A Situation Theoretic Approach to Computational Semantics

PhD Thesis

Alan W Black

Department of Artificial Intelligence
Faculty of Science and Engineering
University of Edinburgh

1992

Foreword

This document consists of re-edited version of my PhD thesis, submitted to Department of Artificial Intelligence, University of Edinburgh, December 1992.

An implementation of the computational language ASTL, described and used throughout this thesis is available by anonymous ftp. The software is free and may be freely redistributed (subject to the attached GNU General public licence). The software is available from

scott.cogsci.ed.ac.uk [129.215.144.3]:pub/awb/astl-0.4.tar.Z

Note that this is still an experimental version, later versions will be better. The above version includes the example ASTL descriptions described in Chapters 4, 5 and 6 (which are also reproduced in Appendix A).

Alan W Black
awb@ed.ac.uk
Centre for Cognitive Science
University of Edinburgh
March, 1993

Abstract

This thesis presents an approach to the description of natural language semantic theories within a situation theoretic framework. In recent years, research has produced a number of semantic theories of natural language that primarily deal with very similar phenomena, such as quantification and anaphora. Although these theories often deal with similar data it is not always possible to see differences between theories' treatments due to differences in the theories' syntax, notations and definitions. In order to allow better comparison of theories, the idea of a general semantic meta-language is discussed and a suitable language in presented.

ASTL is a computational language which is formally defined. It is based on fundamental aspects of situation theory. It offers representations of individuals, relations, parameters, facts, types and situations. It also offers inter-situation constraints and a set of inference rules is defined over them. In order to show ASTL's suitability as a computational meta-language three contemporary semantic theories are described within it: Situation Theoretic Grammar—a situation semantic based theory, Discourse Representation Theory and a form of dynamic semantics.

The results show that at least core parts of these semantic theories can be described in ASTL. Because ASTL has an implementation, it directly offers implementation of the theories described in it. The three descriptions can be closely compared because they are described in the same framework. Also this introduces the possibility of sharing treatments of semantic phenomena between theories.

Various extensions to ASTL are discussed but even in its simplest form it is powerful and useful both as an implementation language and specification language. Finally we try to identify what essential properties of ASTL make it suitable as a computational meta-language for natural language semantic theories.

Acknowledgements

Firstly I would like to thank Robin Cooper. His comments and guidance through this work has added greatly to it as well as helping me understand what I am doing. Graeme Ritchie has now for many years given me help in my research (and career) both at a high and low level, for which I thank him. Ian Lewin has also contributed to my work through long discussions in which I would try to explain to him what I was trying to do and more often he could tell me.

I am also indebted to various funding bodies who have made my studies possible. The SERC funded the majority of this work through a postgraduate studentship (number 89313458). Also towards the end of this work I have been more than adequately funded by Esprit Basic Research Action Project 6852 (DYANA-2). In addition to the major contributions I also wish to acknowledge funding for travel from Esprit Basic Research Action Project 3175 (DYANA) and Department of Artificial Intelligence which allowed me to attend conferences and workshops at which some of this work was presented. At these events I gained much useful experience and background. Thanks also go to Gail Anderson of AIAI for arranging use of a workstation during most of this project.

I would also like to thank Richard Tobin and Jeff Dalton who have put up with me for some years now offering cheap accommodation, food and home based computing services (both for work and diversion). I also cannot forget my fellow students who I have served my time with: John Beaven, Matt Crocker, Flávio Corrêa da Silva, Carla Pedro Gomes, Ian Frank, Ian Lewin, Nelson Ludlow, Suresh Manandhar, Dave Moffat, Keiichi Nakata, Brian Ross, Rob Scott, Wamberto Vasconcelos and others who have passed through E17. Without them my time in Edinburgh would not have been so enjoyable. There are others too who have contributed to my views and work in my previous incarnations in the Edinburgh research community, I thank them. I am grateful to have had the opportunity to be part of such a very stimulating community.

Contents

Chapter 1

Introduction

Since the development of computers one of the many areas of research has been the automatic processing of human language. In the beginning it was hoped that natural language processing would not be too difficult and the expectations were high. The translation of one natural language automatically into another was thought to be possible and many projects were started. However, it was quickly discovered that it would not be as simple as first thought. First, better theories of natural language were needed and secondly better theories of programming were needed in order to implement language theories efficiently. It is not unconnected that during this time there was an increase in the study of theoretical linguistics which offered theories of language more suitable for computer implementation. Work in Artificial Intelligence however often tried to develop its own computational theories of language, which were concerned more with computation than with linguistics.

Although the overall goal of high performance automatic natural language processing was shared between the theorists and the pragmatists differences of opinion did exist. Many implementors believed that linguistic theory was not relevant to building practical computational systems. It was felt that too much theory in an implemented system would do little to improve performance. There is the story, probably apocryphal, of the speech processing group who would sack a linguist to make their system run faster. Although both sides have their extremists what is really necessary is knowing which parts of linguistic theory can benefit practical applications and which should be ignored for the present. However with the steady improvement in power of computer systems more and more aspects can be reasonably implemented.

Initial theoretical work in natural language processing has concentrated on syntax, and even today that area is probably the most studied. Although there are still many problems to solve, practical syntactic grammars, which have a firm theoretical grounding, exist for significant fragments of some natural languages. Semantics, the meaning of language, is still trailing a little behind, maybe because it is more difficult or because it is prerequisite to have a basic theory of syntax in which a semantic theory may be described. Formal philosophy and logic has worried about the meaning of language for thousands of years but it is only in the last thirty years or so that computational issues have started to influence these theories.

It is important that formal semantics not be ignored in the development of practical natural language systems. Although an implementation may ignore certain aspects it is important to understand exactly what the consequences are in ignoring certain aspects of theoretical semantics. Even if theories are not directly embodied in systems, theories of semantics are important in order to give a better understanding of what the limitations of implementations are.

Today, there are a number of computational semantic theories offering treatments of a few interesting semantic phenomena. Mostly these theories concentrate on similar aspects of language. Although many theories seem to be addressing similar issues it is not always possible to give a detailed comparison of them because of differences in notation, differences in emphasis, and even differences in versions of each theory. It would aid the development of semantic theories of natural language greatly if there were a theoretically based system in which contempory semantic theories could be compared more easily. Also it is important to realise that developing computational theories of natural language semantics is not obvious. Understanding the consequences of an abstract definition is not easy. Computers should not just be seen as the ultimate delivery agent, computers can also act as a useful tool in the development and experimentation of theories.

This thesis takes a theoretical approach to the description and implementation of aspects of contemporary semantic theories of natural language. Situation theory ([Barwise & Perry 83], [Barwise 89b]) is used as the basis for a computational language called ASTL. Semantic theories can be described in ASTL and because ASTL has an implementation, it offers an implementation for theories described in it. Because these semantic theories are described within the same environment differences in notation, syntax etc. can be factored out and a very detailed comparison can be made between them. Also as theories are described within the same environment the prospect of sharing treatments of semantic phenomena becomes possible.

1.1 Outline of chapters

Chapter 2 discusses aspects of computational semantics. It gives brief descriptions of some current semantic theories of natural language and some of the currently investigated semantic phenomena. Situation theory is introduced and the notion of general meta-language for semantics theories is discussed. Various possible frameworks within which such a language could be developed are discussed and reasons for choosing situation theory are given.

Chapter 3 introduces the situation theoretic language ASTL. It defines ASTL's formal syntax and semantics as well as its inference rules. Simple examples are given and one possible implementation of this language is described.

In order to justify ASTL as a computational meta-language for describing aspects of semantic theories it is necessary to give detailed examples. Chapter 4 shows how simple language processing is possible in ASTL and a simple syntactic framework is introduced which will be used in later examples. Then the first of three ASTL descriptions of semantic theories is given. The first is Situation Theoretic Grammar (STG) [Cooper 89] which is a situation semantic theory. Although this theory is closest to the ideas built into ASTL it is important to show that ASTL is capable of describing basic situation semantics. The next two chapters deal with theories that address much of the same phenomena and hence are suitable for close comparison. Chapter 5 gives a detailed description of Discourse Representation Theory (DRT) ([Kamp 81] [Kamp & Reyle 93]). Chapter 6 discusses dynamic semantics as in Dynamic Predicate Logic (DPL) [Groenendijk & Stokhof 91b]. A description in ASTL is given for the logic DPL as well as a dynamic semantic treatment of the same simple language fragment used in the preceding two examples. Comparisons are made between the DRT and dynamic semantics descriptions showing how closely they compare and what the actual differences between these two theories are.

Chapter 7 shows how once in a framework of situation theory aspects of it can be easily adopted into semantic theories described within in it. This chapter not only discusses extensions to described theories but also useful extensions to ASTL itself to make descriptions easier.

Finally, Chapter 8 again discusses why a situation theoretic based language like ASTL is suitable as a meta-language and exactly what properties make it so. Also it re-iterates the basic thesis arguments. The contributions are identified and conclusions drawn.

Chapter 2

Computational Semantics

2.1 Introduction

In processing natural language by computer a number of techniques have been used to try to capture the *meaning* of natural language utterances. In early natural language processing systems meanings were often computed in a rather *ad hoc* fashion. SHRDLU [Winograd 72], an early system, translated sentences into procedures whose evaluation (i.e. execution) would achieve the desired interpretation of the utterance. As such there was not really an abstract semantic representation language, let alone a formal definition of it. Semantics in natural language processing and artificial intelligence systems were typically very specific to the task and embedded within the actual implementation. (A good description of the issues at the time is given in [Charniak & Wilks 76].) Representational formalisms from that time have survived but much work has been done to characterise these formalisms and give them a more formal semantics. For example semantic nets at first were fairly arbitrary until [Woods 75] began to try to define them. Eventually they developed into KL-ONE [Brachman & Schmolze 85] which does have a detailed formal semantics.

Another thread in the field of computational semantics, though not always separate, is the substantial work already done in philosophy and linguistics on formal logic. With the advent of computers computational systems based on logics appeared. The whole area of *logic programming* was developed, part of which is devoted to language processing. The idea of using a computer to translate natural language utterances into a logical form is best typified by CHAT-80 [Pereira 82]. CHAT-80 is a Prolog program which parses English queries about world geography, the queries are converted into simple logical forms and after some manipulation to optimise the query, checked against a geographical database. However, early on it was feared that first order predicate logic was not rich enough to capture all the various semantic phenomena found in natural language utterances. Higher order logics would probably be required although they are significantly more difficult to deal with computationally.

Within this chapter (and this thesis) the term *computational semantics* will be used for the field of study that primarily is interested in using formal logics in computational natural language systems. By *computational* we mean those systems that are developed with, at least, possible computer implementation in mind but more often those systems that have actual implementations. Computational semantics can be seen as a bridge between formal semantics (typically logic) and applied natural language processing. There could be two aspects to computational semantics, first the translation of natural language utterances into a semantic representation (and the choice of representation language) and secondly the use of the translation and inferences we can draw from it. Although we will touch on the second aspect primarily we will be dealing with the translation and representation.

As the ultimate goal in computational semantic is a computational treatment which we can actually use on a computer, semantic theories should be specificied in such a way so that this is possible. The specifications we give of theories in later chapters do meet that criterion.

However before we lay out the aims and methodology of this thesis we will outline some of the major areas in computational semantics that have been studied, both theories and phenomena. Montague Grammar [Montague 74] provided a basis from which much of the current work in computational semantics derives (or at least is inspired by). Specifically we will look at the areas of quantification and anaphora. Characteristic problems in semantics will be listed which are used as targets for theories. Some specific theories will be described and which of the identified problems they address (and fail to address) will be given. The second part of this chapter will describe the work on Situation Semantics and Situation Theory ([Barwise & Perry 83], [Barwise 89b]) its motivation and its current state in the field of computational semantics. Then the idea of a semantic theory meta-language is introduced and possible areas from which such a language may be found are discussed. Finally a short discussion will be given justifying the direction taken in the rest of this thesis.

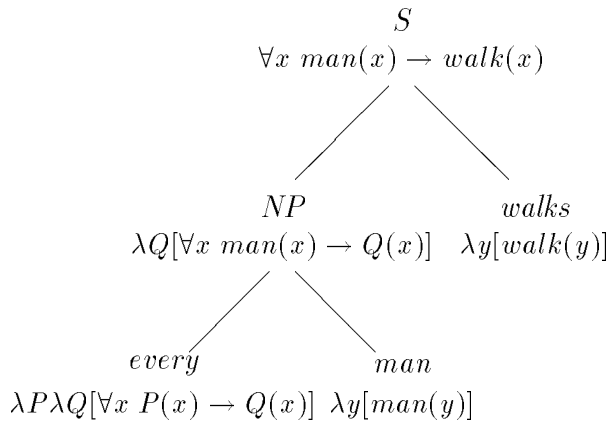
2.2 Montague Grammar

Montague Grammar was probably the first example of a semantic system for natural language which had a detailed formal definition. It shows how a formal logic treatment of language can be made for a non-trivial subset of English. Although, of course, not fully comprehensive it still has set a "standard" for more contemporary systems for certain semantic phenomena. Montague's original papers from the 60's and 70's are unfortunately difficult to read (many are collected together in [Thomason 74]). Later introductions ([Thomason 74], [Partee 75]) helped make the rest of the logic and philosophy community aware of his work. However [Dowty *et al* 81] is probably the most accessible description.

A short description of Montague Grammar is given here as it is a good example of the basic model for computational semantics used within this thesis. The basic idea is that a natural language utterance can be translated into an expression in a *semantic representation language*. Using an interpretation function defined for that semantic representation language the *meaning* of the utterance can be found. In the case of Montague Grammar the representation language is Intensional Logic, while the interpretation function is the semantics for Intensional Logic. The basic notion in Montague Grammar is that the meaning of a sentence is a function from worlds to truth values. That is in order to know the meaning of a sentence one must know the circumstances in which it is true or false. The semantics of the Intensional Logic translation of an utterance reflects this. Note that the Intensional Logic translation of an utterence is merely an intermediate form but not in itself the semantics.

A much simplified example follows. Here we simply use first order logic and the lambda calculus rather than full Intensional Logic in order to make the example a little more readable.

An important aspect of Montague Grammar is that the semantic rules are related to syntactic grammar rules thus offering a strict *compositional* treatment of semantics. That is for every syntactic constituent in the grammar fragment there exists a semantic translation. In order to achieve this a liberal use of lambda abstraction is necessary. A typical analysis of a simple sentence *"every man walks"* would be as follows. The syntax and semantic translations are shown for each node.



In this example the translation of the mother node is achieved by functional application of the translation of the daughters. Computationally this can be implemented by applying the semantic translation of one daughter to the other and using beta reduction to find the normalised form.

Montague Grammar actually uses Intensional Logic for its semantic representation. This includes modal operators, intensional operators (up arrow and down arrow) and lambda abstraction. [Montague 74] presents a fragment of English with both syntax and semantics. In many ways this example sets the target for later semantic theories. Montague's work showed not only how to represent some examples of natural language utterances in logic but also how to construct a logical translation from syntactic parse trees. Montague concentrated on a number of specific semantic phenomena. Within his fragment he gave treatments for simple declarative sentences, quantifiers, bound anaphora and others. Treatments of intensional aspects of language were also included.

Montague's fragment is by no means fully comprehensive but does offer a firm ground. Much development has since taken place both in its formal aspects and increasing its coverage. Although inference in Intensional Logic is in general computationally undecidable, Montague Grammar does offer a method for implementation and has been used as a semantic basis in a number of implemented systems (e.g. [Clifford 90]).

Since the original work on Montague Grammar a number of new theories and extensions have been developed. Some of the motivation of this later work was to address specific problems in semantics which could not be dealt with in Montague Grammar's original form. Sometimes these have been extensions to Montague Grammar itself, as in [Muskens 89] where partiality is added to possible worlds, or new theories as in Discourse Representation Theory [Kamp 81].

2.3 Some semantic phenomena

Many of these extensions and new theories were motivated by particular problems in semantics. We will look at two particular areas of semantics: quantification and anaphora, and identify some problems. These problems were either treated by Montague's original fragment, and hence have become points with which other theories are compared; or were missing or inadequately treated, and required extensions or new theories.

Quantifiers, such as *"every"*, *"a"*, *"at least three"* etc. are common in natural language utterances but their interpretation is sometimes tricky. When more than one quantifier appears in an utterance there can be an ambiguity.

Every man loves a woman

is normally taken to be ambiguous between there being one particular woman who all men love and all men loving some (possibly different) woman. This ambiguity is shown clearly in the two possible logical forms for this sentence. The first represents the case where each man loves some woman (but not necessary all the same) while the second case is where there is one particular woman loved by all.

$$\forall x \; man(x) \rightarrow [\exists y[woman(y) \wedge love(x, y)]]$$
$$\exists y \; woman(y) \wedge [\forall x[man(x) \rightarrow love(x, y)]]$$

As we can see the order of the quantifiers is crucial in differentiating the two cases. This phenomenon is referred to as *quantifier scope*.

Unfortunately it is not simply the case that all readings of a sentence can be found by finding all permutations of the quantifiers in its resultant translation. Some combinations are not permitted. Various solutions have been proposed to find all possible scopings. In the original work of Montague different scopings were achieved by different syntactic analyses of the same utterance. Later work, [Cooper 83], proposed that the alternative scopings could be generated non-deterministically during semantic analysis. Even later work has further partitioned off the work of finding quantifier scopings from building semantic representations. The idea of a representation that does not yet have its scopings resolved has been used in a number of actual systems. Most typical is the Core Language Engine (CLE) where a *quasi-logical form* (QLF) is generated and later processed to find the possible scopings [Alshawi 92]. Various algorithms have been proposed for finding the possible scopings given a QLF or similar representation ([Lewin 90], [Hobbs & Shieber 87]).

A second problem in quantification can be shown as follows. In basic Montague Grammar the representation for the determiners *"every"* and *"a"* are

$$\begin{array}{ll} every & \text{—}\ \lambda P \lambda Q[\forall x\ P(x) \rightarrow Q(x)] \\ a & \text{—}\ \lambda P \lambda Q[\exists x\ P(x) \wedge Q(x)] \end{array}$$

There are other quantifiers, *"few"*, *"most"*, *"at least three"* etc. If the above framework were to be followed all would require their own unique form. In order to make the representation of quantifiers more consistent we can view all quantifiers as two-place relations between properties. Thus determiners would be represented as

$$\begin{array}{ll} every & \text{—}\ \lambda P \lambda Q[forall(P,Q)] \\ a & \text{—}\ \lambda P \lambda Q[exists(P,Q)] \\ most & \text{—}\ \lambda P \lambda Q[most(P,Q)] \end{array}$$

where P and Q would be some form of property such as lambda abstractions as in $\lambda x[man(x)]$ and $\lambda x[mortal(x)]$. This representation for quantifiers removes the need for specialised logical operators within the representations (i.e. $\rightarrow$ and $\wedge$ in the examples above). This allows a more consistent treatment. It also makes possible a treatment for quantifiers like *most*—as *most* must be defined as a relation between the sets defined by the two arguments rather than a simple logical operator between the two. This form of representation for quantifiers is called *generalised quantifiers*. The first argument (P) is sometimes called the *range* while the second (Q) is sometimes called the *body* (or *scope*) of the quantifier. Generalised quantifiers are more fully discussed in [Barwise & Cooper 82].

There are other phenomena which although not directly related to quantification can be treated in a similar form. Comparatives have been given a treatment within a framework of generalised quantifiers [Pulman 91]. Also a number of adverbs can also be treated like quantifiers (e.g. *"usually"*, *"sometimes"*, *"always"* etc.) [Chiercha 92].

The second area of phenomena we will identify is various forms of anaphora (or pronoun use). There is already a large selection of data on various forms of anaphora found in natural language (see [Hirst 81] for a good review). One of the simplest forms of anaphora can been seen in the following examples.

A man_1 *walks.* He_1 *talks.*

The *"He"* in the second sentence can refer to the man identified in the first[1]. This we will call *inter-sentential anaphora* where a pronoun refers to an object in the discourse introduced in an earlier sentence.

A second form of anaphora is what is termed *bound anaphora* where a pronoun appears within the scope of a quantifier and refers to the object(s) introduced by the quantifier. That is the pronoun acts like a bound variable. For example

Every student$_1$ *revised his*$_1$ *paper.*

where *"his"* refers to each student. This relation between anaphora and quantifiers they are within the scope of, is a major part of some syntactic theories—often termed *binding theory* as in GB [Chomsky 81].

Another form of anaphora which has inspired a lot of study is what has come to be called *donkey anaphora* due to the classic example sentence.

Every farmer who owns a donkey beats it.

Originally discussed in [Geach 62], the *"it"* in the above sentence does not (under at least one reasonable interpretation) refer to one particular donkey but to the donkey(s) belonging to each farmer. That is the referent for *"it"* is dependent on the quantifier introduced by *"a donkey"* which in turn is dependent on the quantifier introduced by *"every farmer"*. This again shows how anaphora can be closely related to the treatment of quantification.

The problem can be further explained in looking at potential logical forms of the sentence. One possible (and correct) form is

$$\forall x \forall y\ [[farmer(x) \wedge donkey(y) \wedge own(x,y)] \rightarrow beat(x,y)]$$

Note that here we require a universal quantifier to represent the indefinite noun phrase *"a donkey"* while in a simple sentence like

$$\text{“}a\ farmer\ walks\text{”} \leadsto \exists x\ [farmer(x) \wedge walk(x)]$$

the indefinite is represented by an existential quantifier. Naive attempts to give a more unified treatment fail, as the simple translation of *"Every farmer who owns a donkey beats it"* as

$$\forall x\ [[farmer(x) \wedge \exists y\ donkey(y) \wedge own(x,y)] \rightarrow beat(x,y)]$$

[1]We will sometimes use the convention of subscripting to show the referent for anaphora.

is not a valid expression as the y in the right hand side of the implication lies outwith the scope of the existential quantifier that introduces y. Another possible translation might be

$$\forall x \exists y\ [[farmer(x) \wedge donkey(y) \wedge own(x,y)] \rightarrow beat(x,y)]$$

but this, although logically well-formed does not capture the meaning of the English utterance. The above is true in the following model

$$\begin{array}{ll} farmer(a) & own(a,b) \\ donkey(b) & cat(c) \end{array}$$

where a owns a donkey but does not beat it.

As we can see we need to translate indefinites to either universal quantifiers (when already within the scope of a universal quantifier) or existential quantifiers otherwise. It would be more convenient if a uniform treatment of indefinites could be given.

As well as simple anaphora for noun phrases there is also the phenomenon of verb phrase ellipsis. As in

Hanako met Noriko and so did Taro.

Normally we would wish to treat this as Hanako met Noriko and also Taro met Noriko[2]. However things are more complex when the verb phrase in the first clause contains a quantifier or a pronoun.

Hanako ate a pizza and so did Taro.
Hanako met her mother and so did Noriko.

The first is ambiguous as to whether Hanako and Taro ate the same pizza or not (differentiated by the scope of existential introduced by the indefinite *"a pizza"*). The second is ambiguous as to whether Noriko met Hanako's mother (called the *strict* reading) or her own mother (called the *sloppy* reading). This is to do with whether the the verb phrase representation that is "re-used" contains the pronoun or its referent from the first use. Examples like these are discussed in detail in [Gawron & Peters 90]. Their descriptions are within the framework of situation semantics and hence it is not always easy to see their relationship with the work on VP ellipsis in DRT (e.g. [Partee 84]) and dynamic logic [Gardent 91].

In addition to semantic phenomena there are also aspects of computational semantics that are to do with technique rather than merely linguistic adequacy. A characteristic

[2]Throughout this thesis instead of the classic example proper names of *"John"* and *"Mary"* for variety we will use Japanese examples. *"Hanako"* and *"Noriko"* are common Japanese female names while *"Taro"* is a common male name.

which many consider to be essential in a semantic theory is *compositionality*. Basically compositionality means that the meaning of an utterance is made from the meaning of its parts. However it is actually difficult to find any computational treatment for semantics where this can be untrue (in general) (see [Zadronzy 92] for some formal discussion on this point). A stronger definition that is sometimes imposed is that for each *syntactic* constituent of an utterance there is a corresponding semantic translation and that that translation is solely composed from a function of the *semantic translations* of the syntactic parts of that constituent. Even with this stricter definition it is possible to convert almost any theory to this form by simply complicating either the semantic components or the conjoining process (e.g. by checking for different cases). Making the constituents more complex is not the intention of the proponents of compositionality. In fact, compositionality is a property that is difficult to define satisfactorily. Its status as a desired property is probably because it is a property of Montague Grammar where semantic rules are directly linked to syntactic ones, while in more contemporary theories an emphasis on compositionality should perhaps not be so necessary or appropriate.

Another phenomenon which is often considered abstractly from the actual semantic theory used is *incrementality*. This is where there is a representation for each initial substring of an utterance. Again, like compositionality, it seems that incrementality can always be achieved at the expense of the complexity of the representation. More detailed definitions can specify that the representation for each initial substring must have a semantic denotation. Again it is unclear what the ultimate purpose is in achieving incrementality. Such a direction really needs other justifications such as psychological or human performance issues (see [Crocker 91] for more discussion of this point).

As we have stated there are a number of aspects of quantification and anaphora that are closely related: quantifier scope, various quantifiers, plurals, inter-sentential anaphora, VP ellipsis etc. Various solutions to these problems have been proposed but often in quite different frameworks. This can make comparisons of solutions to problems difficult as well as sometimes requiring duplicate research.

2.4 Some semantic theories

Now that we have seen a number of semantic phenomena we will briefly describe some semantic theories which have been designed to treat such phenomena. Each of the three theories described are described in more detail in later chapters, so only a high level overview of them and their motivation is given here. Also we try to highlight aspects of them which justify the direction taken in this thesis.

2.4.1 Discourse Representation Theory

Discourse Representation Theory (DRT), as its name suggests, offers a representation for discourses [Kamp 81], [Kamp & Reyle 93]. Only a brief description is given here, a more in-depth description being given in Chapter 5. The "state" of a discourse is represented by a *Discourse Representation Structure* (DRS). DRSs are typically drawn as boxes consisting of two parts: the top section contains *discourse markers* which are introduced by nouns; and the bottom section consists of conditions about those markers. A typical DRS for the sentence *"A man walks"* is

```
 X
------------
 man(X)
 walk(X)
```

Two important aspects of DRT can be shown by a simple example of how pronoun resolution is achieved: that is the *structural* and *dynamic* aspects of the theory. Given the context of the above sentence, a following sentence *"He talks"* would extend the above DRS such that it would look like

```
 X Y
------------
 talk(Y)
 is(X,Y)
 man(X)
 walk(X)
```

(For the purpose of this example we will ignore that fact that sometimes we cannot deal with the words in a sentence in exactly the same order as they appear.) In the second sentence, the *"He"* introduces a new discourse marker (`Y`) and finds some previous discourse marker (of the right type) (`X`) which it can be related to, then the processing of the verb adds the condition `talk(Y)`. The extending of a DRSs through the processing of the discourse shows the *dynamic* aspect of DRT. Effectively we can view this as the sentence adding to an incoming DRS to produce an outgoing DRS (which is the treatment we will adopt in Chapter 5).

In Montague Grammar the denotations are simply truth values and functions. In DRT there is a *structural* aspect to the semantics. DRSs themselves are said to be not just intermediate representations built as a convenience in processing but as representations of psychologically real structures necessary in the analysis of language. Although this may be an extreme way to put it, it is true that the DRS structure is actively used in analysis. In pronoun resolution, possible candidate referents are found by looking at the current DRS itself. This *structural* (or it could be called *informational*) aspect is relatively new to computational semantic theories.

As well as offering a representation for the content of discourses DRT also offers a *construction algorithm* which shows how a DRS can be constructed from a parse tree of an utterance. This aspect is important as DRT is not only concerned about semantic representation but also with the computational processing required to construct such a representation.

So we can see that DRT does offer something new to computational semantics. It offers both dynamic and structural aspects which are missing in the Montague Grammar framework. It includes a construction algorithm as part of the theory, noting that construction of representation is as important as the representation itself.

DRT does not just offer representations for simple sentences: even in its simplest form it deals with simple quantifiers. *"Every"* is translated as a conditional, as a relation between two sub-DRS. Indefinite noun phrases are translated with implicit existentials. This, and the way universals are treated, allows a uniform treatment of indefinites both within the scope of universal quantifiers and without. Thus DRT offers a clean treatment of donkey anaphora.

Later extensions to DRT [Kamp & Reyle 93] have included a treatment for generalised quantifiers which introduces a diamond-shaped box identifying a discourse marker and relating two sub-DRSs. DRT has also been used as a basic framework for other phenomena. Temporal anaphora, where events are introduced as discourse markers has been described by [Partee 84] and others. However, also more general semantic phenomena which do not directly depend on the basic features of DRT have been described within a DRT framework (e.g. [Lascarides & Asher 91] on commonsense entailment).

2.4.2 Dynamic semantics

Dynamic semantics follows the basic idea that the meaning of an utterance transforms some input "context" to produce a output "context" which will form the input "context" of the next part of the discourse. This idea has come from the techniques in theoretical computer science in defining the semantics of computer languages ([Harel 84]). For example, a "program" {`x := x + 1`} can transform an input state g to an output state h that differs only from g such that the value of x in h is 1 larger than the value of x in g. This transforming of state is the reason for the use of the term *dynamic*.

Dynamic Predicate Logic [Groenendijk & Stokhof 91b] was developed as a reply to DRT. DRT had been a move away from the classical logical perspective (and more precisely away from Montague Grammar). Dynamic semantics is an attempt to bring the semantic coverage of DRT back into a standard logical framework. To do this the conventional syntax of logical expressions is used but the semantics is changed. A DPL expression denotes a set of pairs of input and output states which represent the valid input and output contexts the expression can appear in.

A typical example of a DPL expression representing the utterance *"a man walks"* is

$$\exists x[man(x)] \wedge walk(x)$$

Although the second x would in a conventional (non-dynamic) logic lie outwith the scope of the existential quantifier this is not the case in DPL. The semantics of DPL is such that variable bindings introduced by an existential quantifier are held in *assign-*

ments which can be referred to later in the expression—the details of this are described in Chapter 6.

DPL is a very simple logic which is basically first order. Dynamic Montague Grammar (DMG) [Groenendijk & Stokhof 91a] is an attempt to deal with a richer logic (called Dynamic Intensional Logic—DIL) in a dynamic way. Unlike DPL, DMG relates natural language utterances to logical translations.

An important aspect of dynamic logic is that it offers a simple compositional treatment for sentences. In a conventional logic in order for a variable (or discourse marker) introduced in one sentence to be referred to in the next (e.g. in the use of inter-sentential anaphora) it is necessary that the second sentence appears within the scope of any existential quantifier introduced in the first. In dynamic logic two sentences can simply be conjoined by a (dynamic) conjunction operator. For example, if we have the following discourse

A man_1 *walks.* He_1 *talks*

a conventional (non-dynamic) logic representation of the first sentence must allow for the possibility of including the succeeding sentence within the scope of the existential introduced in the first. This might look something like the following

$$\lambda p[\exists x\ [man(x) \wedge walk(x) \wedge p]]$$
$$talk(x)$$

But this is still inadequate as although we can now get the second sentence within the scope of the existential in the first, the x in the second sentence is free and there is no reason that it should be the same x as in the first sentence even after application. Because of the dynamic treatment of existentials in DPL we can represent the first sentence in DPL as

$$\exists x\ [man(x) \wedge walk(x)]$$

and represent the two sentences in DPL as

$$\exists x\ [man(x) \wedge walk(x)] \wedge talk(x)$$

and still have the x in $talk(x)$ be the same as the x introduced by the existential. We can compare this with the non-dynamic expression, which for both sentences would be

$$\exists x\ [man(x) \wedge walk(x) \wedge talk(x)]$$

Crucially we can see that in the non-dynamic case there is no sub-expression which represents the first sentence. This aspect is argued as a reason why the non-dynamic treatment is non-compositional. Of course in the dynamic case a redefinition of the conjunction operator is necessary in order to achieve compositionality.

With respect to DRT, DPL offers a conventional logical treatment of one of the major difficult semantic phenomenon covered by DRT—donkey anaphora. But unlike Montague Grammar, DPL keeps the dynamic aspect of the translation.

2.4.3 Situation Theory

In the early 80s there was a group who proposed a new theory to natural language semantics. Situation Semantics and what has later become known as Situation Theory ([Barwise & Perry 83, Barwise 89b]) were devised as an alternative to possible world semantics. It was a move away from conventional logics which only have relatively simple objects in the semantic domain to having much more complex semantic objects. Within this movement, although at first there was little distinction, today there is a split between *situation theory*: the formal aspects of the theory, mathematical, logical, philosophical, logical, proof theoretic etc.; and *situation semantics*: the application of situation theory to the semantics of natural language.

Some early motivation for the development of the situation theory was sentences of the form

John saw Mary walk.
John saw Mary walk and Bill talk or not talk.

In a conventional classical logic (one that is rich enough to represent embedded sentences), there is no way to distinguish between these two examples, they are semantically equivalent—while intuitively there seems to be a difference.

Situation theory introduces the notion of a *situation*. Situations can intuitively be thought of as parts of the world. Unlike possible worlds, situations are *partial*—they do not define the truth/falsity of all relations on all objects in the domain. Situations *support* facts[3]. Facts have a polarity (1 or 0) representing whether the fact is positive or negative (in some situation). A simple example would be

$$S_1 \models \ll walk, mary; 1 \gg$$

which is used to represent the fact that Mary walks in the situation S_1. With a notion of polarity the truth and falsity of a fact is not dependent on the supports relation, so that

[3] There is sometimes some confusion in the terms *infon*, *fact*, *possible fact* and *soa* (state of affairs). Some proponents of situation theory make distinctions between these depending on whether they are actual (part of the real world) or not. To continue this confusion I will not distinguish between these terms but will typically use the term *fact*.

$$S_2 \not\models \ll walk, mary; 1 \gg$$

does not imply

$$S_2 \models \ll walk, mary; 0 \gg$$

This allows the two linguistic examples above to be have distinct representations

"John saw Mary walk." $\leadsto$

$$S_1 \models \ll walk, mary; 1 \gg$$
$$S_2 \models \ll see, john, S_1; 1 \gg$$

"John saw Mary walk and Bill talk or not talk." $\leadsto$

$$S_3 \models \ll walk, mary; 1 \gg$$
$$S_3 \models \ll talk, bill; 1 \gg \vee \ll talk, bill; 0 \gg$$
$$S_4 \models \ll see, john, S_3; 1 \gg$$

Thus the notion of a situation offers a way to deal with partial information and a way to hold information in distinct places (a fact may be positive in one situation but negative (or unknown) in another). An important aspect of the theory is that situations are *first class* objects. They may be used as arguments in relations. This offers an important level of power to the theory as relations can not just be *in* situations but also hold *between* situations and other objects.

A second important aspect of situation theory is that of *parameters*. The idea of parameters is to allow the representation of under-defined objects. In logic, variables are syntactic expressions but in situation theory the idea is that these "variables" should be in the semantic domain—although this has been considered by some to be a difficult direction to go in. This use of parameters and *anchoring* (analogous to assignments to variables) allows situation theory to describe what would be called variables and assignment within the theory itself rather than only in the meta-theory used to describe the logic.

The distinction of situation theory versus situation semantics occurred as the area matured. Situation theory concerns itself with the philosophical, mathematical and logical aspects of the field while situation semantics concerns itself with defining a situation theoretic account of natural language semantics. Although we talk about *situation semantics* as a theory it is not true that there is one clearly defined situation semantic theory but a collection of theories which are all defined in (or at least appeal to) aspects of situation theory. As we have a natural language semantic theory (situation semantics) defined in terms of a general theory (situation theory) there is the question whether other (non-situation semantic) theories might also be able to be defined within a situation theoretic framework.

Although this question seems an appealing and interesting question to investigate, there are problems. Situation theory is still a young area and it is constantly changing. The early work [Barwise & Perry 83] is notoriously difficult to read and not formally fully specified. As the field is still new there are many views on its best courses and many options even for the most fundamental aspects of the theory. So much so that there is even a paper defining some of the possible questions about the basic theory [Barwise 89a]. However there is progress albeit sometimes slowly, both on situation theory such as the work on inference [Barwise & Etchemendy 90], and in situation semantics such as [Gawron & Peters 90] which shows a situation semantic treatment of quantification and anaphora. Other work in the field has shown a treatment of classically difficult logical representation problems like paradoxes as in [Barwise & Etchemendy 87] where a treatment of the liar paradox is discussed. A more detailed and formal description of some aspects of situation theory is given in Chapter 3.

Computationally, situation semantics is even more in its infancy. Because of its youth firm definitions have not been possible, making it difficult to extract a fragment that is suitable for implementation. However some small systems have been attempted. The language *Determiner-Free Aliass* outlined in [Barwise & Perry 83, Ch 6] has been implemented [Braun *et al* 88, Polzin *et al* 89].

One computational use of situation theory which has gained a number of followers is *situation schemata*. Situation schemata [Fenstad *et al* 87] are a method for encoding a form of fact (or infon) in an attribute-value matrix. A typical example for the sentence *"John walks"* may look like

$$\left[\begin{array}{ll}
SITSCHEMA & \left[\begin{array}{ll}
REL & walk \\
ARG.1 & \left[\begin{array}{ll} IND & John \\ FOCUS & \left[\begin{array}{ll} IND & []_{John} \end{array}\right] \end{array}\right] \\
LOC & \left[\begin{array}{ll} IND & IND.1 \\ COND & \left[\begin{array}{ll} REL & \circ \\ ARG.1 & []_1 \\ ARG.2 & l_d \end{array}\right] \end{array}\right] \\
FOCUS & \left[\begin{array}{ll} IND & []_{John} \end{array}\right] \\
POL & 1
\end{array}\right] \\
FSTRUC & \left[\begin{array}{ll}
SUBJ & d_1 \left[\begin{array}{ll} PRED & 'John' \\ NUM & SG \end{array}\right] \\
TENSE & PRESENT \\
PRED & 'walk < []_{d1} >'
\end{array}\right]
\end{array}\right]$$

Situation schemata can be built up in a conventional feature grammar using unification of partial schemata. This is similar to the technique used to build a conventional logical forms in a unification feature grammar (as in [Shieber 86a]), but the semantics of schemata is given in a situation theoretic way. Depending on the instantiation of situation schemata it is possible to view schemata as equivalent to QLFs (quasi-logical forms [Alshawi 92]) as they can have unresolved aspects such as quantification. Situation schemata are probably the most accessible implementational device available in situation semantics and have been used in a number of applications (e.g. see [Rupp 89] or [Cooper 90]).

The above computational treatments of situation semantics are interesting in that they do not use their resulting representation in any active way. They use some other processing (or computational formalism) to build situation theoretic representations, but they do not define any situation theoretic concept of inference.

A second class of computational situation theoretic systems are those that use situation theory as their computational base rather than just using aspects of situation theory in a representation formalism within some other theory. The prime example (before the work presented here) is PROSIT ([Nakashima *et al* 88],[Frank & Schütze 90]). PROSIT is designed to be a general knowledge representation/programming language based on situation theory in a similar way that Prolog is based on first order logic. PROSIT offers a representation of situations, facts, parameters and intra-situation constraints. (A detailed description is given in Section 3.7.2.) What makes PROSIT different from the other treatments of situation theory is that it deals with more than simply representation. PROSIT offers a inference mechanism within a situation theory framework. Thus it allows queries to be proved about systems of situations and constraints.

Primarily PROSIT has been used to look at problems of self-reference in knowledge representation rather than its use for representing natural language semantics. The fact that situation theory allows situations as arguments to facts means it is easy to represent self-referential statements. For example suppose we wish to represent a card game where there are two players. Hanako has the 3♡ and Taro has the 5♣. Both players are displaying their cards, so both can see each other's cards and both can see that they can see each other's cards, etc. This infinite regression can be easily modelled by self-reference.

$$
\begin{aligned}
S_1 &\models \ll has, h, 3\heartsuit; 1 \gg \\
S_1 &\models \ll has, t, 5\clubsuit; 1 \gg \\
S_1 &\models \ll see, h, S_1; 1 \gg \\
S_1 &\models \ll see, t, S_1; 1 \gg
\end{aligned}
$$

It is this self-reference and ability to represent other's belief states that is exploited in PROSIT examples such as the description of the "three wise men and the colour of their hats" problem described in [Nakashima *et al* 91].

2.5 A general computational semantic language

Since the development of Montague Grammar a number of new semantic theories have been developed either to augment Montague Grammar itself or as alternate theories to deal with some problem not dealt with in the original definition. There are many such theories but within this thesis we will be looking at only a few: Discourse Representation Theory (DRT) [Kamp 81], Situation Semantics ([Cooper 89, Gawron & Peters 90] and others) and Dynamic Logics ([Groenendijk & Stokhof 91a, Groenendijk & Stokhof 91b]). These theories, as we have seen above, use widely different notations to describe many of the same phenomena. For example a simple sentence like *"a man walks"* might have representations as

Situation Semantics	$S \models \ll man, X; 1 \gg$
	$S \models \ll walk, X; 1 \gg$
Dynamic Logic	$\exists x\ [man(x)] \wedge walk(x)$
DRT	$\begin{array}{\|l\|}\hline \texttt{X} \\ \hline \texttt{man(X)} \\ \texttt{walk(X)} \\ \hline \end{array}$

Even in the case of dynamic logic, the apparent similarity to standard logic is only superficial (as we will see in Chapter 6). Also, even after we look through the different syntactic form of these expressions there still are differences. Note how the dynamic semantics translation has an explicit existential quantifier while the others do not.

The problem with having a number of semantic theories all attempting to describe similar phenomena (especially when their notations are so different) is that treatments of various phenomena may be given in one theory but cannot (at least not obviously) be adopted by others. Also although these theories sometimes purport to deal with the same phenomena they may do so in subtly different ways which are not obvious due to the notations and semantics of the theory. In order to efficiently cross-pollinate ideas and treatments as well as investigate the exact differences it would be useful to have a computational environment in which such semantic theories could be described, implemented and tested.

The idea of a general meta-theory for a number of apparently different theories covering very similar phenomena has already successfully been developed in the field of computational syntax. In the early 80s a number of syntactic theories were developed which although apparently different were offering treatments of similar syntactic phenomena. These included Lexical Functional Grammar (LFG) [Bresnan 82], Generalised Phrase Structure Grammar (GPSG) [Gazdar *et al* 85] and Categorial Grammar [Ades & Steedman 82]. Functional Unification Grammar (FUG) [Kay 84] was the first to try and find a general formalism in which other grammar theories could be described, but PATR-II [Shieber 84] was really the first system in which specific grammar theories were written (as opposed to borrowing ideas from others to form a new theory). Descriptions of GPSG [Shieber 86b] and Categorial Grammar [Uszkoreit 86] in PATR-II helped to determine future grammatical theories in that it allowed them to see what features of these theories are really significant. Even though the descriptions were rarely complete it was useful to identify which aspects of the theories were easy to describe and which were not. HPSG [Pollard & Sag 87] which was developed later has benefited from this comparison. Although it itself has not been described in PATR-II it has been influenced by earlier comparisons of theories in PATR-II. It is easy to see aspects of GPSG and Categorial Grammar within HPSG.

It is perhaps too early in the development of semantic theories to hope for such a well defined "PATR-II for semantics". The field of computational semantics is perhaps not as stable as computational syntax was then. However there are strong analogies between the two fields. Today we have different semantic theories covering similar semantic phenomena in the same way we had ten years ago with computational syntax. And if it is not possible to find such a language then it would be interesting to know why not.

It should be noted that a general language in which other theories can be described is already the subject of a number of pieces of research. Obviously general logic programming languages in some sense offer this. If it is possible to implement a semantic theory at all it can be done in Prolog (and it may even be easy to do so) but of course that is not quite what we are looking for. Prolog is too general and does not constrain itself to features suitable for semantic theories of natural language. In implementations of semantic theories in Prolog it is often difficult to differentiate between parts of the theory and the programming language itself. Something more specific to the task is desired.

Within the field of logic programming there has already been work on defining general systems in which various logics can be defined. Particularly SOCRATES is a system in which logics such as first order, modal, etc. can be abstractly defined and the system will generate theorem provers from these definitions [Jackson *et al* 89].

With a view to finding a general semantic meta-language let us look at some existing frameworks which could offer a framework within which such a meta-language might be defined.

2.5.1 Feature systems

Feature systems (sometimes called attribute-value logics [Johnson 88] or feature logics [Smolka 88]) are often used as a general mechanism for syntactic representation in natural language systems. However they have also been used for semantic representation too. Feature systems in their simplest form allow a representation of sets of features (categories) where each feature can take either an atomic value or a category value. This allows a simple but powerful representation which has been used in many syntactic theories (e.g. GPSG [Gazdar *et al* 85]) and also for simple semantic representation of logical forms (e.g. in [Shieber 86a]). Originally their use was quite informal but much work has been done on formalising the theory of features. Also many enhancements have been added to the basic form as it was found not powerful enough to easily represent many syntactic phenomena (let alone semantic phenomena).

Various extensions to features have been considered. Apart from simple atomic or category-valued features we can now have disjunctive features and set-valued features. Also the specification of feature structures can be made as sets of path equations (possibly including regular expressions) instead of simple attribute-value matrices. For example the following two descriptions represent the same feature structure. They are representations of the sentence *"Hanako seems to sleep"*.

$$\begin{bmatrix} SUBJ & \begin{bmatrix} AGR & \begin{bmatrix} PERS & 3rd \\ NUM & sing \end{bmatrix} \\ PRED & hanako \end{bmatrix}_1 \\ & \\ COMP & \begin{bmatrix} SUBJ & []_1 \\ PRED & sleep \\ TENSE & none \end{bmatrix} \\ & \\ PRED & seem \\ TENSE & pres \end{bmatrix}$$

In path equation form (as used in PATR-II) the above could be written as

<SUBJ AGR PERS> = *3rd* $\wedge$
<SUBJ AGR NUM> = *sing* $\wedge$
<SUBJ PRED> = *hanako* $\wedge$
<COMP SUBJ> = <SUBJ> $\wedge$
<COMP PRED> = *sleep* $\wedge$
<COMP TENSE> = *none* $\wedge$
<PRED> = *sleep* $\wedge$
<TENSE> = *pres*

Also note the cross indexing such that part of the structure is shared between two parts of the feature structure.

At first it was felt that feature structures could only be acyclic but later it was realised that cycles were useful and there were no reasons to exclude them. Later work made more distinction between the syntactic expression of a feature matrix (or equations) and the feature structure it denotes.

Work on the representation of set values for features posed a number of problems. There is in fact a number of ways of interpreting set-valued features. First we can view the values in a *disjunctive* way. That is the value of such a feature is one of a set of values but at this stage it is not known which—this is consistent with the view of a feature structure being an underdetermined description of feature graphs. The second view is to deal with the values of a set-valued feature in a *conjunctive* way. That is the feature value is all of the values in the set. These distinctions are detailed in [Rounds 88]. But it turns out that these distinctions are not enough. Another treatment of set values is possible and indeed useful in linguistic representation. [Pollard & Moshier 90] show a treatment of set values that allows some values to be collapsed into one member which they use in the treatment of SLASH categories (see [Gazdar *et al* 85, Ch 7]). All this shows that there are many treatments possible and the required treatment can be selected as required.

