = all(Z:e,all(J:s,ppty(V,Z,J)<=>A)).
|
| relation( in,         X:e, V:p, Y:e, I:s, in(X,V,Y,I) ).
```

Although **in** is often described as an extensional preposition we see that in SMG it is partly intensional despite having a transparent object. The predication in(X,V,Y,I) holds iff X does V in Y at I.

EXAMPLE 16

```
<John walks in a park.

  f4| john
    | f0| walk
    |   | f5| in
    |   |   | f0| a
    |   |   |   | park

>exists(X1:e,park(X1,!)&exists(V2:p,in(j,V2,X1,!)&
 all(X3:e,all(I4:s,ppty(V2,X3,I4)<=>walk(X3,I4)))))

 ∃y [(PARK y !) ∧ (IN y WALK J !)]
```

This holds iff there are an entity X1 and a property V2 such that X1 is a park, John does V2 in X1, and V2 is to walk.

3.6.4. Intensional Preposition

```
| % S1, T1a
| about         := tav           : about* .
|
| R *= [[[Z,J|ppty(V1,Z,J)],J|A],[Y,K|B],X,I
|        |exists(V:p,exists(W:q,Phi&Psi)&Chi)] :-
|          relation( R, X:e, V:p, W:q, I:s, Phi ),
|          Psi = all(V1:p,all(J:s,qtty(W,V1,J)<=>A)),
|          Chi = all(Y:e,all(K:s,ppty(V,Y,K)<=>B)).
|
| relation( about,      X:e, V:p, W:q, I:s, about(X,V,W,I) ).
```

This is the fully intensional preposition. As a four-place relation between entities, properties, quantities, and indices it has the most complex type in the fragment. The predication about(X,V,W,I) holds iff X does V about W at I.

EXAMPLE 17

```
<John talks about a unicorn.

  f4| john
    | f0| talk
    |   | f5| about
    |   |   | f0| a
    |   |   |   | unicorn

>exists(V1:p,exists(W2:q,about(j,V1,W2,!)&
 all(V3:p,all(I4:s,qtty(W2,V3,I4)<=>
 exists(X5:e,unicorn(X5,I4)&ppty(V3,X5,I4)))))&
 all(X2:e,all(I3:s,ppty(V1,X2,I3)<=>talk(X2,I3))))
```

(ABOUT λQ λi $\exists$x [(UNICORN x i) $\wedge$ (Q x i)] TALK J !)

This holds iff there are a property V1 and a quantity W2 such that John does V1 about W2, V1 is to talk, and W2 is to be a property of a unicorn.

Comparing the SMG and the Prolog for **in** and **about** we can observe a certain complementarity. In SMG **about** $\Rightarrow$ ABOUT but **in** translates to the complicated expression **in'** given in T1i. Yet the mechanism of **in'** is primarily to retract the default opaque expectation. This appears implausible since surely the transparent case is the unmarked one. In the implementation however things are the right way around because here the hard work is done to establish an opaque context.

RULES of functional application

$F_5(\alpha, \beta) = \alpha\ \beta$ unless β has the form $\mathbf{he}_n$;*
$F_5(\alpha, \mathbf{he}_n) = \alpha\ \mathbf{him}_n$

S5. If $\delta \in P_{TV}$ and $\alpha \in P_T$ then $F_5(\delta, \alpha) \in P_{IV}$.

T5. if $\delta \Rightarrow \delta'$ and $\alpha \Rightarrow \alpha'$ then $F_5(\delta, \alpha) \Rightarrow (\delta'\ \alpha')$.

S6. If $\zeta \in P_{TAV}$ and $\alpha \in P_T$ then $F_5(\zeta, \alpha) \in P_{IAV}$.

T6. if $\zeta \Rightarrow \zeta'$ and $\alpha \Rightarrow \alpha'$ then $F_5(\zeta, \alpha) \Rightarrow (\zeta'\ \alpha')$.
□

```
| % S5, T5
| f5 >> tv(Tense):[PP|P] + t(Gend,obj):PP => iv(Tense):P.
|
| % S6, T6
| f5 >> tav:[PP|P_Q] + t(Gend,obj):PP => iav:P_Q.
```

* See page 60

3.7. Coordination

The coordination of sentences and of predicates is very straightforward and could easily have been introduced much earlier to no benefit. However the coordination of terms is difficult to achieve using unification and highlights its difference from lambda-reduction.

3.7.1. Sentential Coordinators

Like all coordinators these are basically extensional. There is nothing to do but spread the index. In a lambda-expression spreading involves the abstracted variable occurring multiply within the body distributed at regular positions.

RULE of conjunction and disjunction

$F_8(\ \alpha,\ \beta\ ) = \alpha$ **and** β

$F_9(\ \alpha,\ \beta\ ) = \alpha$ **or** β

S11. If $\phi,\ \psi \in P_S$ then $F_8(\phi,\ \psi),\ F_9(\phi,\ \psi) \in P_S$.

T11. if $\phi \Rightarrow \phi'$ and $\psi \Rightarrow \psi'$ then $F_8(\phi,\ \psi) \Rightarrow \lambda i\ [(\phi'\ i) \wedge (\psi'\ i)]$,
$F_9(\phi,\ \psi) \Rightarrow \lambda i\ [(\phi'\ i) \vee (\psi'\ i)]$.

□

```
| and          := c(and)        : [].
| or           := c(or)         : [].
|
| % S11, T11
| Fn >> s:[I|A] + c(Co):[] + s:[I|B] => s:[I|C]
|    <- co_ordinate( Co, Fn, A, B, C ).
|
| co_ordinate( and, f8, A, B, A & B ).
| co_ordinate( or,  f9, A, B, A v B ).
```

The condition introduced by '<-' is declaratively identical to one introduced by a Prolog ':-' but it is called from within the parser for greater efficiency.

EXAMPLE 18

```
<John walks and Bill talks.

 f8| f4| john
   |   | walk
   | f4| bill
   |   | talk

>walk(j,!)&talk(b,!)

 [(WALK J !) Λ (TALK B !)]
```

3.7.2. Predicate Coordinators

These involve spreading the subject and the index. The only predicates which Montague coordinated were intransitive verbs but a similar rule could easily be given for common nouns.

RULE of conjunction and disjunction

S12. If α, $\beta \in P_{IV}$ then $F_8(\alpha, \beta)$, $F_9(\alpha, \beta) \in P_{IV}$.

T12. if $\alpha \Rightarrow \alpha'$ and $\beta \Rightarrow \beta'$ then $F_8(\alpha, \beta) \Rightarrow \lambda x\ \lambda i\ [(\alpha'\ x\ i) \wedge (\beta'\ x\ i)]$,
$F_9(\alpha, \beta) \Rightarrow \lambda x\ \lambda i\ [(\alpha'\ x\ i) \vee (\beta'\ x\ i)]$.

□

```
| % S12, T12
| Fn >> iv(Tense):[X,I|A] + c(Co):[] + iv(Tense):[X,I|B] => iv(Tense):[X,I|C]
|     <- co_ordinate( Co, Fn, A, B, C ).
```

EXAMPLE 19

```
<John walks and talks.

 f4| john
   | f8| walk
   |   | talk

>walk(j,!)&talk(j,!)

 [(WALK J !) Λ (TALK J !)]
```

3.7.3. Term Coordinator

Noun phrase coordination poses problems for the unification approach. Montague's second-order translation is not convertible to a workable semantic template. There is also doubt (in English itself) about the gender of a disjunction when the disjuncts have different genders. Montague restricted term coordination to disjunction merely to avoid the plurality of conjoined terms.

RULE of disjunction

S13. If α, $\beta \in P_T$ then $F_9(\alpha, \beta) \in P_T$.

T13. if $\alpha \Rightarrow \alpha'$ and $\beta \Rightarrow \beta'$ then $F_9(\alpha, \beta) \Rightarrow \lambda P\ \lambda i\ [(\alpha'\ P\ i)\ V\ (\beta'\ P\ i)]$.
□

```
* % S13, T13
* f9 >> t(Gend,Case):[P,I|A] + c(or):[] + t(Gend1,Case):[P,I|B]
*     => t(Gend,Case):[P,I|A v B].
```

This naive attempt to implement term coordination following SMG fails to yield a template for **John or Mary** because 'P' cannot be unified with both [j|F] and [m|G] at once. Although a predication such as walk(X) can be spread, this only provides walk(X) v walk(X) whereas **John or Mary walks** seems to require walk(X) v walk(Y). The problem arises because I use open templates where Montague used closed lambda-expressions for which variables having different binding occurrences can be made distinct by alpha-conversion.

Fortunately there is an adequate first-order semantics. It involves an entity X such that X=j or X=m. There is no need to spread the predicate.

```
| % S13, T13
| f9 >> t(Gend1,Case):[[Y,I|X=Y],I|A] + c(or):[] + t(Gend2,Case):[[Z,I|X=Z],I|B]
|     => t(Gend,Case):[[X,I|C],I|exists(X:e,(A v B)&C)]
|     <- or_gender( Gend1, Gend2, Gend ).
|
| or_gender( Gend,  Gend,  Gend ).
| or_gender( Gend1, Gend2, neut ) :-
|          Gend1 \= Gend2.
```

The problem with the gender of a coordination such as **John or Mary** is solved by arbitrary stipulation as neuter. Something must be chosen and I will choose **it** because the solution using singular **they** is way beyond this fragment.

EXAMPLE 20

```
<John or Mary walks.

 f4| f9| john
   |   | mary
   | walk

>exists(X1:e,(X1=j v X1=m)&walk(X1,!))

 [(WALK J !) V (WALK M !)]
```

* Buggy code, not part of the program.

The plural term **John and Mary** is not given in Montague Grammar but it is clear what translation **John and Mary walk** would have under this quantificational analysis.

```
>all(X1:e,(X1=j v X1=m)=>walk(X1,!))
```

3.8. Parsing

The phrase structural component of the grammar has now been covered. The parser interprets those phrase structure rules. As Montague's schematic rules cannot be stated in the same format, they will be programmed directly into the parser. An understanding of the parsing mechanism will therefore be required in order to follow the subsequent exposition of the handling of scope and bound anaphora.

The parser employed in this system uses a bottom-up, right to left, shift and reduce algorithm. Its bottom-up nature avoids the problem of looping on parsing left-recursive structures suffered by top-down algorithms such as the direct Prolog interpretation of definite clause grammars.

In Montague Grammar the left-recursive structures are postmodified phrases such as the IV **run rapidly voluntarily allegedly** or the CN **man such that he walks such that he talks** and coordinate structures. Furthermore the rules of quantifying-in which will be used to treat the phenomena of scope and bound anaphora are syntactically vacuous rewrite rules. With a top-down approach it would not be clear what semantic information to associate with them but while doing a bottom-up analysis all the information which could be relevant is available.

Despite this requirement to treat left-recursion most of the structures in Montague Grammar are right-recursive. Therefore a right to left parse will be pursued purely for the sake of efficiency. All parses would be found eventually in either direction but this way the usual parses will be found faster.

A shift and reduce, stack and buffer parser is the simplest method of implementing bottom-up analysis of grammatical phrases. Even after more efficient methods have been found I think that this remains the most compelling conceptual model. Since I am primarily interested in semantics rather than parsing I will be content with programming it directly. The essential non-determinacy of parsing will be captured by relying upon Prolog's own built-in non-determinacy implemented as backtracking.

The procedure call parse(Stack, Buffer, N, Answer) succeeds when the arcs already parsed on the Stack, followed by the words remaining in the Buffer, with N subscripted pronouns, parse as the arc Answer.

Every arc has the form

syntactic category # syntactic structure # semantic template # Cooper store

and contains all the information needed about one constituent phrase. The syntactic category and the semantic template above are exactly like the *syntax:semantics* to be found in the combination rules.

The syntactic structure contains inessential information useful for tracing which corresponds to Montague's analysis tree. It holds a record of the parse of a constituent. The labels 'fn' from the statements *fn>>rule* are used to cross reference analysis trees, rules, and their SMG definitions. A syntactic structure either is a root word or has the form *fn'arguments* where 'fn' is applied to a list of arguments which are syntactic structures recursively.

The Cooper store contains binding information about quantified terms. Its operation will be described fully in the section on quantification.

The initial call to the parser is encapsulated in a convenient three argument procedure:

```
| % PARSE SENTENCE
| parse( Buffer, Str, A ) :-
|         parse( [], Buffer, 0, s#Str#[!|A]#[] ).
```

There is nothing on the stack, the Buffer is the list of words to be analysed, it should contain zero subscripted pronouns*, it should be a sentence with the syntactic structure Str and (letting the local index be '!') with the translation A where all variables have been bound.†

* although SMG overgenerates with phrases containing them.

† To run the extensional fragment of Chapter 2 substitute this procedure:

```
| parse( Buffer, Str, A ) :-
|         parse( [], Buffer, 0, s#Str#A#[] ).
```

The termination state of the parser is

```
| % PARSE FOUND
| parse( [Arc], [], _, Arc ).
```

The stack holds only Arc which is the constituent arc of the whole sentence and this is the final answer, no more words remain in the buffer, and we don't care how many subscripted pronouns became necessary.

The fundamental non-determinate operation of the parser is either to shift a word off the buffer, find out its lexical insertion arc, and place it on the stack, or to reduce some arcs on top of the stack according to a combination rule selected non-determinately. Since this is to be a right to left parser shifting will be favoured over reducing.

```
| % SHIFT S1, T1
| parse( Stack, [Word|Buffer], N, Answer ) :-
|           an_check( Word, Buffer ),
|           lexicon( Word, Syn, Str, Sem ),
|           ante_check( Syn, Str, Sem, Stack ),
|           parse( [Syn#Str#Sem#[]|Stack], Buffer, N, Answer ).
```

The purpose of 'an_check' is to check whether **a** or **an** has been properly chosen as it must agree with the next word in the buffer.

```
| an_check( a, [Word|_] ) :-
|           \+ an( Word ).
| an_check( an, [Word|_] ) :-
|           an( Word ).
| an_check( Word, _ ) :-
|           Word \= a,
|           Word \= an.
```

The actual lexicon is represented by the 'lexicon' procedure which expands out the abbreviations employed with the ':=' procedure. The words given lexical entries are required to be exactly the possible members of the buffer. A single ':=' clause may correspond to several actual entries in the lexicon. The syntactic structures are explicit in the lexicon but they must be inferred from each clause of the ':=' procedure.

```
| lexicon( Word, Syn, Str, Sem ) :-
|           WordAb := Syn:SemAb,
|           select_form( WordAb, Word, Str ),
|           expand_star( SemAb, Sem ).
```

The 'select_form' procedure is responsible for expanding the horizontal abbreviation of vertical redundancy in the lexicon. Along with each word its root is supplied. In SMG the root of a word is a basic phrase unmodified by any combination rules. The root provides the syntactic structure in the lexical entry for a word.

```
| select_form( Word, Word, Word ) :-
|         atom( Word ).
| select_form( Word pn, Word, he(N) ).
| select_form( Word/_, Word, Word ).
| select_form( Root/Word, Word, Root ).
| select_form( v(Word,_,_,base), Word, Word ).
| select_form( v(Root,Word,_,pres), Word, Root ).
| select_form( v(Root,_,Word,past), Word, Root ).
```

The 'expand_star' procedure expands semantic abbreviations which use the superscript '*' notation.