The main computational operation used with feature structures is *unification*. Unification allows the conjunction of two feature descriptions in order to find a description of a new object (or objects) which are described by both descriptions. Unification is well researched but can be a computationally expensive operation, especially when sets and other extensions are admitted, although various relatively efficient implementations have been found (e.g. [Ait-Kaci & Nasr 85]). In addition to unification the use of *constraints* has also been introduced. Originally only simple forms were required by grammatical theories. Feature Cooccurrence Restrictions in GPSG [Gazdar *et al* 85] are simple constraints *within* categories about which features may appear (or not appear) together. Others have considered constraints *between* categories ([Kilbury 87] [Frisch 86]) which are more powerful, but computationally more expensive. Later work [Hegner 91] has proven decidability for constraints restricted to horn clauses.

Head-driven Phrase Structure Grammar (HPSG) [Pollard & Sag 87] is a theory that requires (probably) the richest form of feature systems. Although it is currently not specified in a fully formalised way it probably requires, at least, conjunctive and disjunction features, set values, negation, cycles, and constraints—[King 89] gives a logical formalisation of major parts of the theory. HPSG even requires representations of situations for its semantic forms. With such a rich representation, it is quite possible that aspects of an implementation of HPSG would be undecidable (either constraint satisfaction and/or parsing) however cut down versions do exist (e.g. [Franz 90], [Popowich & Vogel 91]).

Thus with all these various facilities a feature system (given the right choice of options) could offer a rich enough formalism within which semantic representation would probably be possible. However it would require careful selection of the right combination.

2.5.2 Semantic abstraction

Another approach to developing a general semantic meta-theory is semantic abstraction [Johnson & Kay 90]. In semantic abstraction a number of basic operators (called *constructors*) are defined. The operators can be used within some grammar to define the operations necessary in building a semantic representation of an utterance. The important aspect of this method is that depending on the semantic theory desired (hopefully) only the definition of the operators need be redefined. The application of the operators remains the same.

[Johnson & Kay 90] define a (non-exhaustive) set of six basic operators: *external*, *atom*, *conjoin*, *new_index*, *accessible_index* and *compose*. Each syntactic grammar rule is related to some set of basic semantic operations. The evaluation of these define the construction of the semantic translation for that syntactic constituent. Definitions for these operators have been made for predicate logic, discourse representation structures and a simple form of situation semantics.

This method does seem attractive and does seem to work for these simple cases, although it must be said that the given examples are very simple and constructed so that they illustrate the technique but in their present form would not scale up. Semantic abstraction as it is defined in [Johnson & Kay 90] only concerns itself with the construction of semantic forms and not with the semantics interpretation of these forms but this is also true of most treatments of semantics in feature systems. It should not be expected that *all* semantic constructors be used for *all* theories as it is expected that there are some differences between these theories. However it should be the case that a large part of each theory would use the same constructors. Intuitively there does seem to be overlap in the theories. For example *conjoin* might be simple unification in one theory and application in another, but the basic notion of joining two objects exists in both.

What is important in this method is to ensure that the "core" constructors are used in each description. If a non-intersecting set of constructors are used in the description of different theories this method ceases to have any interesting comparative properties and is reduced to a usefulness like a general (though appropriate) programming language. Also although considered, a semantics for the abstract constructors themselves is not given.

2.6 Thesis aims

We have described a number of semantic phenomena and described a number of semantic theories aimed at describing these phenomena. Then we identified a number of possible frameworks in which a general computational description of these theories could be made.

Because situation theory has been proposed not only as a framework for natural language semantics but also as a general all-encompassing theory of information content, it was decided to investigate its use as a general semantic meta-theory in which other semantic theories can be formally specified, implemented and compared. Situation theory seems to offer more power than simple first order logic and because it offers intensional objects, abstract descriptions should be possible. A language based on situation theory is also unlike simple Prolog as it already offers a formal semantics and should restrict its descriptions to aspects of semantics and less to do with implementation. This is not to say that a semantic meta-language could not be achieved in a logic programming language, a feature system or in a framework of semantic abstraction, in fact it may be the case that a situation theoretic language can be defined within these frameworks themselves. However as an initial stage we will try to develop a meta-language based on situation theory but we will return to the wider issues of what the necessary properties of a semantic meta-language are in Chapter 8.

First, it is necessary to define a computational fragment based on situation theory. This is done in Chapter 3 with the definition of the language ASTL. This requires careful selection of various properties of situation theory and the definition of an inference mechanism in order to obtain a usable language. In order to show that ASTL is suitable as a general meta-theory for semantic theories it is necessary to show some detailed examples. To be completely formal we should find full formal definitions of semantic theories and prove equivalence with them and their formalisation within ASTL. This extreme has not been done. Finding full formal specifications of natural language semantic theories is not always easy and even when found that specification may not be the current accepted version. Instead we will encode theories by looking at paradigmatic analyses. This is justified because many natural theories typically concentrate on some specific semantic phenomena, and it is treatments of those phenomena which are important to the theory. However this is not to say that the formalisation of a theory within ASTL is just some arbitrary "program". Descriptions of theories in ASTL are still a formal specification but they are also suitable for execution. Because we are concerned with computational semantic theories is seems reasonable, or even necessary, that formal specifications can be used to show analyses of paradigmatic utterances exhibiting specific semantic phenomena. The formal descriptions presented in the later chapters of this thesis are directly executable via ASTL's implementation and they can used to derive that theory's semantic representation from a given utterance.

Three theories are considered in detail, each is given an executable formalisation of key aspects of the theory. Chapter 4 describes a form of situation semantics called Situation Theoretic Grammar (STG) [Cooper 89] which shows that ASTL is at least suitable for describing "conventional" situation semantics. Chapter 5 shows how Discourse Representation Theory (DRT) [Kamp 81] can be described in ASTL. Not only can Discourse Representation Structures (DRSs) be represented but also a "*construction algorithm*" (the method of generating DRSs from utterances) can be defined within such a situation theoretic language. Third, a description of dynamic semantics is given. A description of Dynamic Predicate Logic (DPL) [Groenendijk & Stokhof 91b] is given in ASTL. This differs from the other descriptions in that DPL is a logic rather than a treatment of natural language. A separate description called DPL-NL shows how DPL can be related to natural language utterances and offer a dynamic logic treatment of them.

The STG description is given really as a basic building block, showing how both syntactic and semantic processing may be done in ASTL. The later two descriptions, DRT and dynamic semantics, are given in order to allow specific comparisons between them. Both these later theories deal with some specific problems in semantics and therefore it seems justifiable (and interesting) to compare them closely. That such comparisons are possible (and easy) shows one of the advantages of a general meta-theory for semantic theories. That these descriptions are not just static descriptions but can actually be run in an implementation of ASTL also allows comparisons to be made about their computability and suitability in practical natural language processing systems. Examples of the descriptions are given throughout the chapters but detailed examples are also given in Appendix A.

At this stage is is worth noting that although ASTL is designed as a general language, any such language will impose certain restrictions on the descriptions encoded within it. Some of these restrictions are arbitrary and just factors which are necessary when dealing detailed formalisations. Other restrictions are directly to do with the underlying aspects of ASTL and situation theory (the framework ASTL is in) and are worth noting. At this stage, before ASTL is presented, we will not discuss examples of this but will return to this issue in the final chapter.

2.7 Summary

In this chapter we have identified a number of semantic phenomena currently under investigation in the area of formal and computational semantics. These lie primarily in the area of quantification and anaphora. A number of contemporary semantic theories are briefly described. The general idea of a computational mechanism in which these semantic theories can be described and run is introduced and a number of possible areas where such a mechanism might be found are described (logic programming, feature structures and semantic abstraction). Finally the basic aim of the thesis is proposed. That is to define a computational situation theoretic language which is adequate to describe formal (and executable) specifications of other semantic theories and illustrate this by describing a number of theories in it.

Chapter 3

A Computational Situation Theoretic Language

3.1 Introduction

In this chapter we will formally define a computational language in which a number of semantic theories of natural language can be defined. This language is called ASTL. ASTL relies heavily on basic aspects of situation theory[1]. This language is not simply an abstract theoretic one but is designed specifically to be run on a computer. Formal specifications of natural language semantic theories can be written in ASTL and run on a conventional computer. An implementation of ASTL exists and results from descriptions in ASTL will be shown throughout this thesis. The basic intended use of ASTL is that aspects of semantic theories are specified in ASTL such that it is possible to at least derive the semantic representation for utterances with respect to that theory. Although we do not spend much time discussing the interpretation and inference based on the resulting semantic representations, ASTL does seem suitable for further investigation in that area.

The suitability of ASTL as a language for describing natural language semantic theories relies on the fact that it exploits some fundamental aspects of situation theory. The concept of the *situation* and its status as a first class object which can be used as an argument to arbitrary relations allows ASTL to offer a high level of structure in its representation of objects. Secondly, ASTL also exploits situation theory's mechanisms for representing parameters and anchoring. This allows ASTL a method for describing variables and assignments. It is this level of description which normally only exists in a meta-language used to described semantic theories that makes ASTL a suitable tool in the formal specification and implementation of a number of natural language semantic theories.

However, a language which only offers a method for representation of semantic objects

[1]Note that the name "ASTL" is not intended to be an acronym, although a number have been suggested.

is not powerful enough in itself to allow computation. As well as a representation for, individuals, parameters, relations, facts and situations, ASTL also offers a representation of *constraints*. Constraints allow generalisations between situations to be described. Finally in order for computation to be possible ASTL also includes a definition of inference with respect to basic situations and constraints.

ASTL is not the first computational language to be based on situation theory, but it is probably the first to be specifically designed for processing natural language utterances. PROSIT ([Nakashima *et al* 88], [Frank & Schütze 90]) is another example of a language based on aspects of situation theory. (PROSIT is described in detail in Section 3.7.2.) In the work presented here we are primarily concerned with language processing and representing semantic translations, and in its current form PROSIT does not include an easy way to deal with grammar and language processing. Rather than extend PROSIT to include such mechanisms it was felt better to define a new language from the start which would offer only the facilities that appear necessary for a semantic meta-language. ASTL is the result.

3.2 ASTL—a situation theoretic language

In some uses of situation semantics the language in which the examples are given is not fully defined. The reader is expected to build up an idea of the language just based on the given examples. Equally so in AI, specialised programming languages are also often poorly defined, specified only by their syntax with no formal (or often even informal) specification of their semantics. To try to counter that, here we will give both the formal syntax and semantics of ASTL. As we are dealing with a computational language which has an implementation there will be times when the abstract definition differs from the actual operational semantics—these occasions are indicated in the text and justification for the difference is given.

Here we continue with the idea used in model theoretic semantics for conventional logics where the denotation of expressions in a language are objects in a model. However here our model consists of more complex semantic objects, such as facts, types and situations, where in the case of simple first order predicate logic the semantic objects are simpler.

The language ASTL is fairly conservative in its use of situation theoretic objects, and in fact fairly simple. Rather than define a complicated language at this stage we will start simply. Extensions are discussed later but we will see that even this simple form is sufficient for the basic aspects of the semantic theories we are interested in.

The following two sections on the syntax and semantics of ASTL are rather formal and perhaps difficult to read. It is necessary to formally define ASTL before we can discuss it in any detail. However it is not necessary to follow these sections closely at this stage. They may be skimmed and later referred to when it is necessary to understand the semantics in more detail. Section 3.4 gives a full example of a description in the language and shows how it can actually be used. The basic ideas of ASTL and its use can be understood from that section.

3.2.1 Syntax of ASTL

This section describes the syntax of terms and sentences in ASTL. Unlike many AI programming languages which use typographical conventions (e.g. upper case letters to identify variables) or context to distinguish types of symbol, ASTL requires its symbols to be declared before their use. However for ease of reading ASTL expressions some typographical conventions will be used.

Terms in ASTL fall into two classes:

atomic: individuals, relations, parameters and variables.

complex: i-terms, types, and situations.

Although there are no built in naming conventions for atomic terms we will use the following conventions:

individuals: lower case letters (i.e. `a`, `b`, `c`, ...).

relations: lower case words (i.e. `walk`, `man`, etc.).

parameters: upper case letters (i.e. `A`, `B`, `C`, ...).

variables: upper case letters preceded by an asterisk (i.e. `*A`, `*B`, `*C`, ...).

The syntax of complex terms are

if rel is a relation of arity n, and $arg_1, \ldots, arg_n$ are terms and the polarity p is 0 or 1 then `<<`$rel, arg_1, \ldots, arg_n, p$`>>` is an **i-term**.

if Par is a parameter and $i_1, \ldots, i_n$ are i-terms then

`[`Par `!` Par `!=` i_1
...
Par `!=` i_n `]`

is a **situation type**. (Later we will refer to the sub-parts of a type of the form Par `!=` i as *conditions*—although conditions are not terms.) Also if α and β are situation types α & β is also a **situation type**.

if σ is a situation name and τ is a type then σ is a **situation term** and σ`::`τ is a **situation term**.

A few comments seem relevant at this stage. Currently there is no syntactic specification of *appropriateness* for arguments to a relation, although such a restriction could be considered. Here we allow any term to be an argument to a relation and only state that the number of arguments must be equal to the declared arity of the relation. A second comment is about types. These are limited to situation types, although a more general type system is described in terms of a generalised abstraction extension details of which are given in Section 7.4.1.

Sentences in ASTL have the following syntax

If σ is a situation name and τ a situation type then σ:τ. is a **proposition**.

If $\sigma_0, \sigma_1, \ldots, \sigma_n$ are situation names and $\tau_0, \tau_1, \ldots, \tau_n$ are situation types then
σ_0:τ_0<= σ_1:$\tau_1, \ldots, \sigma_n$:τ_n. is a **constraint**.

Note that these sentences (propositions and constraints) are not terms and cannot be used as arguments to relations. In Chapter 4 we will add to this a number of sentences which are convenient abbreviations for propositions and constraints that make the specification of natural language processing easier but as they can all be defined in terms of the propositions and constraints, the above represents a proper core of the language.

Now that we have given the complete syntax for all the terms and sentences in ASTL we can give the syntax for an ASTL *description*. A *description* is a set of declarations and sentences which describe a system of situations and constraints—in some ways this can be thought of as a program in the ASTL language. Optional sections are contained within `[...]`.

`Individuals`	$\{i_1, \ldots, i_n\}$
`Relations`	$\{rel_1/arity, \ldots, rel_n/arity\}$
`Parameters`	$\{p_1, \ldots, p_n\}$
`Variables`	$\{v_1, \ldots, v_n\}$
`Situations`	(<*situation term*>
	...
	<*situation term*>)
`[ Constraints`	<*constraint*>
	...
	<*constraint*>]

A full example ASTL description would look like

```
Individuals   {h,t}
Relations     {happy/1, smiles/1}
Parameters    {S}
Variables     {*S, *Y}
Situations
```

```
        ;; Basic situations -- i.e. who smiles where
        (SIT1 :: [S ! S != <<smiles,h,1>>
                      S != <<smiles,t,1>>]
         SIT2 :: [S ! S != <<smiles,t,1>>] )
    Constraints
       ;; if they smile they are happy
       *S : [S ! S != <<happy,*Y,1>>]
       <=
          *S : [S ! S != <<smiles,*Y,1>>].
```

Situation declarations really serve a dual purpose. They define what the basic situations are in a description and also assert that these situation are of the specified type. That is, when we declare a situation

```
SIT1 :: [S ! S != <<smiles,h,1>>
             S != <<smiles,t,1>>]
```

we are also asserting the proposition

```
SIT1 : [S ! S != <<smiles,h,1>>
            S != <<smiles,t,1>>]
```

Comments may be included in descriptions in a Lisp comment style: characters from a semi-colon to the end of a line are ignored.

3.2.2 Semantics of ASTL

This section describes the semantics of ASTL in terms of a model. A model M for an ASTL description consists of the following

I a set of individuals.

R a set of relations.

P a set of parameters.

S a set of situations.

T a set of types.

F a set of facts.

Supports a set of pairs of situations and facts[2] constructed from the set $\mathbf{S} \times \mathbf{F}$.

[2]In other definitions of situation theory it would be normal to restrict facts to those that do not "contain" parameters. This has deliberately not been done here.

Relation_of a function that given a fact will return its relation.

Argument_of a function that given a fact f and an integer i will return the member of **I**, **R**, **P**, **S**, **T** or **F**, which is the ith argument of f.

Polarity_of a function that given a fact will return a 0 or 1, the polarity of the fact.

Facts_of a function that given a type will return a subset of **F**.

Param_of a function that given a type will return a member of **P**.

Func a function assigning, names, relation names, parameters names and situation names, to members of **I**, **R**, **P** and **S** respectively.

It is possible to construct the members of **F** (the facts) and **T** (the situation types) from the sets **I**, **R**, **P**, **S**, and **T** and **F** themselves but this has not been done here. We also could have a notion of *appropriateness* for arguments to facts. In this definition we will only enforce the number of arguments of a member of **F** to be the same as the arity declared for its relation. *Appropriateness* is left as a matter for each description within ASTL rather than the language itself.

The semantic values for expressions in ASTL can be described with respect to a model M and a variable assignment function g.

The semantic values of basic terms are

If u is a variable then $[\![\ u\ ]\!]^{M,g} = g(u)$.

If ι is an individual name, relation name, parameter name, situation name then $[\![\ \iota\ ]\!]^{M,g} = Func(\iota)$.

If χ is 0 or 1 then $[\![\ \chi\ ]\!]^{M,g} = 0$ or 1 respectively.

The semantic values for complex terms are

If ϕ is an i-term `<<`ρ, *args*, *polarity* `>>` then $[\![\ \phi\ ]\!]^{M,g}$ is a fact f, a member of **F** where $[\![\ \rho\ ]\!]^{M,g} = Relation\_of(f)$, for each argument in $args$ $\alpha_1, ..., \alpha_n$ for $i = 1$ to n, $[\![\ \alpha_i\ ]\!]^{M,g} = Argument\_of(f,i)$, and $[\![\ polarity\ ]\!]^{M,g} = Polarity\_of(f)$. If any argument $[\![\ \alpha_i\ ]\!]^{M,g}$ is undefined then ϕ is undefined.

If τ is a (situation) type of the form `[` π `|` π `|=` ϕ_1 ... π `|=` ϕ_n `]` then $[\![\ \tau\ ]\!]^{M,g} = t$, where $t \in \mathbf{T}$ such that $\{[\![\ \phi_1\ ]\!]^{M,g}, \ldots, [\![\ \phi_n\ ]\!]^{M,g}\}$ is equal[3] to $Facts\_of(t)$ $\{Param\_of(t)\ /\ [\![\ \pi\ ]\!]^{M,g}\}$, where $Facts\_of(t)$ $\{Param\_of(t)\ /\ [\![\ \pi\ ]\!]^{M,g}\}$ is $Facts\_of(t)$ except that all occurences of $Param\_of(t)$ are substituted with $[\![\ \pi\ ]\!]^{M,g}$. If $\tau_1 \& \tau_2$ is a (situation) type then $[\![\ \tau_1 \& \tau_2\ ]\!]^{M,g} \equiv [\![\ \tau_3\ ]\!]^{M,g}$ such that τ_3 is the conjunction of conditions in τ_1 and $\tau 2$.

[3] The sets need to be *equal* rather than subset to ensure there is a unique t for each type. We use sets rather than lists so that the order of the conditions is not significant.

If $\sigma :: \tau$ is a typed situation term then $[\![\sigma :: \tau]\!]^{M,g} = [\![\sigma]\!]^{M,g}$ if $[\![\sigma]\!]^{M,g}$ is of type $[\![\tau]\!]^{M,g}$ otherwise it is undefined. $[\![\sigma]\!]^{M,g}$ is of type $[\![\tau]\!]^{M,g}$ iff where $[\![\sigma]\!]^{M,g} = s$ and $[\![\tau]\!]^{M,g} = t$, for each fact f_i in $Facts\_of(t)$, $< s, f_i\{Param\_of(t)/s\} > \in$ **Supports** for each i, where $f_i\{Param\_of(t)/s\}$ is f_i except that all occurrences of $Param\_of(t)$ in f_i are substituted with s.

The semantic values of sentences in ASTL are

If π is a proposition $\sigma : \tau$ then $[\![\pi]\!]^{M,g}$ is true if $[\![\sigma]\!]^{M,g}$ is of type $[\![\tau]\!]^{M,g}$. (The definition of a situation being of a type is defined above in the semantics of typed situations.) If any term in π is undefined π is false.

If κ is a constraint of the form $\sigma_0 : \tau_0$ `<=` $\sigma_1 : \tau_1, ..., \sigma_n : \tau_n$ and V_R is the set of all variables in the right hand side of κ and V_L is the set of all variables in κ minus V_R then $[\![\kappa]\!]^{M,g}$ is true if for all g' such that g' is exactly the same as g except possibly in the values it assigns to the variables in V_R and such that $[\![\sigma_1 : \tau_1]\!]^{M,g'}$, ...,$[\![\sigma_n : \tau_n]\!]^{M,g'}$ are true, then there exists g'' such that g'' is exactly the same as g' except possibly in the values it assigns to the variables in V_L such that $[\![\sigma_0 : \tau_0]\!]^{M,g''}$ is true.

Some comment on this may be useful. In this framework situations may support parametric facts. This perhaps is a little unusual in that most versions of situation theory would not allow this. This is allowed here in order not to complicate the language—such restrictions are left to descriptions in ASTL. Also there is no special treatment of parameters built in to ASTL—except in their use as in identifying the "abstracted" situation in a situation type. Parameters are just another simple class of atomic individuals. The use of parameters as "variables" is possible within a user's description but the notion is not built into the language itself.

Another aspect which is not covered by the semantics but deserves some comment is *coherence*. Specifically, the above definition does not make any restrictions on situations which support conflicting facts. For example a situation of the following type

```
SIT1 :: [S ! S != <<happy,h,1>>
             S != <<happy,h,0>>].
```

would be inconsistent within some definitions of situation theory but it is acceptable in ASTL. Currently coherence is a notion which can only be specified within an ASTL description and is not defined within the model. It is true that the issue of coherence is not yet dealt with in ASTL but future extensions may have some notion of coherence built in.

A little further discussion about free variables will help clarify the semantics. Free variables in propositions or the left hand side of constraints are effectively skolemized. That is they are given an arbitrary constant as a value, hence we are treating free variables as being existentially quantified. Queries however are treated slightly differently. Although we have not given a formal semantics for queries, they are important to the operational semantics of any implementation of ASTL. Free variables in queries in the implementation discussed later in this chapter are given a universal treatment. When a query is asked we want to find all ways that it can be made true, taking the analogy of Prolog.

A further point that needs some mention is that in the version of ASTL defined above, propositions and constraints are not first class objects. They could be but are not due to the complexities this would introduce into any simple implementation, and it was felt that it was not necessary for the examples given in later chapters.

3.2.3 Inference in ASTL

A static definition of the language ASTL is not sufficient to allow it to be used in computation. We would like to compute what the consequences of a given set of constraints and propositions are. The following set of inference rules are defined to achieve this. Each rule is described as an equation, which should be read as, if we have the sentences above the line we can conclude (prove) those on the bottom. Greek characters are used as variables over expressions in the language (rather than variables in the language).

Constraints containing (ASTL) variables can be viewed as shorthand for a number of fully explicit non-variable constraints. The following definitions are given with respect to fully expanded constraints so that variable binding need not be specified in the rules.

Type reduction

A type with more than one *condition* may be broken down.

$$\frac{\sigma : [\ \pi\ !\ c_1, \ldots, c_n]}{\sigma : [\ \pi\ !\ c_1], \ldots, \sigma : [\ \pi\ !\ c_n]}$$

For example if we have a situation

```
SIT1:[S ! S != <<happy,h,1>>
          S != <<sings,h,1>>].
```

We can conclude that `Sit1` is *also* of the following two types.

```
SIT1:[S ! S != <<happy,h,1>>].
SIT1:[S ! S != <<sings,h,1>>].
```

Type combination

This rule is the reverse of type reduction above. Single condition propositions of the same situation can be combined.

$$\frac{\sigma\ :[\ \pi_1\ !\ c_1],\ \ldots,\ \sigma\ :[\ \pi_n\ !\ c_n]}{\sigma\ :[\ \pi_x\ !\ c_1, \ldots, c_n]\ \ \{\ \pi_1, \ldots, \pi_n/\pi_x\ \}}$$

That is the parameters of the individual types are replaced with a new parameter used in the new combined type.

This rule and the previous one effectively define a subsumption relation between types. Types can be broken down and built up such that any "subset" of conditions (in any order) of a situation's type is still a type of that situation. For example if the following proposition is true

```
SIT1:[S ! S != <<happy,h,1>>
          S != <<sings,h,1>>
          S != <<dances,h,1>>].
```

the following are also true

```
SIT1:[S ! S != <<happy,h,1>>].
SIT1:[S ! S != <<dances,h,1>>
          S != <<sings,h,1>>].
```

As well as all other combinations and orders of conditions. Note that the result of these two inference rules is that the order of conditions in types is semantically irrelevant.

Modus ponens

We have two cases, one when there is only one type on the right hand side of a constraint and the other when there is more than one

$$\frac{\begin{array}{l}\sigma_0 : \tau_0 \texttt{ <= } \sigma_1 : \tau_1 \\ \sigma_1 : \tau_1\end{array}}{\sigma_0 : \tau_0} \qquad \frac{\begin{array}{l}\sigma_0 : \tau_0 \texttt{ <= } \sigma_1 : \tau_1,\ \ldots,\ \sigma_{i-1} : \tau_{i-1},\ \sigma_i : \tau_i,\ \sigma_{i+1} : \tau_{i+1}, \ldots,\ \sigma_n : \tau_n \\ \sigma_i : \tau_i\end{array}}{\sigma_0 : \tau_0 \texttt{ <= } \sigma_1 : \tau_1,\ \ldots\ \sigma_{i-1} : \tau_{i-1},\ \sigma_{i+1} : \tau_{i+1}, \ldots,\ \sigma_n : \tau_n}$$

For example if we have

```
SIT1:[S ! S != <<happy,h,1>>]
  <= SIT2:[S ! S != <<sings,h,1>>].
SIT2:[S ! S != <<sings,h,1>>].
```

we can deduce

```
Sit1:[S ! S != <<happy,h,1>>]
```

It should be emphasized at this point that there is a distinction between ASTL the language and ASTL the implementation. Although the implementation attempts to come as close as possible to the formal definition it does take some short cuts for the sake of efficiency. In particular the second part of the above definition allows the subparts of the right hand side to be eliminated in any order. In the actual implementation the rule is

$$
\begin{array}{l}
\sigma_0 : \tau_0 \ \texttt{<=}\ \sigma_1 : \tau_1,\ \sigma_2 : \tau_2, \ldots, \sigma_n : \tau_n \\
\sigma_1 : \tau_1 \\
\hline
\sigma_0 : \tau_0 \ \texttt{<=}\ \sigma_2 : \tau_2, \ldots, \sigma_n : \tau_n
\end{array}
$$

That is the clauses on the right hand side need to be proved in order. The effect of this is that there may exist some sentences of the form $\sigma_0 : \tau_0 \ \texttt{<=}\ \sigma_1 : \tau_1, \ldots, \sigma_n : \tau_n$ which are true but cannot be proved because of the ordering of clauses in the right hand side. This is not seen as a problem as the possible goals in the implementation are propositions rather than constraints.

Argument promotion

This is a rather unusual rule to allow a treatment of (typed) situations as arguments in facts.

$$
\begin{array}{l}
\sigma_0 : \tau_0 \ \texttt{<=}\ \sigma_1 : \texttt{[}\pi\ \texttt{!}\ \pi\ \texttt{!=}\ \texttt{<<}rel, arg_{0..i},\ \sigma_2 :: \tau_2\ , arg_{i+2..n}, pol\texttt{>>]} \\
\sigma_1 : \texttt{[}\pi\ \texttt{!}\ \pi\ \texttt{!=}\ \texttt{<<}rel, arg_{0..i},\ \sigma_2\ , arg_{i+2..n}, pol\texttt{>>]} \\
\hline
\sigma_0 : \tau_0 \ \texttt{<=}\ \sigma_2 : \tau_2
\end{array}
$$

That is the argument $\sigma_2 :: \tau_2$ is *promoted* from an argument to a clause. The informal motivation is to move conditions about situations out of arguments and into the top level of a constraint. For example from

```
SIT1:[S ! S != <<sees,h,SIT2,1>>].
SIT0:[S ! S != <<happy,h,1>>]
<=
  SIT1:[S ! S != <<sees,h,
                      SIT2::[S ! S != <<happy,t,1>>],
                      1>>].
```

we can deduce

```
SIT0:[S ! S != <<happy,h,1>>]
<=
   SIT2:[S ! S != <<happy,t,1>>]
```

As with the modus ponens rule above there is also a rule to deal with multiple clauses on the right hand side

$$\frac{\begin{array}{l}\sigma_0 : \tau_0 \texttt{ <= } \sigma_1 : \tau_1,\ \ldots,\ \sigma_{i-1} : \tau_{i-1}, \\ \qquad\qquad \sigma_i : \texttt{[}\pi \texttt{ ! } \pi \texttt{ != <<}rel, arg_{0..i},\ \sigma_2 :: \tau_2\ , arg_{i+2..n}, pol\texttt{>>]} \\ \qquad\qquad \sigma_{i+1} : \tau_{i+1}, \ldots, \sigma_n : \tau_n \\ \sigma_i : \texttt{[}\pi \texttt{ ! } \pi \texttt{ != <<}rel, arg_{0..i},\ \sigma_2\ , arg_{i+2..n}, pol\texttt{>>]}\end{array}}{\sigma_0 : \tau_0 \texttt{ <= } \sigma_1 : \tau_1,\ \ldots,\ \sigma_{i-1} : \tau_{i-1},\ \sigma_{i+1} : \tau_{i+1}, \ldots,\ \sigma_n : \tau_n,}$$

Again, as with the modus ponens rule, the implementation differs from the formal specification in that the clauses are proved in order, and hence some constraint sentences may not be provable.

Comment

It is important that cyclic structures be treated properly and not cause the inference system to go into an infinite loop. The consequences of the above definition of ASTL and its inferences are that such loops will not occur. Because we use names to refer to situations direct loops do not occur in expressions.

A simple example proof shows the use of the above inference rules. Suppose we have the following (rather specific) constraint and basic proposition.

```
SIT1:[S ! S != <<happy,h,1>>]            (C1)
<=
   SIT1:[S ! S != <<see,h,*S,1>>
           S != <<happy,t,1>>].

SIT1:[S ! S != <<happy,t,1>>            (P1)
          S != <<see,h,SIT1,1>>].
```

A proof for the proposition

```
SIT1:[S ! S != <<happy,h,1>>
          S != <<happy,t,1>>].
```

would include the following steps

```
                          // by type reduction from (P1)
SIT1:[S ! S != <<happy,t,1>>].         (P2)
SIT1:[S ! S != <<see,h,SIT1,1>>].      (P3)
                          // by type combination from (P3) and (P2)
SIT1:[S ! S != <<see,h,SIT1>>          (P4)
          S != <<happy,t,1>>].
                          // by modus ponens from (C1) and (P4)
SIT1:[S ! S != <<happy,h,1>>].         (P5)
                          // by type combination from (P5) and (P2)
SIT1:[S ! S != <<happy,h,1>>
          S != <<happy,t,1>>].
                                       QED
```

3.3 Extended Kamp Notation

The basic notation for ASTL is not always easy to read, especially when an expression includes embedded situations. In other treatments in situation theory (e.g. in [Gawron & Peters 90]) expressions are also difficult to read because much use is made of sub-scripting to add restrictions to objects, as in

$$\ll [x_{subj}, y | \ll EATING, x, y_{(s \models \ll BISCUIT, y \gg)} \gg], x; 0 \gg$$

This is used to represent a state of affairs where x is eating a biscuit. Wishing to make ASTL more readable, an alternative formalism has been developed. A graphical notation for situation theoretic objects is described in [Barwise & Cooper 93]. *Extended Kamp Notation* (EKN) as it is called, owes something to the use of boxes in DRT. Within the implementation of ASTL this graphical notation is available as an output mechanism. It would be useful if EKN were also available as an input mechanism but that would require significant work on building some form of graphical editor which was not felt worthwhile at this stage in the development of the language.

EKN is in fact a rich notation for representing many different situation theoretic objects including situations, abstractions, anchoring environments, etc. many of which are not part of basic ASTL. In fact, in ASTL's box notation, there are only two forms which are displayed as boxes. First is the form for situations, which actually are not given their own notation in EKN as such. In ASTL, when the EKN output option is selected, situations are displayed as boxes with double lines at the top and bottom, the situation name appears in an inset box at the top left and facts that it supports are displayed in a more convention predicate form. For example the situation

```
SIT56::[S ! S != <<happy,h,1>>
            S != <<happy,t,0>>]
```

would be displayed in ASTL's box notation as

```
SIT56

 happy(h)
 ¬happy(t)
```

In EKN, as defined in [Barwise & Cooper 93] the above would be as a situation with a particular restriction. In their notation the above situation could be written as

```
          SIT56
              S
          S

 SIT56     happy(h)
           ¬happy(t)
```

The ASTL notation can be viewed as an abbreviation for the above. The second form which will be displayed as a box is situation types. Here the form is the same as that used in EKN. If we have the situation type

```
[P1 ! P1 != <<happy,h,1>>
      P1 != <<happy,t,1>>]
```

when the EKN output option is selected the above would appear as

```
   P1
P1

 happy(h)
 happy(t)
```

The ASTL implementation prints the boxes using ASCII characters (`|`, `-` and `=`) but here we will use special macros to display the boxes. The usefulness of a graphical notation becomes much more apparent when facts include situations as arguments. For example

```
SIT1 :: [S ! S != <<happy,h,1>>
             S != <<happy,t,0>>
             S != <<sees,h,SIT2 :: [P ! P != <<happy,t,1>>
                                        P != <<happy,h,1>>],
                            1>>
             S != <<sees,h,SIT1,1>>]
```

can also be displayed as

```
 SIT1

  happy(h)
  ¬happy(t)
           SIT2

  sees(h,   happy(t)    )
            happy(h)

  sees(h,SIT1)
```

However although EKN expressions are easier to read than the simple linear form of ASTL expressions it must be added that expressions in descriptions can easily become so complex that even the EKN notation is not really a help. This is especially true of representations of syntactic and semantic translations for utterances. In that case other techniques must be used to make the output readable—the implementation allows the user to specify that facts with certain relations should not be displayed when a situation is printed.

3.4 Simple example

It is not always easy to understand the practical use of a formal system simply from its formal specification. Here we will give a short example. The example shows what a full ASTL description looks like and what sort of computations are possible. This example only uses simple propositions and one constraint, but is sufficient to illustrate the basic capabilities of the system.

```
Individuals   {h,t}
Relations     {happy/1, smiles/1}
Parameters    {S}
Variables     {*S, *Y}
Situations
    ;; Basic situations -- i.e. who smiles where
    (SIT1 :: [S ! S != <<smiles,h,1>>
                  S != <<smiles,t,1>>]
     SIT2 :: [S ! S != <<smiles,t,1>>] )
Constraints
   ;; if they smile they are happy
   *S : [S ! S != <<happy,*Y,1>>]
   <=
      *S : [S ! S != <<smiles,*Y,1>>].
```

That is we have defined two basic situations (SIT1 and SIT2). Both Hanako (h) and Taro (t) smile in SIT1 but only Taro smiles in SIT2. Also we have a constraint that states that if some object in some situation smiles then that object is also happy in that situation.

We can load the above description, then at the top level we can ask to prove propositions about the system of situations and constraints. Thus if we ask the query

```
astl> query SIT2 : [S ! S != <<happy,t,1>>].
```

rather than just a simple "yes" or "unknown" answer the implementation prints out any situations for which this proposition is true. In this case the only possible situation concerned is SIT2. This proposition is true because the constraint is appropriate and hence Taro is happy because he smiles in SIT2, so the system prints

```
SIT2
  happy(t)
  smiles(t)
```

If we ask the query

```
astl> query *S : [S ! S != <<happy,t,1>>].
```

This time we are asking about *any* situation where Taro is happy. In this case this is true for both SIT1 and SIT2. Thus the result would be.

```
SIT1
  happy(t)
  smiles(t)
```

```
SIT2
  happy(t)
  smiles(t)
```

Notice that in SIT1 only the facts which were used in the proof are printed out. SIT1 also supports the fact that Hanako smiles, and by way of the constraint that she is happy too. That is when a situation is printed not all facts that it supports are included, only those that are directly to do with the proving of the stated goal. However an option is available in the implementation to allow all currently known facts in a situation to be printed. It is important to realise that finding *all* facts supported by a situation may be undecidable, or at least inefficient. The system is goal directed and although it may find other facts not directly related to the actual goal it does not attempt to find all true propositions. This allows the system to still produce results even when some aspects of the described system may be undecidable. This issue is returned to in Section 3.6 which discusses the implementation method used.

A second short example shows the cyclic use of situations (this example was also discussed above in Section 2.4.3). Suppose we have a position in a card game where there are two players (Hanako and Taro) and each are holding one card and both players can see that they are holding their cards and what the other player is holding, and that they can both see that they can see this. We can represent this as an ASTL description

```
Individuals      {h,t,5c,3h}
Relations        {has/2, sees/2}
Parameters       {S,P,R,Q}
Variables        {*S, *Y, *T, *U, *V}
Situations
    (SIT1 :: [S ! S != <<sees,h,SIT1,1>>
                  S != <<sees,t,SIT1,1>>
                  S != <<has,t,5c,1>>
                  S != <<has,h,3h,1>>]  )
```

We can then ask "cyclic" queries such as, does a situation support the fact that Hanako can see a situation in which she has the three of diamonds

```
query *S : [S ! S != <<sees,h,*T :: [P ! P != <<has,h,3h,1>>],1>>].
```

This gives a result of

```
SIT1

 sees(h,SIT1)
 sees(t,SIT1)
 has(t,5c)
 has(h,3h)
```

Notice that although `SIT1` appears as an argument to a fact only its name is printed and not its full specification. The full situation with its known facts is only printed once. All later occurrences of it in a display appear as names. This stops the printing mechanism from going into a loop.

We can continue our querying of the above situation with a query about a situation in which Hanako can see a situation in which Hanako can see a situation in which Hanako has the three of hearts.

```
query *S : [S ! S != <<sees,h,*T::[P ! P != <<sees,h,
                      *U::[R ! R != <<has,h,3h,1>>],1>>],1>>].
```

And we of course can continue deeper

```
query *S:[S ! S != <<sees,h,*T :: [P ! P != <<sees,h,
                    *U::[R ! R != <<sees,h,*V::
                        [Q ! Q != <<has,h,3h,1>>],1>>],1>>],1>>].
```

This shows how such self-referential situations can be represented in ASTL and how an infinite number of queries can be proved about them.

3.5 Some formal properties

After that short interlude on the actual use of ASTL let us return to the formal properties of the language. In this section we will discuss a soundness proof for ASTL and discuss ASTL's computational complexity.

3.5.1 Soundness of ASTL

In order to show that all propositions provable from a set of axioms using the inference rules given above in Section 3.2.3 are in fact true with respect to the semantics of ASTL we give the following *soundness* proof.

ASTL has four inference rules:

(1) Type reduction
(2) Type combination
(3) Modus ponens
(4) Argument promotion

We will deal with each inference rule in turn.

Type reduction states that if a proposition consisting of some situation and a complex type is true the propositions for that situation with a simple type for each condition in the complex type are also true. For example if

σ : [π ! $c_1 \ldots c_n$]

is true the rule states that for each condition

σ : [π ! c_i]

is also true. This can be proved from the definition of the semantics of propositions and types. The proposition $\sigma\ :\ \tau$ is true if $[\![\ \sigma\ ]\!]^{M,g}$ is of type $[\![\ \tau\ ]\!]^{M,g}$, while each reduced form of the proposition σ : `[` π `|` π `!=` ϕ_i `]` will be true if $[\![\ \sigma\ ]\!]^{M,g}$ is of type $[\![\ \tau_i\ ]\!]^{M,g}$. These will both be true in all models because for $[\![\ \sigma\ ]\!]^{M,g}$ to be of type $[\![\ \tau\ ]\!]^{M,g}$ it is necessary, by definition, that for each condition in τ, `[` π `|` π `|=` ϕ_1 ... π `|=` ϕ_1 `]`, $<[\![\ \sigma\ ]\!]^{M,g}, [\![\ \phi_i\{\pi/\sigma\}\ ]\!]^{M,g}> \in$ **Supports**, where $\phi_i\{\pi/\sigma\}$ is ϕ_i except that all occurrences of π in ϕ_i are substituted with σ. Therefore in any model where $\sigma : \tau$ is true, the reduced propositions σ : `[` π `|` π `|=` ϕ_i `]` for each condition in τ must also be true. Hence type reduction is sound.

Type combination states that propositions about the same situation may have their types combined to produce new propositions. Again by the definition of semantics of propositions, types are ultimately broken done into individual conditions. Therefore a combined type made from conditions from true propositions about the same situation will also be true. Therefore type combination is also sound.

Modus ponens, in its simplest form, states that given a constraint of the form $\sigma_0 : \tau_0$ `<=` $\sigma_1 : \tau_1$ and a proposition $\sigma_1 : \tau_1$ we can infer $\sigma_0 : \tau_0$. The definition of the semantics of a constraint states that for a constraint $\sigma_0 : \tau_0$ `<=` $\sigma_1 : \tau_1$ be true when $\sigma_1 : \tau_1$ is true $\sigma_0 : \tau_0$ is true also. Therefore by definition if a constraint $\sigma_0 : \tau_0$ `<=` $\sigma_1 : \tau_1$ and a proposition $\sigma_1 : \tau_1$ is true then $\sigma_0 : \tau_0$ must also be true. This argument likewise applies to constraints with more than one proposition on their right hand side. Therefore modus ponens is sound.

Finally, *argument promotion* states that if a fact in a proposition has a typed situation as an argument then if that proposition can be proved with the argument as a situation and the proposition formed from the typed situation can also be proved then the original proposition is true. In order for a proposition of the form

σ_1 `:[P ! P != <<`$rel, \sigma_2 :: \tau, 1$`>> ]`

to be true by definition the argument $\sigma_2 :: \tau$ denotes what σ_2 denotes iff the proposition $\sigma_2 : \tau$ is true. Therefore for the above proposition to be true it is necessary for the following propositions to be true

σ_1 `:[P ! P != <<`$rel, \sigma_2, 1$`>> ]`
$\sigma_2 : \tau$

This is exactly what the *argument promotion* inference rule states therefore it is sound. The above shows briefly how the four inference rules of ASTL are sound showing that all inferences possible for a set of axioms using these will be true with respect to the given semantics.

Proving *completeness* of ASTL is a lot harder. Completeness, for ASTL, would mean that every proposition which logically follows from a basic ASTL description can be derived by some application of the inference rules given above. Although ASTL is probably in this sense complete, we will not give a formal proof.

3.5.2 Computational complexity

The actual computational complexity of the ASTL system is difficult to discuss in isolation from the algorithms used in the actual implementation although perhaps some definition of the theoretically "best" treatment might be made. In this section we will outline a method for encoding arbitrary Turing machines in ASTL hence implying that the system is, in general, undecidable (see [Hopcroft & Ullman 79, Ch. 7] for a full discussion of Turing machines and proving equivalence).

The following outline of a Turing machine encoding is fairly standard. Similar descriptions are given for encoding Turing machines in feature grammars (e.g. [Ritchie 85]). A stage in the computation can be encoded as a situation. Each stage will support facts about the current state of the Turing machine, the symbol in the current position on the tape and encodings of the tape itself. The state of the tape will be encoded as what are effectively two stacks related by the relations `LeftTape` and `RightTape`. The relations take two arguments the first is the current stage-situation while the second is a two-place relation `Tape`. Its first argument is a value on the tape at that point. Its second argument is either the constant `Nil` or another `Tape`-fact. The representation of the tape can be expanded whenever it finds `Nil` at the left or rightmost point on the encoded tape. The following ASTL constraints specify this

```
*T : [S ! S != <<LeftTape,S,<<Tape,Blank,Nil,1>>,1>>
          S != <<RightTape,S,*Right,1>>
          S != <<State,S,*State,1>>
          S != <<Value,S,*Value,1>>]
<=
  *S : [S ! S != <<LeftTape,S,Nil,1>>
            S != <<RightTape,S,*Right,1>>
            S != <<State,S,*State,1>>
            S != <<Value,S,*Value,1>>]

*T : [S ! S != <<RightTape,S,<<Tape,Blank,Nil,1>>,1>>]
          S != <<LeftTape,S,*Left,1>>
          S != <<State,S,*State,1>>
          S != <<Value,S,*Value,1>>]
<=
  *S : [S ! S != <<RightTape,S,Nil,1>>
            S != <<LeftTape,S,*Left,1>>
            S != <<State,S,*State,1>>
            S != <<Value,S,*Value,1>>]
```

The information about the Turing machine transitions themselves can be encoded in a situation called `TM`. `TM` supports facts of the five place relation `Transition`. The arguments are current state, current tape value, new state, new tape value and direction (one of `MoveRight`, `MoveLeft` or `Halt`). For example a very simple machine may be encoded thus. It moves left over `a`'s on the tape until a `b` is found.

```
 TM :: [M ! M != <<Transition,1,a,1,a,MoveLeft,1>>
            M != <<Transition,1,b,2,b,Halt,1>>]
```

In addition to the tape expansion constraints we need two constraints to specify the cases when the operation is move left or move right.

```
*T : [S ! S != <<LeftTape,S,<<Tape,*NewValue,*Left,1>>,1>>
          S != <<RightTape,S,*Right,1>>
          S != <<State,S,*NewState,1>>
          S != <<Value,S,*NextValue,1>>]
<=
  *S : [S ! S != <<LeftTape,S,*Left,1>>
            S != <<RightTape,S,<<Tape,*NextValue,*Right,1>>,1>>
            S != <<State,S,*State,1>>
           S != <<Value,S,*Value,1>>],
  TM : [M ! M != <<Transition,*State,
                                *Value,
                                *NewValue,
                                *NewSate,
                                MoveRight,1>>].

*T : [S ! S != <<LeftTape,S,*Left,1>>
          S != <<RightTape,S,<<Tape,*NewValue,*Right,1>>,1>>
          S != <<State,S,*NewState,1>>
          S != <<Value,S,*NextValue,1>>]
<=
  *S : [S ! S != <<LeftTape,S,<<Tape,*NextValue,*Left,1>>,1>>
            S != <<RightTape,S,*Right,1>>
            S != <<State,S,*State,1>>
            S != <<Value,S,*Value,1>>],
  TM : [M ! M != <<Transition,*State,
                                *Value,
                                *NewValue,
                                *NewSate,
                                MoveLeft,1>>].
```

The initial tape can be specified as a basic situation and we can ask queries about the existence of some state-situation which is in a halting state.

This encoding is simple and does not require computationally complex functions to build the representation of any Turing machine and hence shows that ASTL is Turing computable. This fact need not worry us and it is probably "a good thing" rather than "a bad thing". One might want to consider such computational power as bad because it implies that the theories of (human) knowledge representation and natural language semantic comprehension are Turing computable which is unlikely to be true as humans have only finite resources. However when we consider ASTL as a meta-theory for implementing theories its computational power is an advantage. Theories encoded within it need not necessarily be Turing computable but it is convenient, if not necessary, for a tool to provide a high level of computational power.

3.6 Implementation

Some description of the implementation is useful. As we are interested not only in describing theories but also in giving actual computational treatments on a computer, an implementation of ASTL is useful or even necessary to show the suitability of ASTL as an actual computational system.

This section describes a particular implementation of ASTL. It is important to realise that there is a distinction between ASTL, the abstract situation theoretic language and ASTL the implementation. Certain aspects of the abstract language have been modified to allow for simpler implementation (some of which were mentioned in the inference section above). The implementation described here is in Common Lisp but it should not be read that ASTL could not or should not be implemented in any other programming language (e.g. Prolog or C). Lisp was chosen for reasons such as history and the author's personal preference in writing Lisp rather than any computational reason.

The technique used in the implementation for theorem proving deserves some mention. Again it must be stressed that this is not the only way to implement ASTL but is a relatively interesting way to do it.

ASTL, as a computational system, basically deals with a set of propositions and constraints. Using the inference rules, queries (goal propositions) are proved by applying the inference rules where appropriate and generating new propositions. All propositions are actually held as basic propositions consisting of a situation followed by a type with *one* condition (i.e. the type reduction inference rule is applied before propositions are "asserted" to the database).

Originally an implementation was attempted that closely followed standard Prolog implementations. Proof trees were searched using depth first search with backtracking. Although this initially seemed like a good strategy, because ASTL is different from standard first order logic, two basic problems exist. First, ASTL descriptions are more likely to contain "left-recursive" constraints, and secondly constraints may contain cyclic references—these two points are closely related. Both these conditions are likely to put a simple depth first search with backtracking strategy into an infinite loop. A more robust strategy is necessary.

The implementation described here uses a technique reminiscent in many ways to a chart parser. In fact the following description relies heavily on such a technique (see [Winograd 83, pp 116-127] for a more general description of chart parsing). Edges in this "chart" implementation represent sentences in ASTL (i.e. either basic propositions or constraints). Complete edges represent proved propositions while incomplete ones represent constraints—which can be viewed as conditional propositions. For example given the following proposition and constraint

```
Sit1:[S ! S != <<smiles,h,1>>].
Sit1:[S ! S != <<happy,h,1>>]
<=
   Sit1:[S ! S != <<smiles,h,1>>].
```

The above two ASTL sentences give rise to the following two edges.

```
Edge1:
        Label: Sit1:[S ! S != <<smiles,h,1>>]
        Requires: nil
Edge2:
        Label: Sit1:[S ! S != <<happy,h,1>>]
        Requires: (Sit1:[S ! S != <<smiles,h,1>>])
```

Edges have a label, and a required list (as well as other fields—see later). The label and each member of the required list are propositions. Notably we do not (at this stage) have any equivalent for vertices in this "chart". Edges effectively start and end at the same point. Two global structures of edges are kept: the `chart` a list of edges already processed and the `agenda` a list of edges that have yet to be processed. To prove a query the following algorithm **ProveProp** is used

```
ProveProp
    construct basic edges from basic situations
        and add them to the chart
    construct initial edge from the query
        and add it to the agenda
    while agenda is not empty do
        remove top edge from agenda and make it current
        if current is not already in chart
            add current to chart
            if current's required list is non-nil
                find all constraints that might allow
                the first proposition in current's
                required list to be proved.
                make edges from them and add to agenda
            if current's required list is nil
                (current is a complete proposition)
                for all incomplete edges i in the chart
                combine(current,i)
            else if current's required list is non-nil
                for all complete edges i n the chart
```

```
20                    combine(i,current)
21    find all situations that make the initial query true and
22    print them.
```

The check for constraints that could be used to prove the current proposition (lines 10-13) deserves a little more explanation. As we are proving top down, a check for appropriate constraints only occurs when the current edge has a non-null required list. The first proposition in the required list of `current` is looked at, all constraints that could prove a proposition with the same relation as `current`'s first required proposition are used to create new edges on the agenda. (If `current`'s first required proposition's relation is a variable then, of course, all constraints must be selected.) In this way slightly more work may be done than is absolutely necessary in proving the goal. It may be that a proposed constraint may not in fact contribute to the proof but this cannot be determined until we have actually proved the goal. However, this is not always bad. Because all proved propositions are recorded in the chart, they may turn out to be required later in the proof. This is directly analogous to what happens in a conventional chart parser used with a syntactic grammar. For example when a noun phrase is required at some point, all grammar rules which can form noun phrases are proposed to the chart even though it may be only a third person singular noun phrase that we are looking for.[4]

A second part of **ProveProp** that requires further explanation is the **Combine** routine (lines 17 and 20). This routine is the equivalent of the *fundamental rule* in a standard chart parser. It is this function that applies the inference rules.

```
Combine(complete-edge, incomplete-edge)
    if complete-edge's label matches the first proposition
        in incomplete-edge's required list then
        build a new edge from incomplete-edge minus the
            first proposition from its required list
            (modus ponens inference rule)
            add new edge to agenda
    if complete-edge's label matches the first proposition
        in incomplete-edge's required list but
        only up to situation arguments then
        build a new edge from incomplete-edge minus the
            first proposition from its required list plus
            the unmatched situation arguments
            (argument promotion inference rule)
            add new edge to agenda
```

This "chart" technique has the advantage that because a check can easily be made if a particular edge/proposition already exists infinite loops can be avoided in many

[4]Of course, not all chart parsers do this. Some do use top down prediction by instantiating some variables to cut down parse time but the advantages and disadvantages of this depend very much on the particular grammar being used and particular utterance being be parsed.

cases where a simple depth first search strategy would not terminate. For example if we have a basic situation and constraint of the form

```
*S : [S ! S != <<happy,h,1>>]
<=
   *S : [S ! S != <<happy,*X,1>>].
SIT1 : [S ! S != <<happy,t,1>>].
```

(This can be summarised as Hanako is happy in a situation if someone is happy in that situation.) The following query

```
*S : [S ! S != <<happy,h,1>>].
```

would succeed in finding `SIT1` as a solution but would not loop indefinitely. In a depth first search strategy, as used by default in Prolog, the above constraint could cause a loop if naively implemented because in trying to prove the right hand side of the constraint we could try to re-use the constraint (so called *left recursive rule*).

There are cases, however, where loops are unavoidable, that is where constraints actually do predict an infinite number of distinct propositions. For example the basic situation and constraint

```
Sit1 : [S ! S != <<happy,h,1>>].
*S : [S ! S != <<happy,h,1>>]
<=
   *T : [S ! S != <<happy,h,1>>].
```

(This can be summarised as if Hanako is happy in one situation she is also happy in another.) And we have a goal proposition

```
*S : [S ! S != <<happy,h,1>>].
```

(In which situations is Hanako happy.) The result is an infinite number of situations as the consequence of applying the constraint introduces a new situation to which the constraint may apply again. Such a query would cause this ASTL implementation to loop.

However the main advantage of a chart based proof strategy is that all propositions and constraints proved during a proof remain in the chart so that if they are required again during the same proof they do not need to be re-proved. This should provide for a more efficient proof. This technique of keeping proofs and partial proofs in a table is termed the *tableau method* in the theorem proving literature ([Reeves 83]). In the AI literature this technique has been described as *Earley Deduction* ([Pereira & Warren 83]). However there they are only showing how to treat grammars and utterances in a logical way rather than using such a theorem proving technique for proving arbitrary logical propositions.

Although not exploited in the current version, another advantage of a chart based theorem prover is that traces can be kept of how propositions have been proved, showing the dependencies between propositions and constraints. This could be used to generate a simple explanation of why a proposition is true. Also looking further ahead, if a treatment of default constraints, or *defeasible constraints* are added to the language it is important to keep track of interdependencies between propositions and constraints so that consequences of retracted propositions may be dealt with efficiently.

In the above explanation we stated that our edges in the ASTL chart do not have vertices as in a normal chart parser. This is true for normal propositions and constraints. However as was mentioned before ASTL is designed as a tool for language processing. In the next chapter we will introduce extensions to ASTL to allow efficient processing of grammars and utterances. Although these extensions can be modelled completely within the basic ASTL system special treatment has been built into the implementation so that a more efficient treatment can be given. These extensions are implemented such that propositions about utterances have a notion of start and end points which in this chart system are implemented with vertices. The addition of vertices to (some) edges allows more efficient indexing of propositions and constraints thus less searching for appropriate edges is necessary.

Also to aid efficiency, edges in the chart (both for utterance situations—i.e. with vertices—and others) are indexed by their situation and relation name. We have not yet really said anything about variables in edges. All edges are also associated with a bindings list for variables that occur in them. Thus it is more correct to talk about instances of constraints being represented by edges than the constraints themselves.

All unbound variables in complete edges are skolemised. This is justified from the semantics of a simple ASTL proposition containing a variable. The proposition

```
SIT1 : [S ! S != <<happy,*X,1>>]
```

states that `SIT1` supports the fact that something is happy. As the scope of variables is the sentence in which they appear this variable must be unique. The implementation will assign an arbitrary constant to this variable. This means that even if the previous proposition is true it is not sufficient proof that some particular object is happy in `SIT1`. That is if we have

```
SIT1 : [S ! S != <<happy,*X,1>>]
```

we cannot prove (based only on this evidence)

```
SIT1 : [S ! S != <<happy,h,1>>]
```

Situation theory (and ASTL) also offers *parameters* as a means of representing indeterminate objects. However no specific support is included within ASTL for parameters thus the way they are treated is solely dependent on the particular ASTL description. But it must be emphasised that variables in the language of ASTL are quite distinct from the concept of parameters. Variables will denote some non-variable object in the model while parameters denote parameters in the model.

3.7 Comparison with other systems

Now that we have introduced ASTL and given some discussion of its various properties it seems useful to compare it with other systems. Here we will specifically compare it in three ways. First with situation theory itself, then with PROSIT, an alternative computational situation theoretic language, and finally with the general computational systems of feature systems.

3.7.1 ASTL and situation theory

ASTL is designed as a situation theoretic language. ASTL can be summarised as consisting of the following features: a representation for individuals, relations, parameters, situations, situation types, propositions and constraints. It also offers a set of inference rules (and inference mechanism) to prove propositions about a system of propositions and constraints. From a situation theoretic point of view this collection of objects is rather conservative. All of these, except perhaps constraints, are part of almost any definition of situation theory. A notion of constraints is described in [Barwise & Perry 83] and later in [Barwise 89b, Ch 5] but these descriptions are concerned more with the philosophical aspects than the computational ones. More recent work, has introduced the concepts of *channel theory* which offers an abstract characterisation of constraints and information flow([Barwise 92], [Barwise 93], [Barwise & Seligman 93]).

Although we can claim that all parts of ASTL have a situation theoretic basis we would also like to claim that ASTL embodies all the fundamental aspects of situation theory. However this is probably not the case. It is true that, as it will be seen in later chapters, ASTL is sufficiently powerful to allow a number of other semantic theories to be described in it but there are still aspects normally associated with situation theory that are not contained within the current version of ASTL.

One aspect which is not dealt with within ASTL is *coherence*. It is normally stated as part of situation theory that a fact and its dual (the fact with opposite polarity) may not both be supported by a situation. In ASTL terms this would require the following expression to always be false.

```
Sit1 : [S ! S != <<happy,h,1>>
            S != <<happy,h,0>>]
```

But within the current version the above can be true. However it may be possible to give a useful treatment within a particular ASTL description. That is, some constraint of the form[5]

```
*S : [S ! S != <<actual,S,0>>]
<=
   *S : [S ! S != <<*R,*A,1>>
             S != <<*R,*A,0>>]
```

Although something more complex is really needed that would probably require actual extensions to the ASTL language. The whole area of coherence in situations is non-trivial and the ideas in it are closely related to those in other aspects of knowledge representation: belief revision, non-monotonicity and truth maintenance.

Another important aspect of situation theory which is not explicitly part of ASTL is that of *parameters* and *anchoring*. One of the major motivations for situation theory was a requirement for a representation of parametric objects. The introduction of *parameters* which have a denotation in the model—rather than simply variables—allows the description of parametric objects. *Anchoring* is a facility which allows parameters to be related to other objects in a way analogous to variable assignment. ASTL does offer parameters and simple aspects of anchoring can be modelled within ASTL by constraints and representing *anchoring environments* as situations. In Chapter 4 we will see such a technique but it is not really adequate in general. In Section 7.4.1 we will outline an extension to ASTL which allows for a better treatment of this phenomenon.

Thus although ASTL is firmly grounded within situation theory there are still some aspects which are missing from the language. However ASTL does offer a firm base on which we can build extensions and offer a language which encompasses more of the theory.

3.7.2 ASTL and PROSIT

ASTL is not the only attempt at building a computational language based on situation theory. PROSIT has very similar goals, [Nakashima *et al* 88],[Frank & Schütze 90]. PROSIT is a programming/knowledge representation language based on situation theory in a similar way that Prolog is based on first order logic. It offers a representation of individuals, relations, parameters, situations and constraints. As it is currently implemented it does not offer a representation for types or abstractions but such extensions are being considered.

PROSIT is written in Common Lisp. When run it gives a new top level which offers a Prolog-like interface (although PROSIT's syntax is Lisp-like). Statements can be

[5] Of course, this is inadequate. It only deals with one form of fact—relations with one argument—and requires that situations supporting an `actual`-fact be treated specially throughout the rest of the description. However hopefully the general idea is illustrated.

asserted or queried. Unlike Prolog, statements (basically infons) have to be situated. That is, assertions and queries are with respect to particular situations (unlike Prolog, which effectively only has one "situation"—the whole database). For example

```
<top> ? (! (!= S (sings hanako)))
```

This asserts in the global situation (*top*) that the situation S supports the (positive) fact `(sings hanako)` So that,

```
<top> ? (!= S (sings hanako))
yes
<top> ? (!= S (sings taro))
unknown
```

Note that we assert the support infon to the global situation. As all things are situated we could assert infons to different situations say $S1$ and $S2$ or even we could assert these assertions to different situations thus S would or would not support `(sings hanako)` depending on the situation it was queried in. As all assertions are situated, queries depend very much on the "current" situation. One can view the "database" as a tree of situations with the global situation at the top. One can traverse this tree explicitly using the *in* and *out* relations, or implicitly using nested support relations. The paths are always disjoint, thus there is no way to *jump* to a situation as the viewpoint is always situated. This treatment of the supports relation as non-absolute distinguishes PROSIT from other descriptions of situation theory. Their argument for taking this direction is that it means that truth is locally determined and there is no need to search some (probably very large) global list of supports relations.

PROSIT also supports a form of *constraint*. Unlike ASTL which effectively offers global constraints between situations, PROSIT's constraints are between facts *within* situations. Suppose we wish to state that in situation S anything that sings also dances.

```
<top> ? (! (resp S (<= (dances *X) (sings *X))))
yes
<top> ? (!= S (dances hanako))
yes
```

resp is a special relation used to cause the first argument (a situation) to respect the second argument (a constraint). Notice that the fact that a situation supports a constraint is also situated (in this case in *top*) thus depending on "where" a query is made the constraint may or may not apply.

In ASTL, constraints are global over all situations so that if some situation is of the appropriate type the constraint will apply. In PROSIT the situation is different. Constraints will only apply if the fact that the situation respects a constraint is explicitly asserted. This has the advantage that unnecessary searching for appropriate constraints/situations does not occur but the disadvantage that some mechanism must assert *resp* facts for each appropriate situation. These two views are extreme ends of a spectrum. Ideally we would like some constraints to be (as in ASTL) global—using the distinctions made in [Barwise & Perry 83, Ch 5] these would be called *necessary constraints*. We would also like some to be local to a situation (as in PROSIT's constraints). But more importantly we would also like something in between. Barwise also describes *conditional constraints* which apply to general classes of situations. Neither ASTL or PROSIT offer such constraints directly—though such constraints can be modelled in either system. However merely modelling may not be enough. To check that a constraint is appropriate requires work. If the domain of situations that the constraint may apply to is pre-defined a more efficient use of constraints should be possible.

PROSIT also offers a simple form of anchoring for parameters although not really any form of anchoring environment. Unlike ASTL, PROSIT offers a number of features which are not normally within situation theory but do make the language easier to use. Such operations as union and intersection of facts in a situation allow a more explicit and procedural interpretation. Moreover, constraints can be specified to be forward or backward chaining thus stating if application should occur at assert time or query time.

In summary, PROSIT also offers a computational language based on situation theory but it also admits other features which, although useful to the language, are not normally associated with situation theory. ASTL has been developed for experiments with natural language processing while PROSIT is primarily aimed at the more general problems of knowledge representation and probably fits closer to the work of logic programming than natural language processing (for example typical work in PROSIT can be seen in [Nakashima *et al* 91]). Although initially an attempt was made to extend PROSIT to include language processing features too many additions were necessary. Both the treatment of constraints and the fact that PROSIT does not have a concept of abstraction made it difficult to augment PROSIT satisfactorily. This is not to say that the two systems ASTL and PROSIT could never be combined, ideas from each could be used together to build a language which would offer the advantages of both.

3.7.3 ASTL and feature systems

In the initial investigation for a general computational system for implementing semantic theories some time was spent looking at the possibility of using some form of feature system to do this. Feature systems are now very general ([Johnson 88], [Smolka 88]). They have been used in many theories mainly for syntactic representation (e.g. GPSG [Gazdar *et al* 85]), but also they have been used for semantic representation. However there now comes the question exactly what do we mean by feature systems. The problem is that there are many facilities which one can include in a feature system and depending on your needs you can vary your selection.

With hindsight we can look at the definition of ASTL given above and find a particular feature system which has the same computational and descriptive power. ASTL individuals, parameters, and facts can all be represented simply in a standard feature system. The difficult cases are types, situations and constraints.

Situation types require a form of set-valued feature. Representing facts within a type by some form of list representation (using features like `FIRST` and `REST` cf. [Shieber 86a, p 29]) is not powerful enough, as manipulation and testing of facts against situations becomes too difficult (if at all possible in general) because facts in situations are not ordered while a `FIRST/REST` encoding enforces an order. What is needed is a representation whose interpretation is not dependent on the order in which facts occur. This can be done by set-valued features. In feature logics there are (at least) two interpretations of set-valued features. Intuitively, first we can think about the value as being underdetermined. That is the value for the feature is one of the values of the set but as yet we cannot tell which. Alternatively we can view the value of a set-valued feature to be all values of the set. The definition of unification of such values also differs. *Hoare unification* must be used in the multi-valued case (effectively the values are unioned by unification) and *Smythe unification* is used in the case where the value is underdetermined (values are intersected). [Rounds 88] discusses these issues in more detail. (There are other definitions of set-valued features too that differ from this, as in [Pollard & Moshier 90].) For a representation of situation types we require to use set-values in a multi-valued way. A situation supports some set of facts. Hence the multi-value definition and Hoare unification are required.

Using set-valued features only offers a representation for types but not quite for situations. In situation theory, situations are first class objects. To represent situations in feature logics, there are two possible alternatives. We can either have a structure that represents a situation, containing a set-valued feature representing the facts it supports. All references to that situation would refer to that very structure. This would introduce cycles in our structures where situations referred (directly or indirectly) to themselves. Or, the second technique is to have names for situations and rely on the model to equate names to structures. The second of these has a problem that assertion (or unifying) has to trace names of situations to the situations themselves. Both these are possible although the first seems closer to standard definitions of feature structures.

The third construction in ASTL which does not directly map on to any particular feature structure are constraints. Some grammar theories which are described using feature logics require constraints between features in a category (e.g. Feature Cooccurrence Restrictions in GPSG [Gazdar *et al* 85]). Obviously grammar rules are really a form of constraint but others have considered other constraints between categories ([Kilbury 87], [Frisch 86]) which is closer to the kind of constraints defined in ASTL. Later work ([Hegner 91]) has proven decidability for constraints restricted to horn clauses, while HPSG ([Pollard & Sag 87]) requires more powerful constraints. As we can see constraints in feature logics are not new and we can easily find a definition that comes close to the required definition of ASTL constraints.

What this comes down to is that a feature systems with sets, cyclic structures and a form on inter-category constraint would come very close to what ASTL is. That is a language of the computational and descriptive power of ASTL could be defined as a feature system. This begs the question: if we could have used such a feature system, why do we actually describe the system in terms of situation theory rather than features? The answer to that is in the end a subjective one. However even if ASTL were described solely in terms of a feature system it should also be pointed out that its characteristics are the fundamental characteristics of situation theory so even as a feature system the close relationship with situation theory would still be there. If such a definition were made it seems acceptable to describe it as an implementation of ASTL in a feature system. Also because situation theory is described as a semantic theory it seems consistent to describe our general semantic theory in such terms rather than simply as a feature system.

3.8 Summary

This chapter has introduced a computational language called ASTL which is based on aspects of situation theory. ASTL is formally defined and some simple examples of the language are given showing how it can actually be used. One possible implementation is described. ASTL is then described with respect to three other systems. First, it is contrasted with situation theory itself showing that the basic parts of ASTL can be found in all of the general descriptions of situation theory placing it legitimately in that paradigm. Secondly, another situation theoretic language, PROSIT, is described comparing it with ASTL and showing where they differ. Finally, feature systems are discussed identifying which aspects of feature systems would be required to define a particular feature systems that would have the same computational and descriptive properties as ASTL.

It should be noted that at this stage we have not yet justified ASTL's characteristics as necessary (or sufficient) for a system that is suitable for describing aspects of general semantic theories. This we will do in the following three chapters—where we will look at how three semantic theories can be specified within ASTL. In Chapter 8, we will also return to this issue of why the characteristics of ASTL are those that are necessary in a system for describing semantic theories.

Chapter 4

Processing Natural Language and STG

4.1 Introduction

In this chapter we will show how the situation theoretic based language, ASTL, can be used as a medium for describing natural language linguistic grammars and how it offers a mechanism that allows parses of utterances to be found with respect to such grammars. A way of representing linguistic entities is also described. Grammar rules can be defined as constraints over such objects. A simple fragment of English is given within such a framework. That fragment is simple but includes enough syntax to allow certain interesting semantic phenomena to be displayed. The syntax is sufficient for simple examples of donkey anaphora, inter-sentential anaphora and quantification.

The second part of this chapter is the first example of using ASTL to describe another semantic theory, Situation Theoretic Grammar (STG) [Cooper 89]. Admittedly STG should be one of the easiest theories to describe in ASTL as STG's ideas of grammar processing are those that are embodied in ASTL. But STG is an example of situation semantics and it is necessary that ASTL can at least deal with the semantic theories closest to itself if ASTL is to be treated as a general semantic theory.

4.2 Situations and language processing

In this section we will explain how a situation theoretic representation can be given to natural language utterances (and do so in terms of ASTL). We will show that language and a conventional computational syntactic view of it can be naturally described within a situation theoretic framework. Note that here we are *not* proposing a new syntactic theory, only a method of encoding existing theories within situation theory and ASTL. Some of the basic ideas of syntactic processing in a situation theoretic framework presented here are basically those in Situation Theoretic Grammar (STG) given in

[Cooper 89].

The basic idea is that the actual utterance of a piece of language can be described as a situation. That situation supports facts about the utterance such as the start and end points as well as what was actually said (the phonology). The situation can also support facts about the linguistic content of the utterance—its syntactic category, its number, gender etc. as well as the semantics of the utterance itself. For example the utterance of the noun phrase *"Hanako"* can be represented as

"Hanako" $\rightsquigarrow$

```
SIT123

cat(SIT123,NounPhrase)
use_of(SIT123,"Hanako")
start(SIT123,t234)
end(SIT123,t235)
```

Likewise the utterance of the verb *"sings"* in the same location, immediately following the above utterance of "*Hanako"* may give rise to the following situation

"sings" $\rightsquigarrow$

```
SIT135

cat(SIT135,VerbPhrase)
use_of(SIT135,"sings")
start(SIT135,t235)
end(SIT135,t236)
```

Of course we can now consider these two utterances as part of a larger one. They were both uttered at the same time (the `end` of the first is at the `start` of the second) and hence we can also state that

```
                         SIT143
                         cat(SIT143,Sentence)
                         start(SIT143,t234)
                         end(SIT143,t236)
                                            SIT123
                                            cat(SIT123,NounPhrase)
                         daughter(SIT143,   use_of(SIT123,"Hanako")   )
                                            start(SIT123,t234)
                                            end(SIT123,t235)
"Hanako sings" ~>
                                            SIT135
                                            cat(SIT135,VerbPhrase)
                         daughter(SIT143,   use_of(SIT135,"sings")   )
                                            start(SIT135,t235)
                                            end(SIT135,t236)

                         lp(SIT143,SIT123,SIT135)
```

If we wish to state that a sentence-type situation occurs when we have a noun phrase and verb phrase together we can, in the same way as we state normal (conventional phrase structure) grammar rules, write an ASTL constraint which says just that

```
*S : [S ! S != <<cat,S,Sentence,1>>
          S != <<start,S,*Start,1>>
          S != <<end,S,*End,1>>
          S != <<daughter,S,*NP,1>>
          S != <<daughter,S,*VP,1>>
          S != <<lp,S,*NP,*VP,1>>]
<=
   *NP : [NP ! NP != <<cat,NP,NounPhrase,1>>
               NP != <<start,S,*Start,1>>
               NP != <<end,S,*M,1>>],
   *VP : [VP ! VP != <<cat,VP,VerbPhrase,1>>
               VP != <<start,S,*M,1>>
               VP != <<end,S,*End,1>>].
```

There is nothing new or special going on here. Basically we are effectively encoding conventional grammar rules within ASTL in a very similar way to how grammar rules can be encoded in feature systems. There are perhaps some differences—specifically we include an explicit reference to the start and end points of an utterance in its encoding while in conventional feature grammars this would not normally be the case.

As discussed earlier (in Section 3.7.3) it may be possible to define a feature system which has most of the properties of ASTL. But we can also look at this in reverse, ASTL is not unrelated to feature systems and there may be ways to encode feature systems within ASTL. Let us briefly look at this possibility. We can represent feature structures as situations and features as facts but with one important difference. Features are functional while facts are relational thus there is nothing that restricts a feature structure situation from supporting conflicting feature facts as in

```
SIT276::[S ! S != <<cat,S,sentence,1>>
             S != <<cat,S,verbphrase,1>>]
```

What is necessary is a definition of *feature relations*. Although this cannot be done in basic ASTL as it is currently defined, a simple extension could achieve this. We could add the restriction that a situation may only support at most one positive fact for any particular feature relation. The other difference between an ASTL grammar and a conventional feature grammar is "unification". In a conventional feature grammar, constituent feature structures "match" grammar rule daughters using unification, that is as long as there are no conflicting features they may combine (more formally if the sets of feature graphs of which they are types have a non-null intersection). In contrast, in an ASTL grammar, daughters only "match" constituents when a situation has *all* the "features" mentioned in the right hand side of rule. In unification terms the ASTL constituent must be an *extension* of ASTL grammar rule daughter.

Furthermore ASTL can not only offer encodings for conventional phrase structure rules. Syntactic theories such as GPSG [Gazdar *et al* 85] do not contain conventional phrase structure rules but get the same effect through Immediate Dominance and Linear Precedence rules. These rules collectively constrain a syntactic structure over the same syntactic constituents rather than in a conventional phrase structure grammar where rules define a single hierarchical structure. Because of the form of ASTL constraints it should not be very difficult to define ID/LP rules within ASTL.

In the simple example ASTL grammar rule above we state a `daughter` relationship between the mother node and the daughters. We also specify a linear precedence relation (via the relation `lp`). We do not, but perhaps should, explicitly state the exact number of daughters allowed.

It is not the purpose of this thesis to introduce a new syntactic theory. All we wish to show is that existing syntactic theories can naturally be modelled within ASTL. It is no harder to specify a computational description of a syntactic theory in ASTL than it is to do so in a feature system. In fact, because ASTL offers very general constraints, it may even be easier to describe some theories which do not limit themselves to phrase structure rules (e.g. GPSG and GB) in ASTL than in a feature system. It must be noted that we are not really concerned with strictly defining feature systems within ASTL and only wish to point to how this could be achieved. The grammar system given here is simple and does not require the full power of a general feature system to describe it, but it is adequate for the semantic phenomena we wish to examine.

There is an important extension to basic ASTL which makes the treatment of phrase structure grammars easier and more efficient. Above we showed how a simple phrase structure rule could be encoded as a constraint. This technique is also used in logic programming. Definite Clause Grammars (DCGs) [Pereira & Warren 80] use a very similar method to the one outlined above for encoding phrase structure grammars in first order logic (or more specifically in Prolog). In DCGs instead of a start and end point the extra conditions are with respect to a list of words. More particularly a category, represented by a predicate indicates its start position as one point in a list structure and its end further down that list. This difference between this modelling of the string of words uttered and that used by ASTL (points) is not important here.

However, more importantly what has been copied from Prolog DCGs is that DCGs in Prolog can be specified using a special syntax such that the utterance start and end points need not be explicitly stated in the rules. This offers a much more readable notation for rules. In ASTL we have also added a special syntax for grammar rules. For example the ASTL constraint above that represents a phrase structure rule can be re-written as the following ASTL grammar rule

```
*S : [S ! S != <<cat,S,Sentence,1>>
          S != <<daughter,S,*NP,1>>
          S != <<daughter,S,*VP,1>>
          S != <<lp,S,*NP,*VP,1>>]
->
   *NP : [NP ! NP != <<cat,NP,NounPhrase,1>>],
   *VP : [VP ! VP != <<cat,VP,VerbPhrase,1>>].
```

Specifically we need not include the `start` and `end` relations and we have a different form of arrow.

Unlike DCGs, ASTL grammar rules are *not* translated into their underlying form with explicit start and end points. In the implementation discussed here grammar rules are interpreted directly. This allows for a much more efficient implementation. The start and end points, because they are known to exist for these situations can be built in and used for efficient indexing in the theorem prover.

In addition to a special form of constraint that acts as a grammar rule we also have a special form which introduces basic situations that are used to represent utterances of words. Word entries specify what type of situation is introduced by the utterance of a word. As in

```
Hanako - [S ! S != <<cat,S,NounPhrase,1>>
              S != <<use_of,S,"Hanako",1>>].
```

If we were to translate this to its expanded form it would look something like

```
*S : [S ! S != <<cat,S,NounPhrase,1>>
          S != <<use_of,S,"Hanako",1>>
```

```
            S != <<start,S,*Start,1>>
            S != <<end,S,*End,1>>]
<=
     use of "Hanako"
```

We include a way to "utter" words in a sequence which will introduce situations of the type declared for each word.

A consequence of this is that effectively there can exist two types of situation in an ASTL description *utterance situations* and *normal situations*. Situations introduced by word entries and those predicted by grammar rules are called *utterance situations*. Grammar rules can only apply to utterance situations but normal constraints can apply to both types of situation. In the current system it is *not* possible to make a normal situation into an utterance situation.

Both grammar rules and word entries are optional parts of an ASTL description. If included they appear at the end of a description.

4.3 A simple grammar fragment

Throughout most of the rest of this thesis we will be looking at how particular semantic theories can be described within ASTL. In order to do this we need some form of syntactic backbone with which we can realise our descriptions and actually use them to produce semantic representations of natural language utterances. To do this we will use the same simple grammar fragment. The following fragment is based on that in [Rooth 87] and includes sufficient syntax to give examples of the semantic phenomena we are interested in.

As a conventional phrase structure grammar the Rooth fragment can summarised as

$$
\begin{array}{ll}
S & \rightarrow NP,\ VP. \\
VP & \rightarrow V,\ NP. \\
NP & \rightarrow Det,\ N. \\
N & \rightarrow N,\ PP. \\
PP & \rightarrow P,\ NP. \\
D & \rightarrow S. \\
D & \rightarrow D,\ S.
\end{array}
$$

We also include the category D for discourse to allow utterances of more than one sentence. Lexical entries will slightly vary from description to description (pronouns are not included in the basic STG description, though they are in the later descriptions of DRT and dynamic semantics), however overall the following classes will be used.

Det: a, every.

N:	man, donkey.
NP:	he, she, it, Hanako, Taro.
VP:	walks, talks, smiles, smiles.
V:	beats, likes.
P:	with, on.

The grammar rules can be simply translated into ASTL grammar rules as in

```
[S ! S != <<cat,S,Sentence,1>>
     S != <<daughter,*NP,1>>
     S != <<daughter,*VP,1>>]
->
 *NP : [NP ! NP != <<cat,NP,NounPhrase,1>>],
 *VP : [VP ! VP != <<cat,VP,VerbPhrase,1>>].
```

Likewise we can translate the other six rules. The full ASTL description is shown in Appendix A.2. This grammar is sufficient to describe examples like the following

Hanako sings.
A man walks. He talks.
Every man with a donkey beats it.

The following is a example analysis for *"a man walks"* (this is shown for a sentence rather than a discourse so that it can reasonably fit on a page).

```
SIT5397
  cat(SIT5397,sentence)
  daughter(SIT5397, SIT5155
                      cat(SIT5155,nounphrase)
                      daughter(SIT5155, SIT4991
                                          use_of(SIT4991,"a")
                                          cat(SIT4991,determiner)   )
                      daughter(SIT5155, SIT4981
                                          use_of(SIT4981,"man")
                                          cat(SIT4981,noun)   )
                                                                      )
  daughter(SIT5397, SIT5001
                      use_of(SIT5001,"walks")
                      cat(SIT5001,verbphrase)   )
```

4.4 Situation Theoretic Grammar

Situation Theoretic Grammar (STG) [Cooper 89] is a situation semantic theory. Its description includes a computational treatment in Prolog.[1] Not only does STG offer a semantic treatment of simple utterances but it includes a situation theoretic treatment of syntax. As ASTL was developed partly as an attempt to generalise the computational situation theoretic properties of STG it is not surprising that ASTL's treatment of syntax is essentially the same. However although we have shown in the previous section how to treat syntactic grammars in ASTL we have not yet dealt with describing treatments of natural language semantics. In this section we will describe how STG can be described within ASTL showing how situation semantic representations can be constructed for simple utterances.

Although the term "situation semantics" is often used as if it refers to a single semantic theory this is not actually the case. The term has been used to describe many quite different theories of natural language semantics—such as [Barwise & Perry 83] [Gawron & Peters 90], situation schemata [Fenstad *et al* 87] etc. Probably the only aspect that these theories have in common is the use of a *situation* object in their

[1] Confusingly Cooper's implementation is called ProSit (note capitalisation to distinguish it from PROSIT). Cooper's framework offers operators and predicates to deal with situation theoretic objects within standard Prolog rather than the design of a whole new language.

description. STG offers both a situation theoretic treatment of syntax as well as semantics. Unlike other situation semantic theories STG is given with respect to a particular grammar fragment thus making it easier to compare with more conventional semantic theories (e.g. Montague grammar).

In the previous section we showed how ASTL can be used to describe syntactic theories. The Rooth grammar fragment detailed above will be used as the basis for this description of STG. In the Rooth fragment utterances are represented by situations but the semantics of these utterances (i.e. what these utterances describe) are not included in the description.

Here, as in Cooper's original STG description, we will include a relation in each *utterance situation*, relating the situation to a situation theoretic object representing its semantics. An intransitive verb's semantics will be represented by a *parametric fact* with one argument. Parameters in situation theory can be used to represent partially determined objects. For example the representation for the intransitive verb *"smile"* would be

```
<<smile,A1,1>>
```

(It is possible for the polarity to also be parametric but we shall ignore that possibility in these examples.) In other semantic theories this would be similar to the simple lambda expression

$$\lambda A1\ [smile(A1)]$$

However, unlike variables in a lambda expression, there is no explicit identification of the parameters in the parametric fact case. In a transitive verb's representation there would be two parameters. The lambda representation is required to order those in some way while the parametric fact representation is not. Within ASTL as we have defined it, a parametric i-term simply denotes a fact containing parameters. How a parametric semantic object (in ASTL's model) relates to the real world is not defined here. The above case could be defined as the set of all possible *smile*-facts or as an abstraction over them. Such philosophical issues do not impinge on the descriptive or computational aspects of ASTL therefore we can ignore them for the present. However the issue of whether ASTL's representation as a parametric fact or extending ASTL to include an explicit representation for *abstractions* is returned to in Section 7.4.1.

In STG, we allow parameters to be *anchored* and *labelled*. These are ways of relating parameters to other objects. We hold anchoring and labelling facts in a situation that we call an *anchoring environment*. Anchoring is analogous to variable assignment (or substitution) in other theories. Each utterance situation is related to both a parametric semantic fact and an anchoring environment. For example, in a verb phrase utterance situation all parameters except the one representing the subject of the sentence will be anchored (as the subject parameter is as yet undetermined). In order to identify which parameter is related to which grammatical argument, parameters are *labelled* with grammatical functions[2] (e.g. `subj`, `obj`, etc.). According to these definitions the basic lexical entry for *"smiles"* would be

```
smiles -
  [VP ! VP != <<cat,VP,VerbPhrase,1>>
        VP != <<use_of,VP,"smiles",1>>
        VP != <<sem,VP,<<R1,A1,1>>,1>>
        VP != <<env,VP,SmileEnv::[S ! S != <<label,R1,pred,1>>
                                      S != <<anchor,R1,smile,1>>
                                      S != <<label,A1,subj,1>>],1>>]
```

The semantic entry is fully parametric and the associated anchoring environment anchors the parameter `R1` to the relation `smile`.

The semantic content of an utterance is defined with respect to the utterance's anchoring environment and parametric fact. The *content* is that fact where all the parameters in it are replaced by the objects anchored to them by the anchoring environment. This is analogous to beta reduction. For example the content of the *"smiles"* entry above is

```
<<smile,A1,1>>
```

In an utterance situation representing *"Hanako smiles"* we wish the content of the related parametric fact and anchoring environment to be

```
<<smile,h,1>>
```

To do this the semantics of the sentence utterance situation can be the same parametric fact as that in the verb phrase utterance situation but the anchoring environment also needs an anchoring for the parameter `A1`. There are two ways to consider this. The first is to have a constraint that adds to the anchoring environment that is related to the verb phrase the extra anchoring relation for `A1` such that the anchoring environments on the sentence and verb phrase utterance situations are the same. A second view is for the anchoring environment on the verb phrase to remain the same but state that the anchoring environment on the sentence utterance situation support the same anchoring and labelling facts as that on the verb phrase *plus* the new anchoring fact for the parameter `A1`. That is we extend the anchoring environment of the verb phrase with the anchoring relation creating a new situation. In situation theoretic terms we can say the verb phrase's anchoring environment is *part-of* the sentence's anchoring environment.[3]

[2] In [Cooper 89] parameters are labelled with their grammatical function *and* their respective utterance situation.

[3] Technically there is a possible distinction here between passing situation types and a *part-of* relation. A *part-of* relation between two situations A and B would be that all facts supported by A are also supported by B, while linking situation types does not necessarily entail this. As although the basic type is copied and all appropriate constraints will apply to both situations A and B it may be that A will actually support more than B by virtue of A appearing in some relation in some other situation which B does not.

Both these forms can be specified in ASTL. The first where the sentence and verb phrase have the *same* anchoring environment is easier to specify

```
[S ! S != <<cat,S,Sentence,1>>
     S != <<sem,S,*Fact,1>>
     S != <<env,S,*Env ::
                  [Env ! Env != <<anchor,*X,*Y,1>>],1>>]
->
 [NP ! NP != <<cat,NP,NounPhrase,1>>
       NP != <<sem,NP,*Y,1>>],
 [VP ! VP != <<cat,VP,VerbPhrase,1>>
       VP != <<sem,VP,*Fact,1>>
       VP != <<env,VP,*Env ::
              [Env ! Env != <<label,*X,subj,1>>],1>>].
```

Note how we select the parameter to be anchored by finding the one labelled by `subj` in the anchoring environment. Also we are assuming that the semantics of the noun phrase is simply a constant. The environments related to the verb phrase and sentence will be the same because we name them with the same variable `*Env`.

The second method where we extend the environment is the actual one we use throughout the following description. In this case the verb phrase's anchoring environment does not contain any facts anchoring the parameter labelled `subj`. However this is a little harder to specify in ASTL. The rule below uses a simple extension which allows multiple types to be specified for situations (separated by an ampersand). The rule would be

```
[S ! S != <<cat,S,Sentence,1>>
     S != <<sem,S,*Fact,1>>
     S != <<env,S,*SEnv ::
                  *VPEnvType &
                  [Env ! Env != <<anchor,*X,*Y,1>>],1>>]
->
 [NP ! NP != <<cat,NP,NounPhrase,1>>
       NP != <<sem,NP,*Y,1>>],
 [VP ! VP != <<cat,VP,VerbPhrase,1>>
       VP != <<sem,VP,*Fact,1>>
       VP != <<env,VP,*VPEnv ::
                      *VPEnvType &
                      [Env ! Env != <<label,*X,subj,1>>],
                  1>>].
```

In this case the two environments are distinct because they are referred to by different names[4] (`*SEnv` and `*VPEnv`). However we state that the type of `*SEnv` is `*VPEnvType`

[4] Formally this may not be true. The rule only states that they are not necessary the same rather

which is also the type of `*VPEnv` therefore all facts that are supported by `*VPEnv` will also be supported by `*SEnv`, but not necessarily the reverse. We also specify that the type of `*SEnv` includes not only the type of `*VPEnv` but also the fact anchoring the parameter labelled `subj` to the semantics of the noun phrase.

To see how an anchoring environment is built up from example utterances consider the utterance situations for *"Hanako"* and *"smiles"*.

```
SIT1007
  sem(SIT1007,h)
  use_of(SIT1007,"Hanako")
  cat(SIT1007,nounphrase)
```

```
SIT1011
                  SMILEENV
  env(SIT1011,      label(A1,subj)
                    anchor(R1,smile)      )
                    label(R1,pred)
  sem(SIT1011, R1(A1) )
  use_of(SIT1011,"smiles")
  cat(SIT1011,verbphrase)
```

Using the above grammar rule we get a sentence utterance situation of the form

```
SIT1211
  cat(SIT1211,sentence)
  sem(SIT1211, R1(A1) )
                  SIT1213
                    anchor(A1,h)
  env(SIT1211,      label(R1,pred)      )
                    anchor(R1,smile)
                    label(A1,subj)
```

The result is a situation related to a semantics which is a parametric fact, `R1(A1)` and an anchoring environment where `R1` is anchored to the relation `smile` and `A1` is anchored to the individual `h`.

We can view extending the anchoring environment as analogous to lambda application in a lambda calculus based system. But application is not the whole story, we still need something analogous to reduction to find the *content* of the parametric fact and anchoring environment. To do this what we will do is use the information in the

than that they are different. Also even if they have different names within ASTL they may actually denote the same situation within the model. However for the purposes of this explanation we can think of them as being different.

parametric fact and anchoring environment to define the *described situation*, that is what the utterance describes.

```
*S : [S ! S != <<described,S,*DS :: [D ! D != <<*VR1,*VA1,1>>],1>>
<=
   *S : [S ! S != <<cat,S,sentence,1>>
             S != <<sem,S,<<*R1,*A1,1>>,1>>
             S != <<env,S,*SEnv ::
                      [Env ! Env != <<anchor,*R1,*VR1,1>>
                             Env != <<anchor,*A1,*VA1,1>>],1>>].
```

That is we are finger the values anchored to the parameters in the parametric fact. `*VR1` and `*VA1` will be the values (e.g. `smile` and `h`). This technique is, of course, inadequate for general beta reduction. We will return to this issue shortly. In the above constraint we define the described situation in terms of the parametric and the anchoring environment. The full utterance situation for the sentence *"Hanako smiles"* is

```
SIT1909
  cat(SIT1909,sentence)
  sem(SIT1909, R1(A1) )
                    SIT1911
                      anchor(A1,h)
  env(SIT1909,        label(R1,pred)      )
                      anchor(R1,smile)
                      label(A1,subj)

                         SIT2310
  described(SIT1909,                    )
                           smile(h)
```

4.4.1 Quantification

The above describes how simple declarative utterances involving proper nouns can be treated in STG. In this section we discuss the treatment of the simple quantifiers `every` and `some`.

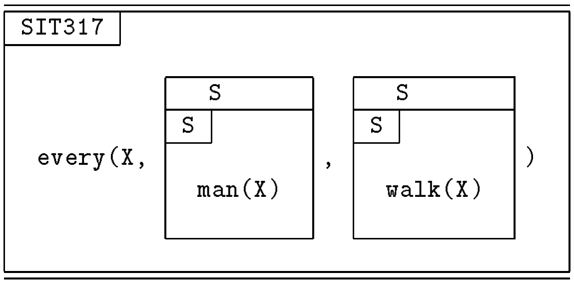
In Cooper's original description quantifiers were represented as a relation between two *properties*. Properties are a form of abstraction over propositions.

$$[X \mid s \models \ll happy, X, 1 \gg]$$

X is a parameter. Informally, for some object to have a property it must be the case that the proposition in the property is true if the parameter is replaced with that object. For example the above property is true for h if the proposition

$$s \models \ll happy, h, 1 \gg$$

is true. There is no direct equivalent of properties in ASTL (though the concept of *abstractions* discussed in Section 7.4.1 is very similar but a little more general). In the ASTL description of STG we must find an alternative representation. Here we use situation types (with parametric facts) to represent abstractions. As we may wish to extend the description later to deal with generalised quantifiers our ASTL representation for quantifiers is a three place relation between a variable (represented by a parameter), and two (parametric) situation types. Thus the desired ASTL semantic representation for the utterance *"every man walks"* is



Each utterance situation is related to a *described situation* whose type is defined by the (parametric) semantic fact and anchoring environment. This makes it possible to deal with cases where more than one fact is necessary to represent the sentence. Alternatively, we could introduce "logical" relations `and` and `or` and use these as in *"a man walks"*

```
<<and,<<man,X,1>>,<<walk,X,1>>,1>>
```

but this seems to complicate the issue when even just a limited form of types are available. It seems useful that an utterance describes some situation and that the utterance determines the type of that described situation.

There is also question about what it means for a situation to support an `every`-fact. We can paraphrase this with a constraint. Given the above situation `SIT317` we could capture its interpretation by the following ASTL constraint.

```
*S : [D ! D != <<walk,*X,1>>]
<=
   *S : [D ! D != <<man,*X,1>>
           D != <<every,*X,
                   [S ! S != <<man,*X,1>>],
                   [S ! S != <<walk,*X,1>>],1>>
```

That is the interpretation of the *every* fact effectively quantifies over its first argument (a parameter) treating it like a variable.