```
| expand_star( Ab*, Sem ) :-
|         Ab *= Sem.
| expand_star( Sem, Sem ) :-
|         Sem \= _* .
```

Note that because of the poor treatment of negation in Prolog, the goal Sem \= _* above means \+Ab^(Sem = Ab*) declaratively where '^' is Prolog's existential quantifier. There are other similar goals below.

The purpose of 'ante_check' is to find the antecedents of pronouns. For non-pronouns it always succeeds vacuously.

```
| ante_check( _, Str, _, _ ) :-
|         Str \= he(_).
```

This 'parse' clause tries each combination rule in turn attempting to reduce arcs on the top of the stack which match the left-hand side of a combination rule to the arc of the right-hand side of that rule match.

```
| % REDUCE {PHRASE STRUCTURE RULES}
| parse( Stack, Buffer, N, Answer ) :-
|         (Fn >> Rule),
|         get( Rule, Seq, Syn:Sem, Cond ),
|         match( Seq, Stack, Args-[], Stores-[], StackTail ),
|         Cond,
|         appendlist( Stores, Sto ),
|         parse( [Syn#Fn‘Args#Sem#Sto|StackTail], Buffer, N, Answer ).
```

Combination rules may include an optional condition to be tested after a successful matching. This is more efficient than using Prolog's own primitive conditional.

```
| get( (Lhs => Rhs), Lhs, Rhs, true ).
| get( (Lhs => Rhs <- Cond), Lhs, Rhs, Cond ).
```

The 'match' procedure attempts to match its first argument which is a sequence of items to combine, against the beginning of its second argument which is a stack of arcs. Like a unification, a matching is unique if it succeeds. A successful matching returns the entire tail of the stack after the matched arcs. The third and fourth arguments to 'match' are queues (difference lists) for the syntactic structure and the Cooper store respectively so that they can be built up in the natural left to right order. A one-item sequence is handled by the base cases of 'match' and a sequence concatenated to an item is handled by its recursive cases, where the item is sought in the arc on the top of the stack. The first clause is the base case for items which in SMG are syncategorematic and so have a null semantics. Syncategorematic items contribute nothing to the queues.

```
| match( Syn:[], [Syn#_#[]#_|Stack], ArgQ, StoQ, Stack ) :-
|         empty_queue( ArgQ ),
|         empty_queue( StoQ ).
```

The second clause is the base case for items having full semantic templates.

```
| match( Syn:Sem, [Syn#Str#Sem#Sto|Stack], ArgQ, StoQ, Stack ) :-
|         Sem \= [],
|         unit_queue( Str, ArgQ ),
|         unit_queue( Sto, StoQ ).
```

The third clause is the recursive case for syncategorematic items.

```
| match( Seq+Syn:[], [Syn#_#[]#_|Stack], ArgQ, StoQ, StackTail ) :-
|         match( Seq, Stack, ArgQ, StoQ, StackTail ).
```

And the fourth clause is the recursive case for items having full semantic templates.

```
| match( Seq+Syn:Sem, [Syn#Str#Sem#Sto|Stack], ArgQ1, StoQ1, StackTail ) :-
|         Sem \= [],
|         match( Seq, Stack, ArgQ, StoQ, StackTail ),
|         join_queue( Str, ArgQ, ArgQ1 ),
|         join_queue( Sto, StoQ, StoQ1 ).
```

StoQ becomes bound to a queue of the stores of the subconstituents. What is actually required is their set-theoretic union which is computed by 'appendlist' only if the rule condition succeeds. The empty store is generated upon lexical insertion so I have yet to specify how stores become filled.

For perspicuity, separate queue manipulation procedures replace further complication of the 'match' procedure.

```
| empty_queue( List-List ).
|
| unit_queue( Elem, [Elem|List]-List ).
|
| join_queue( Elem, List-[Elem|Last], List-Last ).
```

3.9. Quantification

As far as the lexicon is concerned this only involves the addition of subscripted pronouns to the set of basic terms of SMG. These correspond to free variables in translation. Real pronouns will be substituted for the subscripted ones by the rules of quantification which bind the variables in translation. Since subscripted pronouns are not English words SMG overgenerates.

LEXICON

B_T = {**John**, **Mary**, **Bill**, **Kate**, **he**$_0$, **he**$_1$, **he**$_2$, ...}

T1e. **he**$_n$ $\Rightarrow \lambda P\ (P\ x_n)$

where $x_n \equiv v_{2n,e}$

□

Note that free variables are implicitly universally quantified over the whole sentence.* For example, **he**$_0$ **talks** is true if and only if everyone talks! Therefore **John walks and he talks** cannot be translated as

$$\lambda i\ [(\mathrm{WALK}\ J\ i) \wedge (\mathrm{TALK}\ x_0\ i)]$$

UG, (Cooper 1979), and others to the contrary notwithstanding.

```
| % S1, T1e
| he    pn   := t(masc,subj) : [[X|F]|F].
| him   pn   := t(masc,obj)  : [[X|F]|F].
| she   pn   := t(fem, subj) : [[X|F]|F].
| her   pn   := t(fem, obj)  : [[X|F]|F].
| it    pn   := t(neut,Case) : [[X|F]|F].
|
| anaphorise( X, [[X|F]|F] ).
```

The marking of pronouns with the 'pn' operator is necessary because, for example, **he** and **John** have unifiable lexical assignments but the latter is not anaphoric. Subscripted pronouns lack lexical entries because they cannot occur in the buffer. Instead they get their semantics by means of the 'anaphorise' procedure, which makes the variable X available so that it can be associated with any subscript.

* SMG agrees with PTQ and Prolog.

RULE SCHEMA of quantification

$F_{10,n}(\ \alpha,\ \beta\ ) = \beta 1$ unless α has the form $\mathbf{he}_k$, where $\beta 1$ comes from β by replacing the first occurrence of $\mathbf{he}_n$ or $\mathbf{him}_n$ by α and all other occurrences of $\mathbf{he}_n$ or $\mathbf{him}_n$ by {**he**, **she**, **it**} or {**him**, **her**, **it**} respectively, according as the gender of the first B_{CN} or B_T in α is {masculine, feminine, neuter};

$F_{10,n}(\ \mathbf{he}_k,\ \beta\ ) = \beta 1$ where $\beta 1$ comes from β by replacing all occurrences of $\mathbf{he}_n$ or $\mathbf{him}_n$ by $\mathbf{he}_k$ or $\mathbf{him}_k$ respectively.

S14. If $\alpha \in P_T$ and $\phi \in P_S$ then $F_{10,n}(\alpha,\ \phi) \in P_S$,

T14. if $\alpha \Rightarrow \alpha'$ and $\phi \Rightarrow \phi'$ then $F_{10,n}(\alpha,\ \phi) \Rightarrow (\alpha'\ \lambda x_n\ \phi')$.
□

The first clause of the definition of $F_{10,n}$ is the syntactic analogue of beta-reduction. Applications of it are vacuous when neither $\mathbf{he}_n$ nor $\mathbf{him}_n$ occurs in β. Vacuous quantification can be illustrated by a specification such as

$F_{10,0}($ **a unicorn**, **John walks** $)$ = **John walks**.

The semantics for the sentence **John walks** derived in this manner would give it the reading "There is a unicorn such that John walks" which would be totally wrong. However this pitfall can easily be avoided by founding the syntactic structure on material which actually occurs in the sentence being parsed. The opposite error would arise if a term which did occur were lost thus freeing a variable.

The second clause of the definition of $F_{10,n}$ was included by Montague in order to make the quantification rules total over all terms. It is the syntactic analogue of alpha-conversion. It will be avoided in the implementation by numbering the subscripts consecutively (starting from zero) so that renumbering is unnecessary.

The subscripts chosen for pronouns are *entirely distinct* from the names of the variables in the translation into first-order logic. This is unlike the situation with Montague's translation into intensional logic where they coincide. In the program, the variables which appear during the translation process are meta-variables, strictly speaking. These can be instantiated to any term. When the translation of the whole sentence has been obtained the remaining uninstantiated meta-variables are all instantiated to consecutive variables of the object language (starting from one, to emphasise the difference from pronoun subscripts).

EXAMPLE 21

```
<No woman loves her.

f10| 0
   | f0| no
   |   | woman
   | f4| he(0)
   |   | f5| love
   |   |   | he(0)

>~exists(X1:e,woman(X1,!)&love(X1,X1,!))

 ¬∃x [(WOMAN x !) ∧ (LOVE x x !)]
```

Montague Grammar does not distinguish reflexive and non-reflexive pronouns. In the above example, **no woman loves her(self)** is derived faithfully to Montague as

$$F_{10,0}(\textbf{ no woman, } \textbf{he}_0 \textbf{ loves him}_0 \ ).$$

This is the only possible derivation for the above sentence if vacuous and renumbering rule applications are disallowed.

EXAMPLE 22

```
<A woman doesn't love her.

f10| 0
   | f0| a
   |   | woman
   | f1| he(0)
   |   | doesnt
   |   | f5| love
   |   |   | he(0)

>exists(X1:e,woman(X1,!)&~love(X1,X1,!))

 ∃x [(WOMAN x !) ∧ ¬(LOVE x x !)]
```

This is the sole derivation again. These two examples serve to demonstrate that the rules of quantification have two usages which are inseparable. Firstly, to bind anaphors and secondly, to give a quantified term wide scope. In order to bind an anaphor it is necessary to quantify its antecedent in, but that scopes the quantified term in a way which may not be desirable. Consequently no synonymous reading can be produced for those two examples.

The formulation which I am about to suggest improves upon Montague Grammar by ruling out vacuous quantification, free variables, and renumbered variables. The idea is to use an auxiliary data structure in the syntax which constitutes a mild violation of the principle of compositionality.

The auxiliary data structure will be a *store* rather like the store in (Cooper 1983). However Cooper's store was semantic.

Each arc on the stack has associated with it a syntactic category, syntactic structure, and semantic template as already required. Each arc also includes an associated Cooper store as required for quantification. In this theory the store is a set of bindings between pronouns and arcs of the form

$$syntax : semantics = arc.$$

A pronoun need only be represented by its syntactic subscript and semantic meta-variable. Arcs in store are quadruples as on the stack.

In (Cooper 1983) the store is a set of bindings between pronouns and term denotations. My semantic template in each stored arc corresponds to Cooper's term denotation. Therefore my analysis differs from Cooper's in two essential respects. Stored arcs also have their own stores. I will argue that this is necessary yet Cooper simply flattens the proper nested structure of the store. Also, agreement is a syntactic phenomenon. The syntactic category field of quadruples in the store could contain the full category and features of its origin. However it will be more parsimonious to store only sufficient information to carry out gender agreement. Furthermore the inessential syntactic structure is also stored away and turns up in the right place to add credibility to my approach.

Stores can, of course, be empty which is the initial store state associated with arcs upon lexical insertion. During phrase structural combination the set theoretic union of stores is taken. The option for storage occurs whenever a term has been built up. If it is taken then that term becomes available to be the antecedent of a pronoun. The stored term arc is itself given a pronominal semantics.

```
| % STORE S14; S16; S15
| parse( [Syn#Str#Sem#Sto|Stack], Buffer, N, Answer ) :-
|         Syn = t(Gend,Case),
|         Str \= he(_),
|         anaphorise( X, Seml ),
|         succ( N, M ),
|         parse( [Syn#he(N)#Seml#[N:X=Gend#Str#Sem#Sto]|Stack],
|                 Buffer, M, Answer ).
```

The storage rule is only applicable to terms not having the syntactic structure of subscripted pronouns. That case corresponds to renumbering and permitting it would cause the rule to cycle on its own output. When storage has applied the store is a singleton because any other stored material should not be available at this depth of the analysis tree.

Suppose the parser is at the point of lexically inserting a pronoun off the buffer onto the stack. What is the subscript in the analysis of that pronoun? Under the assumptions that there are to be no unbound pronouns and that cataphora is to be avoided, a strategy for the subscripting of pronouns is immediately apparent.

Since the stack contains the analysis of the words so far and the buffer contains words remaining up to the end of the sentence only the stack is relevant to anaphoric reference. The strategy is to search down the stack looking in all the stores for a possible antecedent of the correct gender. There may be more than one candidate so this is a non-determinate part of the algorithm. If an antecedent cannot be found then the parser should clearly *fail immediately* and backtrack. The pronoun is subscripted with the number with which its antecedent is associated in store.

```
| % S14; S16; S3; S15
| ante_check( t(Gend,_), he(N), Sem, Stack ) :-
|         member( _#_#_#Sto, Stack ),
|         stored( N:X=Gend#_, Sto ),
|         anaphorise( X, Sem ).
```

Stores can be nested so the search through a store is recursive.

```
| stored( Binding, Sto ) :-
|         member( Binding1, Sto ),
|         ( Binding1 = Binding;
|           Binding1 = (_=_#_#_#Sto1),
|           stored( Binding, Sto1 ) ).
```

The option for retrieval occurs whenever a sentence or a predicate (ie. intransitive verb or common noun) has been built up. When it is taken the pronoun in the antecedent position and any subsequent pronouns are bound and the term is given its scope. All stored terms must be retrieved eventually because the goal of parsing specifies a closed sentence with an empty store.

```
| % RETRIEVE S14; S16; S15
| parse( [Syn#Str#Sem#Sto|Stack], Buffer, N, Answer ) :-
|         quantify_in( Syn, X, TermSem, Sem, Seml ),
|         select( M:X=Gend#TermStr#TermSem#TermSto, Sto, RestSto ),
|         append( TermSto, RestSto, Stol ),
|         parse( [Syn#f10'[M,TermStr,Str]#Seml#Stol|Stack],
|                 Buffer, N, Answer ).
```

The calls to 'quantify_in' and 'select' are conceptually in the opposite order. They are implemented the unclear way round so that the first argument to 'quantify_in' can tell immediately whether this rule is applicable. The following call to 'select' then causes the quantification to become fully instantiated. Note that retrieval is the place to make the embedded store available. In this respect it is like a combination rule in which the term and the sentence or predicate are just being combined.