Returning to our definition of quantifiers, our lexical entries for the words *"every"* and *"a"* in this schema would be

```
every -
 [DET ! DET != <<cat,DET,Determiner,1>>
        DET != <<use_of,DET,"every",1>>
        DET != <<sem,DET,<<Q1,A1,A2,A3,1>>,1>>
        DET != <<env,DET,EveryEnv::
                  [Env ! Env != <<label,Q1,quantifier,1>>
                         Env != <<anchor,Q1,every,1>>
                         Env != <<label,A1,var,1>>
                         Env != <<label,A2,range,1>>
                         Env != <<label,A3,body,1>>],1>>]

a -
 [DET ! DET != <<cat,DET,Determiner,1>>
        DET != <<use_of,DET,"a",1>>
        DET != <<sem,DET,<<Q1,A1,A2,A3,1>>,1>>
        DET != <<env,DET,AEnv::
                  [Env ! Env != <<label,Q1,quantifier,1>>
                         Env != <<anchor,Q1,some,1>>
                         Env != <<label,A1,var,1>>
                         Env != <<label,A2,range,1>>
                         Env != <<label,A3,body,1>>],1>>]
```

The parameters `A2` and `A3` labelled `range` and `body` will be anchored to the type of the described situation of the nounphrase (without determiner) and the verb phrase. The grammar rule for sentences with quantifiers in their subject noun phrases is

```
[S ! S != <<cat,S,Sentence,1>>
     S != <<tense,S,pres,1>>
     S != <<sem,S,*Qexpr,1>>
     S != <<env,S,*SEnv ::
                 *EnvType &
                 [Env ! Env != <<anchor,*Y,*Var,1>>
                        Env != <<anchor,*Body,*DSType,1>>],
              1>>]
->
 [NP ! NP != <<cat,NP,NounPhrase,1>>
       NP != <<sem,NP,*Qexpr,1>>
       NP != <<env,NP,*Env ::
                    *EnvType &
                     [Env ! Env != <<label,*Body,body,1>>
```

```
                          Env != <<label,*X,var,1>>
                          Env != <<anchor,*X,*Var,1>>],1>>],
      [VP ! VP != <<cat,VP,VerbPhrase,1>>
            VP != <<tense,VP,pres,1>>
            VP != <<described,VP,*DS::*DStype,1>>
            VP != <<env,VP,*VPEnv ::
                           [Env ! Env != <<label,*Y,subj,1>>],
                        1>>
           VP != <<sem,VP,*Fact,1>>].
```

Also we need a rule for quantified noun phrases

```
   [NP ! NP != <<cat,NP,NounPhrase,1>>
         NP != <<sem,NP,*Qexpr,1>>
         NP != <<env,NP,*NPEnv ::
                      *EnvType &
                      [Env ! Env != <<anchor,*X,*A1,1>>
                             Env != <<anchor,*Range,*DSType,1>>],
                   1>>]
   ->
      [DET ! DET != <<cat,DET,Determiner,1>>
             DET != <<sem,DET,*Qexpr,1>>
             DET != <<env,DET,*DetEnv ::
                       *EnvType &
                       [Env ! Env != <<label,*Range,range,1>>
                              Env != <<label,*X,var,1>>],1>>],
      [N ! N != <<cat,N,noun,1>>
           N != <<env,N,*Env ::
                        [Env ! Env != <<label,*A1,arg1,1>>
                               Env != <<anchor,*R1,*pred,1>>],1>>
           N != <<sem,N,<<*R1,*A1,1>>,1>>
           N != <<described,N,*DS::*DSType,1>>].
```

Thus a full analysis of the utterance *"every man walks"* would be,

```
SIT3817

cat(SIT3817,sentence)
tense(SIT3817,pres)
sem(SIT3817, Q1(A1,A2,A3) )

              SIT3819

                anchor(WA1,MA1)
                anchor(A3, walk(WA1) )
                anchor(A1,MA1)
                anchor(A2, man(MA1) )
env(SIT3817,    label(Q1,quantifier)       )
                anchor(Q1,every)
                label(A1,var)
                label(A2,range)
                label(A3,body)

                     SIT4317

                                   P12              P15
                                   P12              P15
described(SIT3817,   every(MA1,                 ,                 )   )
                                   man(MA1)         walk(WA1)
```

There are a number of comments that should be made about the above. Here the parameters used have names based on which word entry introduced them—i.e. `WA1` is introduced by the word *"walks"*. Actually we should really have unique parameters for each use of that word.

However there is a major problem with the above. If you look closely you will see that the above is not quite right. The `WA1` in the situation type whose parameter is `P15` should appear as `MA1` as the anchoring environment states that `WA1` is anchored to `MA1`. The reason it does not is that the sentence rule given above states that the parameter labelled `body` in the quantifier relation should be anchored to the type of the described situation of the verb phrase. At that point the anchoring of `WA1` to `MA1` is not stated so no reduction takes place. This points to a fundamental problem in using simple constraints to model reduction of parametric facts and anchoring environments. Even to get the reductions needed for the STG description given here requires a large number of specific rules. Basically a constraint is needed for each utterance type (sentence, nounphrase, etc.) and for each possible arity of semantic parametric facts. However in order to get the above right a constraint (or more probably a large number of them) would be required that goes further than this and checks not only the fact and its arguments but checks the values within arguments too. For example the key constraint for the basis of a sentence utterance situation is

```
*S : [S ! S != <<described,S,*DS ::
                [DS ! DS != <<*VR1,*VA1,*VA2,*VA3,1>>],1>>]
<=
   *S : [S ! S != <<cat,S,sentence,1>>
             S != <<sem,S,<<*R1,*A1,*A2,*A3,1>>,1>>
             S != <<env,S,*SEnv ::
                        [Env ! Env != <<anchor,*R1,*VR1,1>>
                               Env != <<anchor,*A1,*VA1,1>>
                               Env != <<anchor,*A2,*VA2,1>>
                               Env != <<anchor,*A3,*VA3,1>>],1>>].
```

To get the above example correct we should really have the value of `*VA3` (a type from the described situation of the verb phrase) *also* reduced with respect to the anchoring environment `*SEnv`. Although it may actually be theoretically possible to specify such constraints in ASTL, it is obviously not easy. This fact suggests that some new function should be added to the basic definition of ASTL to cover such reduction. Such an extension is discussed in Section 7.4.1.

As we can see that although tricky in some cases, a basic treatment of STG in ASTL is possible—the full ASTL description is given in Appendix A.3. We can encode the ideas of anchoring and labelling of language and build simple situation semantic translations of utterances. This treatment is reminiscent of using the lambda calculus in Montague grammar where we have an expression with explicitly named parts that have yet to get a value. In contrast with, say, a unification approach where variables are used or as in the next chapter where expressions are built only when all information is available.

STG in its original form does not deal with anaphora. We could try to add some form to it but instead of a designing yet another treatment it seems sensible to borrow from the work of other theories. The next chapter deals with Discourse Representation Theory which has an adequate treatment of both bound and inter-sentential anaphora. The relationship between DRT and STG will be discussed then.

Cooper's description also includes more complex syntactic (and semantic) forms than those available in the Rooth fragment. Particularly it offers simple treatments for embedded sentences and reflexive pronouns. This part is not reconstructed here although should not be very difficult to add.

4.5 Summary

In this chapter we have shown how a situation theoretic treatment of conventional syntax can be given in ASTL. We showed how conventional phrase structure grammars can be naturally encoded as situation theoretic constraints. Although we can explicitly encode these "grammar rules" as constraints, ASTL also offers a built in mechanism for such rules offering a more efficient implementation. A small syntactic fragment is introduced with some examples which is used as the syntactic backbone of the later

Situation Theoretic Grammar description and also will be used in the following two chapters.

A description of Cooper's Situation Theoretic Grammar is given. Although many of the ideas in ASTL come directly from STG is it useful to see that a description of STG, both its syntactic and semantics treatments of utterances can be fully described in ASTL. Some examples and problems with the description are also identified. The above description in ASTL is just to show basic adequacy for ASTL. The following two chapters discuss two other semantic theories which concentrate on the same semantic phenomena and hence descriptions of them can appropriately be closely compared. The STG description does not include a treatment of anaphora and hence cannot be as closely compared but we will consider an extension to STG in Section 7.3 adopting the treatment of pronouns from the following ASTL description of Discourse Representation Theory.

Chapter 5

Discourse Representation Theory and Threading

5.1 Introduction

In this chapter we look at a semantic theory which was originally thought of as being quite distinct from situation theory. Kamp's Discourse Representation Theory (DRT) is a general semantic theory aimed at offering a general semantic representation for natural language discourses ([Kamp 81, Kamp & Reyle 93]). First, we will give a general description of DRT and identify its essential properties. Then we will give a treatment of DRT in ASTL showing how ASTL can be used to describe a non-situation semantic theory. In this description of DRT in ASTL we will introduce and define the concept of *threading* showing how a structure other than the basic syntactic structure may be defined over a discourse.

After the description of DRT in ASTL we compare it with other implementations of DRT. Also we discuss some of the criticisms that have been made of DRT in terms of this new description and see if a situation theoretic treatment offers any advantages.

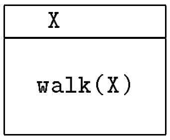
Some of the discussion in this chapter has also appeared in [Black 92].

5.2 Discourse Representation Theory

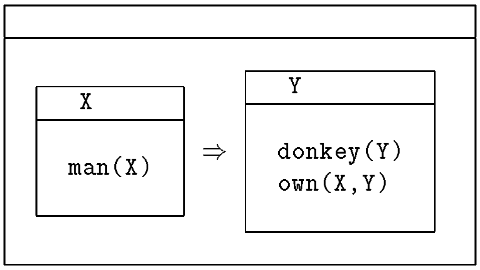
Among the original motivations for DRT was a treatment for the problem of donkey anaphora and a consistent treatment for bound anaphora and inter-sentential anaphora. Tense also played an important role at the beginning. Later work has expanded DRT in a number of different directions, offering solutions to a number of semantic phenomena.

The essential structure in DRT is the *discourse representation structure* (DRS) which is characteristically written as a box. A DRS consists of two parts: a set of *discourse* *markers* called the *domain*; and a set of *conditions*. A DRS is used to represent the current state of information obtained by processing a discourse. As the discourse progresses more information will be entered in the corresponding DRS. Kamp has made claims that an intermediate representation between the syntax and the meaning of an utterance is a *necessary* part of natural language understanding and posits DRSs as that level of representation. Hence DRSs, to him, are not just a convenient form but psychologically real. However, such arguments are not relevant to the description here.

A DRS is usually written as a box with discourse markers in the top part and conditions on these markers on the bottom. A simple example will help illustrate their use. A DRS for the utterance *"a man walks"* could be written as



Discourse markers are used to represent objects introduced in the discourse. In the simplest form of DRT (for example as described in [Johnson & Klein 86]) conditions can be simple predicates over discourse markers or can be the relation $\Rightarrow$ over two sub-DRSs. This relation is used in the representation of the determiner *"every"*. A DRSs for the utterance *"every man owns a donkey"* might be



The interpretation of DRSs is as follows. A DRS is said to be true in a model if there exists a binding for the discourse markers to objects in the model that makes all the conditions true. In the special case of the $\Rightarrow$ condition: it is true if for all ways that the left hand sub-DRS can be made true there exists an extension of the bindings that makes the right hand sub-DRS true.

Another important aspect of DRSs is that they implicitly define an accessibility condition on markers—that is accessibility can be derived from the structure of a DRS. In order for a discourse marker to be a candidate for pronominal reference it must be the case that the marker is *accessible* from the point in the DRS where the discourse marker corresponding to the pronoun is entered. The *accessible* relation can be defined as

- A marker is accessible in the DRS whose domain it appears in.

- All markers accessible in a DRS are accessible in its sub-DRSs.
- All markers accessible in the left sub-DRS of the $\Rightarrow$ relation are also accessible in the right sub-DRS.

The above definition of accessibility allows such anaphoric references as (co-indexing is marked by subscripts)

> *A* man_1 *walks.* He_1 *talks.*
> *Every man with a* $donkey_2$ *likes* it_2.

but (properly) disallows the following (in the case where there is more than one woman—as in $\forall x[man(x) \rightarrow [\exists y[woman(y) \wedge loves(x,y)]]]$ that is wide scope for the universal quantifier introduced by *"every"*)

> $*$*Every man loves a* $woman_3$. She_3 *is happy.*

Specifically the marker introduced by *"a woman"* lies within the scope of the *"every"* and hence is not available for later anaphoric reference.[1]

One basic problem that DRT solves is that in conventional logics a different translation is required for indefinite noun phrases depending on their context. In a simple declarative sentence like "*a man walks*" any translation must introduce some form of existential quantifier for the indefinite noun phrase. A first order logic translation would be

$$\exists x[man(x) \wedge walk(x)]$$

In the case of an indefinite noun phrase embedded within a universal the required translation is different. A first order translation for "*every man with a donkey walks*" is

$$\forall x \forall y\ [[man(x) \wedge donkey(y) \wedge with(x,y)] \rightarrow walk(x)]$$

That is the translation for the indefinite "*a donkey*" introduces a universal quantifier. DRT offers a uniform translation for indefinites in either context (within or outwith the scope of a universal). The DRS for *"every man with a donkey walks"* is

[1] This restriction is often cited in the literature and is done so here as part of the description of DRT. The "closing off" of the scope of a universal quantifier is important in DRT and also, as we will see later in dynamic semantics. However there are good exceptions to these sentences, although it seems that in the starred example there can only be one woman (i.e. wide scope for the existential introduced by *"a woman"*) and not the wide scope for the universal quantifier introduced by *"every"* there are good examples like

> *Every real man owns a* car_1. It_1 *is red.*

which naturally seems to be discussing more than one car.

```
+------------------------------------------+
|                                          |
+------------------------------------------+
|                                          |
|  +------------+                          |
|  |  X Y       |      +-------------+     |
|  +------------+      |             |     |
|  |            |      +-------------+     |
|  | man(X)     |  =>  |             |     |
|  | donkey(Y)  |      |   walk(X)   |     |
|  | with(X,Y)  |      |             |     |
|  |            |      +-------------+     |
|  +------------+                          |
|                                          |
+------------------------------------------+
```

and for *"a man walks"* the DRS is

```
+------------+
|  X         |
+------------+
|            |
| man(X)     |
| walk(X)    |
|            |
+------------+
```

Because DRSs have implicit existential quantifiers on introduced discourse markers and universal quantifiers are not over variables but over DRSs (effectively properties or types) the same translation for indefinites is possible. This is one of the major advantages of DRT.

Another important characteristic of DRT is that it is not just a representation formalism, it also offers a *construction algorithm* which defines how a DRS may be formed from a simple syntactic parse tree. The basic algorithm is specified as a conversion from a syntactic tree to a DRS. The processing is basically done top-down through the tree, rewriting parts of the tree into DRSs. The exact form of construction algorithm changes from implementation to implementation. We can identify three different ways in which the construction is achieved. First there is the "original" method where a syntactic tree is re-written into a DRS. The intermediate structures consist of a DRS box which may contain both conditions and partially converted syntactic trees. The second method is the use of lambda abstraction and application (e.g. in [Pinkal 91]) where each syntactic constituent of a parse is related to an abstraction over a DRS. Lambda application and beta reduction can be used to form the DRS component of the mother node of a syntactic local tree from the DRS components of its daughters. Unfortunately simple lambda application is not sufficient to compose all structures and a special composition operator is also needed. The third method of implementing the construction algorithm is with the use of *threading* (as in [Johnson & Klein 86]). This is the method we also use here and is detailed below. However what should be noted is that in DRT the construction of the representation from a natural language utterance is rightly considered important enough to be part of the theory and not just a footnote.

In the description of DRT in ASTL given in the next section we will only consider the basic parts of DRT. That is just enough coverage to deal with donkey anaphora. Particularly we will deal with simple declarative sentences (and discourses) of transitive and intransitive verbs whose arguments are proper nouns, pronouns or quantified noun phrases optionally followed by prepositional phrases. Only the determiners *"every"* and *"a"* are included. In other words the syntax is exactly that of the Rooth fragment described before in Section 4.3.

This coverage is essentially that in [Johnson & Klein 86]. But it should not be thought that [Johnson & Klein 86] is the most complete description of DRT. That coverage was aimed for here because it is adequate to show the translation of DRT in ASTL (as well as the basic adequacy of DRT itself). There have been many extensions to DRT. Generalised quantifiers in DRT are detailed in [Kamp & Reyle 93]. Propositional attitudes are described in [Kamp 91]. Work on temporal anaphora has also been carried out within a DRT framework ([Partee 84]). Also DRT has been used merely as a framework in which to cast solutions of other semantic problems (for example [Lascarides & Asher 91] on commonsense entailment). This shows DRT is not just a semantic theory for donkey anaphora but does stand as a suitable semantic theory for general semantic representation in its own right.

5.3 DRT in ASTL

There is other work on DRT in situation semantics particularly [Cooper & Kamp 91]. In that description they identify three possible ways in which we can consider DRT in a situation theoretic way which can be paraphrased as:

1. Give a situation semantics for an already existing language for DRSs.
2. Give a model for DRSs as objects in situation theory.
3. Start from an existing situation semantics and incorporate the dynamic aspects of DRT

Although any description of DRT in situation theory/semantics may relate to more than one of these points, the description in [Cooper & Kamp 91] primarily takes the approach given in 1 and give a situation semantics for an existing language of DRSs. Here we will effectively take the second approach and define DRSs as situation theoretic objects.

The following description is in two parts. First we give a description of the representation of DRSs as situation theoretic objects. Second we introduce and formally define a notion of *threading* which relates utterance situations in a way necessary for the construction of DRSs throughout a discourse.

5.3.1 DRSs in ASTL

There are a number of possible ways to represent a Discourse Representation Structure (DRS) in ASTL. The first consideration is the representation of discourse markers. As these are objects which can be bound to objects in the world there is an obvious relationship to a *parameter*. Conditions can be represented as i-terms (facts). DRSs themselves will be treated as parametric situation types. DRSs as types, as we will see below, allows an obvious route to interpretation of DRSs.

An example DRS for the utterance *"a man walks"* is

```
    X
-----------
 man(X)
 walk(X)
```

while the ASTL representation of the same DRS is

```
    P1
-----------
P1 |
 man(X)
 walk(X)
```

In the basic ASTL syntax this would be written as

```
[P1 ! P1 != <<man,X,1>>
      P1 != <<walk,X,1>>]
```

The most important difference between a standard DRS and its ASTL representation is that the discourse markers and conditions are not partitioned. We are treating parametric types effectively as abstractions over types and a more correct representation should perhaps be (using the EKN notation)

```
    P1 X
-----------
P1 |
 man(X)
 walk(X)
```

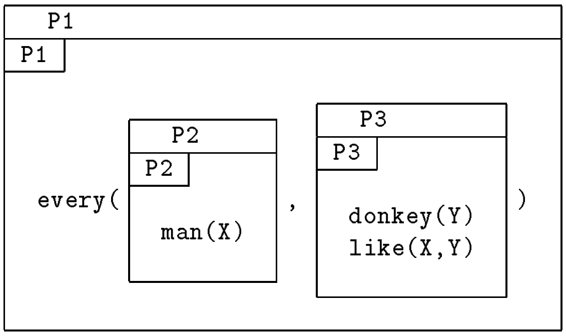
but at present ASTL does not support such objects (an extension that would introduce such objects is discussed in Section 7.4.1). Treating discourse markers explicitly in the abstraction gives a better representation of how we intend to treat them, but it is not necessary.

We can discuss the interpretation of DRSs as parametric types in some model. For example an utterance situation which is related to the above DRS would also be related to a *described situation* (or model). We can capture that relationship by the following ASTL constraint.

```
*S : [S ! S != <<described,S,
                 *DS :: [D ! D != <<man,*X,1>>
                             D != <<walks,*X,1>>],1>>
          S != <<drs-anchor,S,
                 *A :: [AE ! AE != <<anchor,X,*X,1>>],1>>]
<=
   *S : [S ! S != <<DRS,S,[D ! D != <<man,X,1>>
                               D != <<walk,X,1>>],1>>].
```

That is there exists an anchoring for the discourse marker in the DRS such that anchoring the markers in the DRS makes it a type of the described situation. Note that by changing the parameters (discourse markers) into ASTL variables we get the existential treatment of the markers. It should be noted that as ASTL is currently defined the above translation from DRS to described situation cannot be given in general (in ASTL) very easily—you cannot state one constraint to deal with all DRSs—but the concepts of interpretation are expressible as ASTL objects.

Above we only gave an example with conventional conditions, we also treat the special condition used to represent the universal quantifier as a fact. Also if we were to extend this description to include other non-basic conditions of DRT they may also require the introduction of other special facts. An example ASTL DRS containing an `every`-fact for the utterance *"every man likes a donkey"* is



Again we can represent the interpretation of such a fact by ASTL constraints. Again the translation of parameters to ASTL variables gives the required treatment.

```
*S : [S ! S != <<drs-anchor,S,
                *A :: [AE ! AE != <<anchor,X,*X,1>>
                            AE != <<anchor,Y,*Y,1>>],1>>]
<=
   *S : [S !
         S != <<described,S,
                *DS :: [D ! D != <<man,*X,1>>],1>>
         S != <<DRS,S,
                [D ! D != <<every,
```

```
                          [D1 ! D1 != <<man,X,1>>],
                          [D2 ! D2 != <<donkey,Y,1>>
                                D2 != <<like,X,Y,1>>],
                          1>>],
                 1>>].

*DS : [DS ! DS != <<donkey,*Y,1>>
            DS != <<like,*X,*Y,1>>]
<=
   *S : [S ! S != <<described,S,
                    *DS :: [D ! D != <<man,*X,1>>],1>>
             S != <<drs-anchor,S,
                *A :: [AE ! AE != <<anchor,X,*X,1>>
                            AE != <<anchor,Y,*Y,1>>],1>>
             S != <<DRS,S,
                  [D ! D != <<every,
                              [D1 ! D1 != <<man,X,1>>],
                              [D2 ! D2 != <<donkey,Y,1>>
                                    D2 != <<like,X,Y,1>>],
                              1>>],
                  1>>].
```

That is for all ways that we can anchor `X` to something that is a man in the described situation (i.e. all ways that the first argument of `every` can be made a type of the described situation) there exists an anchoring for `Y` to a donkey which is liked by the anchor of `X`. This will mean that an utterance situation may be related to a number of anchoring environments but it will still only be related to one described situation. Also note the obvious parallel here with the semantic representation used in the STG description in the previous chapter (page 70)

In addition to a DRS we will also relate utterance situations to an accessibility situation. This will identify all discourse markers which are currently accessible. In some ways it may be thought of as the domain, in that its facts are all about domain markers. Accessible markers are used primarily for identifying antecedents for pronouns hence we also related them to a type, one of `male`, `female` or `neuter` reflecting the gender of English pronouns. However unlike the domain of a standard DRS the accessibility situation may also include markers from the domains of DRSs which are accessible from the current one. More will be said on this construct later in Section 5.3.4.

5.3.2 Threading

Before we can give the definition of how DRSs are related to each other at each stage in a discourse we must introduce the concept of `threading`. Conventionally we think of an utterance being made up of a single hierarchical tree of syntactic categories, but here we wish to specify other structures over the same set of utterance situations.

The general idea is that as a discourse progresses a new DRS is constructed from the DRS of the previous part of the discourse plus information form the current part of the utterance. The path of utterance situations over which this DRS is extended is called a *thread*. Threads are defined by the binary relation `t-in`. For example given the following relation

```
<<t-in,S1,S2,1>>
```

we would say `S1` is *threaded* to `S2`. The *daughter* relation defines a syntactic tree over the utterance situations stating immediate dominance and linear precedence of the parts of the utterance, while the threading relation states a different structure over the same set of utterance situations.

Each utterance situation appears exactly once as the second argument to the `t-in` relation. That is each utterance situation has exactly one incoming thread. There is one exception to this, the special utterance situation which is used to denote the start of a discourse. It is basically a null context and is used at the initial thread of a discourse (and the start of some sub-threads). There are no cycles in the threads but as we will see there may be more than one thread in a discourse. The actual construction of the threads will be discussed later (Section 5.3.3).

Although we have not yet fully defined threading, in order to justify it as a useful structure to define over utterances we will show how it can be used to define DRSs for a set of utterance situations. The basic idea in DRT is that as a discourse progresses information is added to a DRS about the content of the current utterance. We can define this relationship using threading. Basically each utterance situation is related to two DRSs, through the relations `DRSIn` and `DRSOut`. In addition the simple type of the DRS we also relate each utterance situation to an incoming and outgoing accessibility situation which supports facts about which discourse markers are accessible in that utterance situation. An incoming DRS for an utterance situation is the outgoing DRS for the utterance situation previous in the thread. This is represented by the following constraint.

```
*S:[S ! S != <<DRSIn,S,*Access,*DRS,1>>]
 <=
   *TS:[TS ! TS != <<t-in,*S1,*S,1>>],
   *S1:[S1 ! S1 != <<DRSOut,S1,*Access,*DRS,1>>].
```

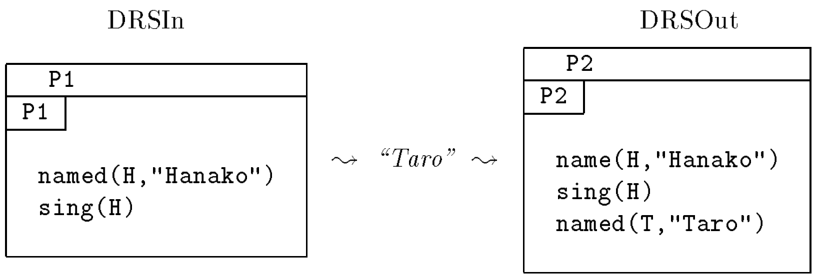
The outgoing DRS of an utterance situation is the information in its incoming DRS *plus* the information contributed by that part of the discourse itself. In the simplest case consider a proper noun. A constraint can be written showing the relationship between the incoming and outgoing DRSs.

```
*S:[S ! S != <<DRSOut,S,
             *A :: *AType &
```

```
                    [A ! A != <<accessible,*X,*TYPE,1>>],
              *DRSIn &
               [D ! D != <<named,*X,*N,1>>],1>>]
 <=
  *S:[S ! S != <<cat,S,ProperNoun,1>>
         S != <<use_of,S,*N,1>>
         S != <<sem,S,*X,1>>
         S != <<type,S,*TYPE,1>>
         S != <<DRSIn,S,
               *A1 :: *AType,
               *DRSIn,1>>].
```

That is we add the DRS condition `named` for that proper noun. The output DRS is defined to be an extension of the input DRS. The output is the type of the input (via the variable `*DRSIn`) plus the new fact about the proper noun. Secondly we have also added the discourse marker for the proper noun to the outgoing accessibility situation, stating that that proper noun is available as an antecedent.

Consider the discourse *"Hanako sings. Taro dances"*. Particularly consider the utterance situation representing *"Taro"*, the incoming DRS would be transformed as follows



Information is monotonically increasing in DRSs as we traverse along a thread. Note that we do not modify the incoming DRS but specify a new DRS with the type of the incoming DRS plus information that may be added at that point.

We also have the condition that any argument or relation that appears in a DRS condition must be related to some utterance situation by the relation `sem` previously in the thread. The consequence of this is that arguments are threaded before predicates—even though they may syntactically appear in a different order. Thus object noun phrases must be threaded before the predicate.

Although the DRT description in [Johnson & Klein 86] also includes a notion of threading the description given here is a little different. We have tried to abstract the notion of threading such that we have thread relations independent of the representation of DRSs. DRSs are defined in terms of threading but the threads exist without the DRSs. In [Johnson & Klein 86] each syntactic component also consists of an incoming DRS and outgoing DRS but the basic threads still directly follow through the syntactic structure, in a depth-first left-to-right manner. This is not the case here, although we do hold "threading" information on each syntactic component about its constituents the `t-in` relation itself defines a structure independent of the syntactic one. It could be said that here we thread the threads.

Unfortunately threading is not quite as simple as is described above. When only simple verbs, proper nouns and pronouns occur there exists only one simple thread through all the utterance situations. But with the introduction of quantifiers the threading must be a little more complex. There is a third form of structural relation apart from `daughter` and `t-in`. Each determiner utterance situation appears in exactly one `range`-relation and one `body`-relation. The second argument of each of these relations is an utterance situation that does not appear as a first argument to any `t-in`-relation (i.e. they are ends of sub-threads). The relationship between the outgoing and incoming DRSs related to a determiner utterance situation is that the outgoing DRS includes the incoming DRS plus the information from the sub-threads. In the case of the determiner *"every"* we have the following relation

```
*S:[S ! S != <<DRSOut,S,
                *Access,
                *DRSIn &
                 [DS ! DS != <<every,*RangeDRS,
                                     *BodyDRS,1>>],1>>]
<=
  *S:[S ! S != <<cat,S,Determiner,1>>
          S != <<DRSIn,S,*Access,*DRSIn,1>>
          S != <<sem,S,every,1>>],
  *TS:[TS ! TS != <<body,*S,*Body::
                  [S ! S != <<DRSOut,S,*A1,*BodyDRS,1>>],1>>
            TS != <<range,*S,*Range::
                  [S ! S != <<DRSOut,S,*A2,*RangeDRS,1>>],1>>].
```

That is we add an `every` condition to the outgoing DRS whose arguments are the outgoing DRSs of the utterance situations related by the relations `range` and `body`. Note that the incoming accessible markers are simply passed forward unchanged as markers introduced within the scope of the `every`-relation are not available for pronominal reference.

The indefinite article is actually easier (no sub-DRSs need be created). The following constraint captures the relationship. In the indefinite case the outgoing DRS consists of the incoming DRS plus the outgoing DRSs of the range sub-thread and of the body sub-thread. The accessible markers from the end of the range sub-thread are passed on out of the determiner utterance situation because unlike the `every` case, markers introduced within the scope of the indefinite are available for future pronominal reference.

```
*S:[S ! S != <<DRSOut,S,
```

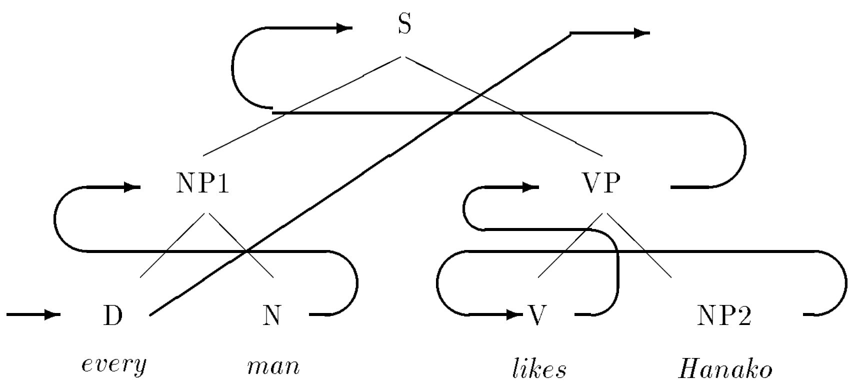
```
                 *Access,
                 *DRSIn & *DRSRange & *DRSBody,1>>]
   <=
     *S:[S ! S != <<cat,S,Determiner,1>>
             S != <<DRSIn,S,*A1,*DRSIn,1>>
             S != <<sem,S,some,1>>],
     *T1:[TS ! TS != <<body,*S,*Body::
                        [S ! S != <<DRSOut,S,
                                  *A2,*DRSBody,1>>],1>>],
     *T2:[TS ! TS != <<range,*S,*Range::
                         [S ! S != <<DRSOut,S,
                                   *Access,*DRSRange,1>>],1>>].
```

However what is actually done in the description here (as shown fully in Appendix A.4) is that the beginning of the body sub-thread is threaded to the end of the range sub-thread and the beginning of the range sub-thread is threaded to the incoming thread of the determiner itself. This means the following simpler constraint achieves the same result.

```
   *S:[S ! S != <<DRSOut,S,*Access,*DRSOut,1>>]
   <=
     *S:[S ! S != <<cat,S,Determiner,1>>
             S != <<sem,S,some,1>>],
     *T1:[TS ! TS != <<body,*S,*Body::
                        [S ! S != <<DRSOut,S,
                                  *Access,*DRSOut,1>>],1>>].
```

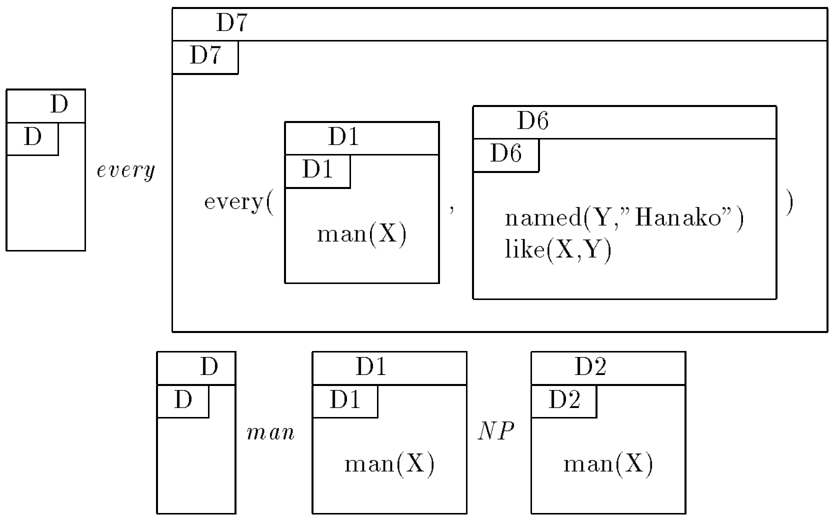
As it is not completely clear from the above examples how the syntactic structure relates to the threaded structure a few simple but detailed examples are given.

Below is an annotated syntactic tree for *"Every man likes Hanako"* showing the threading relation. The `t-in` relation is shown as bold arrows.

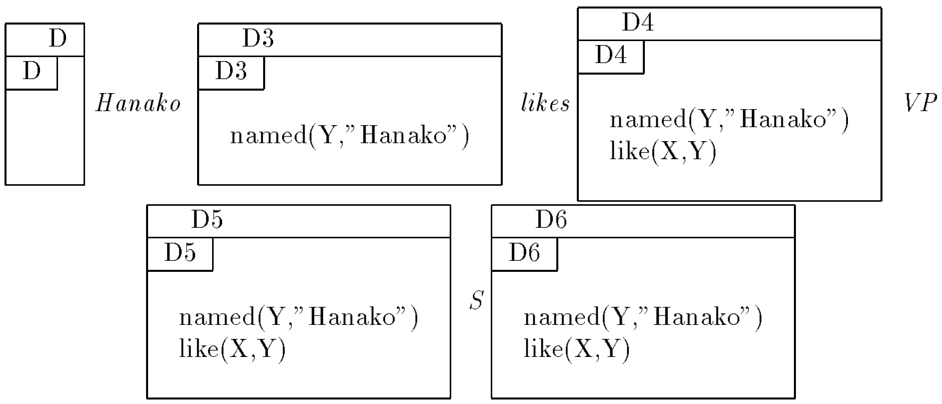


In addition, `DS` (the discourse start situation) is threaded to `D`, `N` and `NP2`. The main discourse thread will go through `D`. There are two other threads ending at `NP1` and `S`. `D` will be related to `NP1` by the relation `range` and to `S` by the relation `body`.

That is there are three threads in this discourse. The main thread goes from the previous utterance in the discourse through the determiner *"every"* and on to the next utterance. Two sub-threads also exist dealing with the range and body of the determiner. DRSs are defined with respect to these threading relations. Abstractly the DRS progression through these threads can be shown as



and finally the body sub-thread



The obvious question is how come the outgoing DRS of the sentence utterance situation does not contain any information about the determiner. The next section describes how these threads are built and the exact relationships between each utterance situation.

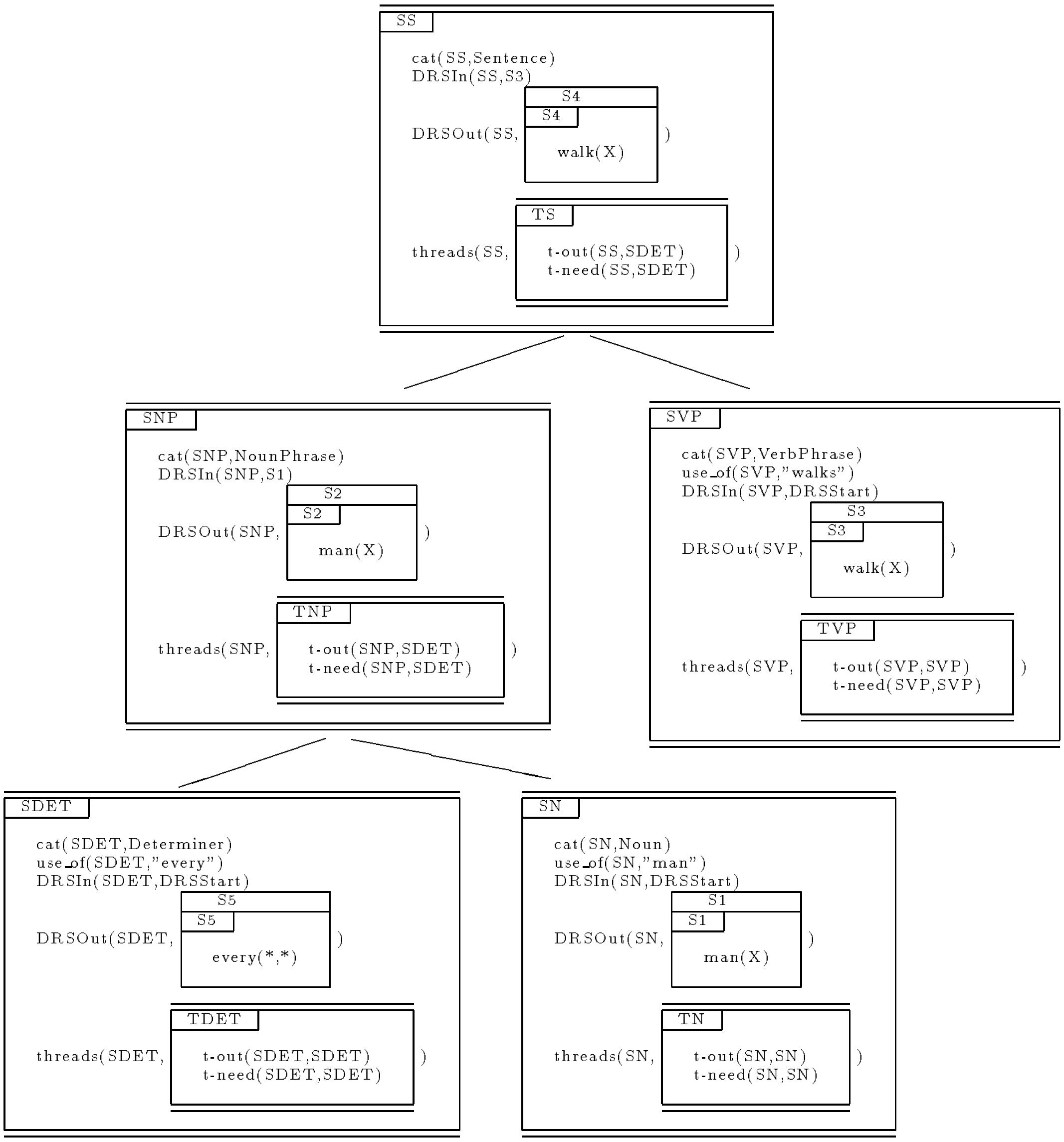
5.3.3 Constructing the threading information

The threading structure is defined with respect to the syntactic structure such that the grammar rules include constraints about how the utterance situations can be threaded. Each utterance situation is related to a "threads" situation which holds the facts that are specific to threading. Note that a `t-in`-relation for a particular utterance situation may not be locally determined and hence the `t-in`-relation for a particular utterance may not be in the "threads" situation for that utterance or even in the utterance situation immediately dominating it. For example the incoming thread for a subject proper noun will be some utterance situation in the previous sentence. The `t-in`-relation for that proper noun will be determined somewhere further up the syntactic hierarchy. Thus there will be a number of "thread" situations around and any one of them may contain the `t-in`-relation for a particular utterance but the conditions stated above for these relations will still be true—only one incoming thread for each utterance situation.

In addition to the actual threading information there are other relations supported by "thread" situations. The `body` and `range` relations are used to relate determiner utterance situations to their sub-threads. Also there are a number of relations which are used in the construction of the `t-in` relations. Facts with any of the relations `t-out`, or `t-need` may also appear in "threads" situation. All utterance situations' thread situation will contain a `t-out` and `t-need` relation. The `t-out` relation identifies an utterance situation which is syntactically dominated by the current utterance situation. This `t-out`-situation is the last situation in the thread in that sub-tree. The `t-need` relation identifies the utterance situation that requires an incoming thread. This is best illustrated by looking at the information in the threading situation related to a sentence utterance situation for the sentence *"Hanako sings"*.

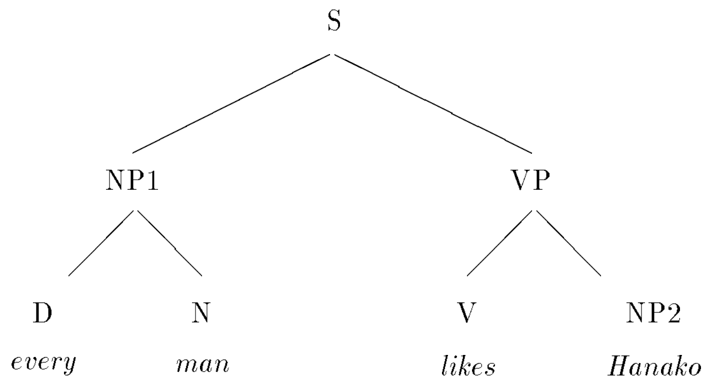
```
SIT1

  cat(SIT1,sentence)
  sem(SIT1, R1(A1))
  daughter(SIT1,SITNP)
  daughter(SIT1,SITVP)
                  TS

  threads(SIT1,     t-out(SIT1,SIT1)      )
                    t-need(SIT1,SITNP)
```

That is the `t-need` of the sentence utterance situation (the situation that needs a thread) is the proper noun and the output, the situation that is to be threaded to the next part of the discourse is the sentence utterance situation itself. In the case of a sentence containing a quantifier the threading is different. In the case of *"every man walks"*, the `t-out` and `t-need` of the sentence is the determiner as it is that utterance which adds the single condition (with relation `every`) to the DRS. That is the main discourse thread goes through the determiner utterance situation.

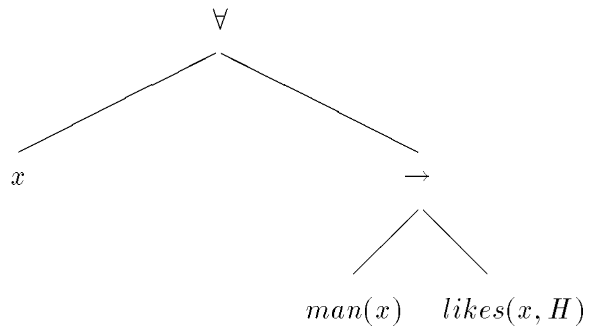


In [Johnson & Klein 86] they also have a notion of threading. A comparison between their implementations may help this description. In J&K each syntactic component (among other features) has in fact two incoming DRSs and two outgoing DRSs. They call them *current* and *super*. In the ASTL implementation the standard incoming and outgoing DRS are analogous to same as J&K's *current* DRSs, while `t-need` and `t-out` relations exist in some way to capture the information made available in J&K's *super*. Basically where J&K pass down the input DRS to an utterance, we pass up the utterance that requires the input. J&K (and others) offer a top-down definition of the construction algorithm while the ASTL description is essentially bottom-up.

The apparent complexity of the threading is directly to do with the fact that the syntactic structure of an utterance does not closely correspond to the semantic structure—particularly in terms of of how quantifiers are scoped. The syntactic structure of the utterance *"every man likes Hanako"* can be written as



while the logical structure is more like



which puts the quantifier node (which is related to the syntactic determiner node) as the dominating node. As we are building a logical structure which is in essence closer to the second from the first we must find ways to related the nodes of one to the other. The threading relation is intended to capture this alternative structure.

5.3.4 Pronouns and accessibility

So far our examples have not included any pronouns. The basic constraint for a pronoun is that it can only refer to something that has already "appeared" in the discourse. In DRT a pronoun introduces a new discourse marker but it also adds a condition relating this new marker to an already existing *accessible* marker. Accessibility is defined in terms of the incoming DRS (and the DRSs of which this may be a sub-DRS). Because accessible markers are defined in terms of existing markers, pronoun referents (in DRT) must appear earlier in the discourse and hence standard definitions of DRT cannot deal with cataphora. The basic notion of pronoun use in our ASTL description can easily be captured by the following constraint.

```
*S:[S ! S != <<DRSout,S,
                *A,
                *DRSIn &
                [DS ! DS != <<is,*X,*Y,1>>],1>>]
<=
 *S:[S ! S != <<cat,S,Pronoun,1>>
         S != <<type,S,*TYPE,1>>
         S != <<sem,S,*X,1>>
         S != <<DRSIn,S,
                 *A::[A ! A != <<accessible,*Y,*TYPE,1>>],
                 *DRSIn,1>>].
```

That is we add an `is`-relation for the two discourse markers, one that was introduced by the pronoun itself and the other some marker of the right type that is accessible from the current context. In DRT the accessibility relation is defined as in Section 5.2 above. In the ASTL description, each utterance situation in addition to a DRS, is related to an accessibility situation which supports facts about which discourse markers are accessible at the point in the discourse. We can check the accessible situation to find candidate antecedents.

The accessible situation is added to when any new discourse marker is introduced, (i.e. by a common noun or proper noun). However it should be noted that the accessible situation will not just contain the markers that have been introduced in the current DRS, it will also contain markers from the DRSs which the current DRS is contained within. For example the incoming assignment situation for the utterance situation representing *"it"* is the utterance *"every man with a donkey likes it"* will be

```
ACC1
 accessible(X,male)
 accessible(Y,neuter)
```

even though the incoming DRS for that utterance situation will be the null situation type representing the start of the sub-DRS that will be the second argument to the `every` relation in the final representation of the whole sentence.

The whole ASTL DRT description is given in Appendix A.4. We can summarize the description by identifying the following parts:

The basic Rooth fragment provides the syntactic backbone. The grammar rules define a syntactic structure over a set of utterance situations using the `daughter` relation.

A set of `t-in` relations defining a threading structure through the same set of utterance situations. Each utterance situation is related to one incoming thread. Determiner utterance situations are be related to range and body sub-threads.

Each utterance situation is related to an incoming DRS and outgoing DRS. The outgoing DRS is defined as the incoming DRS *plus* information contributed by that utterance situation. The DRSs are defined over the threading relation.

Each utterance situation is also related to a situation which supports facts about which discourse markers are *accessible* as potential referents for pronouns.

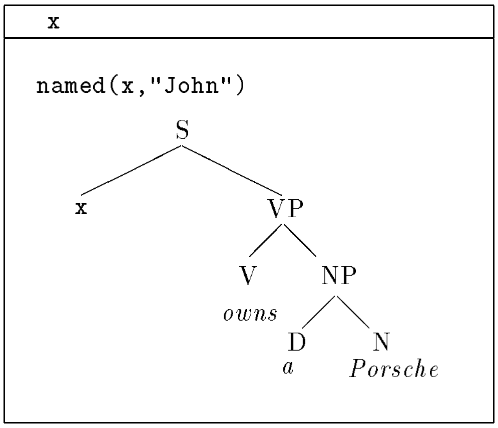
Each utterance situation is related to an utterance situation by the `t-out` relation. An utterance situation's `t-out` situation is either itself or one it syntactically dominates. The outgoing DRS of the `t-out` situation is a representation of the discourse after that whole syntactic tree has been processed.

We will see in Chapter 6 that the description of dynamic semantics in ASTL reuses the same syntactic fragment and threading relation as the DRT description. Only the constraints dealing with the definitions of DRSs and accessibility change. This shows that threading is a worthwhile notion to describe abstractly unlike other implementations which only represent it implicitly.

5.4 Other instantiations of DRT

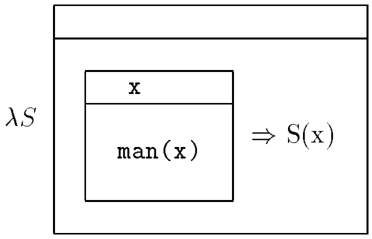
There are other instantiations of DRT both in description and actual implementation. It is worth looking at these with a view to comparing them to the ASTL description described above.

First we will compare the various methods of building a DRS from a syntactic structure. In [Kamp 81] and [Kamp & Reyle 93] the construction algorithm works by rewriting a syntactic tree into a DRS. For example we can see how this works in the following intermediate structure. Using the definitions in [Kamp & Reyle 93], after processing the subject noun phrase in the sentence *"John owns a Porsche"* the DRS would be.

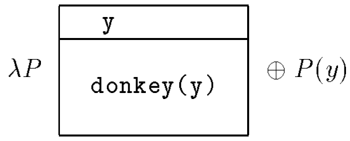


Note how this intermediate structure is not a valid DRS (or tree) itself, but is some combination of both. Various rules are defined which rewrite the tree into the DRS box structure.

The implementation described in [Pinkal 91] is different. Their implementation treats seriously the notion of compositionality. For each syntactic component of a discourse there is a related DRS object. Unlike the ASTL DRT description that DRS object only contains information from the parts the node syntactically dominates. (In the ASTL description the `t-out` DRS will contain information up to that point in the discourse so may contain information from syntactic nodes that are not dominated by that node but appear earlier in the discourse.) The DRS objects in Pinkal's implementation are not simple DRSs. They are essentially lambda abstractions over DRSs. For example the DRS representation for the noun phrase *"every man"* would be



And for the indefinite noun phrase *"a donkey"* would be



The operator $\oplus$ is used to merge two DRSs. Also beta reduction in this framework is not quite the same as in the conventional lambda calculus. A DRS object, may, because it is an abstraction, contain unresolved pronouns. It is necessary for any reduction function to check that unresolved pronouns are either appropriately resolved by the reduction or their conditions are passed up for further reductions. A valid DRS for a discourse must have all its pronouns resolved. The checking of pronouns and their possible resolutions must also be part of the definition of the $\oplus$ DRS merge operator. The result is that Pinkal's system gives a compositional treatment of DRT—"compositional" in its classic sense of a semantic representation for each syntactic component a simple compose function (albeit slightly more complex that lambda application and beta-reduction). Their ideas of a compositional DRT are similar to [Zeevat 89]. Pinkal's system has many other aspects, it is not solely designed as an example of DRT it is designed as a general natural language processing system. The system are also concerned with quantifier scope and have added a version of Cooper Storage, and an algorithm for finding possible scopings based on [Hobbs & Shieber 87].

The third treatment of the construction algorithm in DRT, apart from the one presented earlier in this chapter, is that described in [Johnson & Klein 86]. As a syntactic backbone they use a simple DCG. Like the ASTL description, J&K use the technique of threading—in fact the Johnson and Klein description was used as a base in designing the ASTL description. However, J&K thread differently from our own description. Specifically DRSs are build up as the parse is made, from left to right. Therefore at certain points in the construction of the DRS the representation at a node may not be a valid DRS (in the strict sense). For example at the verb node in an analysis of the utterance *"a man likes a donkey"* the DRS object would be

`x`
`like(x,*Y)` `man(x)`

where `*Y` is a variable. The problem is that because `*Y` is some meta-variable rather than an object within the DRS language itself, this structure has no denotation. This in itself may not be considered a problem but if a formal semantic view is taken this is an issue. It may be possible to recast such free meta-variables in structures into some form of lambda expressions—probably similar to Pinkal's system.

This description of others' interpretation of the construction algorithm enables us to look closer at the one used in the ASTL description. In the ASTL definition given above all DRSs related to utterance situations (either by the `DRSIn` or `DRSOut` relation) are valid DRSs. They do not contain any meta-variables or any form of lambda abstractions. At least not as part of their "structure"—discourse markers are represented by parameters which could be considered a form of meta-variable but that is considered different. This fact could be a basis for an argument that the description of DRT in ASTL is not compositional in the strict sense. The DRS related to an utterance situation may contain information from both what it syntactically dominates and whatever appears before it in the discourse. However this does not seem to be wrong. The semantics of an utterance does naturally seem to depend on not just its sub-parts but also the context it appears in.

Although a basic idea of DRT is that utterances transform some input context to some output context plus information from that utterance, it is not always made explicit in implementations. However the J&K description does make this concept more explicit. The ASTL description of DRT because of its explicit representation of threading also takes the same viewpoint. As we will see in the next chapter on dynamic semantics the idea of changing context through a discourse is common to both DRT and dynamic semantics and we show how this can be abstracted from both semantic descriptions and shared between them. It could be said that of the three other DRT descriptions, [Kamp & Reyle 93], [Pinkal 91] and [Johnson & Klein 86] described above, J&K could be said to be the most dynamic (in the dynamic semantic sense) because of the idea of incoming and outgoing DRSs is built in while the ASTL description is (deliberately) even more so.

There is also a question of incrementality. That is is there a DRS for all initial substrings of a discourse. With such a strict definition of incrementality on the word level it has to be said that DRT in general is not incremental, as there cannot be a DRS representation for the substring *"every man"* without some concept of abstraction over DRSs. However DRT, and the implementations discussed here, including the ASTL one, is incremental at the sentence level. That is a DRSs exists (and can be calculated) for each initial substring of sentences of a discourse.

An important aspect of DRT is that it is not only a representation for discourses but also a mechanism that can build these representations from syntactic parse trees. With respect to the construction algorithm it seems necessary for it to be carried out in a left to right manner—markers must be introduced before they can be referred to by pronouns. This processing aspect, specifically the order of processing, is effectively included in the ASTL description. This is not achieved by defining an algorithm or procedure but because of the dependencies of the constraints used to definition DRT in ASTL we have given a declarative definition which requires a left to right ordering.

5.5 Summary

In this chapter we have described Discourse Representation Theory (DRT) which offers a semantic theory for natural language utterances. A simple description is given followed by a description of how DRT can be defined within ASTL. The basic representational object in DRT is the Discourse Representation Structure (DRS). DRSs are modelled as parametric situation types in ASTL. This seems a natural translation which allows an easy method for interpretation of DRSs. A system of *threading* is detailed which defines a different structure over a set of utterance situations. Thus a syntactic structure is defined by the grammar rules (though the `daughter` relation) while threading offers a structure closer to the logical structure of the same utterance. Two DRSs are defined for each utterance situation, one an input and the other an output. The output DRS contains the information from the input DRS plus information added to the discourse by that utterance situation. The DRSs are defined on top of the threading relation which states what order the utterance situations must be taken to correctly build the DRS. An accessibility condition is also defined allowing for the same possible pronoun resolutions as in conventional DRT.

A comparison of this ASTL description of DRT and other instantiations of DRT is given. Particularly showing how *compositionality* is treated in each of the systems. Although the ASTL description does not offer strict compositionality it does provide a complete DRS for each part of the utterance. Other aspects of a situation theoretic treatment of DRT are also discussed.

This chapter shows that ASTL is not just suitable for describing situation semantic theories of natural language such as STG (as shown in Chapter 4) but also suitable for describing other theories which, at least before, were considered quite different.

Chapter 6

Dynamic Semantics and Situation Theory

6.1 Introduction

In this chapter we discuss dynamic semantics, in particular the work of Groenendijk and Stokhof on Dynamic Predicate Logic (DPL) and Dynamic Montague Grammar (DMG) ([Groenendijk & Stokhof 91b, Groenendijk & Stokhof 91a]). As an alternative to the previous two chapters, where we have been looking at how to encode semantic theories for natural languages in ASTL, here we will, at first, be looking at how to encode a logic within ASTL. Dynamic Predicate Logic (DPL) is a simple first order logic which displays the basic properties of dynamic logics. By encoding it in ASTL we will see how its concepts relate to situation theory. Later we define DPL-NL which gives a dynamic semantic treatment for the Rooth language fragment. This description is used to show the differences between DRT and a dynamic semantic treatment of the same phenomena.

First we will give some background and justification for the work in dynamic semantics followed by a formal description of Dynamic Predicate Logic (DPL). We then show how this logic can be best encoded within ASTL. Next we introduce DPL-NL which offers a dynamic treatment of the same syntactic fragment we used in the previous chapters. DPL-NL deliberately re-uses basic parts of the DRT description described in the preceding chapter. This leads to some detailed discussion of where DRT and dynamic treatments differ in compositionality and representation.

6.2 Background and justification

Dynamic Predicate Logic (DPL) was developed in response to Discourse Representation Theory (DRT) as an attempt to create a "logical" theory that approximates the semantic coverage of DRT. Specifically DPL aims to be a *compositional* and *non-*

representational theory for the same semantic phenomena covered by DRT. We will say more about what is meant by the phrases *"compositional"* and *"non-representational"* during this section.

The basic intuitions of DPL are based on the work done in the formal semantics of (computer) programming languages (see [Harel 84]). The general idea is that an instruction in a programming language transforms one state (assignment of values to variables) to another. For example, a "program" {`x := x + 1`} can transform an input state g to an output state h that differs only from g such that the value of x in h is 1 larger than the value of x in g.

It is this changing of state that is the reason for the term *dynamic*. In this framework the denotation of a "program" is a set of pairs of states (assignments) each of which are valid inputs and output states for the "program".

A conventional predicate logic (PL) treatment of an utterance like *"A man$_1$ walks. He$_1$ talks."* would be $\exists x[man(x) \land walk(x) \land talk(x)]$. The problem with this, argued by Groenendijk and Stokhof, is that there is no simple representation for the first sentence *"A man walks"* in that the scope of the quantifier extends outside the expression used to represent that sentence. That is the existential quantifiers introduced by indefinite noun phrases should not be "closed off" at the end of a sentence but extend over the rest of the discourse (to allow for later anaphora). The problem is that there is no simple complete logical expression representing a sentence. Thus they argue that the conventional (PL) translation of the utterance *"A man$_1$ walks. He$_1$ talks"* as

$$\exists x[man(x) \land walk(x) \land talk(x)]$$

is not as good, from the compositional viewpoint, as

$$\exists x[man(x) \land walk(x)] \land talk(x)$$

which we will see later is the DPL translation. Note that in the second example the second occurrence of x apparently lies outside the scope of the existential.

A second example is with the sentence *"every man who owns a donkey beats it"*. The PL translation is

$$\forall x \forall y[[man(x) \land donkey(y) \land own(x, y)] \rightarrow beat(x, y)]$$

which is obviously not compositional as there is no obvious sub-expression which could be seen to come from the relative clause *"who owns a donkey"*. The DPL translation

$$\forall x[[farmer(x) \land \exists y[donkey(y) \land own(x, y)]] \rightarrow beat(x, y)]$$

does offer an apparently more compositional analysis. However, this of course requires a redefinition of the semantics of such expressions.

It is also argued that although DRT offers a treatment of such sentences (particularly donkey-anaphora without the need for a universal quantifier for the translation of the indefinite in the relative clause), DRT is still not compositional in the Groenendijk and Stokhof sense. Translations of the above two example utterances in DRT would be

```
+-----------+
|  X        |
+-----------+
|           |
|  man(X)   |
|  walk(X)  |
|           |
+-----------+
```

and

```
+-------------------------------------------+
|                                           |
+-------------------------------------------+
|                                           |
|  +------------+       +------------+      |
|  |  X Y       |       |            |      |
|  +------------+       +------------+      |
|  |            |  =>   |            |      |
|  | man(X)     |       | beat(X,Y)  |      |
|  | donkey(Y)  |       |            |      |
|  | own(X,Y)   |       +------------+      |
|  |            |                           |
|  +------------+                           |
|                                           |
+-------------------------------------------+
```

But these also do not contain sub-expressions corresponding to *"a man"* and *"who owns a donkey"* respectively.

Apart from compositionality the other explicit goal of DPL is, in contrast with DRT, to give a *"non-representational"* theory. DRT, [Kamp 81], claims that DRSs are not just structures generated during the process of analysing discourse but that DRSs are structures that are psychologically real in human cognitive processing terms and hence are necessary in such an analysis. DRSs are an intermediate representation between the syntactic representation and the actual semantics, DPL claims that this intermediate representation is not necessary (though perhaps useful in an implementation?).

6.3 Definition of DPL

The syntax of DPL is almost the same as that for standard first order predicate logic. It differs only in the definition of open and closed formulae. The semantics is however very different. According to [Groenendijk & Stokhof 91b] an expression in DPL denotes a set of pairs of assignments, where an assignment is a function from DPL variables to individuals.

A model M for DPL is a pair $\langle D, F\rangle$ where D is a non-empty set of individuals and F is a function from constants to members of D and predicates to sets of n-tuples of D. If α is a constant then $F(\alpha) \in D$. If α is a n-place predicate then $F(\alpha) \subseteq D^n$. An assignment g is a function from variables to individuals: $g(x) \in D$. $[\![\ t\ ]\!]_g = g(t)$ if t is a variable and $[\![t]\!]_g = F(t)$ if t is a constant. We will write $k[x]g$ to mean that k and g are assignments, where k differs from g only in that k specifies an assignment for x to some individual in D.

We can now define the semantics of terms in DPL

$$
\begin{array}{l}
[\![R(t_1,\ldots,t_n)]\!] = \{\langle g,h\rangle \mid h = g \wedge \langle [\![t_1]\!]_h,\ldots,[\![t_n]\!]_h\rangle \in F(R)\} \\
[\![\neg\phi]\!] = \{\langle g,h\rangle \mid h = g \wedge \neg\exists k:\ \langle g,k\rangle \in [\![\phi]\!]\} \\
[\![\phi \wedge \psi]\!] = \{\langle g,h\rangle \mid \exists k:\ \langle g,k\rangle \in [\![\phi]\!] \wedge \langle k,h\rangle \in [\![\psi]\!]\} \\
[\![\phi \vee \psi]\!] = \{\langle g,h\rangle \mid h = g \wedge \exists k:\ \langle g,k\rangle \in [\![\phi]\!] \vee \langle g,k\rangle \in [\![\psi]\!]\} \\
[\![\phi \rightarrow \psi]\!] = \{\langle g,h\rangle \mid h = g \wedge \forall k:\ \langle g,k\rangle \in [\![\phi]\!] \Rightarrow \exists j:\ \langle k,j\rangle \in [\![\psi]\!]\} \\
[\![\exists x\phi]\!] = \{\langle g,h\rangle \mid \exists k:\ k[x]g \wedge \langle k,h\rangle \in [\![\phi]\!]\} \\
[\![\forall x\phi]\!] = \{\langle g,h\rangle \mid h = g \wedge \forall k:\ k[x]g \Rightarrow \exists m:\ \langle k,m\rangle \in [\![\phi]\!]\}
\end{array}
$$

Although not immediately obvious from the definition above the semantics does allow expressions like

$$\exists x[man(x)] \wedge walk(x)$$

to have the desired treatment where the second occurrence of x does in fact lie within the scope of the existential. Assignments are effectively threaded through an expression thus the values bound to variables within the syntactic scope of an existential are still available later in the analysis. The consequences of such a semantics are somewhat different from conventional logics. The conventional logical equivalences are no longer necessarily the same. Notably, conjunction is no longer commutative.

Unlike DRT, DPL does not define a translation from natural language utterances to logical form (that is there is no *"construction algorithm"*). It only deals with the logical form itself. Although a natural language gloss is typically given for DPL expressions no translation algorithm is given from one to the other. Also the natural language glosses already have their pronouns resolved. Thus, a typical natural language gloss for a DPL expression would be

A man_1 *walks. He*$_1$ *talks*
$$\exists x[man(x)] \wedge walk(x) \wedge talk(x)$$

Admittedly DPL is only the first step towards a logical treatment of quantification and anaphora and other extensions have been discussed. Dynamic Intensional Logic (DIL) extends DPL to cover (a form of) intensional logic. Dynamic Montague Grammar (DMG), [Groenendijk & Stokhof 91a], is a major step in the direction of a semantics and a relation to natural language syntax. Some discussion of DMG will be given later.

6.4 DPL in ASTL

There are a number of ways in which DPL could be translated into ASTL. The first and most obvious is perhaps not the best but some mention of why may be interesting.

Given a DPL expression

$$\exists x[man(x)] \wedge walk(x)$$

we could directly translate it into an ASTL representation of the form

```
<<and,<<exists,X,<<man,X,1>>,1>>
      <<walk,X,1>>,1>>
```

However this would require definitions of the (dynamic logic) relations `and`, `exists`, etc. To give a semantics for such relations within ASTL would rely a lot on the programming properties rather than the logical properties of ASTL. As we would like DPL expressions in ASTL to have a fairly natural semantics (with respect to the current semantics of ASTL) an alternative representation would be better. But as the semantics of DPL is so radically different from classical logic semantics (and even situation theory) it seems we cannot have both a natural translation for DPL expressions into ASTL and a natural semantics. Therefore instead of representing the DPL expressions themselves in ASTL we can represent the meta-language description of the expressions. In the previous section we gave the semantics for DPL expressions. The language used to describe the semantics is not dynamic but closer to the semantics of classical logic and ASTL. Therefore we can give a representation for DPL expressions in ASTL but not as the expressions themselves but as the meta-language equivalent thus giving a relatively easy translation and allowing for a more natural interpretation of the result.

The technique however requires a more complex representation of a DPL expression. As well as predicates, constants, variables and logical operators we must also give a representation for *assignment* objects used in giving the semantics of DPL expressions.

6.4.1 Assignments

The denotation of a DPL expression is a set of pairs of assignments. An assignment consists of a function from DPL variables to objects in the DPL model. In ASTL we will represent DPL variables as parameters. The assignments themselves will be represented as facts between parameters and individuals held in an assignment situation. (This type of situation has obvious similarities to the *anchoring environments* that were introduced in the STG description in Section 4.4.) Here we will differ from the Groenendijk and Stokhof semantics. Because we do not have a reasonable notation for representing sets of pairs of assignments we will make the denotation a pair of sets of assignments. This is different but the difference is not important with respect to the dynamic aspects of the theory. Effectively we will thread sets of assignments through a DPL expression reducing the set as we advance through the expression. An assignment is represented by a situation while a set of assignments is represented by a situation type. Each term in a DPL expression will be related to an incoming set of assignments and a set of outgoing assignments. The relation between these depends on the meaning of the term. Note that although this is not explicitly stated in DPL the sets of assignments are monotonically decreasing as we progress through the expression. We will still refer to the DPL semantics and the input and output assignments in the first order representations (typically g and h) it should be seen that there is a close relationship between these two denotations.

Looking at the semantics of DPL expressions more closely we find that it is not just the simple assignment of a DPL variable to an object that is important in an assignment function. There are also conditions on what the assignment is to. The semantics of predicate terms and existentially quantified terms is defined as

$$[\![R(t_1, \ldots, t_n)]\!] = \{\langle g, h\rangle \mid h = g \wedge \langle [\![t_1]\!]_h, \ldots, [\![t_n]\!]_h\rangle \in F(R)\}$$
$$[\![\exists x\phi]\!] = \{\langle g, h\rangle \mid \exists k :\ k[x]g \wedge \langle k, h\rangle \in [\![\phi]\!]\}$$

Given the following DPL expression

$$\exists x[man(x)]$$

The denotation (in DPL) of this expression would be a set of pairs of assignments of the form $\langle g, h\rangle$ where the output assignment h would belong to

$$\{h \mid \langle [\![x]\!]_h\rangle \in F(man)\}$$

That is the assignment function assigns x to something *that is a man*. This last condition is something which must also be modelled in the ASTL representation. There are a number of ways to add such a restriction in situation theory. A basic ASTL representation of h might be (we are really specifying a representation for all possible hs)

```
[S ! S != <<assigned,X,1>>].
```

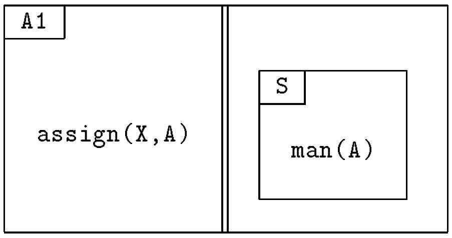
This is the type of any assignment function h. All we can really say about such an assignment function is that the DPL variable `X` is assigned though at present we do not know what to. But the semantics also requires us to place restrictions on what `X` is assigned to. It is of course possible to infer from the above type that there must exist some object that `X` is assigned to so we could expand this to be

```
[S ! S != <<assign,X,A,1>>
     S != <<assigned,X,1>>].
```

We now need a way to restrict what `A` is, (although at this stage we will avoid saying what sort of ASTL object `A` is). There are a number of ways to represent restrictions. One way that has been used in other (non-ASTL) situation theoretic descriptions is (as in [Gawron & Peters 90]) the use of *restricted parameters*. Here the `A` is typed

$A1 \models \ll assign, X, A_{\ll man,A\gg} \gg$

In EKN there is also a notation for restrictions on parameters. In EKN the above would be written as



All of these require some conditions about where (i.e. in which situation) the restriction is required.

In ASTL there are a number of ways to achieve this restriction. If we had some form of abstraction (as outlined in Section 7.4.1) and allowed typing of objects (not defined in the abstraction extension) we might wish to write something like

```
A1::[S ! S != <<assign,X,
                     A::[B ! T != <<man,B,1,>>],1>>].
```

That is we add a type to `A` stating that it must be a man in some situation `T`. Although this looks as if it might be reasonable, some definition of situation `T` is also necessary. Another similar technique which has been used in [Cooper 89] is the introduction of an explicit restriction situation. Thus we would have our assignment plus conditions in the special `Restrictions` situation.

```
A1::[S ! S != <<assign,X,A,1>>].
Restrictions::[S ! S != <<man,A,1,>>].
```

This also seems reasonable but we would need to relate each assignment situation to its relevant restriction situation in order to ensure that the parameter `A` in `A1` is necessarily the same as the `A` in `Restrictions`. Alternatively we would have to allow the restriction to apply everywhere in the description which may not be what is wished.

The problem with all of these representations is that they require an explicit reference to what `X` is assigned to when we do not really know what that is. Therefore what we will do here is merely state that the variable is assigned, that is

```
[S ! S != <<assigned,X,1>>].
```

Then in order to impose restrictions we will do so on the variable, assuming that restrictions apply under the same assignment function thus DPL variables will be anchored to their values. Thus the assignment under discussion will be represented as

```
[S ! S != <<assigned,X,1>>
     S != <<man,X,1>>].
```

That is in an assignment function of this type `X` is assigned but also (with respect to whatever `X` is assigned to) there is a restriction that the assignment is to something that is a `man`. We are using `X` is a slightly different way in each fact. In the `assigned` relation the parameter is in some way *quoted* while in the restriction itself we wish the `X` to be anchored to whatever `X` has been assigned to.

The above specifies a basic type of assignment, but we can state more based on this type. The above is a type of assignment which would act as the representation of the possible output assignments for the DPL expression $\exists x[man(x)]$. Of course we can stipulate constraints on assignments. Any actual assignment function which supports the fact that a DPL variable is assigned will also support an `assign` fact actually relating the variable to its assignment.

```
*S : [S ! S != <<assign,X,*X,1>>]
<=
  *S : [S ! S != <<assigned,X,1>>]
```

Second we must state that the general restrictions in an assignment function are with respect to its own variable assignments. Therefore the described situation that would be related to a situation representing the utterance *"A man walks"* would be captured be the following constraint.

```
*D : [S ! S != <<man,*Y,1>>
          S != <<walks,*Y,1>>
<=
  *S : [S ! S != <<described,S,*D,1>>
            S != <<Assignment,S,
                   *A :: [T ! T != <<man,*X,1>>
                              T != <<walk,*X,1>>
                              T != <<assign,*X,*Y,1>>
                              T != <<assigned,*X,1>>],1>>].
```

That is we are reducing the restrictions with respect to the assignments. At this point it is worth noting the similarities here between assignment functions and the anchoring environments discussed in the STG description in Section 4.4. Assignment functions assign parameters to objects in exactly the way anchoring does. But the inclusion of the restrictions make assignments more like a combination of anchoring environments and the parametric fact used in the semantic translation in STG. Of course assignment functions also have close similarities with DRSs, which we will discuss later.

In addition to simple `assign`-facts and restrictions we will also need a `forall` relation which will be explained below. Formally we also need an `exists` too but this is actually unnecessary as we can achieve the same treatment without it. Because of the semantics of types in ASTL there is effectively an implicit existential before each fact.

6.4.2 DPL expressions in ASTL

Note that as well as a representation for the semantics of a DPL expressions we would like to offer a treatment of DPL syntax within ASTL. Unlike the other descriptions in the two previous chapters this time we are not dealing with a natural language but an artificial language, namely DPL. DPL has its own syntax and we can write grammar rules in ASTL which define that language's syntax. The grammar rules are simple but it is interesting that the syntax of logics typically is not written in the same explicit way as grammar for natural language fragments. Syntax trees for a logical expressions will often mark operators on mother nodes rather than give them their own pre-terminal node. The grammar for DPL is an ASTL grammar based on the following context free grammar

wff → wff `and` wff.
wff → wff `or` wff.
wff → wff `implies` wff.
wff → `exists` *var* wff.
wff → `forall` *var* wff.
wff → *predicate*.

For the sake of simplicity we will only allow predicates with one argument (**`walk`**, **`talk`**, **`sing`**, etc.) and treat them as lexical forms. An ASTL grammar rule for one of the above rules would be of the form

```
*S : [S ! S != <<wff,S,1>>
         S != <<term,S,conj,1>>
         S != <<conjunct,S,*C1,1>>
         S != <<conjunct,S,*C2,1>>]
->
   *C1 : [S ! S != <<wff,S,1>>],
   [S ! S != <<terminal,S,and,1>>],
   *C2 : [S ! S != <<wff,S,1>>].
```

Each of the above context free rules are translated like the above. An example syntactic parse of the DPL expression

$\exists x[man(x)]$

is as below

```
 ________________________________________________
| SIT388                                         |
|                                                |
|  wff(SIT388)                                   |
|  term(SIT388,exists)                           |
|  var(SIT388,X)                                 |
|                   ___________________________  |
|                  | SIT233                    | |
|                  |                           | |
|                  |  wff(SIT233)              | |
|  scope(SIT388,   |  terminal(SIT233,pred)    | ) |
|                  |  arg(SIT233,A)            | |
|                  |  sem(SIT233,man)          | |
|                  |___________________________| |
|________________________________________________|
```

Here we relate each wff situation to two assignment types by the relations `AssignIn` and `AssignOut`. We can define constraints between the input and output assignments related to a wff situation with respect to the semantic definitions of DPL expressions. For the case of predicate terms the DPL semantics is defined as

$$[\![\ R(x)\ ]\!] = \{\langle g, h\rangle \mid h = g \wedge \langle [\![\ x\ ]\!]_h\rangle \in F(R)\}$$

The corresponding ASTL constraint is

```
*S : [S ! S != <<AssignOut,S,
                 *Ain &
                 [T ! T != <<*Rel,*Arg,1>>],1>>]
<=
   *S : [S ! S != <<terminal,S,pred,1>>
             S != <<sem,S,*Rel,1>>
             S != <<arg,S,*Arg,1>>
             S != <<AssignIn,S,*Ain,1>>].
```

Here the output assignment carries forward the type of the input assignment *plus* the restriction contributed by the predicate itself. Again we can see similarities with the constraints used in the definitions of DRSs as described in the previous chapter. (Though perhaps we should say that the DRSs definitions are similar to dynamic logic rather than the reverse.) The above constraint follows the basic concept of dynamic logic in that the expression transforms its input "state" to a new output "state". The types will be monotonically increasing through the expression. Thus the set of assignment situations that are of this type will decrease through the expression.

Let us look at a detailed example to see how the type of the assignment function is built up. We will look at the DPL expression $\exists x[man(x)] \wedge walk(x)$ which might be used to represent the natural language utterance *"a man walks"*.

For the sake of argument we will state that the incoming assignment to this expression is unconstrained (i.e. effectively any assignment function). The type of which can be represented as

```
   A0
A0
```

To be pedantic we could state that it supports the fact that it is an assignment function as in

```
   A0
A0

 assignment-function(A0)
```

thus excluding any random situation as an input but for the sake of space we will just start with an empty type. The DPL semantics for the top level conjunction is

$$[\![\phi \wedge \psi]\!] = \{\langle g,h\rangle \mid \exists k: \ \langle g,k\rangle \in [\![\phi]\!] \wedge \langle k,h\rangle \in [\![\psi]\!]\}$$

Thus the input to the top level expression becomes the input to the first conjunct. The semantics for the first conjunct is

$$[\![\exists x \ \phi]\!] = \{\langle g,h\rangle \mid \exists k: \ k[x]g \wedge \langle k,h\rangle \in [\![\phi]\!]\}$$

This time the constraints for the assignments are a little more complex. First from the definition we see that k is the input assignment for the sub-term ϕ. It is the input assignment to the existential term plus the fact that `X` is assigned. This is captured by the ASTL constraint

```
*T : [S ! S != <<AssignIn,S,*G &
                             [T ! T != <<assigned,*Y,1>>],
                 1>>]
<=
 *S : [S ! S != <<term,S,exists,1>>
           S != <<AssignIn,S,*G,1>>
           S != <<scope,S,*T,1>>
           S != <<var,S,*Y,1>>].
```

So the input assignment (k) to the sub-expression would be

```
   A1
A1

 assigned(X)
```

The term $man(x)$ uses the constraint for predicates shown above. The output assignment for that expression will be the combined type of the incoming type plus the type for the restriction introduced by the predicate

```
   A1                  T
A1                   T
                &
 assigned(X)          man(X)
```

which is reduced to

```
   A2
A2

 man(X)
 assigned(X)
```

The output assignment for the expression $\exists x[man(x)]$ is the same as output from the sub-expression which is captured by the constraint.

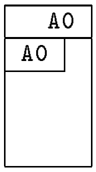
```
*S : [S ! S != <<AssignOut,S,*H,1>>]
<=
   *S : [S ! S != <<term,S,exists,1>>
             S != <<scope,S,
                    *Scope ::
                     [T ! T != <<AssignOut,*Scope,*H,1>>],1>>].
```

The output to this conjunct also acts as the input to the second conjunct $walk(x)$, which in turn gives an output (via the predicate constraint)

```
   A3
A3

 walk(X)
 man(X)
 assigned(X)
```

The above is a simple example but clearly shows the dynamic aspect of the theory. The ASTL constraints are designed to directly reflect the semantics of DPL expressions. The assignments are extended through the expression. Particularly the threading of assignments through the expression allows DPL variables to be referenced outside the apparent scope of the existential quantifier that introduced them.

We will now look at a second example involving the universal quantifier which requires us to use slightly more complex restrictions in assignments. The example is $\forall x[man(x) \rightarrow walk(x)]$ which is a translation of the utterance, *"Every man walks"*. As before the initial incoming assignment will be the empty type



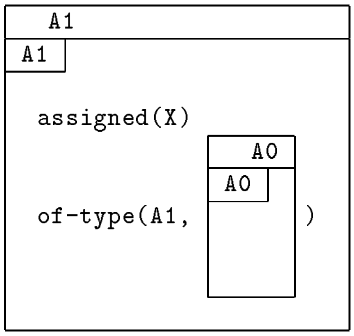
The semantics of the top level expression is stated as

$$[\![\forall x \phi]\!] = \{\langle g, h\rangle \mid h = g \wedge \ \ \forall k : \ k[x]g \Rightarrow \exists m : \ \langle k, m\rangle \in [\![\phi]\!]\}$$

The first stage is that the initial input assignment is extended to state that `X` is assigned and that new assignment k is passed to the sub-expression ϕ. The ASTL constraint that reflects this is

```
*T : [S ! S != <<AssignIn,S, [T ! T != <<assigned,*Y,1>>
                                  T != <<of-type,T,*G,1>>],
                 1>>]
<=
  *S : [S ! S != <<term,S,forall,1>>
            S != <<AssignIn,S,*G,1>>
            S != <<scope,S,*T,1>>
            S != <<var,S,*Y,1>>].
```

The application of this to our initial empty assignment for this example would produce



An important extra relation is used here: `of-type`. Although it might have be thought that the type of input assignment to the sub-expression could be a simple extension of the overall incoming type this is not the case. If we have a discourse *"A man walks. Every woman likes a donkey."* the incoming type to the quantified sentence would be

```
A1
A1

walk(X)
man(X)
assigned(X)
```

By simple extension the incoming assignment to the quantified sub-expression would be

```
A2
A2

assigned(Y)
walk(X)
man(X)
assigned(X)
```

Thus we would have no distinction between variables introduced by the quantifier and existential variables introduced earlier in the discourse. Therefore is it crucial to mark the boundary between the two parts of the type. It is important to quantify only over the changes to the incoming assignment not the whole assignment. Thus we have

```
A2
A2

assigned(Y)
                  A1
                  A1

of-type(A2,        walk(X)       )
                   man(X)
                   assigned(X)
```

All assignment functions that are of this type are also of the type

```
A2
A2

assigned(Y)
walk(X)
man(X)
assigned(X)
```

We partition the type so that we can identify which assignments must be quantified over universally and which (the ones within the type related by `of-type` which must merely be treated existentially.

Returning to our example, the next level of expression is an implication. The DPL semantic definition is

$$[\![\phi \rightarrow \psi]\!] = \{\langle g, h\rangle \mid h = g \wedge \forall k : \langle g, k\rangle \in [\![\phi]\!] \Rightarrow \exists j : \langle k, j\rangle \in [\![\psi]\!]\}$$

We continue feeding assignments through the translation as shown above. The interesting aspect is looking at the output assignment for the whole implication expression. Trivially it is the same as the input expression (as $h = g$), but that ignores the internal condition. As we are including restrictions in assignments as well as the actual assignment of variables themselves, although it is true that no assignment of DPL variables is passed out of this expression there is a condition on the internals of the implication. Thus the output assignment for the implication expression will be the conjunction of the input assignment plus a condition for the implication itself.

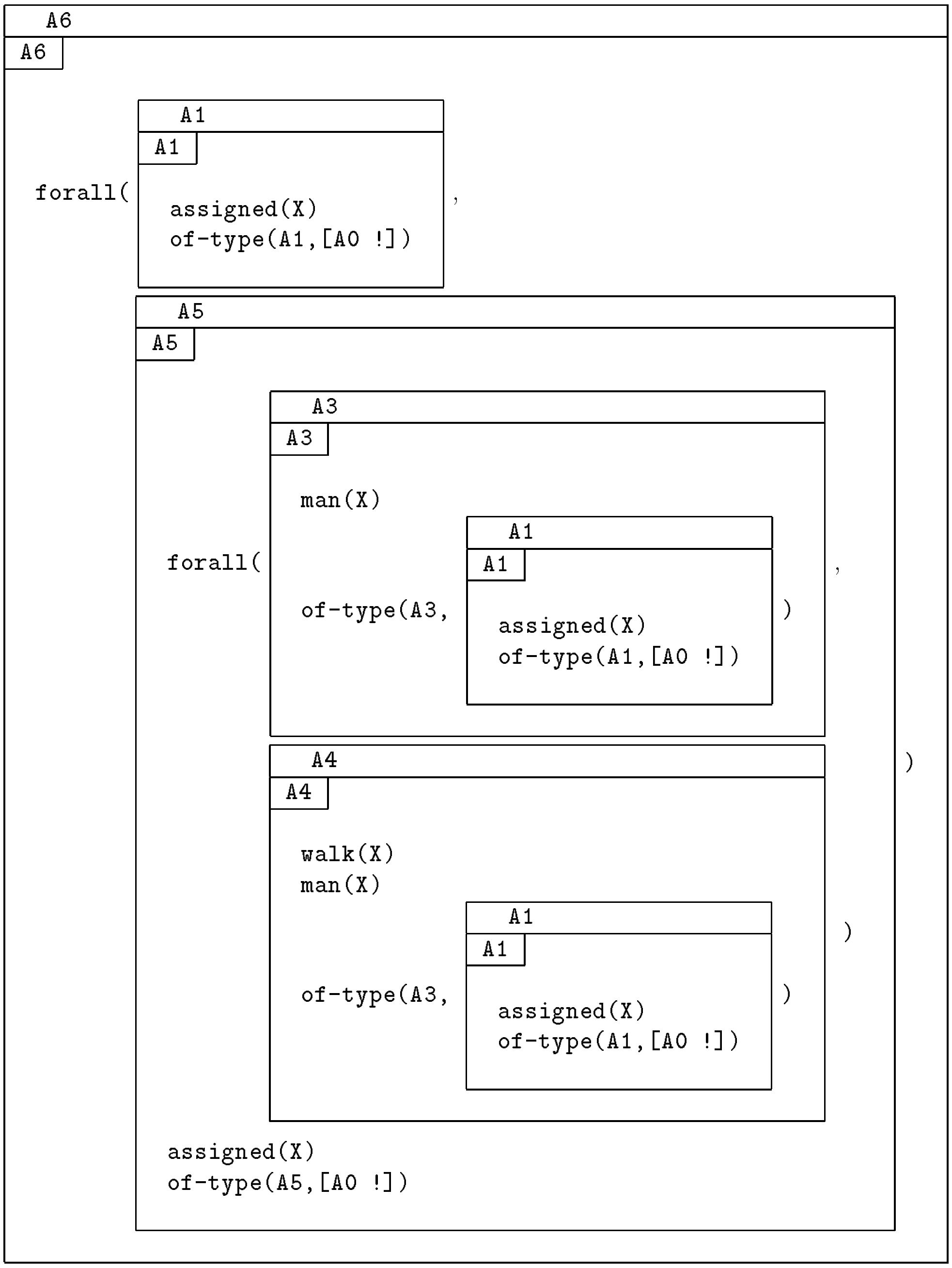
```
+-------------------+
|   A1              |
+----+--------------+
|A1  |              |
+----+              |
|                   |
| assigned(X)       |
| of-type(A1,[A0 !])|
|                   |
+-------------------+
```

```
A2
A2

            A3
            A3

            man(X)
                              A1
  forall(                     A1                     ,

            of-type(A3,                            )
                              assigned(X)
                              of-type(A1,[A0 !])

&
            A4
            A4

            walk(X)
            man(X)
                              A1
                              A1                     )

            of-type(A3,                            )
                              assigned(X)
                              of-type(A1,[A0 !])
```

which can be reduced to form a single type

```
A5
A5

        A3
        A3

        man(X)
                        A1
forall(                 A1                          ,

        of-type(A3,     assigned(X)             )
                        of-type(A1,[A0 !])

        A4
        A4

        walk(X)
        man(X)
                        A1
                        A1                          )

        of-type(A3,     assigned(X)             )
                        of-type(A1,[A0 !])

assigned(X)
of-type(A5,[A0 !])
```

Returning to our analysis of $\forall x[man(x) \rightarrow walk(x)]$, the output of the whole expression is the defined to be the same as the input, *plus* the condition for the sub-expression. Thus the final output type is



This final output assignment is achieved by the following ASTL constraint

```
*S : [S ! S != <<AssignOut,S,
                 *G &
                 [T ! T != <<forall,*K,*M,1>>],1>>]
 <=
   *S : [S ! S != <<term,S,forall,1>>
             S != <<scope,S,
                    *Scope ::
```

```
                    [T ! T != <<AssignOut,*Scope,*M,1>>
                         T != <<AssignIn,*Scope,*K,1>>],1>>
           S != <<AssignIn,S,*G,1>>].
```

Putting the syntactic treatment and semantic definition of assignments together we can now parse DPL expressions and have assignments related to each term in the expression. The following examples show the term situation for complete sentences of DPL. Specifically the output assignment type contains the information necessary to infer the type of the described situation.

```
exists x man(x)

SIT517

cat(SIT517,sentence)
                         A0
                      A0
AssignIn(SIT517,              )

                            A1
                       A1
AssignOut(SIT517,                 )
                       man(X)
                       assigned(X)

forall x man(x)
```

```
SIT595

cat(SIT595,sentence)
                         A0
                        A0
AssignIn(SIT595,             )

                          A3
                         A3

                                      A4
                                     A4
                 forall(                            ,
                                     assigned(X)
                                     of-type(A4,[A0 !])
AssignOut(SIT595,                                        )
                                      A5
                                     A5

                                     man(X)             )
                                     assigned(X)
                                     of-type(A5,[A0 !])
```

We now require a constraint to state the relationship between an assignment supporting a `forall`-fact and the described situation (i.e. the model). The following constraints state such a relationship for the example above

```
*S : [S ! S != <<dpl-assignment,S,
                *A :: [AE ! AE != <<assign,X,*X,1>>],1>>]
<=
   *S : [S !
         S != <<described,S,
                *DS :: [D ! D != <<man,*X,1>>],1>>
         S != <<AssignOut,S,
                [P1 ! P1 !=
                 <<forall,
                   [P2 ! P2 != <<assigned,X,1>>
                         P2 != <<of-type,P2,[P0 !],1>>],
                   [P3 ! P3 != <<walk,X,1>>
                         P3 != <<man,X,1>>
                         P3 |= <<assigned,X,1>>
                         P3 != <<of-type,P3,[P0 !],1>>],
                   1>>],
                1>>].
```

```
*DS : [DS ! DS != <<walk,*X,1>>]
<=
   *S : [S ! S != <<described,S,
                      *DS :: [D ! D != <<man,*X,1>>],1>>
              S != <<dpl-assignment,S,
                      *A :: [AE ! AE != <<assign,X,*X,1>>],1>>
              S != <<AssignOut,S,
                      [P1 ! P1 !=
                       <<forall,
                          [P2 ! P2 != <<assigned,X,1>>
                                P2 != <<of-type,P2,[P0 !],1>>],
                          [P3 ! P3 != <<walk,X,1>>
                                P3 != <<man,X,1>>
                                P3 |= <<assigned,X,1>>
                                P3 != <<of-type,P3,[P0 !],1>>],
                     1>>],
                 1>>].
```

Notice how the above constraints compare with the constraints concerned with the relation `every` for DRSs on page 82. They are effectively the same.

The threading of assignments through an expression is relatively simple when compared with the threading in the DRT example in Section 5.3.2, before it was necessary to build threads to order the parts of a natural language utterance. Here we are dealing with an artificial language which is much better behaved. "Threads", where assignments go, can be determined locally and usually simply go into the first daughter of a rule. Outputs are determined by the type of term the node represents.

6.5 DPL and natural language

DPL as it is basically defined does not give a mechanism for translating natural language utterances into logical forms (analogous to the construction algorithm in DRT). However, in this section we outline such a translation, called DPL-NL, from a natural language fragment to a dynamic predicate logic. Later work in dynamic semantics ([Groenendijk & Stokhof 91a]) describe a more complex form of dynamic semantics, namely Dynamic Montague Grammar for the purposes of our description and comparison with DRT a simpler form is sufficient. Of course, again the translation is for the Rooth syntactic fragment introduced in Section 4.3.

In the DPL (and DMG) literature typically natural language glosses are given to dynamic logic expressions. For example

> *A* man_1 *walks.* He_1 *talks.*
> $\exists x[man(x) \wedge walk(x)] \wedge talk(x)$

Note that the glosses already contain co-indexing of pronouns and their referents as appropriate. In the example presented below we will convert utterances without indexing into a dynamic form similar to the translation of DPL described in the previous section.

There are (at least) two ways to do such a translation. DPL could be used as an intermediate representation between natural language and the semantics (in this case assignments). The description would then be required to construct a DPL expression from the natural language utterance and then rely of something similar to the previous description to relate it to the described situation. A second possible treatment, and the route actually taken, is to translate directly from the natural language utterance to the semantic form—threads of assignments.

The route of a non-explicit intermediate representation (no direct representation of DPL expressions) could be argued by Groenendijk and Stokhof's claim of a *non-representational* theory, although perhaps some of the advantages of DPL, particularly compositionality, may be obscured by this method. Compositionality is proposed as an important aspect of dynamic logic but the examples mainly deal with inter-sentential compositionality rather than intra-sentential. Groenendijk and Stokhof argue that conventional semantic treatments of discourses (including DRT) do not offer simple sub-expressions representing each of the sentences in the discourse. Consider the following discourse

> *A* man_1 *talks.* He_1 *walks*

In simple first order logic would have a translation as

$$\exists x[man(x) \wedge walk(x) \wedge talk(x)]$$

while the DPL translation is

$$\exists x[man(x) \wedge walk(x)] \wedge talk(x)$$

Importantly the DPL translation does have a simple sub-expression which represents each sentence while the first order one does not (without the addition of something like lambda abstractions). This fact should make natural language treatments in DPL easier to specify. Unfortunately this compositionality does not always hold *within* sentences. Particularly there is no DPL sub-expression which directly represents the sub-sentential phrase *"a man"*. The consequence of this is that the translation from utterance to DPL is not as simple as would be hoped but by no means impossible.

Each utterance situation in the analysis of a natural language phrase will be related to an input and output assignment type. Assignments are defined in the same form as described in Section 6.4.1 above. Before the details are given the following example will help to show what the desired treatment is. The analysis for *"a man walks"* is

```
SIT486

cat(SIT486,sentence)
                          A0
                        A0
AssignIn(SIT486,                 )


                           A1
                         A1

AssignOut(SIT486,        walk(X)       )
                         man(X)
                         assigned(X)
```

The output assignment is exactly as it would be for the ASTL DPL treatment of the expression $\exists x[man(x) \wedge walk(x)]$.

Note that we will be treating the sentence *"A man walks"* as $\exists x[man(x) \wedge walk(x)]$ rather than $\exists x[man(x)] \wedge walk(x)$. This is a consequence of the threading relations which are themselves a consequence of the way we wish to deal with DRT. The difference is not significant but we choose the existential to scope over the whole sentence so the treatment can be as similar as possible to the DRT one. Of course the dynamic aspects are still illustrated by inter-sentential anaphora.

The DPL-NL description is built directly using parts of the DRT description. First, it uses exactly the same syntactic grammar (the Rooth fragment). Secondly it uses the same threading relations. Syntactic utterance situations are threaded by the same conditions as DRT description given in Section 5.3.2. The part that does change is the removal of the constraints defining the relationship between incoming and outgoing DRSs, and the definition of accessibility situations. In DPL-NL the semantics is defined by types of assignment which are defined over the threading relations.

Each utterance situation is related to an incoming and outgoing assignment type. The output assignment is the input assignment plus information contributed by the semantics of the utterance situation itself. For example the output assignment of the top node of a quantified noun phrase is

```
*S : [S ! S != <<AssignOut,S,
                 *AssignIn &
                 [DS ! DS != <<*VR1,*VA1,1>>
                       DS != <<type,*VA1,*TYPE,1>>],1>>]
 <=
   *S : [S ! S != <<cat,S,NounPhrase,1>>
             S != <<daughter,S,
                    *DS1 :: [DS !
```

```
                    DS != <<cat,DS,determiner,1>>],1>>
     S != <<daughter,S,
           *DS2 :: [DS !
                    DS != <<cat,DS,noun,1>>
                    DS != <<type,DS,*TYPE,1>>
                    DS != <<sem,DS,<<*R1,*A1,1>>,1>>],1>>
     S != <<env,S,*SEnv ::
          [Env ! Env != <<anchor,*R1,*VR1,1>>
                 Env != <<anchor,*A1,*VA1,1>>],1>>
     S != <<AssignIn,S,*AssignIn,1>>].
```

That is the input assignment type is extended with two conditions. One from the relation introduced by the head noun related to whatever the argument is assigned to. The second fact identifies the type of the noun (male, female or neuter) used in finding pronoun referents.