```
| % T14
| quantify_in( s,     X, [[X|G]|F],     G,      F ).
```

Think of this semantics as abstracting X from Sem and then applying TermSem to the result.

This strategy has the advantage that coreference is determined during the parsing of the sentence rather than by a special tree-walk after the syntactic parse has been found. It should be extensible to a more advanced form of weak syntax/semantics interaction in which the match which determines the antecedent during the search of the stores in the arcs down the stack would succeed depending on some degree of inference rather than just on gender agreement.

Let's consider the possibilities which the quantification rules offer for analysing ambiguities of scope. I have chosen an example with a quantified subject and object, a tense operator, and an intensional transitive verb. Each of the following parses corresponds to a different reading. Note that a syntactically similar sentence without these semantic complexities such as **John loves Mary** would have just as many parses but would always have the same simple translation, love(j,m,!).

Whenever a quantified object has wider scope than an intensional transitive verb the translation of the predicate is essentially extensional. To draw attention to this phenomenon I will use a *sort-lifting* function from entities to quantities written as a postfix '*' and similar to Montague's type-lifting abbreviation but remaining within first-order logic. The sort-lifting function is defined by the universal closure of the equivalence

```
qtty( X*, V, I ) <=> ppty( V, X, I ).
```

The parser does not know anything about this function. I have included it purely for simplification purposes.

EXAMPLE 23

```
<Every man worshipped a woman.

 f2| f0| every
   |   | man
   | f5| worship
   |   | f0| a
   |   |   | woman

>exists(I1:s,I1<!&all(X2:e,man(X2,I1)=>
 exists(W3:q,worship(X2,W3,I1)&
 all(V4:p,all(I5:s,qtty(W3,V4,I5)<=>
 exists(X6:e,woman(X6,I5)&ppty(V4,X6,I5)))))))

 ∃j [(j<!) ∧ ∀x [(MAN x j) →
 (WORSHIP λQ λi ∃x [(WOMAN x i) ∧ (Q x i)] x j)]]
```

The above translation is the one obtainable without quantification rules. To see how the scopes come out as they do, here is its trace:

```
[1] every := det : [[X,I|A],[X,I|B],I|all(X:e,A=>B)].
[2] man := cn(masc) : [X,I|man(X,I)].
[3] every man := t(masc,Case) : [[X,I|B],I|all(X:e,man(X,I)=>B)].
[4] worshipped := tv(past) : [[[Y,J|ppty(V,Y,J)],J|A],X,I|
                    exists(W:q,worship(X,W,I)&all(V:p,all(J:s,qtty(W,V,J)<=>A)))].
[5] a := det : [[Z,K|A],[Z,K|B],K|exists(Z:e,A&B)].
[6] woman := cn(fem) : [Z,K|woman(Z,K)].
[7] a woman := t(fem,Case) : [[Z,K|B],K|exists(Z:e,woman(Z,K)&B)].
[8] worshipped a woman := iv(past) : [X,I|
                    exists(W:q,worship(X,W,I)&all(V:p,all(J:s,qtty(W,V,J)<=>
                    exists(Y:e,woman(Y,J)&ppty(V,Y,J)))))].
[9] every man worshipped a woman := s : [K|exists(I:s,I<K&all(X:e,man(X,I)=>
                    exists(W:q,worship(X,W,I)&all(V:p,all(J:s,qtty(W,V,J)<=>
                    exists(Y:e,woman(Y,J)&ppty(V,Y,J))))))))].
```

Rules of functional application apply to produce [3] from [1] and [2], [7] from [5] and [6], and [8] from [4] and [7]. Although **worshipped** is the past of **worship**, the only record of this is the argument of 'iv' until the rule of tense and sign (instead of a simple rule of functional application) applies to produce [9] from [3] and [8].

EXAMPLE 23 (continued)

```
<Every man worshipped a woman.

f10| 0
   | f0| a
   |   | woman
   | f2| f0| every
   |   |   | man
   |   | f5| worship
   |   |   | he(0)

>exists(X1:e,woman(X1,!)&exists(I2:s,I2<!&all(X3:e,man(X3,I2)=>
 exists(W4:q,worship(X3,W4,I2)&
 all(V5:p,all(I6:s,qtty(W4,V5,I6)<=>ppty(V5,X1,I6)))))))

=exists(X1:e,woman(X1,!)&exists(I2:s,I2<!&all(X3:e,man(X3,I2)=>
 worship(X3,X1*,I2))))
```

$\exists x\ [(\text{WOMAN}\ x\ !)\ \wedge\ \exists j\ [(j<!)\ \wedge\ \forall y\ [(\text{MAN}\ y\ j)\ \rightarrow$
$(\text{WORSHIP}\ \lambda P\ (P\ x)\ y\ j)]]]$

The above translation requires the use of Cooper storage. It is difficult to give a trace of this because it has not been produced compositionally. Let's end lines with '...' when there is an associated non-empty store. This is hardly a general notation but it will serve for all traces here where stores only contain the variable Y bound to the translation of **a woman**.

```
[1] every man := t(masc,Case) : [[X,I|B],I|all(X:e,man(X,I)=>B)].
[2] worshipped := tv(past) : [[[Y,J|ppty(V,Y,J)],J|A],X,I|
                        exists(W:q,worship(X,W,I)&all(V:p,all(J:s,qtty(W,V,J)<=>A)))].
[3] a woman := t(fem,Case) : [[Z,K|B],K|exists(Z:e,woman(Z,K)&B)].
[4] her := t(fem,obj) : [[Y|F]|F]...
[5] worshipped her := iv(past) : [X,I|
                        exists(W:q,worship(X,W,I)&all(V:p,all(J:s,qtty(W,V,J)<=>
                        ppty(V,Y,J)))))]...
[6] every man worshipped her := s : [K|exists(I:s,I<K&all(X:e,man(X,I)=>
                        exists(W:q,worship(X,W,I)&all(V:p,all(J:s,qtty(W,V,J)<=>
                        ppty(V,Y,J)))))))]...
[7] every man worshipped a woman := s : [K|exists(Z:e,woman(Z,K)&
                        exists(I:s,I<K&all(X:e,man(X,I)=>
                        exists(W:q,worship(X,W,I)&all(V:p,all(J:s,qtty(W,V,J)<=>
                        ppty(V,Z,J))))))))].
```

Lines [1], [2], and [3] are also found in the previous trace and need no elaboration. Storage produces [4] from [3], functional application produces [5] from [2] and [4], tense and sign produces [6] from [1] and [5], and finally retrieval produces [7] from [6]. Thus **a woman** is raised to wide sentential scope. Similar analyses can raise **every man** to wide scope, or both **every man** and **a woman** in either order.

EXAMPLE 23 (continued)

```
<Every man worshipped a woman.

f10| 0
   | f0| every
   |   | man
   | f2| he(0)
   |   | f5| worship
   |   |   | f0| a
   |   |   |   | woman

>all(X1:e,man(X1,!)=>exists(I2:s,I2<!&
 exists(W3:q,worship(X1,W3,I2)&
 all(V4:p,all(I5:s,qtty(W3,V4,I5)<=>
 exists(X6:e,woman(X6,I5)&ppty(V4,X6,I5)))))));

 ∀x [(MAN x !) → ∃j [(j<!) ∧
 (WORSHIP λQ λi ∃x [(WOMAN x i) ∧ (Q x i)] x j)]]

f10| 1
   | f0| a
   |   | woman
   |f10| 0
   |   | f0| every
   |   |   | man
   |   | f2| he(0)
   |   |   | f5| worship
   |   |   |   | he(1)

>exists(X1:e,woman(X1,!)&all(X2:e,man(X2,!)=>exists(I3:s,I3<!&
 exists(W4:q,worship(X2,W4,I3)&
 all(V5:p,all(I6:s,qtty(W4,V5,I6)<=>ppty(V5,X1,I6)))))))

=exists(X1:e,woman(X1,!)&all(X2:e,man(X2,!)=>exists(I3:s,I3<!&
 worship(X2,X1*,I3))));

 ∃x [(WOMAN x !) ∧ ∀y [(MAN y !) → ∃j [(j<!) ∧
 (WORSHIP λP (P x) y j)]]]
```

```
f10| 0
   | f0| every
   |   | man
   |f10| 1
   |   | f0| a
   |   |   | woman
   |   | f2| he(0)
   |   |   | f5| worship
   |   |   |   | he(1)
```

```
>all(X1:e,man(X1,!)=>exists(X2:e,woman(X2,!)&exists(I3:s,I3<!&
 exists(W4:q,worship(X1,W4,I3)&
 all(V5:p,all(I6:s,qtty(W4,V5,I6)<=>ppty(V5,X2,I6)))))))

=all(X1:e,man(X1,!)=>exists(X2:e,woman(X2,!)&exists(I3:s,I3<!&
 worship(X1,X2*,I3))))

 ∀x [(MAN x !) → ∃y [(WOMAN y !) ∧ ∃j [(j<!) ∧
 (WORSHIP λP (P y) x j)]]]
```

All anaphora is treated as bound anaphora in Montague Grammar. Therefore some sentences which we would expect to have no scopal ambiguity must be analysed using the same rules of quantification.

EXAMPLE 24

```
<John walks and he talks.

f10| 0
   | john
   | f8| f4| he(0)
   |   |   | walk
   |   | f4| he(0)
   |   |   | talk

>walk(j,!)&talk(j,!)

 [(WALK J !) ∧ (TALK J !)]
```

So far all retrieval considered has been done at sentences ~~sentential~~. There are also rules to retrieve at predicates. The following schematic rule quantifies a term over an intransitive verb.

RULE SCHEMA of quantification

S16. If $\alpha \in P_T$ and $\beta \in P_{IV}$ then $F_{10,n}(\alpha, \beta) \in P_{IV}$.

T16. if $\alpha \Rightarrow \alpha'$ and $\beta \Rightarrow \beta'$ then $F_{10,n}(\alpha, \beta) \Rightarrow \lambda y\ (\alpha'\ \lambda x_n\ (\beta'\ y))$.
□

```
| % T16
| quantify_in( iv(_), X, [[X|G]|F], [Y|G], [Y|F] ).
```

Two more readings of the big example considered above are now provided for. This makes seven readings in all.

```
EXAMPLE 23 (continued)

<Every man worshipped a woman.

 f2| f0| every
   |   | man
   |f10| 0
   |   | f0| a
   |   |   | woman
   |   | f5| worship
   |   |   | he(0)

>exists(I1:s,I1<!&all(X2:e,man(X2,I1)=>exists(X3:e,woman(X3,I1)&
 exists(W4:q,worship(X2,W4,I1)&
 all(V5:p,all(I6:s,qtty(W4,V5,I6)<=>ppty(V5,X3,I6)))))))

=exists(I1:s,I1<!&all(X2:e,man(X2,I1)=>exists(X3:e,woman(X3,I1)&
 worship(X2,X3*,I1))))

 ∃j [(j<!) ∧ ∀x [(MAN x j) → ∃y [(WOMAN y j) ∧
 (WORSHIP λP (P y) x j)]]]
```

The above translation requires the use of retrieval at intransitive verbs. The following trace illustrates how this works:

```
[1] every man := t(masc,Case) : [[X,I|B],I|all(X:e,man(X,I)=>B)].
[2] worshipped := tv(past) : [[[Y,J|ppty(V,Y,J)],J|A],X,I|
                    exists(W:q,worship(X,W,I)&all(V:p,all(J:s,qtty(W,V,J)<=>A)))].
[3] a woman := t(fem,Case) : [[Z,K|B],K|exists(Z:e,woman(Z,K)&B)].
[4] her := t(fem,obj) : [[Y|F]|F]...
[5] worshipped her := iv(past) : [X,I|
                    exists(W:q,worship(X,W,I)&all(V:p,all(J:s,qtty(W,V,J)<=>
                    ppty(V,Y,J))))]...
[6] worshipped a woman := iv(past) : [X,I|exists(Z:e,woman(Z,I)&
                    exists(W:q,worship(X,W,I)&all(V:p,all(J:s,qtty(W,V,J)<=>
                    ppty(V,Z,J)))))].
[7] every man worshipped a woman := s : [K|exists(I:s,I<K&all(X:e,man(X,I)=>
                    exists(Z:e,woman(Z,I)&
                    exists(W:q,worship(X,W,I)&all(V:p,all(J:s,qtty(W,V,J)<=>
                    ppty(V,Z,J))))))))].
```

Lines [1], [2], and [3] are as before. Storage produces [4] from [3], functional application produces [5] from [2] and [4], retrieval produces [6] from [5], and finally tense and sign produces [7] from [1] and [6]. Thus **a woman** is raised over the predicate but not over the sentence. The final analysis is similar and has **every man** raised to wide scope.

```
EXAMPLE 23 (continued)

<Every man worshipped a woman.

f10| 0
   | f0| every
   |   | man
   | f2| he(0)
   |    |f10| 1
   |    |   | f0| a
   |    |   |   | woman
   |    |   | f5| worship
   |    |   |   | he(1)

>all(X1:e,man(X1,!)=>exists(I2:s,I2<!&exists(X3:e,woman(X3,I2)&
 exists(W4:q,worship(X1,W4,I2)&
 all(V5:p,all(I6:s,qtty(W4,V5,I6)<=>ppty(V5,X3,I6)))))))

=all(X1:e,man(X1,!)=>exists(I2:s,I2<!&exists(X3:e,woman(X3,I2)&
 worship(X1,X3*,I2))))

 ∀x [(MAN x !) → ∃j [(j<!) ∧ ∃y [(WOMAN y j) ∧
 (WORSHIP λP (P y) x j)]]]
```

There are also some sentences for which the preferred reading requires quantification over an intransitive verb.

```
EXAMPLE 25

<John tries to catch a fish and eat it.

 f4| john
   | f7| try
   |    |f10| 0
   |    |   | f0| a
   |    |   |   | fish
   |    |   | f8| f5| catch
   |    |   |   |   | he(0)
   |    |   |   | f5| eat
   |    |   |   |   | he(0)

>exists(V1:p,try(j,V1,!)&
 all(X2:e,all(I3:s,ppty(V1,X2,I3)<=>
 exists(X4:e,fish(X4,I3)&catch(X2,X4,I3)&eat(X2,X4,I3)))))

 (TRY λy λi ∃x [(FISH x i) ∧ [(CATCH x y i) ∧ (EAT x y i)]] J !)
```

3.10. Relative Clauses

Montague only analysed what are called such-that relative clauses. His analysis was peculiar in many respects. For rather little benefit it introduced much complexity into the grammar particularly in interaction with other rules. Therefore this section has a length beyond its importance. However some of the matters dealt with below could recur in a syntactically adequate treatment of relative clauses.

A such-that relative clause is a restrictive postmodifier of common nouns introduced by the words **such that** and abstracting from any sentence a predicate to intersect the common noun. This kind of relative clause is popular as logical jargon but is hardly ordinary English.

RULE SCHEMA of relativisation

$F_{3,n}(\alpha, \beta) = \alpha$ **such that** $\beta1$ where $\beta1$ comes from β by replacing each occurrence of **he**$_n$ or **him**$_n$ by {**he**, **she**, **it**} or {**him**, **her**, **it**} respectively, according as the gender of the first B_{CN} in α is {masculine, feminine, neuter}.

S3. If $\alpha \in P_{CN}$ and $\phi \in P_S$ then $F_{3,n}(\alpha, \phi) \in P_{CN}$.

T3. if $\alpha \Rightarrow \alpha'$ and $\phi \Rightarrow \phi'$ then $F_{3,n}(\alpha, \phi) \Rightarrow \lambda x_n\ \lambda i\ [(\alpha'\ x_n\ i) \wedge (\phi'\ i)]$.
□

These are odd rules in that they allow the introduction of pronouns which do not have real terms as their antecedents. This will be a source of complexity in the implementation.

EXAMPLE 26

```
<Every man such that he runs walks.

 f4| f0| every
   |   | f3| 0
   |   |   | man
   |   |   | f4| he(0)
   |   |   |   | run
   | walk

>all(X1:e,man(X1,!)&run(X1,!)=>walk(X1,!))
```

$\forall x\ [[(MAN\ x\ !) \wedge (RUN\ x\ !)] \rightarrow (WALK\ x\ !)]$

In ordinary English this would be **every man who runs walks**. The words **such** and **that** were syncategorematically introduced by $F_{3,0}$. They are implemented as particles, which have null semantic templates. This **that** is the same **that** as was given for the rule F_6.