The constraint for the indefinite determiner utterance situation shows how the translation to the dynamic existential quantifier is treated. The output of the utterance that introduces the existential is the output assignment from the thread it scopes over (i.e. the DPL sub-expression). As the sub-expression already includes the information that was an input to that utterance it does not need to be combined with the input assignment

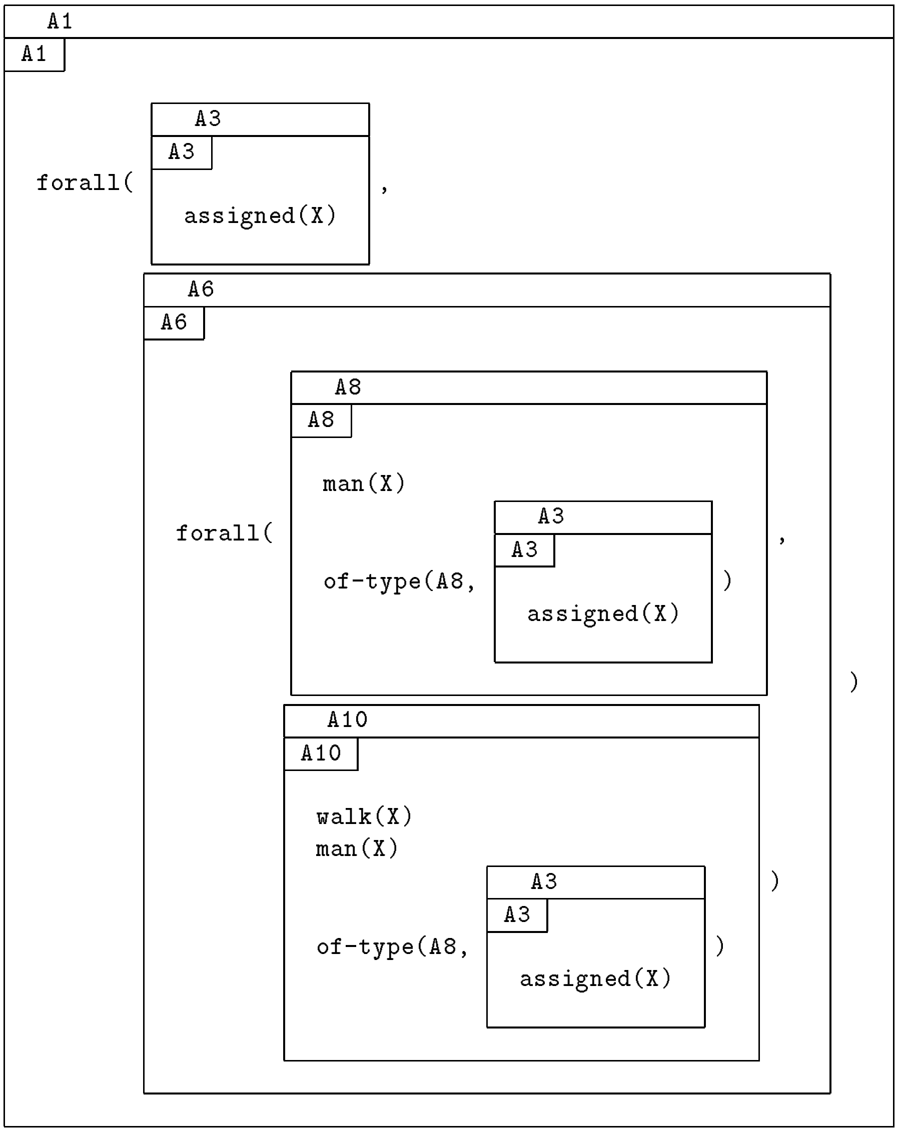
```
*S : [S ! S != <<AssignOut,S,*BodyOut,1>>]
  <=
    *S : [S ! S != <<cat,S,Determiner,1>>
              S != <<sem,S,<<*Q,*X,*Y,*Z,1>>,1>>
              S != <<env,S,*SEnv ::
                      [Env ! Env != <<anchor,*Q,some,1>>],1>>],
    *T2 : [TS ! TS != <<t-body,*S,*Body ::
                       [S ! S != <<AssignOut,S,*BodyOut,1>>],
                       1>>].

*S : [S ! S != <<AssignMid,S,
                 *G &
                 [A ! A != <<assigned,*X,1>>],1>>]
  <=
      *S : [S ! S != <<cat,S,Determiner,1>>
                S != <<sem,S,<<*Q,*X,*Y,*Z,1>>,1>>
                S != <<env,S,*SEnv ::
                       [Env ! Env != <<anchor,*Q,some,1>>],1>>
                S != <<AssignIn,S,*G,1>>
                S != <<ind,S,*I,1>>].
```

The second constraint creates the intermediate assignment which introduces the fact that the DPL variable is assigned. The assignment related by `AssignMid` is threaded into the start of the sub-expression that this quantifies over.

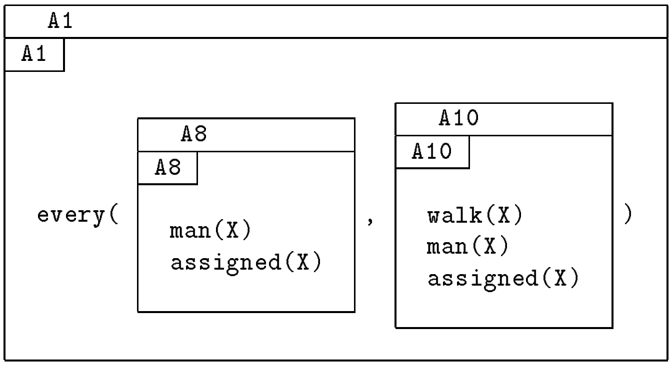
The third example shows the constraint for the universal determiner. This again requires the use `AssignMid` as above, but this time the we must introduce the `of-type` relation.

```
*S : [S ! S != <<AssignMid,S,
                  [A ! A != <<assigned,*X,1>>
                       A != <<of-type,A,*G,1>>],1>>]
  <=
      *S : [S ! S != <<cat,S,Determiner,1>>
                S != <<sem,S,<<*Q,*X,*Y,*Z,1>>,1>>
                S != <<env,S,*SEnv ::
                       [Env ! Env != <<anchor,*Q,every,1>>],1>>
                S != <<AssignIn,S,*G,1>>
                S != <<ind,S,*I,1>>].
```

This time we will use the relation `every` rather than the `forall` relation we used in the DPL translation given in the previous section. `Every` can be viewed as an abbreviation for two `forall` relations. In the previous section we translated $\forall x[man(x) \rightarrow walk(x)]$ as



while using `every` we abbreviate the above to



thus we have reduced one level of quantifier. This is justified by the fact that quantifier representing the natural language word *"every"* has the more specific semantics $\forall x[\alpha \rightarrow \beta]$ rather that $\forall x[\phi]$. This also of course makes the relation more similar to the DRT `every` relation.

The constraint for the *"every"* determiner node in an utterance is

```
*S : [S ! S != <<AssignOut,S,
                 *AssignIn &
                 [DS ! DS != <<every,*RangeAssign,
                                     *BodyAssign,1>>],1>>]
  <=
   *S : [S ! S != <<cat,S,Determiner,1>>
             S != <<sem,S,<<*Q,*X,*Y,*Z,1>>,1>>
             S != <<AssignIn,S,*AssignIn,1>>
             S != <<env,S,*SEnv ::
                        [Env ! Env != <<anchor,*Q,every,1>>],1>>],
   *T1 : [TS ! TS != <<t-body,*S,*Body ::
                        [S ! S != <<AssignOut,S,
                                    *BodyAssign,1>>],1>>],
   *T2 : [TS ! TS != <<t-range,*S,*Range ::
                        [S ! S != <<AssignOut,S,
                                    *RangeAssign,1>>],1>>].
```

Proper nouns will be treated as a form of existential quantifier. That is they introduce a new DPL variable and assign it to a DPL individual that denotes the named object. The constraint for a proper noun utterance situation is

```
*S : [S ! S != <<Assignout,S,
               *AssignIn &
               [A ! A != <<named,*X,*Name,1>>
                    A != <<type,*X,*TYPE,1>>
                    A != <<assigned,*X,1>>],1>>]
  <=
      *S : [S ! S != <<cat,S,ProperNoun,1>>
                S != <<use_of,S,*Name,1>>
                S != <<type,S,*TYPE,1>>
                S != <<sem,S,*X,1>>
                S != <<AssignIn,S,*AssignIn,1>>].
```

This treatment means that (wrongly) proper nouns are not available for pronominal reference outside the scope of a universal quantifier (this is true for the original DPL and DMG descriptions not just of our ASTL description). Some more general treatment of proper nouns should be given where, once introduced, they may be referenced anywhere.

The last interesting aspect of the DPL-NL description that we will discuss is the treatment of pronouns. As co-indexing of pronouns and their antecedents is marked in DPL natural language glosses the treatment of pronoun resolution is not actually discussed within DPL, we could just accept the DPL treatment and label our pronouns and noun phrases but this would require a different grammar (or at least different lexical entries) from the one used in the DRT and STG description. Here, we will try to include a method for selecting possible referents for pronouns.

In this description, pronouns introduce new DPL variables which are related to suitable DPL variable referents. In the following constraint, we add to the outgoing assignment a new DPL variable and add the condition that it (i.e. its denotation) is the same as some accessible referent of the appropriate type.

```
*S : [S ! S != <<AssignOut,S,*AssignIn &
                       [A ! A != <<assigned,*X,1>>
                            A != <<is,*X,*Z,1>>],1>>]
  <=
     *S : [S ! S != <<cat,S,Pronoun,1>>
               S != <<type,S,*TYPE,1>>
               S != <<sem,S,*X,1>>
               S != <<AssignIn,S,*AssignIn,1>>
               S != <<Accessible,S,
                      *Acc :: [A ! A != <<type,*Z,*TYPE,1>>
                                   A != <<accessible,*Z,1>>],
                      1>>].
```

(*TYPE will be one of male, female or neuter.) The type of the situation related by the Accessible relation is that of the incoming assignment

```
*S : [S ! S != <<Accessible,S,*Acc :: *AssignIn,1>>]
   <=
      *S : [S ! S != <<AssignIn,S,*AssignIn,1>>].
```

The important definition is that those items that are assigned in the incoming assignment are exactly those that are accessible as referents. This is captured by

```
*S : [S ! S != <<accessible,*X,1>>]
   <=
      *S : [S ! S != <<assigned,*X,1>>].
```

The whole DPL-NL description gives a dynamic semantics for the Rooth fragment—the full description is given in Appendix A.5. As stated above the description is based on much of the same description as the DRT description, only the definitions of DRSs and accessibility situations have been replaced with definitions for assignments.

The following examples show the output assignments for simple discourses.

A man walks. He talks.

```
SIT587

cat(SIT587,fulldiscourse)
                    P7
                    P7

                     talk(PN1)
                     is(PN1,E1)
AssignOut(SIT587,    assigned(PN1)   )
                     walk(E1)
                     man(E1)
                     type(E1,male)
                     assigned(E1)
```

E1 is the DPL variable introduced by the existential. PN1 is the variable introduced by the pronoun *"it"*.

Every man with a donkey likes it.

```
SIT138

cat(SIT138,fulldiscourse)
AssignOut(SIT138,
  P9
  P9

                                   P8
                                   P8
           P6
           P6                       like(A1,PN3)
                                    is(PN3,E1)
            with(A1,E1)             assigned(PN3)
            donkey(E1)              with(A1,E1)              )
  every(    type(E1,neuter)     ,   donkey(E1)          )
            assigned(E1)            type(E1,neuter)
            man(A1)                 assigned(E1)
            type(A1,male)           man(A1)
            assigned(A1)            type(A1,male)
            of-type(P6,[P0 !])      assigned(A1)
                                    of-type(P8,[P0 !])
```

In this example the main restriction is over two assignments. The interpretation states that for all ways that the first assignment can be made true with an anchoring environment there must be an extension of that anchoring that makes the second argument true. Notice that the full contents of the first type (with parameter `P6`) are contained within the second (`P8`). This is because the type was formed as an extension of the first. This means that assignments contain a "history" of `assign`-relations and restrictions. Three DPL variables are introduced: `A1` from the universal; `E1` from the existential and `PN3` from the pronoun *"it"*.

The DPL-NL description is interesting as it provides a dynamic semantics for a simple natural language fragment showing how such a translation can be made from an utterance to assignments. There is no explicit representation of the dynamic logic expression itself, only the first order representation of the semantics of the DPL expression.

Although in the original work on DPL, [Groenendijk & Stokhof 91b], no translation from natural language to DPL is given there has been other work which attempts to do this. In [Lewin 92], a similar translation to the one above is given. That translation is not given in terms of situation theory but the similarities to the one here are obvious. One difference is that Lewin gives a better treatment of proper nouns such that they have scope over the whole discourse unlike the restrictive treatment given here.

The above translation is for Dynamic Predicate Logic. [Groenendijk & Stokhof 91a] introduces a more complex dynamic logic—Dynamic Montague Grammar (DMG). DMG is the application of Dynamic Intensional Logic to a natural language grammar to give meanings for utterances. The use of lambda abstraction increases the descriptive power significantly and allows for a level of compositional treatment within natural language sentences that is not available in DPL.

The essential differences in DMG are how the semantics of a sentence (and hence a discourse) are treated, and the introduction of the sense of *state*. Again, like DPL, the basic concept in DMG is that the semantics of an utterance transforms a state to a new state. The basic representation of a natural language utterance with one existential can be summarised as

$$[\![\, \phi \,]\!] = \lambda P\, [\phi \,\wedge\, P]$$

where $\wedge$ is a dynamic conjunction operator and hence succeeding sentences can refer to existentials introduced in ϕ. But the whole discourse must be applied to *true* in order to interpret it. This is not the whole story. The denotation of a sentence is with respect to the current state which assigns values to *discourse markers*. Groenendijk and Stokhof distinguish two types of variables, *discourse markers* and conventional variables and justify why this distinction is required. The idea is that the bindings of discourse markers are available outside the normal scope of certain existentials.

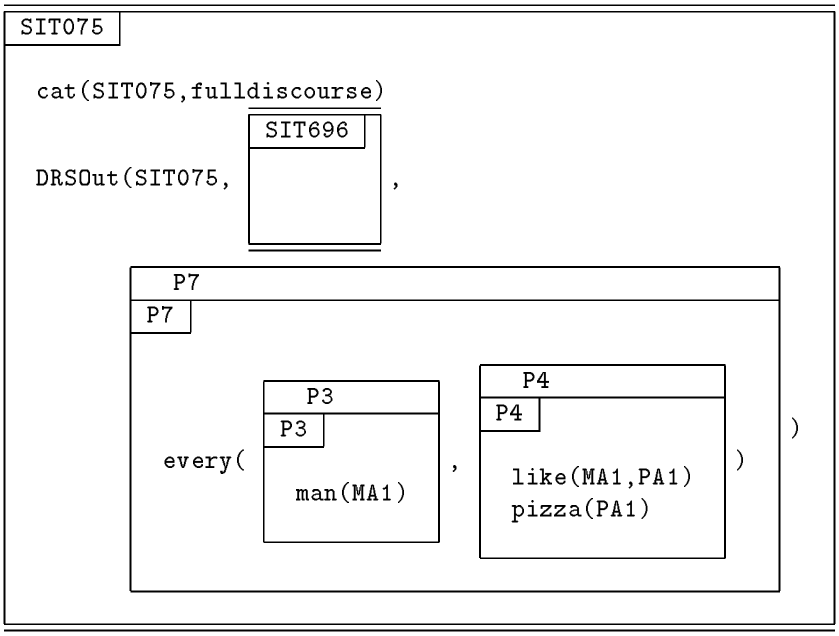
[Beaver 91] gives a reformulation of DMG which in turn has been given a situation theoretic description in [Beaver *et al* 91] where a translation of DMG is given in EKN (Extended Kamp Notation). The result depends crucially on the use of lambda abstraction and application, in situation theoretic terms these are abstraction, anchoring and reduction. We could reformulate this description in ASTL but only if we include general abstraction and a better form of reduction (see Section 7.4.1).

The difference between DPL-NL and a treatment of DMG in ASTL is primarily in power. DMG is a far more complex logic than what DPL-NL is based on and hence can provide a translation for a much wider range of utterances. However that is merely matter of scale. The second and perhaps more important difference is that DMG relies heavily on (lambda) abstraction and hence allows a more "compositional" treatment of meaning formation. The essential characteristic of dynamic logics (DPL and DMG) is the notion of an expression changing state. Secondly, in the case of natural language utterance interpretation both DPL and DMG exhibit the property of threading *discourse markers* introduced by existentials, through the utterance.

6.6 Comparison of DPL-NL and DRT

It is important to remember that DPL was deliberately developed to have the same semantic coverage of DRT so it is not surprising to find how similar these theories are. Also because the definition of such theories always leaves some aspects open to the interpretation of the implementor, there is some freedom in the actual method of implementation. Because of this, and because it is the general intention of this work to show similarities between theories the DRT description in Chapter 5 and the description of DPL-NL given above lead to very similar representations. However there are some important differences which can be highlighted.

First consider the DRT representation and DPL-NL representation of the same utterance. The DRT representation for *"Every man likes a pizza."* is



while the DPL-NL for the same sentence is

```
SIT225

cat(SIT225,fulldiscourse)
AssignOut(SIT225,
    P8
  P8
                                    P7
                                  P7
                P5
              P5                   like(A1,E1)
                                   pizza(E1)                )
               man(A1)             type(E1,neuter)
     every(    type(A1,male)    ,  assigned(E1)          )
               assigned(A1)        man(A1)
               of-type(P5,[P0 !])  type(A1,male)
                                   assigned(A1)
                                   of-type(P7,[P0 !])
```

One interesting aspect of the dynamic translation is that all of the "previous" conditions are held in the assignments. Thus the body of the quantifier (`every`) includes a "history" of conditions, but in the DRT case we only quantify over the minimal number of conditions at that point. In logic terms we can view the DRT case as

$$\forall[\alpha \rightarrow \beta]$$

while in the DPL-NL case we could summarise this as

$$\forall[\alpha \rightarrow [\alpha \wedge \beta]]$$

From an implementation point of view although the above two expressions are logically equivalent the second could be viewed as requiring more work to evaluate. Of course making valid statements about the amount of time it takes to evaluate expressions is difficult. Because it is known that such a relationship exists in the DPL-NL representation we could easily exploit that and optimise the expression (at interpretation time) accordingly. However it should be stated that in the direct interpretation of this dynamic semantic translation it will be the case that assignments will contain the full history of the conditions. Of course it is arguable that this history is a consequence of the representation of assignments, but the method used here does not seem unreasonable.

If DPL were extended to deal with generalised quantifiers this could be a problem. Many quantifiers have equivalent semantics for the following two expressions

$Q(\alpha, \beta)$
$Q(\alpha, \alpha \wedge \beta)$

but this may be a problem with the quantifier `only`—though perhaps that should not be treated as a generalised quantifier. But even if we ignore `only`, [Lewin 92, Chap. 6], who gives a dynamic treatment for generalised quantifiers, argues that unruly quantifiers notwithstanding, the second form above will still cause problems and hence requires an alternative to the standard dynamic semantic treatment. Other dynamic treatments of generalised quantifiers also exist [Chiercha 92].

Although this duplication of information may seem a disadvantage it also has a definite advantage. In the DRT representation we explicitly state the accessible markers at each point in the construction. This is done by threading the information through the analysis in a separate situation. It is the case that the accessibility situation will identify all discourse objects that are accessible at that point in the analysis. That is it will contain the full history of introduced objects in the same way as the DPL-NL assignment situations—though the DRT accessibility situations contain only `accessible` relations not all the conditions. The difference is the accessibility situation in the DRT treatment is not used in the DRS interpretation function only in checking for pronoun referents, and so we do not have extra conditions to check. However it is true that we still have to calculate (or "trace") that information. In the case of DPL-NL assignments no extra calculation or definition of accessible objects is necessary as the information is already directly available in the assignment.

Groenendijk and Stokhof argue that DPL has advantages over DRT as DPL is both a *compositional* and *non-representational* theory. DPL's advantages were displayed by the fact that the relationship between the natural language utterance and the DPL expression were close and that we could easily map DPL sub-expressions to parts of the utterance. For example in

A man walks.
$\exists x[man(x)] \wedge walk(x)$

there is a DPL sub-expressions ($\exists x[man(x)]$) for the subject noun phrase (*A man*). But in the DRS case there is no sub-DRS that can easily be identified with the subject noun phrase

A man walks. $\rightsquigarrow$

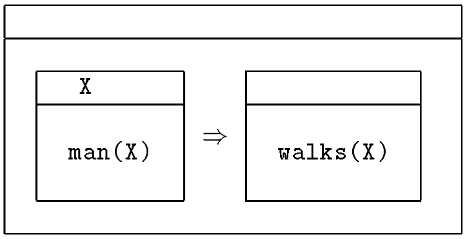
X
`man(X)` `walk(X)`

Because of this mis-match between the structure of expressions and the structure of the utterance Groenendijk and Stokhof claim that DRT is not compositional. However when we consider the following utterance

Every man walks

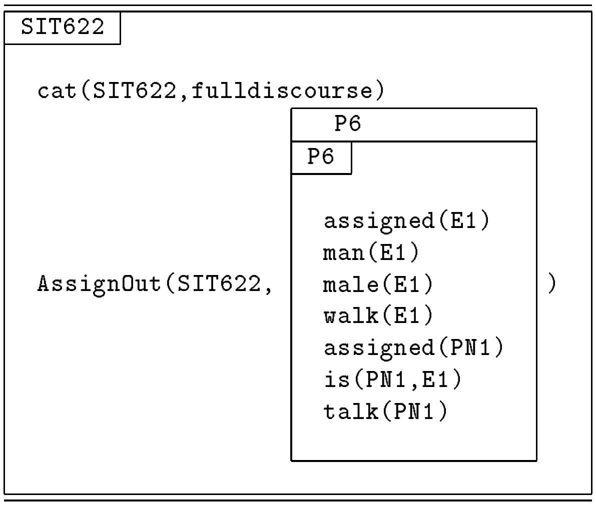
we already lose this direct relationship as neither the DPL representation or the DRS one offers sub-expressions directly representing sub-utterances.

$\forall x[man(x) \rightarrow walk(x)]$



So according to Groenendijk and Stokhof's definition of compositionality DPL itself fails when we look at the universal quantifier. To be fair they only really concern themselves with sentence-sentence compositionality and not intra-sentential. But one of the objects of this exercise is to give a uniform treatment for inter-sentential and intra-sentential anaphora so we cannot simply treat these as different issues.

In DPL-NL the representation itself is not actually DPL but closer to a representation of the first order meta-language that gives the semantics of a DPL statement. Therefore the compositionality of expressions is partially lost. The representation given has the same shortcomings as the DRS representation as in the example *"A man walks. He talks"*: there is no sub-expression of the representation directly related to the initial sentence—unless we discuss the implicit conjunction in types (which is also true for DRSs).



In summary the differences between DPL-NL and DRT seem only to be in the information that is in the assignments/DRSs. The extra information in the assignments is not due to DPL being "non-representational" or "compositional". It would be possible to give a treatment of DRT which would also carry around this extra information in the right hand box of the `=>` relation.

The description of DPL-NL in ASTL shows how close the relationship between DRT and dynamic semantics is. This admittedly has partly been deliberate as the DRT description given in Chapter 5 deliberately emphasizes the dynamic aspects of that theory. Also the representation of DPL assignments has been chosen so that they are very similar to DRSs. Such choices in representation although deliberate are not misrepresenting the close relationship between the two theories. They are both designed to describe the same phenomena and both use the same fundamental techniques to achieve this. Because of this closeness we should not view these as opposing theories but alternative ways to achieve the same result. It should be possible extensions to either theory to be adopted by the other.

6.7 Summary

In this chapter we have described dynamic predicate logic (DPL) and how such a logic may be described in ASTL. Unlike previous chapters which deal with natural language here we describe a logic within ASTL. Then we show how a DPL treatment can be given to the Rooth natural language fragment. The translation re-uses much of the description used in the previous chapter on DRT showing the similarities between dynamic semantics and DRT. Finally a comparison between DPL-NL, the dynamic semantic treatment of natural language, and the ASTL treatment of DRT is given showing exactly the points where the theories differ.

Chapter 7

Extensions

7.1 Introduction

We have proposed situation theory, or more particularly ASTL, as a meta-theory for describing general natural language semantic theories. We have shown how various aspects of contemporary theories can be encoded within ASTL (STG, DRT and dynamic semantics). Given that these encodings are in the same system, detailed comparisons are possible. However, as stated in Chapter 2, one of the ultimate goals in this work is not just to offer a general environment for implementing and comparing theories, which is in itself useful, but also to be able to cross-pollinate ideas and techniques between theories.

In this chapter we will extend our DRT description to include event discourse markers, thus allowing pronouns to have sentence antecedents. Event discourse markers have already been discussed as part of DRT (see [Partee 84], [Kamp & Reyle 93] and others), but here we will show how they naturally fit into our description using properties which are already part of ASTL. The second example shows how we take the treatment of pronouns from our DRT description add it to our STG description, showing how techniques can be re-used in what would previously have been considered different frameworks.

The third part of this chapter discusses what extensions to ASTL itself would be useful to allow a wider coverage of treatments found in semantic theories and making existing descriptions easier.

7.2 Extending DRT in ASTL

In this section we will show how a simple extension can be added to the basic DRT fragment we described in Chapter 5. This extension is simple, and has been considered before (see [Glasbey 91] for a brief history of a treatment of events in DRT), but it shows how we can add to DRT by directly using aspects available in situation theory.

The extension considered here is adding event discourse markers, allowing sentence anaphora. The intention is to deal with such examples as

Hanako sees Taro sing. Anna sees it too.

The important point that we wish to treat is that the referent of *"it"* is *"Taro sing[s]"*, a sentence rather than a simple noun. (We will not try to give any treatment for the word *"too"*.) In order for *"it"* to have a sentence referent we need to state that sentences introduce event discourse markers.

It should be said that the following is not the only way to achieve the desired result there are other possibilities but the exercise illustrates how useful and easy ASTL is in developing theories. In the description of basic DRT we gave in Section 5.2, discourse markers are introduced only for nouns. Here we wish to add that and introduce discourse markers for sentences. We will call this new form of discourse marker *event discourse markers*. Normal discourse markers are, in the interpretation of a DRS, bound to individuals in the model, while event discourse markers need to be bound to more complex objects. Within the ASTL framework we have an obvious candidate, situations. Event discourse markers represent situations of a type as defined by some DRS. Event discourse markers will only be introduced for sentences used as complements rather than all sentences. This seems to be partially linguistically justified but is primarily done to reduce extra ambiguity which would complicate our description.

In this extension to DRT the output DRS for the utterance *"Hanako sees Taro sing"* is

```
+-----------------------------------------------+
|  P6                                           |
+-----------------------------------------------+
|P6 |                                           |
+---+                                           |
|                                               |
|  see(H,E1)                                    |
|                  +-----------------+          |
|                  |      P3         |          |
|                  +-----------------+          |
|                  |P3 |             |          |
|                  +---+             |          |
|  is-of-type(E1,  |  sing(T)        |  )       |
|                  |  named(T,"Taro")|          |
|                  |  type(T,male)   |          |
|                  |                 |          |
|                  +-----------------+          |
|  type(E1,neuter)                              |
|  named(H,"Hanako")                            |
|  type(H,female)                               |
|                                               |
+-----------------------------------------------+
```

This requires a little explanation. First notice that `E1` is a situation name (and also an event discourse marker). Notice we specify the type `neuter` on `E1` so that it may be a referent for the genderless pronoun *"it"*. The special condition `is-type-of` relates situation to the DRS (a parametric situation type). The details of this relation as described below.

To achieve this extension to our simple DRT description we first have to increase the syntax of our fragment to allow for sentence complements. This is simply done by adding an extra VP rule. We also have to worry about the form of the embedded sentence, (it has no agreement). Such syntactic problems are not important to this example and can trivially be dealt with by adding various "features" to the utterance situations.

After adding the necessary syntax and threading information we have to add a constraint for sentence complement utterance situations.

```
*S : [S ! S != <<DRSOut,S,
               *DRSOut :: *DRSIn &
                 [DS ! DS != <<*VR1,*VA1,*EDM,1>>
                       DS != <<is-of-type,*EDM,*DS,1>>
                       DS != <<type,*EDM,neuter,1>>],1>>]
<=
 *S : [S ! S != <<cat,S,sentence,1>>
           S != <<daughter,S, *D1 ::
                  [D ! D != <<cat,D,verbphrase,1>>
                       D != <<daughter,D,*D2 ::
                              [D2 ! D2 != <<cat,D2,sentence,1>>
                                    D2 != <<threads,D2,*Y ::
                         [TS ! TS != <<t-out,*D2,*X ::
                           [T ! T != <<DRSOut,T,*DS,1>>],
                                            1>>],1>>],1>>],1>>
           S != <<DRSIn,S,*DRSIn,1>>
           S != <<sem,S,<<*R1,*A1,*A2,1>>,1>>
           S != <<env,S,*SEnv ::
                     [Env ! Env != <<anchor,*R1,*VR1,1>>
                            Env != <<anchor,*A1,*VA1,1>>
                            Env != <<anchor,*A2,*VA2,1>>],1>>].
```

This apparently rather complex constraint states that the `DRSOut` of a sentence containing an embedded sentence is the input DRS plus a condition about the main sentence. This is made from the main verb phrase relation, the subject argument and an event discourse marker (a situation name). We also use the relation `is-of-type` between the event discourse marker and the output DRS of the embedded sentence. Thirdly we have a condition noting the pronominal type of the event discourse marker.

In addition to a constraint dealing with the incoming and outgoing DRSs we also need to make appropriate changes to the accessibility conditions. Now that a discourse marker for the embedded sentence has been added to the DRS it allows later pronouns to refer to that event. The `DRSout` for the utterance *"Hanako sees Taro sing. Anna sees it too"* is

```
 P11
 P11

 see(A,PN3)
 is(PN3,E7)
 named(A,"Anna")
 type(A,female)
 see(H,E1)
                        P3
                        P3

 is-of-type(E1,         sing(T)            )
                        named(T,"Taro")
                        type(T,male)

 type(E1,neuter)
 named(H,"Hanako")
 type(H,female)
```

Event discourse markers fit neatly into the situation theoretic framework however we do need to add a constraint to say what it means to support an `is-of-type` fact. As with the constraints given on page 81 which describe how the output DRS relates to the described situation we need to state how the DRS parametric situation type relates to the event discourse marker situation. For the above example we can capture this with the following two constraints.

```
*A::[S ! S != <<anchor,T,*T,1>>]
<=
 *S::[S ! S != <<DRSOut,S,
                 [P1 ! P1 != <<is-of-type,*E,
                              [P2 ! P2 != <<sing,T,1>>
                                    P2 != <<named,T,"Taro",1>>
                                    P2 != <<type,T,male,1>>],
                              1>>],1>>
          S != <<drs-anchor,S,*A,1>>].

*E :: [S ! S != <<sing,*T,1>>
           S != <<named,*T,"Taro",1>>
           S != <<male,*T,1>>]
<=
 *S::[S ! S != <<DRSOut,S,
                 [P1 ! P1 != <<is-of-type,*E,
                              [P2 ! P2 != <<sing,T,1>>
                                    P2 != <<named,T,"Taro",1>>
                                    P2 != <<type,T,male,1>>],
                              1>>],1>>
          S != <<drs-anchor,S,
```

```
[A ! A != <<anchor,T,*T,1>>],1>>].
```

This can be read as for every output DRS with a condition `is-of-type` between a situation e and a DRS d as described above there exists an anchoring for the discourse marker `T` such that there exists a situation e which is of the type formed from anchoring `T` in d.

Of course the above example has been simplified drastically. There are obvious problems especially to do with the fact that objects introduced within the embedded sentence will not be accessible to pronouns outside that sentence, but it is issues like this which are easy to investigate in ASTL that make ASTL a useful tool in experimenting with semantic theories.

Importantly we have not had to increase our ontology within the system to add event discourse markers to our simple DRT fragment. The fact that situations are already part of our model mean that they can easily be utilised as event discourse markers.

7.3 Pronouns and Situation Theoretic Grammar

In the work of Situation Theoretic Grammar (STG) there is no treatment of pronouns given in the original definition [Cooper 89]. As stated, describing theories within the framework of ASTL should allow the cross-pollination of treatments between theories. Rather than devise a new treatment for anaphora within STG it would illustrate ASTL's usefulness more if we could take the treatment of anaphora from our DRT description (given in Chapter 5) and add it to the basic STG description (from Chapter 4). In this section we will do just this.

In STG the utterance situation representing the utterance *"A man walks"* is related to a number of semantically relevant objects. The semantics of the phrase itself is captured through the relations `sem` and `env`. The argument to `sem` is a parametric fact. The parameters are anchored by facts in the environment situation which is an argument to the relation `env`. Thus the basic form of the utterance situation for *"A man walks"* is

```
SIT4627

cat(SIT4627,sentence)
tense(SIT4627,pres)
sem(SIT4627, Q1(A1,A2,A3) )

             SIT4589

               anchor(WA1,MA1)
               anchor(A3, walk(WA1) )
               anchor(A1,MA1)
               anchor(A2, man(MA1) )
env(SIT4627,   label(Q1,quantifier)          )
               anchor(Q1,some)
               label(A1,var)
               label(A2,range)
               label(A3,body)
```

In addition to this, this situation is also related to a described situation of the form

```
SIT5376

                P12            P15
              P12            P15
some(MA1,                ,                 )
               man(MA1)        walk(MA1)
```

We can continue with this basic representation for the semantics of a simple sentence even when we treat pronouns. In order to give a treatment for pronouns we need to record which possible referents are available when a pronoun is used. The feature we must then adopt from our DRT description is the concept of *accessibility*. In the DRT description each utterance situation, in addition to an incoming and outgoing DRS, is also related to an explict incoming and outgoing *accessibility situation*. That situation supports facts about all discourse markers which are accessible as pronoun referents at that point in the discourse.

In order to give the same treatment for pronouns in STG we need to copy three things from the DRT description. First we need the threading relations (`t-in`, `t-body` and `t-range`) and the necessary parts that are used in the construction of these relations. These define an alternative structure over the utterance situations in a discourse. Because the threading relation (as discussed in Section 5.3.2) is given abstractly from the DRSs, we can talk about copying it without also having to copy the threaded DRSs too. The second feature we need is that of the accessibility situation. In the DRT treatment we thread DRSs and an accessibility situation through each utterance situation.

Here we need only thread the accessibility situation. That is each utterance situation will be related to an incoming and outgoing accessibility situation. The connection between these will be defined with repsect to the threading relations. The incoming accessibility situation will come from the outgoing accessibility situation previous in the thread. This can be stated by the following constraint

```
*S:[S ! S != <<AccessIn,S,*Access,1>>]
 <=
   *TS:[TS ! TS != <<t-in,*S1,*S,1>>],
   *S1:[S1 ! S1 != <<AccessOut,S1,*Access,1>>].
```

(We also need to ensure the correct threading at determiner nodes in the same way as we do in the DRT (and DPL-NL) descriptions.)

In addition to the actual threading, every utterance situation for nouns has to add a "marker" (actually the parameter introduced by the noun) to the accessibility situation. This again can be copied from the DRT description. Such a constraint for proper nouns would be

```
*S:[S ! S != <<AccessOut,S,
               *A :: *AType &
                     [A ! A != <<accessible,*X,*TYPE,1>>],1>>]
<=
 *S:[S ! S != <<cat,S,ProperNoun,1>>
         S != <<use_of,S,*N,1>>
         S != <<sem,S,*X,1>>
         S != <<type,S,*TYPE,1>>
         S != <<AccessIn,S,
                *A1 :: *AType,1>>].
```

Notice, as with the DRT treatment, we also add typing information for the introduced marker.

The third property we must borrow from DRT is that of determining the referent of a pronoun. That is when we use a pronoun we wish to find possible referents from the accessible markers at that point in the discourse. This can most simply be achieved by the following constraint

```
*S:[S ! S != <<sem,S,*X,1>>]
<=
 *S:[S ! S != <<cat,S,ProNoun,1>>
         S != <<type,S,*TYPE,1>>
         S != <<AccessIn,S,
                *A1 :: [T ! T != <<accessible,*X,*TYPE,1>>],
                1>>]
```

That is we find an accessible marker of the appropriate type (male, female or neuter) and make that parameter the semantics for the pronoun. This of course is a slight simplification of the DRT treatment of pronouns where a new marker is introduced which is related to the referent by the relation `is`. In order to get that generality in our STG description we could introduce an anchor fact in the environment, related to the pronoun utterance situation, anchoring the parameter introduced by pronoun to the referent parameter from the accessibility situation.

After we add the above three aspects from the DRT description to our STG description we have a simple treatment of pronouns. Given the initial sentence *"A man walks"*, the outgoing accessibility situation would be of the form

```
SIT923

  accessible(MA1,male)
```

If the following sentence in the discourse is *"He talks"* the same accessibility situation will be the incoming accessibility situation to the pronoun utterance situation. Using the constraint above this would make the `sem` relation in the pronoun utterance situation

```
<<sem,S,MA1>>
```

The sentence utterance situation for the second sentence would then be.

```
SIT2705

  cat(SIT2705,sentence)
  tense(SIT2705,pres)
  sem(SIT2705, R1(A1) )
                SIT7106

                  anchor(A1,MA1)
  env(SIT2705,    label(R1,pred)    )
                  anchor(R1,talk)
                  label(A1,subj)
```

This is not quite the whole story though. We also have another aspect to deal with in our STG description. In the examples given in Chapter 4 we did not discuss multi-sentential utterances. To give the right treatment for the utterance *"A man walks. He talks"* it is necessary to ensure that the second sentence falls within the scope of any (top-level) existential quantifier introduced in the first.

Note that although we copied over DRT's treatment of pronouns to STG the result is not the same as DRT. Even if we ensure succeeding sentences are in the scope of existential quantifiers, STG still does not provide a reasonable treatment of donkey anaphora. This is due to the way the relation `every` differs in the DRT description and the STG description. Other extensions would be required to deal with donkey anaphora, if we wished to cover them.

However we have shown that we can adopt a treatment of pronouns from one theory into another. The adoption was effectively done by simply copying a number of constraints from the DRT description into the STG description (and some changes to the basic grammar rules). But looking at the original definitions of DRT with its boxes and STG with its situation semantic notation it was not obvious that transfering a treatment from one to the other would be so simple. ASTL has helped show how these theories can be related.

7.4 Extending ASTL

ASTL as described in Chapter 3 is very conservative in what it contains. There are a number of aspects of situation theory which it does not contain. In this section we will discuss some possible extensions. We give details of one particular extension and discuss other in less specific terms.

7.4.1 Abstraction, parameters and anchoring in ASTL

In the basic definition of ASTL, parameters are not treated in any special way. They are only distinguished from individuals in their declaration. The only case where they are treated specially is in situation types where a parameter is used to identify the object (situation) which has been abstracted over, other uses of parameters are only by convention. In the Situation Theoretic Grammar description (in Chapter 4) we used parametric facts to represent abstractions over facts. Specifically we would represent the semantics of intransitive verbs as

```
<<R1,A1,1>>
```

where `R1` and `A1` are parameters. We treat these as something similar to the lambda expression

$$\lambda x \lambda y \; y(x)$$

but in the parametric object case there is no ordering on the abstraction.

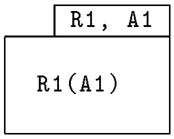
One problem with this current representation of parametric objects is in the scope of the abstraction. For example consider the following expression:

```
<<R1,<<R2,A1,1>>,1>>
```

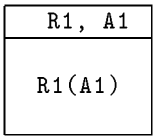
Is this an abstract over a one place relation whose argument is a one place relation, (i.e. there are three parameters) or is this an abstract over a one place relation whose argument is an abstraction itself (i.e. there is only one parameter)? The difference is more obvious in lambda expressions, the above can "mean" either the first or second of these expressions

```
λ R1 [<<R1,(λ R2, A1 [<<R2,A1,1>>]),1>>]
λ R1, R2, A1 [<<R1,<<R2,A1,1>>,1>>]
```

(There are other distinct combinations too.) The difference can (though not always) be important. In EKN ([Barwise & Cooper 93]) abstractions are represented in a form more like lambda abstraction than ASTL ones. They are explicit about the parameters they are abstracting over. There are two notations for abstraction: the first is the most general



The second case is when an abstract can be used as a relation.



There is no real difference between these two notations but the distinction can be useful in identifying their use.

Let us now propose an extension to ASTL and call that extended language ASTL+. The following is directly extending the formal definition of ASTL given in Chapter 3. In addition to the terms we previously defined let us add three new terms: *abstractions*, *anchoring environments* and *reductions*. The first of these, *abstractions*, have the following syntax

$[\ \ P_1, \ldots, P_n \ |\ \langle term \rangle\ ]$

where $P_1, \ldots, P_n$ are parameters. So for example in the STG description we could now represent the semantics of intransitive verbs as

```
[R1, A1 | <<R1,A1,1>>]
```

The denotation for an abstraction term will be an abstraction in the model. This requires us to enrich our model with a more elaborate set of types **T** as the present set **T** contains only situation types, here we need to extend it to include types for all objects. There are some consequences from such a semantics. The following two terms have the same denotation.

```
[R1, A1 | <<R1,A1,1>>]
[R2, A2 | <<R2,A2,1>>]
```

Thus we have alpha equivalence between abstractions. Unfortunately this notation does have a problem. As we wish to "apply" these abstractions to other objects and "reduce" the results there must be some unique way to refer to the parameters appearing before the vertical bar. The problem is, are the following two terms equivalent?

```
[R1, A1 | <<R1,A1,1>>]
[A1, R1 | <<R1,A1,1>>]
```

In ASTL+ we will say that they are. But because we do say that they are and wish to say that the arguments (`A1` and `R1`) are truly unordered there can be no reasonable way to refer to the arguments from outside the abstraction—without actually using the same name. Naming the parameters outside the abstraction and expecting them to have the same semantics seems unreasonable, besides if we have alpha equivalence this implies the parameters names are irrelevant. Using their position in the abstraction requires some concept of ordering (which contradicts the immediate example above). The way we solve this is by allowing the arguments in an abstraction to be *labelled*. As the labels have a scope (actually global) wider than the abstraction themselves this will allow them to be individually referenced.[1] A label (an ASTL individual) can follow the parameter argument separated by an at-sign.

```
[R1@pred, A1@subj | <<R1,A1,1>>]
```

That completes the outline for a definition of abstractions as terms in ASTL+. At this stage it should be stated that this form of abstraction is the same as that described in [Aczel & Lunnon 91]. Although they use a different syntax their treatment of abstractions is effectively that same as the one presentedabove.

We also now need is a second term, that we will call an *anchoring environment*. This is in fact not a new term but a special case of an already existing one. An *anchoring environment* is a situation which supports facts with relation `anchor-to`. (We do not restrict *anchoring environment* to only supporting `anchor-to`-facts as this seems

[1] Examples like

```
[R1@pred, A1@subj | <<R1,A1,1>>]
[R1@verb, A1@arg1 | <<R1,A1,1>>]
```

are equivalent but not identical.

unnecessary but in all examples anchoring environments will only contain `anchor-to` facts.) The relation `anchor-to` takes two arguments, a label and an arbitrary term. An example is

```
A1::[S ! S != <<anchor-to,pred,sing,1>>
         S != <<anchor-to,subj,hanako,1>>]
```

The third term we need is to allow us to relate an abstraction to an anchoring environment and hence the reduced form. We can do this by introducing another term of the form

<*abstraction*> // <*anchoring environment*>

An example is

```
[R1@pred, A1@subj | <<R1,A1,1>>] //
A1::[S ! S != <<anchor-to,pred,sing,1>>
         S != <<anchor-to,subj,hanako,1>>]
```

Basically such a term denotes the same object as its fully reduced form. In this case the above denotes the same as

```
<<sing,hanako,1>>
```

More formally a term of the form

<*abstraction*> // <*anchoring environment*>

denotes the same as the syntactic object that is formed by the following reduction: for each label L_i in the anchoring environment that is related to a term T_i by the relation `anchor-to` and appears as a label to a parameter P_i in the arguments of the abstraction, replace any occurrence of P_i in the body of the abstraction with T_i and remove that parameter from the argument list. If there are no parameters in the abstraction's parameter list the whole expression can be replaced by the body of the abstraction.

But, unfortunately such a simple definition of abstraction, anchoring and reduction requires some more restrictions in computational environment. There are a number of problem cases for which solutions have not been defined. First we have got to add the restriction that a label may only appear at most once as the first argument to an `anchor-to`-fact in any anchoring environment. That is the reduction *must* be functional.

The above definition does not imply that a reduction actually occurs. Because the semantics of the unreduced form has the same denotation as the reduced form there is no need to actually "calculate" it. As an analogy, even if we know $2+2=4$ then a perfectly valid answer to the question what does $2+2$ equal, is $2+2$. Therefore what is also needed is an inference rule which states that if a situation supports the unreduced form it also supports the reduced form. With such a rule we should be able to infer the following. Obviously

```
Sit1:[S ! S != <<sings,hanako,1>>]
```

can be inferred from

```
Sit1:[S ! S != [R1@pred, A1@subj | <<R1,A1,1>>] //
               A1::[S ! S != <<anchor-to,pred,sing,1>>
                        S != <<anchor-to,subj,hanako,1>>]
```

But the following can also be inferred as well

```
Sit1:[S ! S != [A1@subj | <<sing,A1,1>>] //
               A1::[S ! S != <<anchor-to,pred,sing,1>>
                        S != <<anchor-to,subj,hanako,1>>]
Sit1:[S ! S != [R1@pred | <<R1,hanako,1>>] //
               A1::[S ! S != <<anchor-to,pred,sing,1>>
                        S != <<anchor-to,subj,hanako,1>>]
```

Also note that if the following simple proposition is true

```
Sit1:[S ! S != <<sings,hanako,1>>]
```

then from the same inference rule we should be able to deduce

```
Sit1:[S ! S != [A@L1, B@L2 | <<A,B,1>>] //
               Q::[S ! S != <<anchor-to,L1,sing,1>>
                       S != <<anchor-to,L2,hanako,1>>]
```

and infinitely many other propositions. However in a computational system we would wish to restrict inferences in the reduction direction only—at least this would be the simplest way to implement it.

We have given the outline of an extension to ASTL which would make the writing of descriptions of theories easier. Although we have not fully specified the extension it is hoped that the above gives the general idea and that there are not too many real problems. With the above extension the STG description given in Chapter 4 could be improved. In that description "reduction" of parametric objects and anchoring environments was attempted using conventional constraints but this is not general enough. Using the extensions described above we can replace the more complex rules with something a little more readable.

```
[S ! S != <<cat,S,Sentence,1>>
     S != <<sem,S,
                *Fact // *A::*VPEnv &
                          [Env ! Env != <<anchor-to,subj,*Y,1>>],
                1>>]
->
 [NP ! NP != <<cat,NP,NounPhrase,1>>
       NP != <<sem,NP,*Y,1>>],
 [VP ! VP != <<cat,VP,VerbPhrase,1>>
       VP != <<sem,VP,*Fact,1>>
       VP != <<env,VP,*VPEnv,1>>].
```

The semantic form for the verb phrase would now be an abstraction, for example *"walks"* would be of the form

```
[R1@pred, A1@subj ! <<R1,A1,1>>]
```

In addition to making the STG description more succinct the above extensions would allow the description of DMG in [Beaver *et al* 91] to be easily implemented. Also other work in EKN which also makes a heavy use of generalised abstraction and reduction should then be describable. Some of the later work in EKN also appeals to Aczel-Lunnon abstraction and hence such an extension to ASTL would allow more theories to be described more easily.

Another aspect where abstractions would improve a description of a semantic theory in ASTL is in the DRT descriptions. DRSs could perhaps better be represented as abstractions over situation types rather than as at present parametric situation types.

The extensions described above are significant in that they describe a way that may remove the need for parameters to appear in any place other than abstractions. The concept of parametric objects (indeterminates) seemed important. With a definition of abstraction it seems that general "free" parameters are, at least, no longer needed often and perhaps may be completely unnecessary.

The fact that the role of general parameters might be able to be replaced by abstraction is an interesting possibility. One of the original justifications of situation theory was to allow parametric objects in the model. By using abstractions to represent indeterminates we make the problem similar to the use of lambda abstractions for which there is a well understood semantics (see [Barendregt 81]). However it should also be noted that "traditional" parameters are very useful in writing general semantic theory descriptions. The ability to use them as variables within the object theory makes situation theory useful as a meta-language. Although there probably is a way to re-cast all the techniques described in semantic theories that use parameters as variables it is not right for us to force this to be the case. As there is not any computational or implementational problems in having simple parameters in the model its seems unnecessary to remove them from the set of tools provided. This question of parameters versus abstractions will unfortunately remain a philosophical one.

7.4.2 Using semantic translations

Throughout the three major examples described above we have only really been concerned with defining (and constructing) semantic translations for utterances. We have not concerned ourselves with using the results: for example asserting the results to a database and drawing inferences from them. As one of the points, if not the most important point, of building semantic translations, is to actually use them in a computational system it would add further to ASTL's usefulness if we could give such examples.

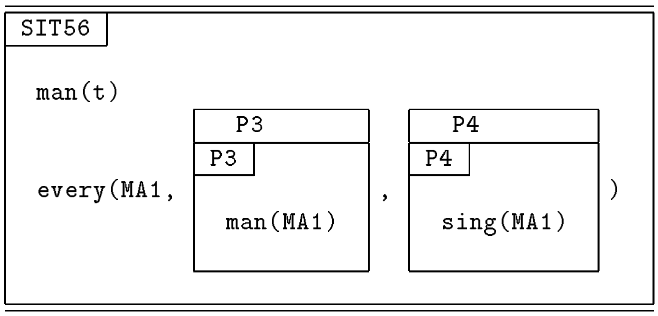
In all three cases the semantic translation is a parametric situation type and constraints are specified as to how it relates to the type of the described situation. Admittedly some manipulation is required in order to treat quantifiers properly but we can for the sake of discussion view the semantic translation simply as a situation type.

ASTL, in order to make using translation easier, requires some extensions. Allowing constraints as terms would allow for more complex descriptions however it is probably adequate to introduce some distinguished relations with a special treatment (e.g. `every`).

Let us briefly look a simple discourse and see what might be required. Suppose we wish to treat the following discourse

> Every man sings.
> Taro is man.
> Who sings?

If we use a STG semantic treatment the described situation after the second sentence would be



We of course need to consider a treatment of questions. As our description currently stands we have no such treatment but we might consider something like the following. Ideally we would like the semantic translation of a sentence to be some form of parametric type. For example *"Who sings"* would translate as

```
+-----------+
|    P1     |
+----+------+
| P1 |      |
+----+      |
|  sing(X)  |
|           |
+-----------+
```

The answer to our question is what can be anchored to `X` such that it becomes a type of the described situation. Using the extension to ASTL described in the previous section we might write

$$S_{ds} : Q \; // \; Answer$$

The anchoring environment *Answer* would contain the information needed to generate an answer. Of course generation of pragmatically useful answers is in its own right a research topic.

Questions are still an active research area in computational semantics, and the above is not intended to be a serious attempt at treating questions only a small illustration. Work has been done in situation semantics of questions but as yet do not have an implementation [Ginzburg 92]. There is also work on questions within a dynamic semantic framework [Groenendijk & Stokhof 92], but they do not depend of dynamic semantics for their treatment of questions and use it only for treatments of quantificiation and anaphora. If descriptions of such theories could be written in ASTL this would directly lead to their implementation.

Because we are in a situation theoretic framework we can take advantage of the objects available—particularly situations. It seems possible to have descriptions in ASTL (or some reasonable extension of ASTL) where our "database" is more complex than simple facts. Situations allow us to more easily represent phenomena like beliefs, attitudes, etc. Perhaps borrowing from the knowledge representational descriptions given in PROSIT would allow interesting experiments to be made in higher level aspects of semantics and discourse modelling. Of course this would be a significant amount of work but implementation should be possible through an ASTL-like language.

Another direction in which ASTL could be extended in *coherence* and *defeasible constraints*. At present there is no built in mechanism for ensuring that situations of the form

```
SIT1 :: [S ! S != <<walks,hanako,1>>
             S != <<walks,hanako,0>>]
```

are given no denotation. A treatment of *coherence*, ensuring a situation does not support a fact and its dual, although not necessary is often included in basic aspects of situation semantics. A treatment of coherence would lead to a better treatment of negation. Somewhat related to this topic are the notions of more complex constraints such as negative constraints and defeasible constraints. The area of defeasible constraints and non-monotonic reasoning is in itself a research topic but it would be useful if treatments could be brought together in the same framework as treatments of natural language discourse.

All of the above show that ASTL has many directions in which it can be extended. Although we have shown its basic competence in the field of computational semantics there is still many useful extensions we could make.

7.5 Summary

In this chapter we have attempted to show two things. First that semantic descriptions in ASTL can easily take ideas and techniques from other theories in order to provide better overall theories. Describing theories in the same environment (i.e. ASTL) allows not only for differences between theories to be identified but for treatments of various semantic phenomena to be copied.

Secondly future changes to ASTL are discussed. An extension to deal with abstractions and reduction is detailed which would allow easier treatment of a number of techniques used in various semantic theories. Also some discussion of how to use semantic translations of utterances is given.

Overall this is intended to show that even as it currently stands ASTL is a useful tool in development of natural language semantic theories but that there are obvious extensions which can be made which would increase ASTL's usefulness.

Chapter 8

Conclusions

In this final chapter we will restate the major points of this thesis and try to draw some conclusions from the work. We will identify what the characteristics of ASTL are and why they are important in a computational language for representing semantic theories. Finally some discussion is given of the future direction of this research and how it contributes to the field of computational semantics.

After some general discussion of contemporary issues in computational semantics, in Chapter 2 we discussed the idea of building a computational framework in which general aspects of computational semantic theories of natural language may be described and experimented with. Because of the broad similarities between some contemporary theories this seems a useful direction in which to head and has been the subject of other research (e.g. [Johnson & Kay 90]). A uniform environment for implementation and experimentation should allow closer comparison of theories and help to identify the exact differences and similarities between them. Also this hopefully will lead to methods for sharing techniques and treatments between theories by extending theories as well as the possibility of creating hybrids where no conflict exists. A number of possible areas from which a basis for a computational language for semantic theories might be found are discussed, including logic programming (as in Prolog), feature structures and situation theory.

In Chapter 3 we introduced the language ASTL. ASTL is designed as a computational language for describing natural language semantic theories. ASTL is defined with respect to basic aspects of situation theory. Its semantics is given in terms of a situation theoretic model. The language offers representations for individuals, relations, parameters, situations, variables, situation types and constraints. Also a set of inference rules are defined in order that we can draw inferences from a system of situations and constraints. Importantly, ASTL is not just a theoretical language, it has an actual implementation. We describe an implementation and give simple examples of how it can be used.

In order to show that ASTL is a suitable implementation device for at least the basic aspects of contemporary natural language semantic theories the following three chapters gave detailed example descriptions of three different theories. By *descriptions* we mean formal specifications of aspects of semantic theories. We could go further as try ensure that formalisations of theories in ASTL are formally equivalent with axiomatizations of the theories we are interested in. This was not done because it is difficult to find axiomatizations of theories due to them constantly changing and improving thus usually making any axiomatization out of date. Instead we have looked at formalizations of classic analyses of phenomena within the theory being described. After all it those analyses of particular phenomena that we actually wish to compare.

First we introduced a method of representing syntactic structures and grammar rules within ASTL. A simple semantics was added to this based on the work of Situation Theoretic Grammar [Cooper 89]. This showed how a situation semantic theory can neatly be described within ASTL. This set the scene showing how we can use ASTL as an environment for describing semantic theories and use those descriptions to derive semantic analyses from utterances. Next we described two different semantic theories which specifically address the same semantic phenomena. Chapter 5 deals with Discourse Representation Theory, [Kamp 81, Kamp & Reyle 93]. DRT offers a representation for natural language discourses. A description of the theory itself is given and an ASTL description is presented which adequately captures the main issues of DRT. The ASTL description is based on the DRT fragment in [Johnson & Klein 86]. The third example of a semantic theory described in ASTL is given in Chapter 6. This deals with the general area of dynamic semantics ([Groenendijk & Stokhof 91b], [Groenendijk & Stokhof 91a]. Two small examples are given, first a treatment of Dynamic Predicate Logic (DPL) is given then a dynamic semantic treatment is given for the same simple natural language syntactic fragment used in the DRT and STG descriptions. The dynamic semantic description re-uses much of the description of DRT. Because DRT and DPL-NL are intended to describe the same semantic phenomena (i.e. aspects of anaphora), once described in ASTL we can easily give a detailed comparison of the theories. The results seems to show that they differ in their representation of the amount of information at each stage in a discourse which may impinge on efficient inference from that result. An important aspect of these two descriptions is how much can actually be directly shared between them. Both DRT and DPL-NL use the same constraints with regards threading of information and only aspects of the information passed along these threads differ. We also showed that the treatment of pronouns from DRT could easily be adopted into Situation Theoretic grammar once both had been described within ASTL, something that would not be obvious when first looking at the semantic representation used in each theory.

Even though we have only partially described three theories in ASTL it seems reasonable to claim that ASTL is a suitable environment for formalizing, implementing, comparing, and developing such theories. Although it should not be very surprising that these theories can be described within the same framework actually showing this is a necessary prerequisite for such a claim. Also although ASTL comprises just very basic aspects of situation theory it is sufficient to give a useful level for semantic description. It should be stated that ASTL is just an experimental system and it is not, in its present form, proposed as a practical system for large scale implementation of theories for the semantic component of practical language processing systems. Chapter 7 describes an extension to ASTL, namely abstraction and reduction, which would make ASTL a much more usable system, but other enhancements would be necessary to make a general implementation system—but these small example descriptions do suggest that these enhancements would be worthwhile.

Another aspect that deserves some mention is how much does ASTL influence the descriptions of theories made within it. Any system of this form will influence formalisations in it. Some of these restrictions are merely arbitrary, such as having to use a linear form to conform to the syntax of ASTL, as opposed to using boxes. Other restrictions are more to do with ASTL itself (and the underlying situation theoretic aspects of the language). ASTL's basic mechanism of constraints means that everything has to be specified in that form, even though an original theory may depend on abstraction and application. These are equivalent (at some level) but it may require looking at a theory in a slightly different way. Although ASTL offers situations as objects it does not require descriptions to use them, though as there are no constructs such as sets or lists there is a certain encouragement. ASTL tries not to constrain descriptions very much, trying to be a tool rather than a specific theory, of course it may be that descriptions (as is the case in the descriptions given in this thesis) can be described in a very similar way but at least some of that is by design rather than forced by the ASTL language. In fact as we wish to use ASTL as an environment for comparing and mixing theories using similar techniques to describe theories is an advantage as long as it does not restrict (too much) the theories that can be described.

A general problem with computational semantics is that there is always a conflict between the "engineers" and "theorists". In this field there is both the temptation to make theories more formal thus making more explicit the underlying properties of the theory, and more computational thus making implementation better (faster, more tractable, easier to use, etc.). This thesis has tried to firmly set itself between these goals paying respect to both sides. However there is always the argument that the theorists may criticise this work because the descriptions of their theories are not complete and the engineers may criticise because they can achieve must faster implementations or greater coverage by resorting to a more general programming language or "bending" a theory a little. Although both critics have valid points it is the whole enterprise that must be judged. Perhaps, they could look from each other's perspective.

From the engineer's point of view we have built a sound system that does work, although it may not have all the debugging aids and efficiency of "real systems". But ASTL has a good theoretical basis and few corners have been cut for the sake of the implementation. This shows that theory can be practical. Also through implementation of these theories we begin to understand the essential properties of these theories such that if short cuts really are necessary in implementation we can more easily identify which short cuts can be made without losing the fundamental treatments of the phenomena we are trying to describe.

From the theorist's point of view this work has tried to use a theoretical basis for the language ASTL. Although we have only described minimal parts of semantic theories within ASTL we have shown that the theories are computational and can have a reasonable implementation. Implementation of a theory allows for a better testing of its computational properties and also allows easy experimentation. Also in describing a theory such that it can be run requires a much more explicit definition that might otherwise be given.

Another theoretical aspect of this work is that of using a situation theoretic language as a "meta-language" for describing natural language semantic theories. Although it would have been possible, or even easier to merely treat "ASTL" (the implementation mechanism) as simply a formalism similar to a feature system or some form of Prolog, using a formalism which is fully given a firm theoretical grounding and a formal semantics makes it clearer about what is going on in descriptions. We can make stronger claims about the descriptions we give as well as having the possibility of using existing formal aspects of situation theory. The fact that we are using situation theory as the basis for ASTL is in itself interesting research as it has not been clear before that situation theory could offer the basis for a computational language but ASTL (and PROSIT ([Nakashima *et al* 88],[Frank & Schütze 90])) has gone some way to add to this claim.

Much work has been done in the area of situation semantics and situation theory but the idea of a computational language based on situation theory, in a similar way that Prolog is based on first order logic, is relatively new. Representational formalisms have been defined (e.g. situation schema) but a language in which computation (i.e. something akin to inference) was new. PROSIT ([Nakashima *et al* 88],[Frank & Schütze 90]) is the only other example. Unlike PROSIT, ASTL tries only to use features which can be described in terms of core aspects of situation theory. This of course restricts the language and perhaps makes it harder to "program" in but means that descriptions in it will have a clear semantics. It is really the definition of constraints and inference that make ASTL computational.

From the other viewpoint showing that aspects of contemporary theories of semantics can be described within a situation theoretic framework shows a use for situation theory which has not really been made explicit before. Situation theory is a very general mathematical framework, this work has helped to show how it can be used *with* current semantic theories rather than as an alternative or opposing framework.

It is worth discussing what alternative language could be used instead of ASTL. This might help identify what properties of ASTL are essential in making it a suitable as a semantic meta-language. The substantial work in the area of feature systems has produced a large number of variant systems each of which is tailored for various tasks. As described in Section 3.7.3 it would be possible to define a feature system which had the necessary properties but this could be reasonably argued as an implementation of ASTL itself. A situation theoretic semantics could be given for such a feature formalism. Also it should be possible to code up ASTL-like descriptions in logic programming languages like Prolog. After all there already exist implementations in Prolog of our three basic example theories, but it is not just the end result of implementation that we are looking for we are also trying to understand what the basic essential computational properties of these theories are. Arbitrary implementations even in the same language will not necessarily help us.

Even if we had the freedom of a general programming language the essential properties that would be used would be those of ASTL. These can be summarised by the following list.

- The ability to represent complex structured objects and allow general relations between them.
- The ability to have constraints between general objects and draw inferences from them.
- A mechanism for reasoning about "variables" in the object language (as distinct from variables in the meta-language) and describing binding mechanisms for these object language variables.

These are the minimum *descriptive* properties that seem necessary. ASTL goes a step further by not only offering these but also offering a formal semantics and not just a formalism. In situation theoretic terms the above properties relate directly to: situations and abstractions; constraints; and parameters and anchoring. All three of these are fundamental properties of situation theory. If we look for such properties in other areas of formal semantics, we can find some but not all very easily. In a Montague-like framework the use of named (partial) possible worlds (as in the work of [Muskens 89]) offers semantic objects similar (or even equivalent for many purposes) to situations.

This thesis is only a first step in a general mechanism for describing and implementing semantic theories of natural language. There are many directions (not all conflicting) in which this work can be continued. From the computational point of view ASTL can be enhanced adding new formal features—for example the general abstraction and anchoring described in Section 7.4.1. There is little in the language that deals with *coherence*: some treatment that deals with ensuring that situations do not support facts and their duals would allow better treatments of negation. Defeasible constraints would lead to better modelling of general knowledge representation and would aid the modelling of belief.

As well as the formal aspects of extensions there are practical aspects too. It is important that a computational system be easy to use. It should not be considered just as an afterthought. Developing computational semantic theories is hard. The full consequences of formal decisions are not always obvious. Experimentation can help enormously but only if the implementation is easy to use. Reasonable speed and debugging facilities are real aids to the computational semanticist. A good method for displaying results and allowing the semanticist to easily see the consequences of their theory makes development significantly easier.

We can consider other directions too. Only minimal descriptions of STG, DRT and dynamic semantics have been given. Larger examples would help confirm the usefulness of the techniques described here. Also it would allow more comparison between theories. Covering extensions to these object theories such as plural anaphora, definites etc. would aid not only the understanding of differences between treatments but also the treatments within the theories themselves as they currently stand. Descriptions of other theories might also be considered. Obvious candidates are Montague Grammar and Dynamic Montague Grammar.

Another direction is in extending what it means to implement the theory. Here we have "implemented" a theory by giving a specification of key aspects of that theory sufficient to derive semantic translations from utterances. Using that translation for database lookup or dialogue modelling would be a better computational test of a theory. ASTL does not, as it currently stands, offer much help in building such active descriptions but it does seem that it would not require many extensions to make the use of semantic translations easier. Moreover because our semantic translations (in some descriptions) are situation types and it is clear what it means for a situation to be of that type, theorem proving with the translation should be relatively easy.

8.1 Final comments

We have described a computational language called ASTL which is given a situation theoretic semantics. ASTL is an example of how situation theory offers a basis for an interesting and powerful language suitable for describing aspects of natural language semantic theories. In order to show this we gave detailed descriptions in ASTL of the basic aspects of three contemporary semantic theories. These are Situation Theoretic Grammar, Discourse Representation Theory and a form of Dynamic Predicate Logic. Because these descriptions are restricted to being in the same environment very detailed comparisons can easily be made identifying exactly the differences between them, particularly in the case of the later two. Also because ASTL has an implementation descriptions of these theories directly offer implementations which can be run to produce semantic translations for utterances.

In conclusion, although we have successfully described a computational language based on situation theory, and showed that at least the core aspects of some contemporary semantic theories of natural language can be neatly described within that language there is still a lot of work to do before we have a reasonably broad computational coverage of natural language semantics. It is important for theoretical semanticists to keep computational and more importantly implementation aspects in mind as much as it is important for implementors of systems to know about theoretical aspects, but neither can do without the other. In this thesis we have taken into consideration both sides of the argument and developed a theoretical approach to the description and implementation of computational semantic theories.

Bibliography

[Aczel & Lunnon 91] P. Aczel and R. Lunnon. Universes and parameters. In *Situation Theory and its Applications, II*, CSLI Lecture Notes Number 26, pages 3–25. Chicago University Press, 1991.

[Ades & Steedman 82] A. Ades and M. Steedman. On the order of words. *Linguistics and Philosophy*, 4:517–558, 1982.

[Ait-Kaci & Nasr 85] H. Ait-Kaci and R. Nasr. LOGIN : A logic programming language with built-in inheritance. Technical Report AI-068-85, MCC, Microelectronics Computer Corporation, 1985.

[Alshawi 92] H. Alshawi. *The Core Language Engine*. MIT Press, Cambridge, Mass., 1992.

[Barendregt 81] H. Barendregt. *The lambda calculus : its syntax and semantics*. North-Holland, Amsterdam, 1981.

[Barwise & Cooper 82] J. Barwise and R. Cooper. Generalized quantifiers and natural language. *Linguistics and Philosophy*, 4:159–219, 1982.

[Barwise & Cooper 93] J. Barwise and R. Cooper. Extended Kamp Notation: a graphical notation for situation theory. In *Situation Theory and its Applications, III*, CSLI Lecture Notes. Chicago University Press, forthcoming 1993.

[Barwise & Etchemendy 87] J. Barwise and J. Etchemendy. *The Liar: an essay on truth and circularity*. Oxford University Press, 1987.

[Barwise & Etchemendy 90] J. Barwise and J. Etchemendy. Information, infons and inference. In *Situation Theory and its Applications, I*, CSLI Lecture Notes Number 22, pages 33–78. Chicago University Press, 1990.

[Barwise & Perry 83] J. Barwise and J. Perry. *Situations and Attitudes*. MIT Press, Cambridge, Mass., 1983.

[Barwise & Seligman 93] J. Barwise and J. Seligman. The rights and wrongs of natural regularity. to be published in *Philosophical Perspectives* vol. 8 or 9 edited by James Tomberlin., forthcoming 1993.

[Barwise 89a] J. Barwise. Notes on branch points in situation theory. In *The Situation in Logic*, CSLI Lecture Notes Number 17, pages 255–276. Chicago University Press, 1989.

[Barwise 89b] J. Barwise. *The Situation in Logic*. CSLI Lecture Notes Number 17. Chicago University Press, 1989.

[Barwise 92] J. Barwise. Information links in domain theory. In *Proceedings of the Mathematical Foundations of Programming Semantics Conference (1991)*, pages 168–192. LNCS 598, Springer, 1992.

[Barwise 93] J. Barwise. Constraints, channels and flow of information. In *Situation Theory and its Applications, III*, CSLI Lecture Notes. Chicago University Press, forthcoming 1993.

[Beaver 91] D. Beaver. DMG through the looking glass. In *Quantification and Anaphora I*, DYANA Deliverable R2.2A, pages 135–153. Centre for Cognitive Science, University of Edinburgh, 1991.

[Beaver *et al* 91] D. Beaver, A. Black, R. Cooper, and I. Lewin. DMG in EKN. In *Partial and Dynamic Semantics III*, DYANA Deliverable R2.1.C, pages 75–91. Centre for Cognitive Science, University of Edinburgh, 1991.

[Black 92] A. Black. Embedding DRT in a Situation Theoretic framework. In *Proceedings of COLING-92, the 14th International Conference on Computational Linguistics*, pages 1116–1120, Nantes, France, 1992.

[Brachman & Schmolze 85] R.J. Brachman and J.G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[Braun *et al* 88] G. Braun, H. Eikmeyer, T. Polzin, H. Rieser, P. Ruhrberg, and U. Schade. Situations in PROLOG. Technical Report Technical Report No. 14, DFG-Research Group 'Kohärenz', Faculty of Linguistic and Literary Studies, University of Bielefeld, 1988.

[Bresnan 82] J. Bresnan. Polyadicity. In *The mental representation of grammatical relations*, pages 149–172. MIT Press, Cambridge, Mass., 1982.

[Charniak & Wilks 76] E. Charniak and Y. Wilks. *Computational Semantics.* Fundamental studies in computer science 4. North-Holland, Amsterdam, 1976.

[Chiercha 92] G. Chiercha. Anaphora and dynamic binding. *Linguistics and Philosophy*, 15:111–183, 1992.

[Chomsky 81] N. Chomsky. *Lectures on Government and Binding.* Studies in Generative Grammar 9. Foris, Dordrecht, Holland, 1981.

[Clifford 90] J. Clifford. *Formal Semantics and Pragmatics for Natural Language Querying.* Cambridge University Press, 1990.

[Cooper & Kamp 91] R. Cooper and H. Kamp. Negation in situation semantics and Discourse Representation Theory. In *Situation Theory and its Applications, II*, CSLI Lecture Notes Number 26, pages 311–333. Chicago University Press, 1991.

[Cooper 83] R. Cooper. *Quantification and Syntactic Theory.* Studies in Linguistics and Philosophy, 21. Reidel, Dordrecht, 1983.

[Cooper 89] R. Cooper. Information and grammar. Technical Report RP No. 438, Dept of Artificial Intelligence, University of Edinburgh, 1989. Also to appear in J. Wedekind, ed. *Proceedings of the Titisee Conference on Unification and Grammar.*

[Cooper 90] Richard Cooper. *Classification-based Phrase Structure Grammar: An Extended Revised Version of HPSG.* Unpublished PhD thesis, University of Edinburgh, Edinburgh, UK, 1990.

[Crocker 91] M. Crocker. Multiple meta-interpreters in a logical model of sentence processing. In C. Brown and G. Koch, editors, *Natural Language Understanding and Logic Programming, III.* Elsevier Science Publishers (North-Holland), 1991.

[Dowty *et al* 81] D. Dowty, R. Wall, and S. Peters. *Introduction to Montague Semantics.* Studies in Linguistics and Philosophy, 11. Reidel, Dordrecht, 1981.

[Fenstad *et al* 87] J. Fenstad, P-K. Halvorsen, T. Langholm, and J. van Bentham. *Situations, Language, and Logic.* Studies in Linguistics and Philosophy, 34. Reidel, Dordrecht, 1987.

[Frank & Schütze 90] M. Frank and H. Schütze. The PROSIT language v0.3. CSLI, Stanford University, 1990.

[Franz 90] A. Franz. A parser for HPSG. Technical Report LCL-90-3, Laboratory for Computational Linguistics, Carnegie Mellon University, Pittsburgh, Penn., 1990.

[Frisch 86] A. Frisch. Parsing with restricted quantification: an initial demonstration. In *Artificial Intelligence and its Applications*, pages 5–22. J. Wiley and Sons, 1986.

[Gardent 91] C. Gardent. *VP anaphora.* Unpublished PhD thesis, University of Edinburgh, Edinburgh, UK, 1991.

[Gawron & Peters 90] M. Gawron and S. Peters. *Anaphora and Quantification in Situation Semantics.* CSLI Lecture Notes Number 19. Chicago University Press, 1990.

[Gazdar *et al* 85] G. Gazdar, E. Klein, G. Pullum, and I. Sag. *Generalized Phrase Structure Grammar.* Blackwell, Oxford, 1985.

[Geach 62] P. Geach. *Reference and Generality.* Cornell University Press, Ithaca, NY, 1962.

[Ginzburg 92] J. Ginzburg. *Questions, Queries and Facts: a semantics and pragmatics for interrogatives.* Unpublished PhD thesis, Stanford University, CA., 1992.

[Glasbey 91] S. Glasbey. Distinguishing between events and times: some evidence from the semantics of 'then'. Technical Report RP No. 566, Dept of Artificial Intelligence, University of Edinburgh, 1991. to appear in *Journal of Natural Language Semantics.*

[Groenendijk & Stokhof 91a] J. Groenendijk and M. Stokhof. Dynamic Montague Grammar. In *Quantification and Anaphora I*, DYANA Deliverable R2.2A, pages 1–37. Centre for Cognitive Science, University of Edinburgh, 1991.

[Groenendijk & Stokhof 91b] J. Groenendijk and M. Stokhof. Dynamic Predicate Logic. *Linguistics and Philosophy*, 14:39–100, 1991.

[Groenendijk & Stokhof 92] J. Groenendijk and M. Stokhof. A note on interrogatives and adverbs of quantification. Technical Report LP-92-07, Institute for Logic, Language and Computation, University of Amsterdam, 1992.

[Harel 84] D. Harel. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic II*, pages 497–604. Reidel, Dordrecht, 1984.

[Hegner 91] S. Hegner. Horn extended feature structures: fast unification with negation and limited disjunction. In *Proceedings of the 5th conference of the European Chapter of the Association for Computational Linguistics*, pages 33–38, Berlin, Germany, 1991.

[Hirst 81] G. Hirst. *Anaphora in Natural Language Understanding: a survey.* Springer-Verlag, Berlin, 1981.

[Hobbs & Shieber 87] J. Hobbs and S. Shieber. An algorithm for generating quantifier scopings. *Computational Linguistics*, 13 numbers 1-2:47–63, 1987.

[Hopcroft & Ullman 79] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley, Reading, Mass., 1979.

[Jackson *et al* 89] P. Jackson, H. Reichgelt, and F. van Harmelen. *Logic-based knowledge representation.* MIT Press, Cambridge, Mass., 1989.

[Johnson & Kay 90] M. Johnson and M. Kay. Semantic abstraction and anaphora. In *Proceedings of the 13th International Conference on Computational Linguistics, Vol. 1*, pages 17–27, Helsinki, Finland, 1990.

[Johnson & Klein 86] M. Johnson and E. Klein. Discourse, anaphora and parsing. In *Proceedings of the 11th International Conference on Computational Linguistics*, pages 669–675, Bonn, West Germany, 1986.

[Johnson 88] M. Johnson. *Attribute-value logic and the theory of grammar.* CSLI Lecture Notes Number 16. Chicago University Press, 1988.

[Kamp & Reyle 93] H. Kamp and U. Reyle. *From discourse to logic: Introduction to Model Theoretic Semantics of Natural Language, Formal logic and Discourse Representation Theory.* Studies in Linguistics and Philosophy, 42. Kluwer, Dordrecht, forthcoming 1993.

[Kamp 81] H. Kamp. A theory of truth and semantic representation. In J. Groenendijk, T. Janssen, and M. Stokhof, editors, *Formal Methods in the Study of Language.* Mathematical Center, Amsterdam, 1981.

[Kamp 91] H. Kamp. Procedural and cognitive aspects of propositional attitudes and contexts. Notes distributed for a course at the Third European Summer School in Language Logic and Information, Universität des Saarlandes, Saarbrücken, 1991.

[Kay 84] M. Kay. Functional unification grammar – a formalism for machine translation. In *Proceedings of the 10th International Conference on Computational Linguistics/22nd Annual Conference of the Association for Computational Linguistics*, pages 75–78, Stanford University, California, 1984.

[Kilbury 87] J. Kilbury. A proposal for modification of the formalism of GPSG. In *Proceedings of the 3rd conference of the European Chapter of the Association for Computational Linguistics*, pages 156–159, Copenhagen, Denmark, 1987.

[King 89] P. King. *A logical formalism for Head-Driven Phrase Structure Grammar.* Unpublished PhD thesis, University of Manchester, Manchester, UK, 1989.

[Lascarides & Asher 91] A. Lascarides and N. Asher. Discourse Relations and Commonsense Entailment. In H. Kamp, editor, *Default Logics for Linguistic Analysis.* Dyana Deliverable R2.5B, 1991.

[Lewin 90] I. Lewin. A quantifier scoping algorithm without a free variable constraint. In *Proceedings of the 13th International Conference on Computational Linguistics, Vol. 3*, pages 190–194, Helsinki, Finland, 1990.

[Lewin 92] I. Lewin. *Dynamic Quantification in Logic and Computational Semantics.* Unpublished PhD thesis, University of Edinburgh, Edinburgh, UK., 1992.

[Montague 74] R. Montague. The proper treatment of quantification in English. In Thomason R., editor, *Formal Philosophy.* Yale University Press, New York, 1974.

[Muskens 89] R. Muskens. *Meaning and Partiality.* Unpublished PhD thesis, University of Amsterdam, Amsterdam, The Netherlands, 1989.

[Nakashima *et al* 88] H. Nakashima, H. Suzuki, P-K. Halvorsen, and S. Peters. Towards a computational interpretation of situation theory. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 489–498. ICOT, 1988.

[Nakashima *et al* 91] H. Nakashima, S. Peters, and H. Schütze. Communication and inference through situations. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, volume 1, pages 75–81, 1991.

[Partee 75] B. Partee. Montague grammar and transformational grammar. *Linguistic Inquiry*, 6:203–300, 1975.

[Partee 84] B. Partee. Nominal and temporal anaphora. *Linguistics and Philosophy*, 7:243–286, 1984.

[Pereira & Warren 80] F. Pereira and D. Warren. Definite Clause Grammars for language analysis. *Artificial Intelligence*, 13:231–278, 1980.

[Pereira & Warren 83] F. Pereira and D. Warren. Parsing as deduction. In *21st Annual Conference of the Association for Computational Linguistics*, pages 137–144, MIT, Massachusetts, 1983.

[Pereira 82] F. Pereira. *Logic for Natural Language Analysis*. Unpublished PhD thesis, University of Edinburgh, Edinburgh, UK., 1982. reprinted as Technical Note 275 Artificial Intelligence Center, SRI International, Menlo Park, Ca.

[Pinkal 91] M. Pinkal. On the syntactic-semantic analysis of bound anaphora. In *Proceedings of the 5th conference of the European Chapter of the Association for Computational Linguistics*, pages 45–50, Berlin, Germany, 1991.

[Pollard & Moshier 90] C. Pollard and D. Moshier. Unifying partial descriptions of sets. In P. Hanson, editor, *Information, Language and Cognition*, volume 1 of *Vancouver Studies in Cognitive Science*. University of British Columbia Press, Vancouver, 1990.

[Pollard & Sag 87] C. Pollard and I. Sag. *Information-based Syntax and Semantics: Volume 1: Fundamentals*. CSLI Lecture Notes Number 13. Chicago University Press, 1987.

[Polzin *et al* 89] T. Polzin, H. Rieser, and U. Schade. More situations in PROLOG. Technical Report Technical Report No. 19, DFG-Research Group 'Kohärenz', Faculty of Linguistic and Literary Studies, University of Bielefeld, 1989.

[Popowich & Vogel 91] F. Popowich and C. Vogel. The HPSG-PL system. Technical Report CSS-IS TR 91-08, School of Computing Science, Simon Fraser University, 1991.

[Pulman 91] S. Pulman. Comparatives and ellipsis. In *Proceedings of the 5th conference of the European Chapter of the Association for Computational Linguistics*, pages 2–7, Berlin, Germany, 1991.

[Reeves 83] S. Reeves. An introduction to semantic tableaux. Technical report, Dept. of Computer Science, University of Essex, 1983.

[Ritchie 85] G. Ritchie. Simulating a Turing machine using functional unification grammar. In T. O'Shea, editor, *Advances in Artificial Intelligence*, pages 285–294. North Holland, 1985.

[Rooth 87] M. Rooth. Noun phrase interpretation in Montague Grammar, file change semantics and situation semantics. In P. Gardenförs, editor, *Generalised Quantifiers*, Studies in Linguistics and Philosophy, 31. Reidel, Dordrecht, 1987.

[Rounds 88] W. Rounds. Set values for unification-based grammar formalisms and logic grammars. Technical Report CSLI-88-129, CSLI, Stanford University, 1988.

[Rupp 89] C. Rupp. Situation semantics and machine translation. In *Proceedings of the Fourth Conference of the European Chapter of the Association for Computational Linguistics*, pages 308–318, Manchester, UK., 1989.

[Shieber 84] S. Shieber. The design of a computer language for linguistic information. In *Proceedings of the 10th International Conference on Computational Linguistics/22nd Annual Conference of the Association for Computational Linguistics*, pages 362–366, Stanford University, California, 1984.

[Shieber 86a] S. Shieber. *An Introduction to Unification Approaches to Grammar*. CSLI Lecture Notes Number 4. Chicago University Press, 1986.

[Shieber 86b] S Shieber. A simple reconstruction of GPSG. In *Proceedings of the 11th International Conference on Computational Linguistics*, pages 211–215, Bonn, West Germany, 1986.

[Smolka 88] G. Smolka. A feature logic with subsorts. LILOG Report 33, IWBS, IBM Deutschland, 1988. To appear in: J. Wedekind and C. Rohrer (eds.), Unification in Grammar; The MIT Press, 1991.

[Thomason 74] R. Thomason. *Formal philosophy: Selected papers by Richard Montague*. Yale University Press, New Haven, 1974.

[Uszkoreit 86] H. Uszkoreit. Categorial Unification Grammars. In *Proceedings of the 11th International Conference on Computational Linguistics*, pages 187–194, Bonn, West Germany, 1986.

[Winograd 72] T. Winograd. *Understanding Natural Language.* Academic Press, New York, 1972.

[Winograd 83] T. Winograd. *Language as a Cognitive Process. Volume I: Syntax.* Addison-Wesley, Reading, Mass., 1983.

[Woods 75] W. Woods. What's in a link: Foundations for semantic nets. In D. Bobrow and A. Collins, editors, *Representation and Understanding: Studies in Cognitive Science*, pages 35–82. Academic Press, New York, 1975.

[Zadronzy 92] W. Zadronzy. On compositional semantics. In *Proceedings of COLING-92, the 14th International Conference on Computational Linguistics*, pages 260–266, Nantes, France, 1992.

[Zeevat 89] H. Zeevat. A compositional approach to Discourse Representation Theory. *Linguistics and Philosophy*, 12:95–131, 1989.

Appendix A

Examples

A.1 Introduction

In this Appendix we will give the full ASTL specification of four different descriptions: the Rooth syntactic fragment (Section 4.3) is the syntactic fragment used as the basis for the following three semantic descriptions; Situation Theoretic Grammar (Section 4.4); Discourse Representation Theory (Chapter 5); and DPL-NL (Chapter 6) which offers a dynamic semantic treatment of the Rooth fragment. These descriptions are, unfortunately, rather difficult to read. They are included here because this is a computational thesis and the full complete descriptions (which are directly "executable") show exactly what is needed in order to compute semantic forms.

A.2 Rooth Fragment

This ASTL description is a for a simple syntactic grammar (described in Section 4.3) based on the fragment in [Rooth 87]. The grammar is large enough to deal with simple declarative sentences, including quantifiers and anaphora. It can analyse sentences like

> *Hanako sings.*
> *A man walks. He talks.*
> *Every man with a donkey likes it.*

Note this only defines the syntax, semantic forms are built on top of this structure in the later STG, DRT and DPL-NL descriptions

```
Individuals
    {}
Relations
    (
```

```
;;;  These are the relations which act like features in a
;;;  conventional attribute value system
      use_of/2 cat/2
;;; Syntactic functions which act as arguments to cat/2
      NounPhrase/1 Noun/1 Determiner/1
      Sentence/1 VerbPhrase/1 Verb/1
      PrepPhrase/1 Preposition/1
      Discourse/1
;;; Structural relation
      daughter/1 )
Parameters
    {D,S,NP,VP,PN,N,PREP,DET,V}
Variables
    {*S,*NP, *VP, *V, *N, *PP, *PREP, *DET, *D }
Situations
    ()
GoalProp
   *S : [S ! S != <<cat,S,discourse,1>>]
Grammar Rules
;;;
;;;  S -> NP VP
;;;
   [S ! S != <<cat,S,Sentence,1>>
        S != <<daughter,*NP,1>>
        S != <<daughter,*VP,1>>]
   ->
    *NP : [NP ! NP != <<cat,NP,NounPhrase,1>>],
    *VP : [VP ! VP != <<cat,VP,VerbPhrase,1>>].
;;;
;;;  VP -> V NP
;;;
    [VP ! VP != <<cat,VP,VerbPhrase,1>>
          VP != <<daughter,*V,1>>
          VP != <<daughter,*NP,1>>]
    ->
       *V  : [V ! V != <<cat,V,Verb,1>>],
       *NP : [NP ! NP != <<cat,NP,NounPhrase,1>>].
;;;
;;;  N -> N PP
;;;
    [N ! N != <<cat,N,noun,1>>
         N != <<daughter,*N,1>>
         N != <<daughter,*PP,1>>]
    ->
      *N : [N ! N != <<cat,N,noun,1>>],
      *PP : [PP ! PP != <<cat,PP,PrepPhrase,1>>].
```

```
;;;
;;;  NP -> Det N
;;;
    [NP ! NP != <<cat,NP,NounPhrase,1>>
          NP != <<daughter,*DET,1>>
          NP != <<daughter,*N,1>>]
    ->
      *DET : [DET ! DET != <<cat,DET,Determiner,1>>],
      *N :   [N ! N != <<cat,N,noun,1>>].
;;;
;;;  PP -> P NP
;;;
    [PP ! PP != <<cat,PP,PrepPhrase,1>>
          PP != <<daughter,*PREP,1>>
          PP != <<daughter,*NP,1>>]
    ->
      *PREP : [PREP ! PREP != <<cat,PREP,Preposition,1>>],
      *NP : [NP ! NP != <<cat,NP,NounPhrase,1>>].
;;;
;;;  D -> S
;;;
    [D ! D != <<cat,D,Discourse,1>>
         D != <<daughter,*S,1>>]
    ->
      *S : [S ! S != <<cat,S,Sentence,1>>].
;;;
;;;  D -> D S
;;;
    [D ! D != <<cat,D,Discourse,1>>
         D != <<daughter,*D,1>>
         D != <<daughter,*S,1>>]
    ->
      *D : [D ! D != <<cat,D,Discourse,1>>],
      *S : [S ! S != <<cat,S,Sentence,1>>].
```

A basic set of lexical entries to allow some simple classic sentences.

```
Lexical Entries

;;;
;;;  Nouns
;;;
Hanako -
   [PN ! PN != <<cat,PN,NounPhrase,1>>
         PN != <<use_of,PN,"Hanako",1>>]
Taro -
```

```
   [PN ! PN != <<cat,PN,NounPhrase,1>>
         PN != <<use_of,PN,"Taro",1>>]
Anna -
   [PN ! PN != <<cat,PN,NounPhrase,1>>
         PN != <<use_of,PN,"Anna",1>>]
man -
   [N ! N != <<cat,N,Noun,1>>
        N != <<use_of,N,"man",1>>]
donkey -
   [N ! N != <<cat,N,Noun,1>>
        N != <<use_of,N,"donkey",1>>]
;;;
;;;  Pronouns
;;;
he -
   [PN ! PN != <<cat,PN,NounPhrase,1>>
         PN != <<use_of,PN,"he",1>>]
she -
   [PN ! PN != <<cat,PN,NounPhrase,1>>
         PN != <<use_of,PN,"she",1>>]
it -
   [PN ! PN != <<cat,PN,NounPhrase,1>>
         PN != <<use_of,PN,"it",1>>]
;;;
;;;  Determiners
;;;
a -
   [DET ! DET != <<cat,DET,Determiner,1>>
          DET != <<use_of,DET,"a",1>>]
the -
   [DET ! DET != <<cat,DET,Determiner,1>>
          DET != <<use_of,DET,"the",1>>]
every -
   [DET ! DET != <<cat,DET,Determiner,1>>
          DET != <<use_of,DET,"every",1>>]
;;;
;;;  Verbs
;;;
smiles -
   [VP ! VP != <<cat,VP,VerbPhrase,1>>
         VP != <<use_of,VP,"smiles",1>>]

sings -
   [VP ! VP != <<cat,VP,VerbPhrase,1>>
         VP != <<use_of,VP,"sings",1>>]
walks -
```

```
   [VP ! VP != <<cat,VP,VerbPhrase,1>>
         VP != <<use_of,VP,"walks",1>>]
talks -
   [VP ! VP != <<cat,VP,VerbPhrase,1>>
         VP != <<use_of,VP,"talks",1>>]
runs -
   [VP ! VP != <<cat,VP,VerbPhrase,1>>
         VP != <<use_of,VP,"runs",1>>]
likes -
   [V ! V != <<cat,V,Verb,1>>
        V != <<use_of,V,"likes",1>>]
beats -
   [V ! V != <<cat,V,Verb,1>>
        V != <<use_of,V,"beats",1>>]
;;;
;;;  Prepositions
;;;
to -
     [PREP ! PREP != <<cat,PREP,Preposition,1>>
             PREP != <<use_of,PREP,"to",1>>]
with -
     [PREP ! PREP != <<cat,PREP,Preposition,1>>
             PREP != <<use_of,PREP,"with",1>>]
on -
     [PREP ! PREP != <<cat,PREP,Preposition,1>>
             PREP != <<use_of,PREP,"on",1>>]
```

A.3 STG description

This adds a Situation Theoretic Grammar semantic treatment for the Rooth fragment [Rooth 87]. This deals with simple declarative sentences, but not anaphora. See Section 4.4 for a full discussion. Note unlike the Rooth ASTL description above here we distinguish between proper nouns, pronouns and noun phrases.