```
| such        := part(such)   : [].
| that        := part(that)   : [].
```

Because the implementation of anaphoric pronouns assumes that they have terms as antecedents and no such term can be identified for the pronoun bound in a such-that relative clause, a kludge is necessary in the implementation of this strange construction. The relative clause rules have been implemented as if their grammar were as follows.

QUASI-RULES of relativisation

$F_{31}(\ \alpha\ ) = \alpha$ **such that**

$F_{32,n}(\ \alpha,\ \beta\ ) = \alpha\ \beta 1$ where $\beta 1$ comes from β as in $F_{3,n}$.

S31. If $\alpha \in P_{CN}$ then $F_{31}(\alpha) \in P_{ST}$.

T31. if $\alpha \Rightarrow \alpha'$ then $F_{31}(\alpha) \Rightarrow \alpha'$.

S32. If $\alpha \in P_{ST}$ and $\phi \in P_S$ then $F_{32,n}(\alpha,\ \phi) \in P_{CN}$.

T32. if $\alpha \Rightarrow \alpha'$ and $\phi \Rightarrow \phi'$ then $F_{32,n}(\alpha,\ \phi) \Rightarrow \lambda x_n\ \lambda i\ [(\alpha'\ x_n\ i) \wedge (\phi'\ i)]$.
□

Naturally it is the proper rules which are manifest in the syntactic structure fields rather than the quasi-rules which are just a fiction to help in understanding the program.

The intermediate category ST of such-that antecedents is introduced in order to store a virtual term for pronouns in the relativised sentence to refer back to. Of course this is not a real term (it is not even a real constituent) so its structure, semantics, and store are ill-defined and are just given as 'dummy'. The following 'parse' clause combines a common-noun phrase and **such that** to give a such-that phrase.

```
| % STORE S31
| parse( [part(that)#_,part(such)#_,cn(Gend)#Str#Sem#Sto|Stack],
|        Buffer, N, Answer ) :-
|          succ( N, M ),
|          parse( [st(Gend)#Str#Sem#[N:X=Gend#dummy|Sto]|Stack],
|                 Buffer, M, Answer ).
```

Once a relativised sentence has been parsed, it can be concatenated onto the such-that phrase to again give a common-noun phrase, losing 'dummy' in the process.

```
| % RETRIEVE S32
| parse( [s#Str2#Sem2#Sto2,st(Gend)#Str1#Sem1#[M:X=_|Sto1]|Stack],
|        Buffer, N, Answer ) :-
|          relativise( X, Sem1, Sem2, Sem ),
|          append( Sto1, Sto2, Sto ),
|          parse( [cn(Gend)#f3'[M,Str1,Str2]#Sem#Sto|Stack],
|                 Buffer, N, Answer ).
```

Think of the semantics expressed by 'relativise' as abstracting X from Sem2 and then intersecting Sem1 with the result.

```
| % T3
| relativise( X, [X,I|A], [I|B], [X,I|A&B] ).
```

Such-that relative clauses allow relativisation of any term position within a sentence. But they leave a personal pronoun in place instead of preposing a relative pronoun.

EXAMPLE 27

```
<A park such that John runs in it lives.

 f4| f0| a
   |   | f3| 0
   |   |   | park
   |   |   | f4| john
   |   |   |   | f0| run
   |   |   |   |   | f5| in
   |   |   |   |   |   | he(0)
   | live

>exists(X1:e,(park(X1,!)&exists(V2:p,in(j,V2,X1,!)&
 all(X3:e,all(I4:s,ppty(V2,X3,I4)<=>run(X3,I4)))))&
 live(X1,!))

 ∃x [[(PARK x !) ∧ (IN x RUN J !)] ∧ (LIVE x !)]
```

In ordinary English this would be **a park which John runs in lives.** The bound pronoun may occur multiply or not at all. In the latter case vacuous abstraction results but it is not a semantic error as was vacuous quantification with $F_{10,n}$ because no information is now being lost.

Relative clauses allow terms to occur within terms and they are unique for this in SMG. If a pronoun is free in a such-that common noun (ie. it is not the relativised pronoun) then it is available to have a previous term as its antecedent. There is then the opportunity for an unfortunate loophole in the strategy for avoiding free variables.

EXAMPLE 28

```
<A man finds a pen such that he walks.

f10| 2
   | f0| a
   |   | f3| 1
   |   |   | pen
   |   |   | f4| he(0)
   |   |   |   | walk
   |f10| 0
   |   | f0| a
   |   |   | man
   |   | f4| he(0)
   |   |   | f5| find
   |   |   |   | he(2)

*exists(X1:e,(pen(X1,!)&walk(X2,!))&exists(X2:e,man(X2,!)&find(X2,X1,!)))
```

$\exists x\ [[(\text{PEN}\ x\ !) \wedge (\text{WALK}\ x_0\ !)] \wedge \exists y\ [(\text{MAN}\ y\ !) \wedge (\text{FIND}\ x\ y\ !)]]$

The variable X2 is free here although there are also bound occurrences of X2. What went wrong is that **he**$_0$ found **a man** as its antecedent but then **a pen such that he**$_0$ **walks** was stored so that when **a man** was quantified in it missed binding **he**$_0$.

Note that the other three parses of the above example are good and that even the bad parse is in SMG. Yet SMG has a semantic error here because free variables are interpreted universally. The implementation has avoided free variables till now and so we would like to close this loophole.

The same problem was noticed by (Friedman and Warren 1978). To avoid it, when deciding to retrieve a term, they always checked that no other stored term contained a free occurrence of the pronoun to which the chosen term was bound. I have described the equivalent of their solution for my methodology. There is a simpler solution. To avoid unbound pronouns, check that each syntactic structure is closed as it is output by the parser. The procedure implementing this closure check is presented in appendix 1.4.

It is a pity that the closure check is necessary at all because it rarely fails. If stores were not nested then the check would become essential to block vacuous quantification as it did in Friedman and Warren's program. However in my program it can be regarded as a luxury which one may prefer to forego at the cost of the very occasional free pronoun within a relative clause.

* Not produced by program with closure check.

As if such-that relative clauses didn't have enough problems of their own, Montague also provided a schematic rule of quantification over common nouns similar to that over intransitive verbs which is only non-vacuous for such-that common nouns.

RULE SCHEMA of quantification

S15. If $\alpha \in P_T$ and $\beta \in P_{CN}$ then $F_{10,n}(\alpha, \beta) \in P_{CN}$.

T15. if $\alpha \Rightarrow \alpha'$ and $\beta \Rightarrow \beta'$ then $F_{10,n}(\alpha, \beta) \Rightarrow \lambda y\ (\alpha'\ \lambda x_n\ (\beta'\ y))$.
□

```
| % T15
| quantify_in( cn(_), X, [[X|G]|F], [Y|G], [Y|F] ).
```

No examples demand this rule and it is hard* to find examples having readings which can only be produced by it. The following is such an example for the given reading.

EXAMPLE 29

```
<An alleged man such that he loves every woman lives.

  f4| f0| a
    |   |f10| 1
    |   |   | f0| every
    |   |   |   | woman
    |   |   | f0| alleged
    |   |   |   | f3| 0
    |   |   |   |   | man
    |   |   |   |   | f4| he(0)
    |   |   |   |   |   | f5| love
    |   |   |   |   |   |   | he(1)
    | live

>exists(X1:e,all(X2:e,woman(X2,!)=>exists(V3:p,alleged(X1,V3,!)&
 all(X4:e,all(I5:s,ppty(V3,X4,I5)<=>man(X4,I5)&love(X4,X2,I5)))))&
 live(X1,!))
```

$\exists x\ [\forall y\ [(\text{WOMAN}\ y\ !) \rightarrow (\text{ALLEGED}\ \lambda x_0\ \lambda i\ [(\text{MAN}\ x_0\ i) \wedge (\text{LOVE}\ y\ x_0\ i)]\ x\ !)] \wedge (\text{LIVE}\ x\ !)]$

In order to give **every woman** narrower scope than **an** but still remove it from the intensional context of **alleged** whilst **loves** remains within that context, it is necessary to use the CN quantification rule.

Let's see how far Montague Grammar gets with the problematic "donkey sentences" as they are called. The prime example is **everyone who owns a donkey beats it** and the best that SMG

* It is even harder in the PTQ fragment which has no adjectives. An example due (almost) to Partee is **every man such that he lost a pen voluntarily such that he doesn't find it will walk slowly** for the reading where **a pen** binds **it** but has narrower scope than **every**. Partee was wrong in giving this example without the **voluntarily**.

can do follows. Only translations in which the indefinite determiner corresponds to a wide scope existential quantifier can be found for such examples.

EXAMPLE 30

```
<Every man such that he catches a fish eats it.

f10| 1
   | f0| a
   |   | fish
   | f4| f0| every
   |   |   | f3| 0
   |   |   |   | man
   |   |   |   | f4| he(0)
   |   |   |   |   | f5| catch
   |   |   |   |   |   | he(1)
   |   | f5| eat
   |   |   | he(1)

>exists(X1:e,fish(X1,!)&all(X2:e,man(X2,!)&catch(X2,X1,!)=>eat(X2,X1,!)))
```

$\exists x\ [(\mathrm{FISH}\ x\ !) \wedge \forall y\ [[(\mathrm{MAN}\ y\ !) \wedge (\mathrm{CATCH}\ x\ y\ !)] \rightarrow (\mathrm{EAT}\ x\ y\ !)]]$

Admittedly this is a possible reading but it is not the preferred one. In the preferred reading the existential quantification has narrow scope.

```
*all(X1:e,man(X1,!)&exists(X2:e,fish(X2,!)&catch(X1,X2,!))=>eat(X1,X2,!))
```

However in this translation attempt an occurrence of X2 is left unbound.

Because the existential quantifier is in the antecedent of an implication it is equivalent to a wide scope universal quantifier.

```
*all(X1:e,all(X2:e,man(X1,!)&fish(X2,!)&catch(X1,X2,!)=>eat(X1,X2,!)))
```

Although this is the best translation it is hard to obtain compositionally because the indefinite determiner corresponds to a different quantifier from that assigned lexically and preserved by the usual rules of quantifying-in.

Donkey sentences are more of a problem for their translation into pure logic than for their implementation in Prolog. This is because the above translation with the free variable is usable. To be precise, the output of the parser is a structure containing meta-variables but quantifiers can only bind variables. Substitutions for meta-variables will ignore quantifier boundaries. So if the quantifiers are removed by substituting free variables and Skolem terms then the resulting translation

* These translations cannot be produced by the program.

will work.

The problem is more acute in the case of disjunctive donkey sentences such as **everyone who owns Pedro or Eeyore beats it**. Here the coordination is essentially a wide scope conjunction but Montague's predicate spreading analysis of term coordination does not introduce any variable which the pronoun could translate as. If my quantificational analysis is adopted then the above technique can be applied here too.

3.11. Summary and Outlook

All the syntactic constructions of Montague Grammar have now been covered. The translations output by the program have been designed to be in a first-order language while still being equivalent to the higher-order translations provided by the specification. The aim is to use a first-order theorem prover on the output. Axioms of comprehension and extensionality are necessary and sufficient to prove this equivalence, because for that purpose the first-order language lacks only the lambda-terms of the omega-order language. Comprehension axioms assure the existence of terms equal to the missing lambda-terms and extensionality axioms assure their uniqueness. Such an axiomatisation is incomplete because other irrelevant set-theoretic axioms (choice, infinity, etc.) are missing. In the next chapter, such proof-theoretic considerations will be studied in detail.

4. Theory of Montague Semantics (TMS)

At first, one might expect to implement SMG using a theorem prover for Ty2 similar to the theorem-proving system described in (Andrews &al. 1984). However these experts warn that they are "a long way from having a reasonable theorem-proving procedure for higher-order logic". Furthermore they have set their sights on covering most of mathematics, for which Ty2 is adequate. For example, they show how Cantor's Theorem* that a set has more subsets than members can be proved completely automatically by their system but this is near the limit of its ability. I believe it to be badly inefficient to use a mathematical theorem prover for a natural language fragment which admits extra constraints.

The hard fact remains that theorem proving for SMG is going to be quite difficult so I feel justified in using any arrangement that can be found to make a system to demonstrate that it can be practical. In earlier chapters it has been shown how to translate sentences of SMG into sentences of the language of TMS. Now that method offers numerous constraints which are very valuable: in addition to being *first-order*, it uses a predicate logic which is entirely *relational* (it lacks functional terms), the relations are either *two*, *three*, or *four-place*, bound variables are all *sorted*, and there are *no free variables* in the translations.

The resources of sentential and quantificational logic alone do not comprise a complete proof theory for TMS. It still remains to provide axioms for the membership relations (true, ppty, and qtty), the equality and temporal anteriority relations, and the sorted constants in order to capture proof-theoretic capabilities which were implicit in translations into Ty2. In the case of the membership relations this amounts to providing a modicum of set theory in the shape of suitable extensionality and comprehension axioms. Comprehension is by far the greater challenge.

It is beyond the scope of this thesis to present a specific theorem prover for first-order logic or to investigate control issues concerning searches for proofs in TMS. However a standard reduction will be presented from five-sorted first-order predicate logic to an unsorted Kowalski-form language which can be processed by most resolution theorem provers†. In that sense, it is fair to say that a

* $\neg\exists R\ \forall P\ \exists x\ ((R\ x)=P)$ proved by substituting $\lambda y\ \neg(R\ y\ y)$ for P, where $P \equiv v_{0,<e,t>}$ and $R \equiv v_{0,<e,<e,t>>}$.

complete theorem-proving system will be precisely specified.

4.1. Extensionality

As the intention is to follow Montague's analysis of propositions, properties, and quantities, they have been installed as sets. Therefore providing an appropriate proof theory for them entails providing at least extensionality and comprehension axioms. Extensionality states that two of these objects are equal if they consist of the same members.

Extensionality Axioms

```
all(U1:o,all(U2:o,all(I:s,true(U1,I)<=>true(U2,I)) => U1=U2))
all(V1:p,all(V2:p,all(X:e,all(I:s,ppty(V1,X,I)<=>ppty(V2,X,I))) => V1=V2))
all(W1:q,all(W2:q,all(V:p,all(I:s,qtty(W1,V,I)<=>qtty(W2,V,I))) => W1=W2))
```

The intensionality obtained, for example, by using propositions instead of truth values as the objects of **believe** has been called "weak" intensionality by (Chierchia 1984) because although **believe** is not extensional its propositional objects are extensional sets of indices. This is the correct interpretation for Montague Grammar and we shall see that the extensionality of sets cannot be withheld in TMS.

4.1.1. The Role of Extensionality

Consider the following translations:

```
<John believes that Kate thinks.
>exists(U1:o,believe(j,U1,!)&all(I:s,true(U1,I)<=>think(k,I)))
<Mary believes that Kate thinks.
>exists(U2:o,believe(m,U2,!)&all(I:s,true(U2,I)<=>think(k,I)))
```

Extensionality is necessary to ensure a unique proposition **that Kate thinks** so that John and Mary believe the same thing.

† Such a language cannot, of course, be directly interpreted by Prolog because Prolog is correct only for Horn clauses and is incomplete even for them.

4.2. Comprehension Introduced

It is more difficult to provide sufficient existential principles for propositions, properties, and quantities than anything else in TMS. In the first-order language used so far it is most natural to postulate the following schematic axioms:

First-Order Comprehension Axioms

```
exists(U:o,all(I:s,true(U,I)<=>φ))                 ; no free U in φ
exists(V:p,all(X:e,all(I:s,ppty(V,X,I)<=>φ)))      ; no free V in φ
exists(W:q,all(V:p,all(I:s,qtty(W,V,I)<=>φ)))      ; no free W in φ
```

The semantics for TMS in Chapter 1 is non-committal about the sizes of D_o, D_p, and D_q but let's suppose that in the intended model they are big enough for these schemas. They must contain *sets*, however, and just as there are rival versions of set theory so there are other comprehension axioms for TMS which should be considered. The preferred axioms given above are analogous to Zermelo-Fraenkel set theory in its usual elementary formulation.

The inclusion of axiom schemas in a theory is an unwelcome source of complexity for its theorem prover. The present formulation has the virtue of exhibiting where the trouble arises and confining it there.

4.2.1. The Role of Comprehension

Consider the following translation:

```
<John doesn't believe that Kate thinks.
>~exists(U:o,believe(j,U,!)&all(I:s,true(U,I)<=>think(k,I)))
=all(U:o,all(I:s,true(U,I)<=>think(k,I))=>~believe(j,U,!))
```

Comprehension is necessary to ensure the existence of a proposition **that Kate thinks** so that John could have believed it and the sentence cannot be vacuously true.

Observe that there is a dual approach to the one adopted here, in which the translations are

```
<John believes that Kate thinks.
>all(U:o,all(I:s,true(U,I)<=>think(k,I))=>believe(j,U,!))
<John doesn't believe that Kate thinks.
>~all(U:o,all(I:s,true(U,I)<=>think(k,I))=>believe(j,U,!))
=exists(U:o,~believe(j,U,!)&all(I:s,true(U,I)<=>think(k,I)))
```

In the dual approach the polarities in which extensionality and comprehension play their roles are reversed. I have adopted the earlier approach because conjunction is commutative and I prefer to write the conjuncts the other way around, with **John believes** before **that Kate thinks.**

4.3. Comprehension Unconstrained

It is clear that the first-order comprehension axioms supply enough propositions, properties, and quantities to fulfill the roles they actually play in TMS. However it has often been observed that whereas there are only $\aleph_0$ propositions definable by an open sentence ϕ, there are $2^{\aleph_0}$ propositions if there are $\aleph_0$ indices (and there are at least $\aleph_0$ indices). Following the idea of (Henkin 1950), the TMS semantics only needs to ask for enough propositions in D_o to satisfy all instances of the first-order comprehension schema but let's briefly aim for $D_o = \mathbf{P}(D_s)$. Using second-order predicate logic, let [S1,...,Sn] be the type of predicates of n-tuples of individuals having the sorts S1 through Sn. These three axioms then suffice:

Second-Order Comprehension Axioms

```
all(P:[s],exists(U:o,all(I:s,true(U,I)<=>P(I))))
all(P:[e,s],exists(V:p,all(X:e,all(I:s,ppty(V,X,I)<=>P(X,I)))))
all(P:[p,s],exists(W:q,all(V:p,all(I:s,qtty(W,V,I)<=>P(V,I)))))
```

In second-order logic a finite axiomatisation is therefore achievable. However these axioms only "work" in so far as second-order logic itself can be formalised. Given the standard semantics for type-theoretic languages they say all that one would wish but the problem of what sets there are is merely thrown back to the problem of what predicates P there are. To appreciate this, note that the usual proof theory would provide the following comprehension schema for predicates:

Arbitrary-Order Comprehension Axioms

```
exists(P:[S1,...,Sn],all(T1:S1,...all(Tn:Sn,P(T1,...,Tn)<=>ϕ)...))
                                               ; no free P in ϕ
```

Clearly from a theorem-proving point of view little progress has been made since the system must still process an infinity of axioms. From an ontological point of view the position is much worse.

In the recommended first-order comprehension schemas the meta-variable ϕ obviously ranges over first-order sentences but in the above schema it is far from obvious what should be allowable in ϕ. Allowing only first-order sentences repeats the proof-theoretic capabilities of the purely first-order system but seems unnatural now that a language is being used in which quantification over predicates is provided. Allowing all second-order sentences yields a more powerful system analogous to Morse-Kelley set theory. Alternatively the arbitrary-order comprehension schema could be considered to have instances in a third-order, omega-order, or even transfinite-order language, yielding extra sets for each increase in order. There is nowhere to stop; this is a consequence of the incompleteness of second-order logic. Furthermore there are other second-order principles, such as the axiom of choice, which are independent of comprehension and which it is unclear whether to take or leave. That these considerations have no relevance to linguistics suggests that second-order logic is too expressive for TMS.

4.4. Metatypes

It is not without precedent in computer science to want to represent an uncountable number of objects: real numbers are approximated by floating-point numbers. So although there are too many second-order predicates, I will proceed by assuming that there are enough *abstract predicates* for the first-order comprehension schemas because these suffice in practice. Let's defer the internal arrangement of abstract predicates and first consider their interface with the rest of TMS. It is obviously necessary to extend the language output by the parser to accommodate them. Note that in the language from the parser all variables are bound and sorted. Note also that in the second-order comprehension axioms the predicate variables are outermostly universally quantified and unsorted*.

* Notwithstanding that they have second-order types, such types are necessarily not sorts. To allow otherwise would make predicates into individuals and hence admit Russell's paradox.

Therefore let free variables in the extended language be implicitly universally quantified and unsorted so that they can have abstract predicates within their range. Now it is possible to cover comprehension without clashing with anything in the language output by the parser.

Furthermore it is very convenient to make relations in the output from the parser more accessible in the extended language. Relations can easily be construed as unsorted objects of another kind. It must be understood immediately that this move is not essential in principle. It is a purely syntactic convenience for reducing a finite number of cases (the semantic vocabulary of the fragment) to only three cases (arity still being kept distinct). All relation variables are free variables because bound variables are always sorted individual variables in TMS. Unlike abstract predicates, relations have no comprehension principles to satisfy and no terms outside of the vocabulary. Thus the semantic importance of construing relations as objects is minimised.

Every relation in the output from the parser is to be retired into the first argument position of the '$' predicate of the appropriate arity.

EXAMPLE 31

```
<Mary found the park and walked in it.

f10| 0
   | f0| the
   |   | park
   | f2| mary
   |   | f8| f5| find
   |   |   |   | he(0)
   |   |   | f0| walk
   |   |   |   | f5| in
   |   |   |   |   | he(0)

>exists(X1:e,all(X2:e,park(X2,!)<=>X1=X2)&
 exists(I2:s,I2<!&find(m,X1,I2)&exists(V3:p,in(m,V3,X1,I2)&
 all(X4:e,all(I5:s,ppty(V3,X4,I5)<=>walk(X4,I5))))))

=exists(X1:e,all(X2:e,$(park,X2,!)<=>$(=,X1,X2))&
 exists(I2:s,$(<,I2,!)&$(find,m,X1,I2)&exists(V3:p,$(in,m,V3,X1,I2)&
 all(X4:e,all(I5:s,$(ppty,V3,X4,I5)<=>$(walk,X4,I5))))))
```

This new notation will be the usual one for the rest of the chapter.

Having both relations and abstract predicates as free and unsorted variables raises the question of how they are to be distinguished. The answer comes in a broader context by being precise about the full extended language suitable for automatic theorem proving in TMS. The usual metalanguage in which pure first-order logic is described only distinguishes between sentences ϕ and terms α but TMS is an applied first-order logic in which terms can be further subclassified in the metalanguage. Altogether nine *metatypes* will be used, comprising: truth valued sentences *sent*; sorts of individuals *sort*; individuals *ind* which are sorted objects; 2-place, 3-place, and 4-place relations between individuals *rel2*, *rel3*, and *rel4*; abstract predicates of lists of individuals *abs*; locations of individuals in lists *loc*; and lists of individuals *indlist* which represent environments. All of these have been met and motivated before except for *loc* and *indlist* which will play a supporting role internal to the arrangement of abstract predicates.

Metatypes can be thought of as intermediate symbols in the phrase structural synthesis of an object language. With their aid the well-formed formulas (sentences and terms) of TMS will be unambiguously specified.

Sentences

```
0, 1, ~ sent, (sent & sent), (sent v sent), (sent => sent), (sent <=> sent),
all(υ:sort,sent), exists(υ:sort,sent), $(rel2,ind,ind), $(rel3,ind,ind,ind),
$(rel4,ind,ind,ind,ind), (abs .. indlist), get(loc,indlist,ind), (sort .: ind).
```

Because the n-place relations *reln* which were previously taken as primitive predicates have now been retired from that role, there is plenty of room for new primitive predicates to be (unconfusingly) introduced to help the theorem prover. Only six will be required altogether. 2-place, 3-place, and 4-place relational predications have already been discussed. Abstract predication (..) and environmental selection (get) help to arrange for axiom schemas. With respect to a list of individuals, '..' associates abstract predicates with sentences and 'get' associates locations with individuals. Finally sortal predication (.:) is not required unless quantifiers are to be eliminated as in the reduction to Kowalski form. Using '.:' free variables and Skolem terms can be assigned sorts.

Individuals are sorted into entities, propositions, properties, quantities, and indices.

Sorts

```
e, o, p, q, s.
```

In the language with sorting quantifiers the only individuals required by the SMG fragment are Here and Now (the local index), Bill, John, Kate, and Mary. In the Kowalski-form language new individuals are introduced by Skolem functions and those associated with the TMS axioms will be characterised and given standard names in due course.

Individuals

```
!, b, j, k, m, ...
```

Binary relations follow the fragment. Note the unexceptional syntax of '=', '<', and 'true'.

2-Place Relations

```
=, <, child, fish, horse, live, man, mermaid, park, pen,
red, run, talk, think, true, unicorn, walk, woman.
```

Ternary relations follow the fragment. Note the unexceptional syntax of 'ppty' and 'qtty'.

3-Place Relations

```
alleged, assert, believe, catch, conceive, date, deny,
eat, find, know, lose, love, ppty, prove, qtty, rapid,
slow, tall, try, voluntary, wish, worship.
```

Quaternary relations follow the fragment.

4-Place Relations

```
about, in.
```

It appears that there are also 1-place and 5-place relations in full English. For example, **it rains** could translate to $(rain,!) and **John bets Mary a pound that it will rain** could roughly translate to $(bet,j,m,w,u,!) where 'w' is a pound and 'u' is the proposition that it will rain. This poses no theoretical problems.

4.5. Equality

One should not lose sight of the fact that the logical extensions proposed in this chapter have the sole purpose of supporting TMS in an automatic theorem proving setting. Therefore the only kind of equality that is interesting is equality between individuals as it arises in the unextended theory. Equality between abstract predicates, for example, is of no concern nor is there any attempt at completeness in these extensions nor even a model theory for them. They remain practical arrangements. A good example of their utility as such is the way in which the retirement of relations into '$' predications simplifies the arrangement of the equality axioms.

Equality Axioms (Version 1)

```
$(=,T,T)

($(=,T1,H1) & $(=,T2,H2)) =>
($(R,T1,T2) => $(R,H1,H2))

($(=,T1,H1) & $(=,T2,H2) & $(=,T3,H3)) =>
($(R,T1,T2,T3) => $(R,H1,H2,H3))

($(=,T1,H1) & $(=,T2,H2) & $(=,T3,H3) & $(=,T4,H4)) =>
($(R,T1,T2,T3,T4) => $(R,H1,H2,H3,H4))
```

Note that R can be instantiated to '=' itself in the second axiom.

4.6. Temporal Anteriority

The main discussion of this business is in section 3.3.

Temporal Anteriority Axioms

```
~ $(<,I,I)

($(<,I,J) & $(<,J,K)) => $(<,I,K)

($(<,I,K) & $(<,J,K)) => ($(<,I,J) v $(=,I,J) v $(<,J,I))
```

This irreflexive, transitive, and left linear relation is of a more general form than that required by the semantics of tensed verbs because its defining axioms are stated using unsorted free variables of metatype *ind* and therefore all individuals are ordered whereas the only individuals which really need to be ordered are those of sort 's', the indices. It is a harmless superfluity which lends simplicity and, I hope, efficiency to the arrangement.

4.7. Comprehension Arranged

Methods of extending TMS for automatic processing have now been developed to a sufficient level for the schematic axioms to be represented in a finite database. The basic idea is that a sentence schema ϕ with free variables υ_1 through υ_n be represented by an abstract predication P..[υ_1,...,υ_n]. The globally free variable P of metatype *abs* is predicated of a single argument which is a list, expressed in Prolog list notation, of individual variables. The internal structure of abstract predicates (and indeed of lists) will soon be disclosed but need not be known to understand this.

The extended system has a pleasing modularity.

Abstract Comprehension Axioms

```
exists(U:o,all(I:s,$(true,U,I)<=>P..[I]))
exists(V:p,all(X:e,all(I:s,$(ppty,V,X,I)<=>P..[X,I])))
exists(W:q,all(V:p,all(I:s,$(qtty,W,V,I)<=>P..[V,I])))
```

This formulation is halfway between the first-order and second-order comprehension axioms presented previously. Its effect is to arrange for the first-order schemas or, equivalently, the three second-order axioms if the arbitrary-order schema has ϕ restricted to first-order sentences. It is analogous to von Neumann-Bernays set theory in its formulation without any class quantifiers.

4.8. Equality Rearranged

There is also an axiom schema of equality which is sometimes used instead of the axioms of equality special to any particular theory. It is

$$T=H \Rightarrow (\phi_{\upsilon}^{T} \Rightarrow \phi_{\upsilon}^{H})$$

where the result of replacing all free occurrences of the variable υ by the term α in the sentence ϕ (with the usual possible renaming of bound variables in ϕ) has been abbreviated to ϕ_{υ}^{α}.

In TMS, this schema can also be given a finite form using abstract predicates.

Equality Axioms (Version 2)

```
$(=,T,T)
$(=,T,H) => (P..[T] => P..[H])
```

It is unclear whether this arrangement is preferable to the version 1 axioms replacing equals for equals in *rel2*, *rel3*, and *rel4* arguments. Perhaps it is a matter for experiment.

4.9. Theory of Abstract Predicates

The three abstract comprehension axioms have been carefully crafted for use in automatic theorem proving. For pragmatic reasons their implicitly universally quantified free variables over abstract predicates are meant to be less demanding than explicit second-order universal quantifications. The arbitrary-order comprehension schema is *impredicative* because ϕ can contain higher-order quantifications. Furthermore it is always *incomplete* because there is no maximal-order logic to exhaust the complexity of predicates. However the aim here is only to arrange for the first-order comprehension schemas; hence abstract predicates have only to cover the *predicative* arbitrary-order comprehension schema with ϕ restricted to first-order sentences. That amounts to a portion of second-order logic for which this section effectively provides a *complete* finite axiomatisation.

Much of TMS can be explained without a structural description of abstract predicates but there need be no mystery about them. They simply abstract over the sentences (closed or open) of the unextended language from the parser. Here are the terms:

Abstract Predicates

and(*abs*,*abs*), do(*rel2*,*loc*,*loc*), do(*rel3*,*loc*,*loc*,*loc*), do(*rel4*,*loc*,*loc*,*loc*,*loc*), every(*sort*,*abs*), if(*abs*,*abs*), iff(*abs*,*abs*), no, not(*abs*), or(*abs*,*abs*), some(*sort*,*abs*), yes.

The most difficult problem in the theory of abstract predicates is that of abstraction over terms, especially bound variables, as they occur in the unextended language. The simplest solution seems to be to use De Bruijn numbers to represent bound variables as described in (Curien 1986) for the lambda calculus. Consider a translation of **Mary found the park and walked in it.**

$\exists x\ [\forall y\ [(\mathrm{PARK}\ y\ !) \leftrightarrow (x=y)] \wedge \exists j\ [(j<!) \wedge [(\mathrm{FIND}\ x\ M\ j) \wedge (\mathrm{IN}\ x\ \mathrm{WALK}\ M\ j)]]]$

De Bruijn dropped the alphabetic variables and used numbers as indicated by the following formula:

$\exists\ [\forall\ [(\mathrm{PARK}\ 1\ !) \leftrightarrow (2=1)] \wedge \exists\ [(1<!) \wedge [(\mathrm{FIND}\ 2\ M\ 1) \wedge (\mathrm{IN}\ 2\ \mathrm{WALK}\ M\ 1)]]]$

The binding for each variable can be found by counting outwards upto its number of lambdas or quantifiers. This technique has been adapted to TMS. The essential idea is to drop variables from quantifiers under abstraction and to abstract each term into the location of an individual in a list of individuals corresponding to the dropped variables in scope. For a bound variable this will be its position in the list expressed as a Peano numeral in the sequence: i, s(i), s(s(i)), etc. Counting starts from position one. For a constant or free variable α it will be the immediate location k(α).

Locations

i, k(*ind*), s(*loc*).

The final metatype in the theory is for lists of individuals. These are the objects of which abstract predicates are predicated. The well-known list notation of Prolog is used with the usual abbreviations such as [α,β] for [α|[β|[]]].

Lists of Individuals

[], [*ind* | *indlist*].

Every well-formed formula in the output from the parser can be encoded into an abstraction term.

EXAMPLE 31

```
<Mary found the park and walked in it.

f10| 0
   | f0| the
   |   | park
   | f2| mary
   |   | f8| f5| find
   |   |   |   | he(0)
   |   |   | f0| walk
   |   |   |   | f5| in
   |   |   |   |   | he(0)

>exists(X1:e,all(X2:e,park(X2,!)<=>X1=X2)&
 exists(I2:s,I2<!&find(m,X1,I2)&
 exists(V3:p,in(m,V3,X1,I2)&
 all(X4:e,all(I5:s,ppty(V3,X4,I5)<=>walk(X4,I5))))))

=some(e,and(every(e,iff(do(park,i,k(!)),do(=,s(i),i))),
 some(s,and(do(<,i,k(!)),and(do(find,k(m),s(i),i),
 some(p,and(do(in,k(m),i,s(s(i)),s(i)),
 every(e,every(s,iff(do(ppty,s(s(i)),s(i),i),do(walk,s(i),i))))))))))))..[]
```

It is not that I want to duplicate the means of expression in TMS for fun. The point is that any sentence can occur for ϕ in the comprehension schemas. The argument list will usually be non-empty because of the initial dependence of ϕ on its environment.

The concretion axioms arrange abstraction terms as an inner logic. They explicitly formalise the Tarskian semantics of the unextended language in the extended language. Those for connectives pass the environment down into subexpressions.

Concretion Axioms (Connectives)

```
(1)  yes..L

(2)  ~ no..L

(3)  not(P)..L       <=>  ~ P..L

(4)  and(P,Q)..L     <=>  (P..L  &  Q..L)

(5)  or(P,Q)..L      <=>  (P..L  v  Q..L)

(6)  if(P,Q)..L      <=>  (P..L  => Q..L)

(7)  iff(P,Q)..L     <=>  (P..L <=> Q..L)
```

Those for quantifiers pass an extended environment down.

Concretion Axioms (Quantifiers)

```
(8)  every(S,P)..L <=>  all( T:S, P..[T|L] )

(9)  some(S,P)..L  <=>  exists( T:S, P..[T|L] )
```

Those for relations dispense with the environment after use.

Concretion Axioms (Relations)

```
(10) (get(O1,L,T1) & get(O2,L,T2)) =>
     (do(R,O1,O2)..L           <=>  $(R,T1,T2))

(11) (get(O1,L,T1) & get(O2,L,T2) & get(O3,L,T3)) =>
     (do(R,O1,O2,O3)..L        <=>  $(R,T1,T2,T3))

(12) (get(O1,L,T1) & get(O2,L,T2) & get(O3,L,T3) & get(O4,L,T4)) =>
     (do(R,O1,O2,O3,O4)..L  <=>  $(R,T1,T2,T3,T4))
```

Those for terms look up individuals at locations in the environment.

Concretion Axioms (Terms)

```
(K)  get( k(T), L, T )

(I)  get( i, [T|L], T )

(S)  get( O, L, T ) => get( s(O), [H|L], T )
```

There are other approaches to the formulation of an inner logic. Instead of De Bruijn numbers it is possible to use inner variables as in (McCarthy 1979) which complicate environments, or to use combinators as in (Barnden 1983) which obscure abstract predicates. It is important to be neither inefficient nor impenetrable and the above method is a happy compromise.

4.9.1. Brace Abstraction

This subsection is concerned with two pieces of notation which are very helpful for understanding the system but which do nothing to increase its power. Let's introduce the terms {L ϕ} and α?L to abbreviate abstract predicates and locations respectively where L is a list of individual variables. They are called the abstraction of L from ϕ, and the location of α in L where ϕ and α are concrete formulae. The fundamental statements of their behaviour are

```
    {L A}..L        <=>  A
and
    get( T?L, L, T )
```

They are not first-order functions but metalanguage operators which are defined from previous simpler terms. The following definitions provide a brace abstraction algorithm for eliminating them from well-formed formulas of the unextended language. Those for connectives recurse down into subexpressions.

Abstraction Cases (Connectives)

```
(1)  {L 1}              ≡   yes
(2)  {L 0}              ≡   no
(3)  {L ~A}             ≡   not( {L A} )
(4)  {L (A & B)}        ≡   and( {L A}, {L B} )
(5)  {L (A v B)}        ≡   or( {L A}, {L B} )
(6)  {L (A => B)}       ≡   if( {L A}, {L B} )
(7)  {L (A <=> B)}      ≡   iff( {L A}, {L B} )
```

Those for quantifiers recurse with an extended variable list.

Abstraction Cases (Quantifiers)

```
(8)  {L all(T:S,A)}     ≡   every( S, {[T|L] A} )
(9)  {L exists(T:S,A)}  ≡   some( S, {[T|L] A} )
```

Those for relations recurse down into the variable list.

Abstraction Cases (Relations)

```
(10) {L R(T1,T2)}          ≡   do( R, T1?L, T2?L )

(11) {L R(T1,T2,T3)}       ≡   do( R, T1?L, T2?L, T3?L )

(12) {L R(T1,T2,T3,T4)}    ≡   do( R, T1?L, T2?L, T3?L, T4?L )
```

Those for terms bottom out eventually with appropriate locations.

Abstraction Cases (Terms)

```
(K)  T?L        ≡   k(T)      if constant(T:S) or L=[]

(I)  T?[T|L]    ≡   i

(S)  T?[H|L]    ≡   s(T?L)    if T≠H
```

These cases cover the entire unextended language suggesting that the arrangement is complete for the first-order comprehension axiom schemas.

4.10. Sort Reduction and Quantifier Removal

Pure sort reduction to standard first-order logic is easily arranged. Introduce a new primitive predicate (.:) corresponding to sortal predication and consider sorted quantifications to be abbreviations for relativised quantifications:

```
all( υ:S, φ )       ≡   all( υ, S.:υ=>φ )

exists( υ:S, φ )    ≡   exists( υ, S.:υ&φ )
```

However pure sort reduction violates the simplifying principle that exactly the bound variables are sorted while exactly the free variables are unsorted. These artificial constraints on binding and sorting are too useful to abandon.

If we want to use the resolution method of theorem proving then quantifiers must be removed entirely and sort reduction can be done simultaneously with that. The resulting quantifierless formulation uses sortal predications, the implicit universal quantification of free variables, and Skolem functions. It is appropriate to identify which Skolem terms arise from the axioms of TMS.

Individuals

!, b, choose(*sort*,*abs*,*indlist*), fail(*sort*,*abs*,*indlist*), io(*ind*,*ind*), ip(*ind*,*ind*), iq(*ind*,*ind*), j, k, m, u(*abs*), v(*abs*), vq(*ind*,*ind*), w(*abs*), xp(*ind*,*ind*).

The extensionality axioms give rise to 'io', 'xp', 'ip', 'vq', and 'iq'. If U1 and U2 are different propositions then io(U1,U2) is an index at which they differ in truth value. Properties are served similarly by 'xp' and 'ip', and quantities by 'vq' and 'iq'. The comprehension axioms give rise to 'u', 'v', and 'w'. If P is an abstract predicate then u(P), v(P), and w(P) are the corresponding proposition, property, and quantity whichever of these notions makes sense. For example, the property of loving Mary is

```
 v({[X,I] love(X,m,I)})
=v(do(love,i,k(m),s(i)))
```

If John has this property then

```
 $(ppty,v(do(love,i,k(m),s(i))),j,!)
=do(love,i,k(m),s(i))..[j,!]
=$(love,j,m,!)
```

Note that if P is "garbage" then not much can be proved about v(P). The concretion axioms (8) and (9) give rise to 'fail' and 'choose' respectively. If S is a sort, P is an abstract predicate, and L is a list of individuals then choose(S,P,L) is an individual of sort S which witnesses P in the environment L; and fail(S,P,L) witnesses not(P) similarly. For example, choose(e,do(child,i,!),[]) is the most childlike entity.

The constants 'j', 'm', 'b', 'k', and '!' also have associated axioms but these are rather trivial. Because constants are basically unsorted in the first-order languages, they must be deliberately assigned sorts as necessary. In the language of the translator the following circumlocutions do the job:

Sortal Assignment Axioms (Sorted Language)

```
exists( X:e, $(=,X,j) )
exists( X:e, $(=,X,m) )
exists( X:e, $(=,X,b) )
exists( X:e, $(=,X,k) )
exists( I:s, $(=,I,!) )
```

These are the last axioms of TMS. In the language of the theorem prover the non-emptiness of each sortal domain is axiomatised by the provision of some terms of that sort. However such terms already come from the TMS axioms by the removal of existential quantification. Although they come from almost everywhere, they are collected together here for exhibition.

Sortal Assignment Axioms (Unsorted Language)

```
e.:j & e.:m & e.:b & e.:k & s.:!
(o.:U1 & o.:U2) => s.:io(U1,U2)
(p.:V1 & p.:V2) => (e.:xp(V1,V2) & s.:ip(V1,V2))
(q.:W1 & q.:W2) => (p.:vq(W1,W2) & s.:iq(W1,W2))
o.:u(P) & p.:v(P) & q.:w(P)
S.:choose(S,P,L)
S.:fail(S,P,L)
```

The correct arrangement of sorts shown above is vital for the consistency of the system. For example, the argument of Russell's paradox can be used to prove that the proposition of {[T] ~true(T,T)} cannot be an index.

Theorem

```
~ s.:u(not(do(true,i,i)))
```

However u(P) is a proposition for all abstract predicates P, that is for all P of metatype *abs*. Note that u(man) is not even a term of the theory because 'man' has the metatype *rel2*. The management of metatypes by the implementation has not been explicitly described but it seems reasonable to suppose that a regime similar to that for polymorphic types (Mycroft and O'Keefe 1983) would not go wrong.

5. Conclusion

It is clear that most difficulties with Montague Grammar can be resolved by the methods of this thesis. The technique of letting the fragment, rather than intensional logic, determine the complexity of the implementation has been very rewarding. The problems of lambda-conversion and substitution of equivalents into opaque contexts have been reduced to easily processed first-order forms. A comprehensive axiom system has been introduced and it was shown how the schematic axioms can be finitely represented in a satisfactory manner. The goal of constraining Montague Grammar for use in computational linguistics is probably unachievable if the above methods are insufficient because it seems impossible to get significantly closer.

The specific axioms proposed here are the easiest to explain although it should not be misunderstood that they are the only ones possible. Minor increases in efficiency may be available with some variations. For example, the well-known fact that a single sentential connective and quantifier suffice could perhaps be used profitably in the theory of abstract predicates. Some experimentation with other axiomatisations may be worthwhile but it is hard to imagine a major advance here.

The remaining place where difficulties can be expected to arise is in the control strategy of the theorem prover. It would not be surprising if the strategies developed for mathematics were inappropriate for semantics. Mathematical proofs tend to involve lengthy deductions from few axioms whereas semantic ones have shorter deductions from more axioms. This characteristic is shared with database applications but these usually restrict the logic so that certain questions which can be asked of the system cannot be asserted to it as facts. For example, disjunctive facts may not be assertable to such a system. By allowing full first-order logic, my system of semantics is somewhere between mathematics and such databases. It does not seem unreasonable to hope that a workable question-answering system will be constructed in its entirity after practical experience with this kind of theorem proving has led to some insight into its desirable behaviour.

Acknowledgements

Many thanks to my supervisors Mark Steedman and Henry Thompson, to David Charles McCarty for supervising me during the Autumn of '86, and to Patrick Blackburn, Robin Cooper, Robert Dale, Nick Haddock, Ewan Klein, Chris Mellish, Dale Miller, Bob Moore, Glyn Morrill, Barry Richards, Ray Turner, and Henk Zeevat for their advice and their direct influence concerning the course of my thesis work.

This work was supported by the Science and Engineering Research Council research quota award 83307952. Thanks are due to Dr. J. A. M. Howe for helping to secure that funding.

Thanks also to Anne De Roeck, Barry Lowden, and Ray Turner for allowing me the time and resources to revise this thesis while I was emplcyed at the University of Essex.

References

Andrews, P. B., Miller, D. A., Cohen, E. L. & Pfenning, F. 1984
"Automating Higher-Order Logic"
American Mathematical Society, Contemporary Mathematics, vol. 29,
Automated Theorem Proving: After 25 Years, pp. 169-192
Providence, Rhode Island

Bainbridge, R. I. 1984
"A Strongly Equivalent Definite Clause Grammar Analogue
for Montague's "Proper Treatment of Quantification" Recursive Syntax"
Teesside Polytechnic, Department of Computer Science
Middlesbrough, Cleveland, England

Barnden, J. A. 1983
"Intensions As Such: An Outline"
Proc. IJCAI 1983, pp. 280-286

Bennett, M. 1974
"Some Extensions of a Montague Fragment of English"
UCLA Ph.D. Dissertation
repr. by Indiana University Linguistics Club

Bronnenberg, W. J. H. J., Bunt, H. C., Landsbergen, S. P. J.,
Scha, R. J. H., Schoenmakers, W. J. & Van Utteren, E. P. C. 1980
"The Question-Answering System PHLIQA 1"
Natural Language Question-Answering Systems, pp. 217-305
ed. Bolc, L.
Macmillan, London

Chierchia, G. 1984
"Topics in the Syntax and Semantics of Infinitives and Gerunds"
University of Massachusetts Ph.D. Dissertation

Church, A. 1940
"A Formulation of the Simple Theory of Types"
Journal of Symbolic Logic, vol. 5, pp. 56-68

Clocksin, W. F. & Mellish, C. S. 1981
Programming in Prolog
Springer-Verlag, Berlin, Heidelberg, New York

Cooper, R. 1979
"Variable Binding and Relative Clauses"
Formal Semantics and Pragmatics for Natural Languages, pp. 131-169
eds. Guenthner, F. & Schmidt, S. J.
Reidel, Dordrecht

Cooper, R. 1983
Quantification and Syntactic Theory
Reidel, Dordrecht

Curien, P. L. 1986
Categorical Combinators, Sequential Algorithms and Functional Programming
Pitman, London

Dowty, D. R., Wall, R. E. & Peters, S. 1981
Introduction to Montague Semantics
Reidel, Dordrecht

Friedman, J. 1978
"Computational and Theoretical Studies in Montague Grammar
at the University of Michigan"
SISTM Quarterly, June 1978, pp. 62-66

Friedman, J. & Warren, D. S. 1978
"A Parsing Method for Montague Grammars"
Linguistics and Philosophy, vol. 2, pp. 347-372

Gallin, D. 1975
Intensional and Higher-Order Modal Logic with Applications
to Montague Semantics
North-Holland, Amsterdam

Goldfarb, W. D. 1981
"The Undecidability of the Second-Order Unification Problem"
Theoretical Computer Science, vol. 13, pp. 225-230

Groenendijk, J. & Stokhof, M. 1982
"Semantic Analysis of WH-Complements"
Linguistics and Philosophy, vol. 5, pp. 175-233

Henkin, L. 1950
"Completeness in the Theory of Types"
Journal of Symbolic Logic, vol. 15, pp. 81-91

Huet, G. P. 1975
"A Unification Algorithm for Typed λ-Calculus"
Theoretical Computer Science, vol. 1, pp. 27-57

Jackendoff, R. 1979
"How to Keep Ninety from Rising"
Linguistic Inquiry, vol. 10, pp. 172-176

Janssen, T. M. V. 1980
"Logical Investigations on PTQ Arising from Programming Requirements"
Synthese, vol. 44, pp. 361-390

Janssen, T. M. V. 1980
"On Problems Concerning the Quantification Rules in Montague Grammar"
Linguistische Arbeiten, vol. 83,
Time, Tense, and Quantifiers, pp. 113-134
ed. Rohrer, C.
Max Niemeyer Verlag, Tubingen

Janssen, T. M. V. 1983
Foundations and Applications of Montague Grammar
Mathematisch Centrum, Amsterdam

Jowsey, H. E. 1985
"Constraining Formal Grammar for Computational Applications:
A Thesis Proposal"
University of Edinburgh, Department of Artificial Intelligence
DAI Discussion Paper: 2

Jowsey, H. E. 1986
"Montague Grammar and First-Order Logic"
University of Edinburgh, Department of Artificial Intelligence
DAI Working Paper: 190
repr. Working Papers in Cognitive Science, vol. 1,
Categorial Grammar, Unification Grammar and Parsing, pp. 143-194
eds. Haddock, N., Klein, E. & Morrill, G.
University of Edinburgh, Centre for Cognitive Science

Keenan, E. L. 1984
"A Boolean Approach to Semantics"
Truth, Interpretation and Information, pp. 65-97
eds. Groenendijk, J. &al.
Foris, Dordrecht

McCarthy, J. 1979
"First-Order Theories of Individual Concepts and Propositions"
Expert Systems in the Micro-electronic Age, pp. 271-287
ed. Michie, D.
Edinburgh University Press, Edinburgh

Miller, D. A. & Nadathur, G. 1986
"Some Uses of Higher-Order Logic in Computational Linguistics"
University of Pennsylvania, Computer and Information Science

Montague, R. 1970
"Universal Grammar"
Theoria, vol. 36, pp. 373-398
repr. (Montague 1974), pp. 222-246

Montague, R. 1973
"The Proper Treatment of Quantification in Ordinary English"
Approaches to Natural Language, pp. 221-242
eds. Hintikka, K. J. J. &al.
Reidel, Dordrecht
repr. (Montague 1974), pp. 247-270

Montague, R. 1974
Formal Philosophy: Selected Papers of Richard Montague
ed. Thomason, R. H.
Yale University Press, New Haven

Mycroft, A. & O'Keefe, R. A. 1983
"A Polymorphic Type System for Prolog"
University of Edinburgh, Department of Artificial Intelligence
DAI Research Paper: 211
repr. Proc. Logic Programming Workshop 1983

Partee, B. H. 1977
"John is Easy to Please"
Linguistic Structures Processing, pp. 281-312
ed. Zampolli, A.
North-Holland, Amsterdam

Pereira, L. M., Pereira, F. C. N. & Warren, D. H. D. 1979
"User's Guide to DECsystem-10 Prolog"
University of Edinburgh, Department of Artificial Intelligence
DAI Occasional Paper: 15

Pereira, F. C. N. 1983
"Logic for Natural Language Analysis"
University of Edinburgh, Department of Artificial Intelligence
Ph.D. Thesis
repr. SRI International, Technical Note 275

Pereira, F. C. N. & Shieber, S. M. 1987
Prolog and Natural-Language Analysis
CSLI Lecture Notes, no. 10
University of Chicago Press, Chicago

Robinson, J. A. 1965
"A Machine-Oriented Logic Based on the Resolution Principle"
Journal of the Association for Computing Machinery, vol. 12, pp. 23-41

Schubert, L. K. & Pelletier, F. J. 1982
"From English to Logic: Context-Free Computation
of 'Conventional' Logical Translation"
American Journal of Computational Linguistics, vol. 8, pp. 26-44

Thomason, R. H. 1980
"A Model Theory for Propositional Attitudes"
Linguistics and Philosophy, vol. 4, pp. 47-70

Appendices

1. Miscellaneous Programs

1.1. Operator Declarations

```
| :- op( 1200, xfx,  >> ).          % Labelled Rule
| :- op( 1150, xfx,  <- ).          % Condition
| :- op(  850, xfx, <=> ).          % Equivalence
| :- op(  850, xfx,  => ).          % Implication, Combination
| :- op(  800, xfy,   v ).          % Disjunction
| :- op(  800, xfy,   & ).          % Conjunction
| :- op(  750,  fy,   ~ ).          % Negation
| :- op(  700, xfx,  .. ).          % Abstract Predication
| :- op(  700, xfx,  .: ).          % Sortal Predication
| :- op(  700, xfx,  := ).          % Lexical Assignment
| :- op(  700, xfx,  *= ).          % Definition of Lexical Abbreviation
| :- op(  500, xfx,   - ).          % List Difference
| :- op(  500, yfx,   + ).          % Concatenation
| :- op(  450, xfx,   : ).          % Translation, Sorting
| :- op(  450, xfy,   # ).          % Arc Construction
| :- op(  400, xf,   pn ).          % Pronoun Marking
| :- op(  400, xfx,   / ).          % Alternative Words
| :- op(  400, xf,    * ).          % Lexical Abbreviation
| :- op(  100, xfx,   ` ).          % Application of Syntactic Combination Function
|                                   % to List of Syntactic Structures
```

1.2. Top Level Driver

```
| go( Opt ) :-                      % Instructions: A procedure call go(s), go(k),
|         abolish( option, 1 ),     % go(a), or go(n) runs the system, showing the
|         assert( option(Opt) ),    % translations in standard form, Kowalski form,
|         go.                       % abstracted form, or no form respectively.
|                                   % After the prompt '<' enter the words to be
| go :-                             % parsed followed by .RETURN or enter just
|         repeat,                   % .RETURN to quit. After a translation enter
|         nl, write( < ),           % ;RETURN to seek another parse or enter RETURN
|         readwords( Buffer ),      % to abandon that sentence. The procedure call
|         process( Buffer ).        % 'go' initially runs as go(s) but subsequently
|                                   % reruns the system as it last ran.
| process( [] ).
| process( Buffer ) :-
|         parse( Buffer, Str, A ),
|         closed( Str ),
|         nl, pp( Str ),
|         option( Opt ),
|         report( Opt, A ),
|         get_return,
|         !,
|         fail.
|
| get_return :-
|        get0( 0'
|                )．
| get_return :-
|        skip( 0'
|                ),
|        fail.
|
| report( s, A ) :-
|        standard( A ),
|        nl, write( > ), write( A ).
| report( k, A ) :-
|        kf( A, K ),
|        kf_pp( K ).
| report( a, A ) :-
|        standard( A ),
|        abstract( A, P ),
|        nl, write( > ), write( P..[] ).
| report( n, A ) :-
|        write( ? ).
|
| option( s ).
```

1.3. English Word Reader

```
| readwords( Ws ) :-
|        get( C ),
|        readchars( C, Cs ),
|        !,
|        words( Ws, Cs, [] ),
|        !.
|
| readchars( 0'., [] ) :-
|        skip( 0'
|                ).
| readchars( C, [C|Cs] ) :-
|        get0( C1 ),
|        readchars( C1, Cs ).
|
| words([W|Ws]) --> word(W), !, blanks, words(Ws).
| words([]) --> [].
|
| word(W) --> [C], {lc(C,D)}, !, alphas(Cs), {name(W,[D|Cs])}.
|
| alphas([D|Cs]) --> [C], {lc(C,D)}, !, alphas(Cs).
| alphas(Cs) --> "'", !, alphas(Cs).
| alphas([]) --> [].
|
| blanks --> [C], {C=<0' }, !, blanks.
| blanks --> [].
|
| lc( C, C ) :-
|        C >= 0'a,
|        C =< 0'z.
| lc( C, D ) :-
|        C >= 0'A,
|        C =< 0'Z,
|         plus( 0'A, N, 0'a ),
|         plus( C, N, D ).
```

1.4. Syntactic Structure Closure Check

```
| closed( Str ) :-
|         closed( Str, [] ), !.
|
| closed( Root, _ ) :-
|         atom( Root ).
| closed( he(N), Env ) :-
|         member( N, Env ).
| closed( Fn`Args, Env ) :-
|         \+ closer( Fn ),
|         closedlist( Args, Env ).
| closed( Fn`[N|Args], Env ) :-
|         closer( Fn ),
|         closedlist( Args, [N|Env] ).
|
| closedlist( [], _ ).
| closedlist( [Arg|Args], Env ) :-
|         closed( Arg, Env ),
|         closedlist( Args, Env ).
|
| closer( f3 ).
| closer( f10 ).
```

1.5. Syntactic Structure Pretty Printer

```
| pp( Str ) :-
|         pp( Str, 0, n, y ), !.
|
| pp( Fn`Args, Depth, X, Y ) :-
|         write_bars( X, Depth ),
|         write_fbar( Fn ),
|         pplist( Args, s(Depth), n, Y ).
| pp( Root, Depth, X, y ) :-
|         write_bars( X, Depth ),
|         write( ' ' ),
|         write( Root ),
|         nl.
|
| pplist( [], _, X, X ).
| pplist( [Arg|Args], Depth, X, Z ) :-
|         pp( Arg, Depth, X, Y ),
|         pplist( Args, Depth, Y, Z ).
|
| write_bars( n, _ ).
| write_bars( y, 0 ).
| write_bars( y, s(Depth) ) :-
|         write( '   |' ),
|         write_bars( y, Depth ).
|
| write_fbar( f10 ) :-
|         write( 'f10|' ).
| write_fbar( Fn ) :-
|         write( ' ' ),
|         write( Fn ),
|         write( '|' ).
```

1.6. First-Order Logic Variable Standardiser

```
| standard( Phi ) :-
|         standard( Phi, 1 ), !.
|
| standard( ~Phi, N ) :-
|         standard( Phi, N ).
| standard( Phi&Psi, N ) :-
|         standard( Phi, N ),
|         standard( Psi, N ).
| standard( Phi v Psi, N ) :-
|         standard( Phi, N ),
|         standard( Psi, N ).
| standard( Phi=>Psi, N ) :-
|         standard( Phi, N ),
|         standard( Psi, N ).
| standard( Phi<=>Psi, N ) :-
|         standard( Phi, N ),
|         standard( Psi, N ).
| standard( all(Var:Sort,Phi), N ) :-
|         variable( Sort, N, Var ),
|         succ( N, M ),
|         standard( Phi, M ).
| standard( exists(Var:Sort,Phi), N ) :-
|         variable( Sort, N, Var ),
|         succ( N, M ),
|         standard( Phi, M ).
| standard( Phi, N ).
|
| variable( Sort, N, Var ) :-
|         prefix( Sort, Char, _ ),
|         name( N, NStr ),
|         name( Var, [Char|NStr] ).
|
| skolem_term( Sort, Args, Alpha ) :-
|         retract( skolem(N) ),
|         succ( N, M ),
|         assert( skolem(M) ),
|         prefix( Sort, _, Char ),
|         name( N, NStr ),
|         name( Fun, [Char|NStr] ),
|         Alpha =.. [Fun|Args].
|
| skolem( 1 ).
|
| prefix( e, 0'X, 0'x ).
| prefix( s, 0'I, 0'i ).
| prefix( o, 0'U, 0'u ).
| prefix( p, 0'V, 0'v ).
| prefix( q, 0'W, 0'w ).
```

1.7. Kowalski Formatter

```
| kf( Phi, K ) :-
|         findall( X, literals( Phi, work(1,[],X-[]/[]) ), K ).
|
| literals( 0, Work ) :-
|         !, empty( Work ).
| literals( 1, Work ) :-
|         !, fail.
| literals( Phi&Psi, Work ) :-
|         !, conjuncts( Phi, Psi, Work ).
| literals( Phi v Psi, Work ) :-
|         !, disjuncts( Phi, Psi, Work ).
| literals( Phi=>Psi, Work ) :-
|         !, disjuncts( ~Phi, Psi, Work ).
| literals( Phi<=>Psi, Work ) :-
|         !, conjuncts( Psi=>Phi, Phi=>Psi, Work ).
| literals( all(Var:Sort,Phi), work(N,F,L) ) :-
|         variable( Sort, N, Var ),
|         succ( N, M ),
|         !, disjuncts( ~{Sort.:Var}, Phi, work(M,[Var|F],L) ).
| literals( exists(Var:Sort,Phi), Work ) :-
|         Work = work(_,F,_),
|         skolem_term( Sort, F, Var ),
|         !, conjuncts( {Sort.:Var}, Phi, Work ).
| literals( ~0, Work ) :-
|         !, fail.
| literals( ~1, Work ) :-
|         !, empty( Work ).
| literals( ~ ~Phi, Work ) :-
|         !, literals( Phi, Work ).
| literals( ~(Phi&Psi), Work ) :-
|         !, disjuncts( ~Phi, ~Psi, Work ).
| literals( ~(Phi v Psi), Work ) :-
|         !, conjuncts( ~Phi, ~Psi, Work ).
| literals( ~(Phi=>Psi), Work ) :-
|         !, conjuncts( Phi, ~Psi, Work ).
| literals( ~(Phi<=>Psi), Work ) :-
|         !, conjuncts( Phi v Psi, ~Phi v ~Psi, Work ).
| literals( ~all(Var:Sort,Phi), Work ) :-
|         Work = work(_,F,_),
|         skolem_term( Sort, F, Var ),
|         !, conjuncts( {Sort.:Var}, ~Phi, Work ).
| literals( ~exists(Var:Sort,Phi), work(N,F,L) ) :-
|         variable( Sort, N, Var ),
|         succ( N, M ),
|         !, disjuncts( ~{Sort.:Var}, ~Phi, work(M,[Var|F],L) ).
| literals( ~Phi, Work ) :-
|         predication( Phi, Psi ),
|         !, antecedent( Psi, Work ).
| literals( Phi, Work ) :-
|         predication( Phi, Psi ),
|         !, consequent( Psi, Work ).
|
| empty( work(_,_,X-X) ).
|
| antecedent( Phi, work(_,_,C/[Phi|A]-C/A) ).
```

```
|
| consequent( Phi, work(_,_,[Phi|C]/A-C/A) ).
|
| conjuncts( Phi, Psi, Work ) :-
|         literals( Phi, Work ).
| conjuncts( Phi, Psi, Work ) :-
|         literals( Psi, Work ).
|
| disjuncts( Phi, Psi, work(N,F,X-Z) ) :-
|         literals( Phi, work(N,F,X-Y) ),
|         literals( Psi, work(N,F,Y-Z) ).
|
| predication( {A}, A ).
| predication( A, $(R,T1) ) :-
|         relation( R, T1:_, A ).
| predication( A, $(R,T1,T2) ) :-
|         relation( R, T1:_, T2:_, A ).
| predication( A, $(R,T1,T2,T3) ) :-
|         relation( R, T1:_, T2:_, T3:_, A ).
| predication( A, $(R,T1,T2,T3,T4) ) :-
|         relation( R, T1:_, T2:_, T3:_, T4:_, A ).
```

1.8. Kowalski Form Pretty Printer

```
| kf_pp( K ) :-
|         kf_pp( n, K ), !.
|
| kf_pp( _, [] ).
| kf_pp( n, [X|K] ) :-
|         literals_pp( X ),
|         kf_pp( y, K ).
| kf_pp( y, K ) :-
|         nl,
|         kf_pp( n, K ).
|
| literals_pp( []/[] ) :-
|         nl,
|         write( 0 ).
| literals_pp( C/A ) :-
|         literals_pp( +, C ),
|         literals_pp( -, A ).
|
| literals_pp( _, [] ).
| literals_pp( S, [Phi|B] ) :-
|         nl,
|         write( S ),
|         write( ' ' ),
|         write( Phi ),
|         literals_pp( S, B ).
```

1.9. Brace Abstracter

```
| abstract( Phi, P ) :-
|         abstract( [], Phi, P ), !.
|
| abstract( L, 0, no ).
| abstract( L, 1, yes ).
| abstract( L, ~Phi, not(P) ) :-
|         abstract( L, Phi, P ).
| abstract( L, Phi&Psi, and(P,Q) ) :-
|         abstract( L, Phi, P ),
|         abstract( L, Psi, Q ).
| abstract( L, Phi v Psi, or(P,Q) ) :-
|         abstract( L, Phi, P ),
|         abstract( L, Psi, Q ).
| abstract( L, Phi=>Psi, if(P,Q) ) :-
|         abstract( L, Phi, P ),
|         abstract( L, Psi, Q ).
| abstract( L, Phi<=>Psi, iff(P,Q) ) :-
|         abstract( L, Phi, P ),
|         abstract( L, Psi, Q ).
| abstract( L, all(T:S,Phi), every(S,P) ) :-
|         abstract( [T|L], Phi, P ).
| abstract( L, exists(T:S,Phi), some(S,P) ) :-
|         abstract( [T|L], Phi, P ).
| abstract( L, Phi, do(R,O1) ) :-
|         predication( Phi, $(R,T1) ),
|         location( T1, L, O1 ).
| abstract( L, Phi, do(R,O1,O2) ) :-
|         predication( Phi, $(R,T1,T2) ),
|         location( T1, L, O1 ),
|         location( T2, L, O2 ).
| abstract( L, Phi, do(R,O1,O2,O3) ) :-
|         predication( Phi, $(R,T1,T2,T3) ),
|         location( T1, L, O1 ),
|         location( T2, L, O2 ),
|         location( T3, L ,O3 ).
| abstract( L, Phi, do(R,O1,O2,O3,O4) ) :-
|         predication( Phi, $(R,T1,T2,T3,T4) ),
|         location( T1, L, O1 ),
|         location( T2, L, O2 ),
|         location( T3, L, O3 ),
|         location( T4, L, O4 ).
|
| location( T, L, k(T) ) :-
|         constant( T:_ );
|         L = [].
| location( T, [T|L], i ).
| location( T, [H|L], s(O) ) :-
|         T \= H,
|         location( T, L, O ).
```

1.10. List Processing Utilities

```
| appendlist( [], [] ).
| appendlist( [H|T], L ) :-
|         appendlist( T, R ),
|         append( H, R, L ).
|
| append( [], L, L ).
| append( [H|T], L, [H|R] ) :-
|          append( T, L, R ).
|
| member( X, [H|T] ) :-
|          X = H;
|          member( X, T ).
|
| select( X, [X|L], L ).
| select( X, [H|T], [H|L] ) :-
|          select( X, T, L ).
```

2. The Axioms of TMS in Kowalski Form

I assume that the theorem prover will process a set of clauses, whose members are conjuncts in Kowalski's normal form:

$$(\phi_1 \ \& \ \dots \ \& \ \phi_m) \Rightarrow (\psi_1 \ \mathrm{v} \ \dots \ \mathrm{v} \ \psi_n)$$

where ϕ_i and ψ_j are predications. Interpret the empty conjunction as truth and the empty disjunction as falsity.

Such clauses will henceforth be printed vertically, thus:

$+ \ \psi_1$
$\ \ \ .$
$\ \ \ .$
$\ \ \ .$
$+ \ \psi_n$
$- \ \phi_1$
$\ \ \ .$
$\ \ \ .$
$\ \ \ .$
$- \ \phi_m$

Clauses from the axiomatisation of TMS have been given mnemonic labels which roughly have the forms '*+name*' for sortal assignments, '*name+*' and '*name-*' for clauses coming in positive-negative pairs, and (when negative, for example) '*-name*' and '*name-*' for clauses coming in left-right pairs.

Clause	Label	Section and Explanation
+ s.:io(U1,U2)	'+io'	4.1. Extensionality: Propositions
- o.:U1		
- o.:U2		
+ $(true,U1,io(U1,U2))	'true+'	
+ $(true,U2,io(U1,U2))		
+ $(=,U1,U2)		
- o.:U1		
- o.:U2		
+ $(=,U1,U2)	'true-'	
- o.:U1		
- o.:U2		
- $(true,U1,io(U1,U2))		
- $(true,U2,io(U1,U2))		

```
+ e.:xp(V1,V2)                       '+xp'     4.1. Extensionality: Properties
- p.:V1
- p.:V2

+ s.:ip(V1,V2)                       '+ip'
- p.:V1
- p.:V2

+ $(ppty,V1,xp(V1,V2),ip(V1,V2))  'ppty+'
+ $(ppty,V2,xp(V1,V2),ip(V1,V2))
+ $(=,V1,V2)
- p.:V1
- p.:V2

+ $(=,V1,V2)                         'ppty-'
- p.:V1
- p.:V2
- $(ppty,V1,xp(V1,V2),ip(V1,V2))
- $(ppty,V2,xp(V1,V2),ip(V1,V2))

+ p.:vq(W1,W2)                       '+vq'     4.1. Extensionality: Quantities
- q.:W1
- q.:W2

+ s.:iq(W1,W2)                       '+iq'
- q.:W1
- q.:W2

+ $(qtty,W1,vq(W1,W2),iq(W1,W2))  'qtty+'
+ $(qtty,W2,vq(W1,W2),iq(W1,W2))
+ $(=,W1,W2)
- q.:W1
- q.:W2

+ $(=,W1,W2)                         'qtty-'
- q.:W1
- q.:W2
- $(qtty,W1,vq(W1,W2),iq(W1,W2))
- $(qtty,W2,vq(W1,W2),iq(W1,W2))

+ o.:u(P)                            '+u'      4.7. Comprehension: Propositions

+ $(true,u(P),I)                     'u+'
- s.:I
- P..[I]

+ P..[I]                             'u-'
- s.:I
- $(true,u(P),I)

+ p.:v(P)                            '+v'      4.7. Comprehension: Properties

+ $(ppty,v(P),X,I)                   'v+'
- e.:X
- s.:I
- P..[X,I]
```

```
+ P..[X,I]                  'v-'
- e.:X
- s.:I
- $(ppty,v(P),X,I)

+ q.:w(P)                   '+w'     4.7. Comprehension: Quantities

+ $(qtty,w(P),V,I)          'w+'
- p.:V
- s.:I
- P..[V,I]

+ P..[V,I]                  'w-'
- p.:V
- s.:I
- $(qtty,w(P),V,I)

+ $(=,T,T)                  '=+'     4.5., 4.8. Equality: Reflexive

+ $(R,H1,H2)                '2=-'    4.5. Equality: 2-Place Substitutive
- $(=,T1,H1)
- $(=,T2,H2)
- $(R,T1,T2)

+ $(R,H1,H2,H3)             '3=-'    4.5. Equality: 3-Place Substitutive
- $(=,T1,H1)
- $(=,T2,H2)
- $(=,T3,H3)
- $(R,T1,T2,T3)

+ $(R,H1,H2,H3,H4)          '4=-'    4.5. Equality: 4-Place Substitutive
- $(=,T1,H1)
- $(=,T2,H2)
- $(=,T3,H3)
- $(=,T4,H4)
- $(R,T1,T2,T3,T4)

+ P..[H]                    '=-'     4.8. Equality: Substitutive
- $(=,T,H)
- P..[T]

- $(<,I,I)                  '-<'     4.6. Anteriority: Irreflexive

+ $(<,I,K)                  '<-'     4.6. Anteriority: Transitive
- $(<,I,J)
- $(<,J,K)

+ $(<,I,J)                  '+<'     4.6. Anteriority: Left Linear
+ $(=,I,J)
+ $(<,J,I)
- $(<,I,K)
- $(<,J,K)

+ e.:j                      '+j'     4.10. Sortal Assignment: John

+ e.:m                      '+m'     4.10. Sortal Assignment: Mary
+ e.:b              '+b'     4.10. Sortal Assignment: Bill

+ e.:k              '+k'     4.10. Sortal Assignment: Kate

+ s.:!              '+!'     4.10. Sortal Assignment: Here and Now

+ yes..L            'yes+'   4.9.(1) Concretion: Truth

- no..L             'no-'    4.9.(2) Concretion: Falsity

+ not(P)..L         'not+'   4.9.(3) Concretion: Negation
+ P..L

- P..L              'not-'
- not(P)..L

+ and(P,Q)..L       'and+'   4.9.(4) Concretion: Conjunction
- P..L
- Q..L

+ P..L              '-and'
- and(P,Q)..L

+ Q..L              'and-'
- and(P,Q)..L

+ or(P,Q)..L        '+or'    4.9.(5) Concretion: Disjunction
- P..L

+ or(P,Q)..L        'or+'
- Q..L

+ P..L              'or-'
+ Q..L
- or(P,Q)..L

+ if(P,Q)..L        '+if'    4.9.(6) Concretion: Implication
+ P..L

+ if(P,Q)..L        'if+'
- Q..L

+ Q..L              'if-'
- if(P,Q)..L
- P..L

+ iff(P,Q)..L       '+iff'   4.9.(7) Concretion: Equivalence
+ P..L
+ Q..L

+ iff(P,Q)..L       'iff+'
- P..L
- Q..L

+ P..L              '-iff'
- iff(P,Q)..L
- Q..L
```

```
+ Q..L                          'iff-'
- iff(P,Q)..L
- P..L

+ S.:fail(S,P,L)                '+fail'   4.9.(8) Concretion: Universality

+ every(S,P)..L                 'every+'
- P..[fail(S,P,L)|L]

+ P..[T|L]                      'every-'
- S.:T
- every(S,P)..L

+ S.:choose(S,P,L)              '+choose' 4.9.(9) Concretion: Existence

+ some(S,P)..L                  'some+'
- S.:T
- P..[T|L]

+ P..[choose(S,P,L)|L]          'some-'
- some(S,P)..L

+ do(R,O1,O2)..L                'do2+'    4.9.(10) Concretion: 2-Place Relation
- get(O1,L,T1)
- get(O2,L,T2)
- $(R,T1,T2)

+ $(R,T1,T2)                    'do2-'
- get(O1,L,T1)
- get(O2,L,T2)
- do(R,O1,O2)..L

+ do(R,O1,O2,O3)..L             'do3+'    4.9.(11) Concretion: 3-Place Relation
- get(O1,L,T1)
- get(O2,L,T2)
- get(O3,L,T3)
- $(R,T1,T2,T3)

+ $(R,T1,T2,T3)                 'do3-'
- get(O1,L,T1)
- get(O2,L,T2)
- get(O3,L,T3)
- do(R,O1,O2,O3)..L

+ do(R,O1,O2,O3,O4)..L          'do4+'    4.9.(12) Concretion: 4-Place Relation
- get(O1,L,T1)
- get(O2,L,T2)
- get(O3,L,T3)
- get(O4,L,T4)
- $(R,T1,T2,T3,T4)

+ $(R,T1,T2,T3,T4)              'do4-'
- get(O1,L,T1)
- get(O2,L,T2)
- get(O3,L,T3)
- get(O4,L,T4)
- do(R,O1,O2,O3,O4)..L
```

```
+ get(k(T),L,T)                 'k+'     4.9.(K) Concretion: Constant

+ get(i,[T|L],T)                'i+'     4.9.(I) Concretion: First Variable

+ get(s(O),[H|L],T)             's+'     4.9.(S) Concretion: Successor
- get(O,L,T)
```

To these permanent initial clauses will be added the clauses coming from the temporary assumptions which happen to have been asserted.

EXAMPLE 31

```
<Mary found the park and walked in it.

f10| 0
   | f0| the
   |   | park
   | f2| mary
   |   | f8| f5| find
   |   |   |   | he(0)
   |   |   | f0| walk
   |   |   |   | f5| in
   |   |   |   |   | he(0)

+ e.:x1

+ $(park,X1,!)
- e.:X1
- $(=,x1,X1)

+ $(=,x1,X1)
- e.:X1
- $(park,X1,!)

+ s.:i2

+ $(<,i2,!)

+ $(find,m,x1,i2)

+ p.:v3

+ $(in,m,v3,x1,i2)

+ $(ppty,v3,X1,I2)
- e.:X1
- s.:I2
- $(walk,X1,I2)

+ $(walk,X1,I2)
- e.:X1
- s.:I2
- $(ppty,v3,X1,I2)
```

The system can then be queried by asserting the negation of the theorem to be proved and attempting to deduce the empty clause by resolution.

QUESTION 1

```
<Mary found an entity.

 f2| mary
   | f5| find
   |   | f0| a
   |   |   | entity

- s.:I1
- $(<,I1,!)
- e.:X2
- $(find,m,X2,I1)
```

The above four goals resolve against the fourth, fifth, first, and sixth clauses respectively in Example 31 to affirm the question "Is it the case that Mary found an entity?" meaning "Did Mary find anything?". An extremely simple example has been chosen but the same system covers all of Montague Grammar.