The semantics of an utterance is represented by a parametric fact and anchoring environment. The anchoring environment is extended with anchor relations as more information is available about the sentence. The described situation is represented by the reduction of the anchoring environment and parametric fact.

```
Individuals
    {a,h,t}
Relations
    ( use_of/2 cat/2
      env/2 sem/2 described/2
      NounPhrase/1 Noun/1 Determiner/1
```

```
      Sentence/1 VerbPhrase/1 Verb/1
      PrepPhrase/1 Preposition/1
      Discourse/1 pform/2
      subj/1 obj/1 comp/1 pred/1
      var/1 range/1 body/1 quantifier/1
      arg/1 arg1/1 arg2/1 prep/1
      label/2 anchor/2
      beat/2 like/2 walk/1 talk/1 smile/1 sing/1 run/1
      man/1 donkey/1
      named/2 )
Parameters
    {R1,Q1,P1,A1,A2,A3,D,S,NP,VP,PN,N,PREP,DET,V,ENV,DS
     SMA1, WA1, SA1, RA1, R1, TA1, LA1, LA2, BA1, BA2, MA1,
     DA1 }
Variables
    {*X, *S, *Y, *Z, *Fact, *Qexpr, *use,
     *DS, *Env,
     *SEnv, *VPEnv, *VEnv, *NPEnv, *NEnv, *DetEnv, *PEnv,
     *PPEnv, *EnvType,
     *Body, *Range, *Type, *Z
     *R1, *A1, *A2, *A3, *VR1, *VA1, *VA2, *VA3,
     *SDS, *DStype, *DS, *DS2, *DS3
     *pred, *Var, *Pvar, *Basis, *pobj, *obj, *prep,
     *PA1, *PA2, *PR1 }
Situations
    (SmileEnv :: [Env ! Env != <<label,R1,pred,1>>
                        Env != <<anchor,R1,smile,1>>
                        Env != <<label,SMA1,subj,1>>]
     SingEnv :: [Env ! Env != <<label,R1,pred,1>>
                       Env != <<anchor,R1,sing,1>>
                       Env != <<label,SA1,subj,1>>]
     WalkEnv :: [Env ! Env != <<label,R1,pred,1>>
                       Env != <<anchor,R1,walk,1>>
                       Env != <<label,WA1,subj,1>>]
     RunEnv :: [Env ! Env != <<label,R1,pred,1>>
                      Env != <<anchor,R1,run,1>>
                      Env != <<label,RA1,subj,1>>]
     TalkEnv :: [Env ! Env != <<label,R1,pred,1>>
                       Env != <<anchor,R1,talk,1>>
                       Env != <<label,TA1,subj,1>>]
     LikeEnv :: [Env ! Env != <<label,R1,pred,1>>
                       Env != <<anchor,R1,like,1>>
                       Env != <<label,LA1,subj,1>>
                       Env != <<label,LA2,obj,1>>]
     BeatEnv :: [Env ! Env != <<label,R1,pred,1>>
                       Env != <<anchor,R1,beat,1>>
```

```
                     Env != <<label,BA1,subj,1>>
                     Env != <<label,BA2,obj,1>>]
     ManEnv :: [Env ! Env != <<label,R1,pred,1>>
                      Env != <<anchor,R1,man,1>>
                      Env != <<label,MA1,arg1,1>>]
     DonkeyEnv :: [Env ! Env != <<label,R1,pred,1>>
                         Env != <<anchor,R1,donkey,1>>
                         Env != <<label,DA1,arg1,1>>]
     AEnv :: [Env ! Env != <<label,Q1,quantifier,1>>
                    Env != <<anchor,Q1,some,1>>
                    Env != <<label,A1,var,1>>
                    Env != <<label,A2,range,1>>
                    Env != <<label,A3,body,1>>]
     EveryEnv :: [Env ! Env != <<label,Q1,quantifier,1>>
                        Env != <<anchor,Q1,every,1>>
                        Env != <<label,A1,var,1>>
                        Env != <<label,A2,range,1>>
                        Env != <<label,A3,body,1>>]
     WithEnv :: [Env ! Env != <<label,P1,prep,1>>
                       Env != <<anchor,P1,with,1>>
                       Env != <<label,A1,arg1,1>>
                       Env != <<label,A2,arg2,1>>]
    )
GoalProp
   *S : [S ! S != <<cat,S,discourse,1>>]
```

A rather exhaustive set of constraints is needed to specify the relationship between the parametric fact and anchoring environment to the described situation. These constraints are really just cases of the same notional constraint. They try to capture the notion of reduction as discussed in Section 7.4.1. Even with this large number of similar constraints the full notion of reduction is actually not captured.

```
Constraints

*S : [S ! S != <<described,S,*DS ::
                  [DS ! DS != <<*VR1,*VA1,*VA2,*VA3,1>>],1>>]
<=
   *S : [S ! S != <<cat,S,sentence,1>>
             S != <<sem,S,<<*R1,*A1,*A2,*A3,1>>,1>>
             S != <<env,S,*SEnv ::
                         [Env ! Env != <<anchor,*R1,*VR1,1>>
                                Env != <<anchor,*A1,*VA1,1>>
                                Env != <<anchor,*A2,*VA2,1>>
                                Env != <<anchor,*A3,*VA3,1>>],1>>].

*S : [S ! S != <<described,S,*DS ::
```

```
                  [DS ! DS != <<*VR1,*VA1,*VA2,*VA3,1>>],1>>]
<=
   *S : [S ! S != <<cat,S,VerbPhrase,1>>
             S != <<env,S,*SEnv ::
                          [Env ! Env != <<anchor,*R1,*VR1,1>>
                                 Env != <<anchor,*A1,*VA1,1>>
                                 Env != <<anchor,*A2,*VA2,1>>
                                 Env != <<anchor,*A3,*VA3,1>>],1>>
             S != <<sem,S,<<*R1,*A1,*A2,*A3,1>>,1>>].

*S : [S ! S != <<described,S,*DS ::
                 [DS ! DS != <<*VR1,*VA1,*VA2,1>>],1>>]
<=
   *S : [S ! S != <<cat,S,sentence,1>>
             S != <<sem,S,<<*R1,*A1,*A2,1>>,1>>
             S != <<env,S,*SEnv ::
                          [Env ! Env != <<anchor,*R1,*VR1,1>>
                                 Env != <<anchor,*A1,*VA1,1>>
                                 Env != <<anchor,*A2,*VA2,1>>],1>>].

*S : [S ! S != <<described,S,
                 *DS :: [DS ! DS != <<*VR1,*A1,*VA2,1>>],1>>]
<=
   *S : [S ! S != <<cat,S,VerbPhrase,1>>
             S != <<env,S,*VPEnv ::
                          [Env ! Env != <<anchor,*R1,*VR1,1>>
                                 Env != <<anchor,*A2,*VA2,1>>],1>>
             S != <<sem,S,<<*R1,*A1,*A2,1>>,1>>].

*S : [S ! S != <<described,S,*DS :: [DS ! DS != <<*VR1,*VA1,1>>],1>>]
<=
   *S : [S ! S != <<cat,S,sentence,1>>
             S != <<sem,S,<<*R1,*A1,1>>,1>>
             S != <<env,S,*SEnv ::
                          [Env ! Env != <<anchor,*R1,*VR1,1>>
                                 Env != <<anchor,*A1,*VA1,1>>],1>>].

*S : [S ! S != <<described,S,*DS :: [DS ! DS != <<*VR1,*A1,1>>],1>>]
<=
   *S : [S ! S != <<cat,S,VerbPhrase,1>>
             S != <<sem,S,<<*R1,*A1,1>>,1>>
             S != <<env,S,*VPEnv ::
                          [Env ! Env != <<anchor,*R1,*VR1,1>>],1>>].

*S : [S ! S != <<described,S,*DS :: *DStype,1>>]
<=
```

```
   *S : [S ! S != <<cat,S,noun,1>>
             S != <<use_of,S,*X,1>>].

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
                       Grammar Rules
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  S -> NP VP
;;;  As proper nouns are not treated like quantifiers (a la
;;;  Montague) two rules are necessary, one to deal with a
;;;  proper noun subject and a second to deal with a quantified
;;;  NP.
;;;
   [S ! S != <<cat,S,Sentence,1>>
        S != <<sem,S,*Fact,1>>
        S != <<env,S,*SEnv :: *EnvType &
               [Env ! Env != <<anchor,*X,*Y,1>>],1>>]
   ->
    [NP ! NP != <<cat,NP,NounPhrase,1>>
          NP != <<use_of,NP,*use,1>>  ;; lexical NP
          NP != <<sem,NP,*Y,1>>],
    [VP ! VP != <<cat,VP,VerbPhrase,1>>
          VP != <<sem,VP,*Fact,1>>
          VP != <<env,VP,*VPEnv :: *EnvType &
                          [Env ! Env != <<label,*X,subj,1>>],
                         1>>].

   [S ! S != <<cat,S,Sentence,1>>
        S != <<sem,S,*Qexpr,1>>
        S != <<env,S,*SEnv :: *EnvType &
                         [Env ! Env != <<anchor,*Y,*Var,1>>
                                Env != <<anchor,*Body,*DSType,1>>],
                   1>>]
   ->
    [NP ! NP != <<cat,NP,NounPhrase,1>>
          NP != <<sem,NP,*Qexpr,1>>
          NP != <<env,NP, *Env :: *EnvType &
                             [Env ! Env != <<label,*Body,body,1>>
                                    Env != <<label,*X,var,1>>
                                    Env != <<anchor,*X,*Var,1>>],1>>],
    [VP ! VP != <<cat,VP,VerbPhrase,1>>
          VP != <<described,VP,*DS :: *DSType,1>>
          VP != <<env,VP,*VPEnv :: [Env ! Env != <<label,*Y,subj,1>>],
                         1>>
```

```
        VP != <<sem,VP,*Fact,1>>].
;;;
;;;  VP -> V NP
;;;  Again two rules, one for a proper noun object and the
;;;  second for quantified NPs.
;;;
    [VP ! VP != <<cat,VP,VerbPhrase,1>>
          VP != <<env,VP,*VPEnv :: *EnvType &
                        [Env ! Env != <<anchor,*Body,
                                        <<*pred,*Y,*Var,1>>,1>>
                               Env != <<label,*Y,subj,1>>],
                    1>>
          VP != <<sem,VP,*Qexpr,1>>]
    ->
       [V ! V != <<cat,V,Verb,1>>
            V != <<env,V,*Env ::
                        [Env ! Env != <<label,*Y,subj,1>>
                               Env != <<label,*R1,pred,1>>
                               Env != <<anchor,*R1,*pred,1>>],
                       1>>
            V != <<sem,V,*Fact,1>>],
       [NP ! NP != <<cat,NP,NounPhrase,1>>
             NP != <<env,NP,*Env :: *Envtype &
                           [Env ! Env != <<label,*Range,range,1>>
                                  Env != <<label,*Body,body,1>>
                                  Env != <<label,*Pvar,var,1>>
                                  Env != <<anchor,*Pvar,*Var,1>>],
                       1>>
             NP != <<sem,NP,*Qexpr,1>>].

    [VP ! VP != <<cat,VP,VerbPhrase,1>>
          VP != <<env,VP,*VPEnv :: *EnvType &
                         [Env ! Env != <<anchor,*X,*Y,1>>],1>>
          VP != <<sem,VP,*Fact,1>>]
    ->
       [V ! V != <<cat,V,Verb,1>>
            V != <<env,V,*Env :: *Envtype &
                                [Env ! Env != <<label,*X,obj,1>>],
                       1>>
            V != <<sem,V,*Fact,1>>],
       [NP ! NP != <<cat,NP,NounPhrase,1>>
             NP != <<use_of,NP,*use,1>>  ;; lexical NP
             NP != <<sem,NP,*Y,1>>].
;;;
;;;  NP -> Det N
;;;
```

```
    [NP ! NP != <<cat,NP,NounPhrase,1>>
          NP != <<sem,NP,*QEXPR,1>>
          NP != <<env,NP,*NPEnv ::
                  *EnvType &
                  [Env ! Env != <<anchor,*X,*A1,1>>
                         Env != <<anchor,*Range,
                                   *DStype &
                                   [DS ! DS != <<*pred,*A1,1>>],1>>],
                  1>>]
    ->
      [DET ! DET != <<cat,DET,Determiner,1>>
             DET != <<sem,DET,*QEXPR,1>>
             DET != <<env,DET,*DetEnv ::
                       *EnvType &
                       [Env ! Env != <<label,*Range,range,1>>
                              Env != <<label,*X,var,1>>],1>>],
      [N ! N != <<cat,N,noun,1>>
           N != <<env,N,*Env ::
                  [Env ! Env != <<label,*A1,arg1,1>>
                         Env != <<anchor,*R1,*pred,1>>],1>>
           N != <<described,N,*DS :: *DStype,1>>
           N != <<sem,N,<<*R1,*A1,1>>,1>>].
;;;
;;;  N -> N PP
;;;  Reduction is done on the fly here
;;;
    [N ! N != <<cat,N,noun,1>>
         N != <<sem,N,*Y,1>>
         N != <<described,N,*DS2 :: *DStype &
                          [DS ! DS != <<*prep,*A1,*Z,1>>],1>>
         N != <<env,N,*NEnv :: *Envtype, 1>>]
    ->
      [N ! N != <<cat,N,noun,1>>
           N != <<env,N,*Env :: *Envtype &
                 [Env ! Env != <<label,*A1,arg1,1>>],1>>
           N != <<sem,N,*Y,1>>],
      [PP ! PP != <<cat,PP,PrepPhrase,1>>
            PP != <<described,PP,*DS :: *DStype,1>>
            PP != <<env,PP,*PPEnv ::
                    [Env ! Env != <<label,*PA1,arg1,1>>
                           Env != <<label,*PA2,arg2,1>>
                           Env != <<anchor,*PA2,*Z,1>>
                           Env != <<label,*PR1,prep,1>>
                           Env != <<anchor,*PR1,*prep,1>>],1>>].
;;;
;;;  PP -> P NP
```

```
;;;  Two are required -- the first for proper nouns, second for
;;;  quantified NPs
;;;
    [PP ! PP != <<cat,PP,PrepPhrase,1>>
          PP != <<described,PP,*DS2 :: *DStype,1>>
          PP != <<env,PP,*PPEnv :: *Envtype &
                  [Env ! Env != <<anchor,*PA2,*Y,1>>],1>>]
    ->
      [PREP ! PREP != <<cat,PREP,Preposition,1>>
              PREP != <<env,PREP,*Penv :: *Envtype &
                       [Env ! Env != <<label,*PA2,arg2,1>>],1>>],
      [NP ! NP != <<cat,NP,NounPhrase,1>>
            NP != <<use_of,NP,*use,1>>  ;; lexical NP
            NP != <<sem,NP,*Y,1>>].
;;;
;;;  D -> S
;;;
    [D ! D != <<cat,D,Discourse,1>>
         D != <<env,S,*Env,1>>
         D != <<described,D,*DS,1>>]
    ->
      [S ! S != <<cat,S,Sentence,1>>
           S != <<env,S,*Env,1>>
           S != <<described,S,*DS,1>>].
;;;
;;;  D -> D S
;;;  This is again a little hacky.  The sentence described is always
;;;  one fact so we can add *it* (we know it's not a them) to the
;;;  discourse described -- but we need a rule for each semantic arity
;;;
    [D ! D != <<cat,D,Discourse,1>>
         D != <<described,D,*DS2 :: *DStype &
                     [DS ! DS != <<*R1,*A1,1>>],1>>]
    ->
      [D ! D != <<cat,D,Discourse,1>>
           D != <<described,D,*DS :: *DStype,1>>],
      [S ! S != <<cat,S,Sentence,1>>
           S != <<described,S,*DS3 ::
                     [DS ! DS != <<*R1,*A1,1>>],1>>].
```

A basic set of lexical entries to allow some the simple classic sentences. Note the constraints above are sometimes used to expand entries (i.e like Lexical Redundancy Rules).

```
Lexical Entries
;;;
```

```
;;;  Nouns
;;;
Hanako -
   [PN ! PN != <<cat,PN,NounPhrase,1>>
         PN != <<use_of,PN,"Hanako",1>>
         PN != <<sem,PN,h,1>>]
Taro -
   [PN ! PN != <<cat,PN,NounPhrase,1>>
         PN != <<use_of,PN,"Taro",1>>
         PN != <<sem,PN,t,1>>]
Anna -
   [PN ! PN != <<cat,PN,NounPhrase,1>>
         PN != <<use_of,PN,"Anna",1>>
         PN != <<sem,PN,a,1>>]
man -
   [N ! N != <<cat,N,Noun,1>>
        N != <<use_of,N,"man",1>>
        N != <<sem,N,<<R1,MA1,1>>,1>>
        N != <<env,N,ManEnv,1>>]
donkey -
   [N ! N != <<cat,N,Noun,1>>
        N != <<use_of,N,"donkey",1>>
        N != <<sem,N,<<R1,DA1,1>>,1>>
        N != <<env,N,DonkeyEnv,1>>]
;;;
;;;  Pronouns
;;;
;;;  Not used in this actual description
;;;
he -
   [PN ! PN != <<cat,PN,NounPhrase,1>>
         PN != <<use_of,PN,"he",1>>
         PN != <<sem,PN,*X,1>>]
she -
   [PN ! PN != <<cat,PN,NounPhrase,1>>
         PN != <<use_of,PN,"she",1>>
         PN != <<sem,PN,*X,1>>]
it -
   [PN ! PN != <<cat,PN,NounPhrase,1>>
         PN != <<use_of,PN,"it",1>>
         PN != <<sem,PN,*X,1>>]
;;;
;;;  Determiners
;;;
;;;  Their semantics is a relation between a variable (a parameter)
;;;  and a range (a type) and a body (a type too).
```

```
;;;
a -
   [DET ! DET != <<cat,DET,Determiner,1>>
          DET != <<use_of,DET,"a",1>>
          DET != <<sem,DET,<<Q1,A1,A2,A3,1>>,1>>
          DET != <<env,DET,AEnv,1>>]
every -
   [DET ! DET != <<cat,DET,Determiner,1>>
          DET != <<use_of,DET,"every",1>>
          DET != <<sem,DET,<<Q1,A1,A2,A3,1>>,1>>
          DET != <<env,DET,EveryEnv,1>>]
;;;
;;;  Verbs
;;;
smiles -
   [VP ! VP != <<cat,VP,VerbPhrase,1>>
         VP != <<use_of,VP,"smiles",1>>
         VP != <<sem,VP, <<R1,SMA1,1>>,1>>
         VP != <<env,VP,SmileEnv,1>>]
sings -
   [VP ! VP != <<cat,VP,VerbPhrase,1>>
         VP != <<use_of,VP,"sings",1>>
         VP != <<sem,VP, <<R1,SA1,1>>,1>>
         VP != <<env,VP,SingEnv,1>>]
walks -
   [VP ! VP != <<cat,VP,VerbPhrase,1>>
         VP != <<use_of,VP,"walks",1>>
         VP != <<sem,VP, <<R1,WA1,1>>,1>>
         VP != <<env,VP,WalkEnv,1>>]
talks -
   [VP ! VP != <<cat,VP,VerbPhrase,1>>
         VP != <<use_of,VP,"talks",1>>
         VP != <<sem,VP,<<R1,TA1,1>>,1>>
         VP != <<env,VP,TalkEnv,1>>]
runs -
   [VP ! VP != <<cat,VP,VerbPhrase,1>>
         VP != <<use_of,VP,"runs",1>>
         VP != <<sem,VP,<<R1,RA1,1>>,1>>
         VP != <<env,VP,RunEnv,1>>]
likes -
   [V ! V != <<cat,V,Verb,1>>
        V != <<use_of,V,"likes",1>>
        V != <<env,V,LikeEnv,1>>
        V != <<sem,V,<<R1,LA1,LA2,1>>,1>>]
beats -
   [V ! V != <<cat,V,Verb,1>>
```

```
        V != <<use_of,V,"beats",1>>
        V != <<sem,V,<<R1,BA1,BA2,1>>,1>>
        V != <<env,V,BeatEnv,1>>]
;;;
;;;  Prepositions
;;;
to -
     [PREP ! PREP != <<cat,PREP,Preposition,1>>
             PREP != <<use_of,PREP,"to",1>>]
with -
     [PREP ! PREP != <<cat,PREP,Preposition,1>>
             PREP != <<use_of,PREP,"with",1>>
             PREP != <<sem,PREP,<<P1,A1,A2,1>>,1>>
             PREP != <<env,PREP,WithEnv,1>>]
on -
     [PREP ! PREP != <<cat,PREP,Preposition,1>>
             PREP != <<pform,PREP,on,1>>
             PREP != <<use_of,PREP,"on",1>>]
```

A.4 DRT description

This is a DRT description in ASTL as described in Chapter 5. Again it is based on the Rooth fragment. Unlike the STG description this deals with pronouns, (including donkey anaphora).

Unlike the STG description the semantics (a DRS) is built up using a threading technique rather than what is effectively lambda application and reduction. Threading relations are set up between utterance situations specifing an ordering. The DRSs are specified as monotonically increasing over threads.

```
Individuals
    {}
Relations
    ( use_of/2 cat/2
      sem/2 env/2 type/2 threads/2
      NounPhrase/1 Sentence/1 VerbPhrase/1 Discourse/1
      ProperNoun/1 ProNoun/1 PrepPhrase/1 Preposition/1
      FullDiscourse/1 DisStart/1 DisEnd/1
      subj/1 pred/1
      label/2 anchor/2
      sing/1 like/2 smile/1
      donkey/1 man/1 hat/1 with/2
      named/2 male/1 female/1 neuter/1
      DRSIn/2 DRSOut/2
      t-in/2 t-out/2 t-feed/2 t-need/2
```

```
      accessible/2 )
Hush  ;; relations not to be displayed on output (by default)
    (daughter threads)
Parameters
    {R1,A1,A2, A3, S,S1,S2,TS,NP,VP,PN,V,Env,DS,Res,
     PN1, PN2, PN3, A, PREP, PP,
     DA1, MA1, HA1, T, H, P  }
Variables
    {*X, *Y, *Z, *S, *U, *Fact, *VPEnv, *SEnv, *VPEnvType, *VEnv,
     *VEnvType, *Env, *PPEnv, *PEnv, *EnvType, *Nenv,
     *R1, *A1, *A2, *VR1, *VA1, *VA2, *DS, *DS1, *PN, *R
     *pred,
     *DRSIn, *DRSout, *AOut, *AIn, *AType, *Access,
     *ThreadS, *ThreadNP, *ThreadVP, *ThreadV, *Thread, *ThreadDS,
     *ThreadDS1, *TYPE, *OUT,
     *A, *OUT1, *M1, *M2, *TD,
     *QEXPR, *Range, *Body, *BodyDRS, *RangeDRS,
     *QUANT, *PQUANT, *PVAR, *PRANGE, *PBODY, *Name
     *T1, *T2, *T3, *S1, *S2, *P1, *P2, *NPSem,
     *NP, *VP, *V, *N, *DET, *PP, *PREP, *N1, *D}
Situations
    (SingEnv :: [Env ! Env != <<label,R1,pred,1>>
                       Env != <<anchor,R1,sing,1>>
                       Env != <<label,A1,subj,1>>]
     SmileEnv :: [Env ! Env != <<label,R1,pred,1>>
                        Env != <<anchor,R1,smile,1>>
                        Env != <<label,A1,subj,1>>]
     WalkEnv :: [Env ! Env != <<label,R1,pred,1>>
                       Env != <<anchor,R1,walk,1>>
                       Env != <<label,WA1,subj,1>>]
     TalkEnv :: [Env ! Env != <<label,R1,pred,1>>
                       Env != <<anchor,R1,talk,1>>
                       Env != <<label,TA1,subj,1>>]
     LikeEnv :: [Env ! Env != <<label,R1,pred,1>>
                       Env != <<anchor,R1,like,1>>
                       Env != <<label,A1,subj,1>>
                       Env != <<label,A2,obj,1>>]
     ManEnv :: [Env ! Env != <<label,R1,pred,1>>
                      Env != <<anchor,R1,man,1>>
                      Env != <<label,MA1,arg1,1>>]
     DonkeyEnv :: [Env ! Env != <<label,R1,pred,1>>
                         Env != <<anchor,R1,donkey,1>>
                         Env != <<label,DA1,arg1,1>>]
     HatEnv :: [Env ! Env != <<label,R1,pred,1>>
                      Env != <<anchor,R1,hat,1>>
                      Env != <<label,HA1,arg1,1>>]
```

```
    AEnv :: [Env ! Env != <<label,Q1,quantifier,1>>
                   Env != <<anchor,Q1,some,1>>
                   Env != <<label,A1,var,1>>
                   Env != <<label,A2,range,1>>
                   Env != <<label,A3,body,1>>]
    EveryEnv :: [Env ! Env != <<label,Q1,quantifier,1>>
                       Env != <<anchor,Q1,every,1>>
                       Env != <<label,A1,var,1>>
                       Env != <<label,A2,range,1>>
                       Env != <<label,A3,body,1>>]
    WithEnv :: [Env ! Env != <<label,P1,prep,1>>
                      Env != <<anchor,P1,with,1>>
                      Env != <<label,A1,arg1,1>>
                      Env != <<label,A2,arg2,1>>]
    AccessStart )
GoalProp
  *S : [S ! S != <<cat,S,fulldiscourse,1>>
            S != <<DRSOut,S,*AOut,*DRSout,1>>]
Constraints
```

These first set of constraints define the relationship between the incoming DRS and the outgoing DRS in the various types of node. The only interesting ones are sentences where a new condition is added, nouns where type information male/female is added and pronouns where the accessible markers are checked a object of the right type that has already been mentioned. The other utterance types simply "copy" the `DRSIn` to `DRSOut`.

```
  *S : [S ! S != <<DRSOut,S,
                   *Access,
                   *DRSIn &
                    [DS ! DS != <<*VR1,*VA1,1>>],1>>]
  <=
     *S : [S ! S != <<cat,S,sentence,1>>
               S != <<DRSIn,S,*Access,*DRSIn,1>>
               S != <<sem,S,<<*R1,*A1,1>>,1>>
               S != <<env,S,*SEnv ::
                     [Env ! Env != <<anchor,*R1,*VR1,1>>
                            Env != <<anchor,*A1,*VA1,1>>],1>>].

  *S : [S ! S != <<DRSOut,S,
                   *Access,
                   *DRSIn &
                    [DS ! DS != <<*VR1,*VA1,*VA2,1>>],1>>]
  <=
     *S : [S ! S != <<cat,S,sentence,1>>
               S != <<DRSIn,S,*Access,*DRSIn,1>>
```

```
                S != <<sem,S,<<*R1,*A1,*A2,1>>,1>>
                S != <<env,S,*SEnv ::
                         [Env ! Env != <<anchor,*R1,*VR1,1>>
                                Env != <<anchor,*A1,*VA1,1>>
                                Env != <<anchor,*A2,*VA2,1>>],1>>].

*S : [S ! S != <<DRSout,S,
                 *Access :: *AType &
                           [AS ! AS != <<accessible,*X,*TYPE,1>>],
                 *DRSIn &
                   [DS ! DS != <<named,*X,*Name,1>>],1>>]
  <=
      *S : [S ! S != <<cat,S,ProperNoun,1>>
                S != <<use_of,S,*Name,1>>
                S != <<sem,S,*X,1>>
                S != <<type,S,*TYPE,1>>
                S != <<DRSIn,S,*A :: *AType, *DRSIn,1>>].

*S : [S ! S != <<DRSout,S,
                 *A1 :: *AType,
                 *DRSIn &
                   [DS ! DS != <<is,*X,*Y,1>>],1>>]
  <=
      *S : [S ! S != <<cat,S,Pronoun,1>>
                S != <<type,S,*TYPE,1>>
                S != <<sem,S,*X,1>>
                S != <<DRSIn,S,
                      *A :: *AType &
                            [A ! A != <<accessible,*Y,*TYPE,1>>],
                      *DRSIn,1>>].

*S : [S ! S != <<DRSOut,S,
                  *Access,
                  *DRSIn &
                  [DS ! DS != <<every,*RangeDRS,*BodyDRS,1>>],1>>]
  <=
      *S : [S ! S != <<cat,S,Determiner,1>>
                S != <<part-of-discourse,S,*TD,1>>
                S != <<DRSIn,S,*Access,*DRSIn,1>>
                S != <<sem,S,<<*PQUANT,*X,*Y,*Z,1>>,1>>
                S != <<env,S,*SEnv ::
                      [Env ! Env != <<anchor,*PQUANT,every,1>>],1>>],
      *T1 : [TS ! TS != <<part-of-discourse,TS,*TD,1>>
                  TS != <<t-body,*S,*Body ::
                          [S ! S != <<part-of-discourse,S,*TD,1>>
                               S != <<DRSOut,S,*A1,*BodyDRS,1>>],1>>],
```

```
      *T2 : [TS ! TS != <<part-of-discourse,TS,*TD,1>>
                  TS != <<t-range,*S,*Range ::
                          [S ! S != <<part-of-discourse,S,*TD,1>>
                               S != <<DRSOut,S,*A2,*RangeDRS,1>>],1>>].

*S : [S ! S != <<DRSOut,S,*Access,*DRSIn,1>>]
  <=
      *S : [S ! S != <<cat,S,Determiner,1>>
                S != <<part-of-discourse,S,*TD,1>>
                S != <<sem,S,<<*PQUANT,*X,*Y,*Z,1>>,1>>
                S != <<env,S,*SEnv ::
                       [Env ! Env != <<anchor,*PQUANT,some,1>>],1>>],
      *T2 : [TS ! TS != <<part-of-discourse,TS,*TD,1>>
                  TS != <<t-body,*S,*Body ::
                         [S ! S != <<part-of-discourse,S,*TD,1>>
                              S != <<DRSOut,S,
                                     *Access,
                                     *DRSIn ,1>>],1>>].

*S : [S ! S != <<DRSOut,S,*Access,*DRSOut,1>>]
  <=
      *S : [S ! S != <<cat,S,NounPhrase,1>>
                S != <<daughter,S,
                       *DS :: [DS ! DS != <<cat,DS,
                                                 determiner,1>>],1>>
                S != <<DRSIn,S,*Access,*DRSOut,1>>].

*S : [S ! S != <<DRSOut,S,*Access,*BodyDRS,1>>]
  <=
      *S : [S ! S != <<cat,S,NounPhrase,1>>
                S != <<part-of-discourse,S,*TD,1>>
                S != <<daughter,S,
                       *DS :: [DS ! DS != <<cat,DS,
                                                 ProperNoun,1>>],1>>],
      *T2 : [TS ! TS != <<part-of-discourse,TS,*TD,1>>
                  TS != <<t-body,*S,*Body ::
                       [S ! S != <<part-of-discourse,S,*TD,1>>
                            S != <<DRSOut,S,*Access,*BodyDRS,1>>],1>>].

*S : [S ! S != <<DRSOut,S,*Access,*BodyDRS,1>>]
  <=
      *S : [S ! S != <<cat,S,NounPhrase,1>>
                S != <<part-of-discourse,S,*TD,1>>
                S != <<daughter,S,
                       *DS :: [DS ! DS != <<cat,DS,
                                                 ProNoun,1>>],1>>],
```

```
      *T2 : [TS ! TS != <<part-of-discourse,TS,*TD,1>>
                  TS != <<t-body,*S,*Body ::
                      [S ! S != <<part-of-discourse,S,*TD,1>>
                           S != <<DRSOut,S,
                                  *Access,
                                  *BodyDRS,1>>],1>>].

*S : [S ! S != <<DRSOut,S,
                 *A :: *AType &
                       [AS ! AS != <<accessible,*A1,*TYPE,1>>],
                 *DRSIn &
                  [DS ! DS != <<*VR1,*A1,1>>],1>>]
  <=
    *S : [S ! S != <<cat,S,Noun,1>>
              S != <<use_of,S,*X,1>>
              S != <<sem,S,<<*R1,*A1,1>>,1>>
              S != <<type,S,*TYPE,1>>
              S != <<env,S,*SEnv ::
                     [Env ! Env != <<anchor,*R1,*VR1,1>>],1>>
              S != <<DRSIn,S,*AIn :: *AType,
                            *DRSIn,1>>].

*S : [S ! S != <<DRSOut,S,
                 *Access,
                 *DRSIn &
                  [DS ! DS != <<*VR1,*VA1,*VA2,1>>],1>>]
  <=
    *S : [S ! S != <<cat,S,Noun,1>>
              S != <<env,S,*SEnv ::
                     [Env ! Env != <<label,*VA1,arg1,1>>],1>>
              S != <<daughter,S,*PP ::
                     [PP ! PP != <<cat,PP,PrepPhrase,1>>
                           PP != <<sem,PP,<<*R1,*A1,*A2,1>>,1>>
                           PP != <<env,PP,*Env ::
                                   [Env ! Env != <<anchor,*R1,*VR1,1>>
                                          Env != <<anchor,*A2,*VA2,1>>
                                   ],1>>],1>>
              S != <<DRSIn,S,*Access,*DRSIn,1>>].

*S : [S ! S != <<DRSOut,S,*Access,*DRSOut,1>>]
  <=
    *S : [S ! S != <<cat,S,VerbPhrase,1>>
              S != <<DRSIn,S,*Access,*DRSOut,1>>].

*S : [S ! S != <<DRSOut,S,*Access,*BodyDRS,1>>]
  <=
```

```
    *S : [S ! S != <<cat,S,Verb,1>>
              S != <<part-of-discourse,S,*TD,1>>
              S != <<sem,S,<<*R1,*A1,1>>,1>>],
    *T2 : [TS ! TS != <<part-of-discourse,TS,*TD,1>>
                TS != <<t-body,*S,*Body ::
                   [S ! S != <<part-of-discourse,S,*TD,1>>
                        S != <<DRSOut,S,*Access,*BodyDRS,1>>],1>>].

*S : [S ! S != <<DRSOut,S,*Access,*DRSOut,1>>]
  <=
    *S : [S ! S != <<cat,S,Verb,1>>
              S != <<sem,S,<<*R1,*A1,*A2,1>>,1>>
              S != <<DRSIn,S,*Access,*DRSOut,1>>].

*S : [S ! S != <<DRSOut,S,*Access,*DRSOut,1>>]
  <=
    *S : [S ! S != <<cat,S,Preposition,1>>
              S != <<DRSIn,S,*Access,*DRSOut,1>>].

*S : [S ! S != <<DRSOut,S,*Access,*DRSOut,1>>]
  <=
    *S : [S ! S != <<cat,S,PrepPhrase,1>>
              S != <<DRSIn,S,*Access,*DRSOut,1>>].

*S : [S ! S != <<DRSOut,S,*Access,*DRSOut,1>>]
  <=
    *S : [S ! S != <<cat,S,Discourse,1>>
              S != <<DRSIn,S,*Access,*DRSOut,1>>].

;;;
;;;  Special DRS used at start of sub-threads
;;;
*S : [S ! S != <<DRSOut,S,*Access,[D !],1>>]
  <=
    *S : [S ! S != <<cat,S,Marker,1>>
              S != <<dominator,S,
                     *D :: [D ! D != <<*R,D,*Access,*X,1>>],
                     *R,1>>].

*S : [S ! S != <<DRSOut,S,*Access,*DRSOut,1>>]
  <=
    *S : [S ! S != <<cat,S,FullDiscourse,1>>
              S != <<threads,S,*T1 ::
                     [TS ! TS != <<t-out,*S,*X  ::
                          [DS ! DS != <<DRSOut,DS,*Access,*DRSOut,1>>]
                          ,1>>],1>>].
```

The relationship between the DRSIn and the previous DRSOut is done by threading. All the situations which support t-in facts are checked for one where information about the threading of that situation is found.

```
*S : [S ! S != <<DRSIn,S,*A,*DRSin,1>>]
<=
  *T1 : [TS ! TS != <<part-of-discourse,TS,*TD,1>>
             TS != <<t-in,*S, *S1 ::
                       [S ! S != <<DRSOut,*S1,*A,*DRSIn,1>>],1>>].
```

This is basically a lexical rule to add the base case of the threads. Lexical items thread through themselves as they have no daughters.

```
*S : [S ! S != <<threads,S,*Thread ::
                 [TS ! TS != <<t-need,*S,*S,1>>],1>>]
<=
   *S : [S ! S != <<use_of,S,*Y,1>>].

*Thread : [TS ! TS != <<t-out,*S,*S,1>>]
<=
   *S : [S ! S != <<cat,S,noun,1>>
             S != <<use_of,S,*Y,1>>
             S != <<threads,S,*Thread,1>>].

*Thread : [TS ! TS != <<t-out,*S,*S,1>>]
<=
   *S : [S ! S != <<cat,S,verbphrase,1>>
             S != <<use_of,S,*Y,1>>
             S != <<threads,S,*Thread,1>>].
```

Spurious partial analyses can occur so it is necessary to identify which discourse an utterance is part of. This is done by relating each utterance to the discourse situation it is part of.

```
*D : [D ! D != <<part-of-discourse,D,*S,1>>]
<=
   *S : [S ! S != <<cat,S,FullDiscourse,1>>
             S != <<daughter,S,*D,1>>].

*D : [D ! D != <<part-of-discourse,D,*X,1>>]
<=
   *S : [S ! S != <<daughter,S,*D,1>>
             S != <<part-of-discourse,S,*X,1>>].
```

```
*T1 : [D ! D != <<part-of-discourse,D,*X,1>>]
<=
   *S : [S ! S != <<threads,S,*T1,1>>
             S != <<part-of-discourse,S,*X,1>>].
```

The actual grammar rules are responsible for building up the syntactic structure and the basic semantic information (relating noun phrases to arguments of verbs) in a very similar way to the the STG description. The grammar rules also build up the threading information.

```
Grammar Rules
;;;
;;;  S -> NP VP
;;;
   *S : [S ! S != <<cat,S,Sentence,1>>
             S != <<sem,S,*Fact,1>>
             S != <<env,S,*SEnv :: *VPEnvType &
                   [Env ! Env != <<anchor,*P1,*NPSem,1>>],1>>
             S != <<daughter,S,*NP,1>>
             S != <<daughter,S,*VP,1>>
             S != <<threads,S,*ThreadS ::
                    [TS ! TS != <<t-need,*S,*Y,1>>
                          TS != <<t-in,*X,*Z,1>>
                          TS != <<t-body,*OUT,*OUT1,1>>
                          TS != <<t-body,*OUT1,*S,1>>
                          TS != <<t-in,*S,*VP,1>>
                          TS != <<t-out,*S,*OUT,1>>],1>>]
   ->
    *NP : [NP ! NP != <<cat,NP,NounPhrase,1>>
                NP != <<sem,NP,*NPSem,1>>
                NP != <<threads,NP,*ThreadNP ::
                        [TS ! TS != <<t-need,*NP,*Y,1>>
                              TS != <<t-out,*NP,*OUT,1>>
                              TS != <<t-feed,*NP,*Z,1>>],1>>
          ],
    *VP : [VP ! VP != <<cat,VP,VerbPhrase,1>>
                VP != <<sem,VP, *Fact ,1>>
                VP != <<env,VP,*VPEnv :: *VPEnvType &
                        [Env ! Env != <<label,*P1,subj,1>>],1>>
                VP != <<threads,VP,*ThreadVP ::
                        [TS ! TS != <<t-need,*VP,*X,1>>
                              TS != <<t-out,*VP,*OUT1,1>>],1>>].
;;;
;;;  VP -> V NP
;;;
   *VP : [VP ! VP != <<cat,VP,VerbPhrase,1>>
```

```
            VP != <<sem,VP,*Fact,1>>
            VP != <<env,VP,*VPEnv :: *VEnvType &
                    [Env ! Env != <<anchor,*P1,*NPSem,1>>],1>>
            VP != <<daughter,VP,*V,1>>
            VP != <<daughter,VP,*NP,1>>
            VP != <<threads,VP,*ThreadVP ::
                    [TS ! TS != <<t-need,*VP,*Y,1>>
                          TS != <<t-in,*X,*Z,1>>
                          TS != <<t-out,*VP,*OUT,1>>
                          TS != <<t-in,*VP,*V,1>>],1>>]
   ->
    *V : [V ! V != <<cat,V,Verb,1>>
              V != <<sem,V,*Fact,1>>
              V != <<env,V,*VEnv :: *VEnvType &
                     [Env ! Env != <<label,*P1,obj,1>>],1>>
              V != <<threads,V,*ThreadV ::
                     [TS ! TS != <<t-need,*V,*X,1>>],1>>],
    *NP : [NP ! NP != <<cat,NP,NounPhrase,1>>
                NP != <<sem,NP,*NPSem,1>>
                NP != <<threads,NP,*ThreadNP ::
                        [TS ! TS != <<t-need,*NP,*Y,1>>
                              TS != <<t-feed,*NP,*Z,1>>
                              TS != <<t-out,*NP,*OUT,1>>],1>>].
;;;
;;;  VP -> Vintrans
;;;
   *VP : [VP ! VP != <<cat,VP,VerbPhrase,1>>
               VP != <<sem,VP,<<*R1,*A1,1>>,1>>
               VP != <<env,VP,*VPEnv,1>>
               VP != <<daughter,VP,*V,1>>
               VP != <<threads,VP,*ThreadVP ::
                       [TS ! TS != <<t-need,*VP,*VP,1>>
                             TS != <<t-out,*VP,*V,1>>],1>>]
   ->
    *V : [V ! V != <<cat,V,Verb,1>>
              V != <<sem,V,<<*R1,*A1,1>>,1>>
              V != <<env,V,*VPEnv,1>>].
;;;
;;;  NP -> Det Noun
;;;
    *NP : [NP ! NP != <<cat,NP,NounPhrase,1>>
                NP != <<sem,NP,*A1,1>>
                NP != <<daughter,NP,*DET,1>>
                NP != <<daughter,NP,*N,1>>
                NP != <<threads,NP,*T2 ::
                    [TS ! TS != <<t-need,*NP,*X,1>>
```

```
                     TS != <<t-in,*Y,*M1 ::
                      [S ! S != <<cat,S,Marker,1>>
                           S != <<dominator,S,*DET,DRSIn,1>>],1>>
                     TS != <<t-in,*NP,*OUT,1>>
                     TS != <<t-range,*DET,*OUT,1>>
                     TS != <<t-feed,*NP,*M2 ::
                      [S ! S != <<cat,S,Marker,1>>
                           S != <<dominator,S,*OUT,DRSOut,1>>],1>>
                     TS != <<t-out,*NP,*DET,1>>],1>>]
->
  *DET : [DET ! DET != <<cat,DET,Determiner,1>>
                DET != <<sem,DET,<<*PQUANT,*PVAR,
                                   *PRANGE,*PBODY,1>>,1>>
                DET != <<env,DET,*Env ::
                        [Env ! Env != <<anchor,
                                        *PQUANT,every,1>>],1>>
                DET != <<threads,DET,*T1 ::
                         [TS ! TS != <<t-need,
                                       *DET,*X,1>>],1>>],
  *N : [N ! N != <<cat,N,noun,1>>
            N != <<env,N,*Env ::
                   [Env ! Env != <<label,*A1,arg1,1>>],1>>
            N != <<sem,N,<<*R1,*A1,1>>,1>>
            N != <<threads,N,*T3 ::
                   [TS ! TS != <<t-need,*N,*Y,1>>
                         TS != <<t-out,*N,*OUT,1>>],1>>].

*NP : [NP ! NP != <<cat,NP,NounPhrase,1>>
            NP != <<sem,NP,*A1,1>>
            NP != <<daughter,NP,*DET,1>>
            NP != <<daughter,NP,*N,1>>
            NP != <<threads,NP,*T2 ::
                    [TS ! TS != <<t-need,*NP,*Y,1>>
                          TS != <<t-in,*NP,*OUT,1>>
                          TS != <<t-range,*DET,*OUT,1>>
                          TS != <<t-feed,*NP,*NP,1>>
                          TS != <<t-out,*NP,*DET,1>>],1>>]
->
  *DET : [DET ! DET != <<cat,DET,Determiner,1>>
                DET != <<sem,DET,<<*PQUANT,*PVAR,
                                   *PRANGE,*PBODY,1>>,1>>
                DET != <<env,DET,*Env ::
                         [Env ! Env != <<anchor,
                                         *PQUANT,some,1>>],1>>
                DET != <<threads,DET,*T1 ::
                        [TS ! TS != <<t-need,*DET,*X,1>>],1>>],
```

```
      *N : [N ! N != <<cat,N,noun,1>>
                N != <<env,N,*Env ::
                       [Env ! Env != <<label,*A1,arg1,1>>],1>>
                N != <<sem,N,<<*R1,*A1,1>>,1>>
                N != <<threads,N,*T3 ::
                       [TS ! TS != <<t-need,*N,*Y,1>>
                             TS != <<t-out,*N,*OUT,1>>],1>>].
;;;
;;;  NP -> Pronoun
;;;
   *NP : [NP ! NP != <<cat,NP,NounPhrase,1>>
               NP != <<sem,NP,*X,1>>
               NP != <<daughter,NP,*PN,1>>
               NP != <<threads,NP,*ThreadNP ::
                       [TS ! TS != <<t-need,*NP,*Y,1>>
                             TS != <<t-out,*NP,*NP,1>>
                             TS != <<t-feed,*NP,*PN,1>>],1>>]

   ->
      *PN : [PN ! PN != <<cat,PN,Pronoun,1>>
                  PN != <<sem,PN,*X,1>>
                  PN != <<threads,PN,*Thread ::
                          [TS ! TS != <<t-need,*PN,*Y,1>>],1>>].
;;;
;;;  NP -> Propernoun
;;;
   *NP : [NP ! NP != <<cat,NP,NounPhrase,1>>
               NP != <<sem,NP,*X,1>>
               NP != <<daughter,NP,*PN,1>>
               NP != <<threads,NP,*ThreadNP ::
                       [TS ! TS != <<t-need,*NP,*Y,1>>
                             TS != <<t-out,*NP,*NP,1>>
                             TS != <<t-feed,*NP,*PN,1>>],1>>]
   ->
      *PN : [PN ! PN != <<cat,PN,ProperNoun,1>>
                  PN != <<sem,PN,*X,1>>
                  PN != <<threads,PN,*Thread ::
                          [TS ! TS != <<t-need,*PN,*Y,1>>],1>>].
;;;
;;;  N -> N PP
;;;
  *N1 :  [N ! N != <<cat,N,noun,1>>
              N != <<daughter,N,*N,1>>
              N != <<daughter,N,*PP,1>>
              N != <<sem,N,<<*R1,*A1,1>>,1>>
              N != <<env,N,*NEnv :: *Envtype, 1>>
```

```
              N != <<threads,N,*T2 ::
                     [TS ! TS != <<t-need,*N1,*X,1>>
                           TS != <<t-in,*N1,*PP,1>>
                           TS != <<t-body,*OUT1,*N1,1>>
                           TS != <<t-in,*Y,*OUT,1>>
                           TS != <<t-out,*N1,*OUT1,1>>],1>>]
    ->
      *N : [N ! N != <<cat,N,noun,1>>
                N != <<env,N,*Env :: *Envtype &
                       [Env ! Env != <<label,*A1,arg1,1>>],1>>
                N != <<sem,N,<<*R1,*A1,1>>,1>>
                N != <<threads,N,*T1 ::
                       [TS ! TS != <<t-need,*N,*X,1>>
                             TS != <<t-out,*N,*OUT,1>>],1>>],
      *PP : [PP ! PP != <<cat,PP,PrepPhrase,1>>
                  PP != <<sem,PP,*Fact,1>>
                  PP != <<threads,PP,*T3 ::
                          [TS ! TS != <<t-need,*PP,*Y,1>>
                                TS != <<t-out,*PP,*OUT1,1>>],1>>].
;;;
;;;  PP -> P NP
;;;
    *PP : [PP ! PP != <<cat,PP,PrepPhrase,1>>
                PP != <<daughter,PP,*PREP,1>>
                PP != <<daughter,PP,*NP,1>>
                PP != <<sem,PP,*Fact,1>>
                PP != <<env,PP,*PPEnv :: *EnvType &
                        [Env ! Env != <<anchor,*P2,*NPSem,1>>],1>>
                PP != <<threads,PP,*T1 ::
                        [TS ! TS != <<t-need,*PP,*Y,1>>
                              TS != <<t-in,*X,*Z,1>>
                              TS != <<t-in,*PP,*PREP,1>>
                              TS != <<t-out,*PP,*OUT,1>>],1>>]
    ->
      *PREP : [PREP ! PREP != <<cat,PREP,Preposition,1>>
                      PREP != <<sem,PREP,*Fact,1>>
                      PREP != <<env,PREP,*PEnv :: *Envtype &
                                [Env ! Env != <<label,
                                                *P2,arg2,1>>],1>>
                      PREP != <<threads,PREP,*T2 ::
                                [TS ! TS != <<t-need,
                                              *PREP,*X,1>>],1>>
              ],
      *NP :  [NP ! NP != <<cat,NP,NounPhrase,1>>
                   NP != <<sem,NP,*NPSem,1>>
                   NP != <<threads,NP,*T3 ::
```

```
                        [TS ! TS != <<t-need,*NP,*Y,1>>
                              TS != <<t-feed,*NP,*Z,1>>
                              TS != <<t-out,*NP,*OUT,1>>],1>>].
;;;
;;;  Discourse -> S
;;;
   *DS : [DS ! DS != <<cat,DS,Discourse,1>>
               DS != <<daughter,DS,*S,1>>
               DS != <<threads,DS,*ThreadDS ::
                       [TS ! TS != <<t-need,*DS,*X,1>>
                             TS != <<t-in,*DS,*S,1>>
                             TS != <<t-out,*DS,*OUT,1>>],1>>]
   ->
     *S : [S ! S != <<cat,S,sentence,1>>
               S != <<threads,S,*ThreadS ::
                      [TS ! TS != <<t-need,*S,*X,1>>
                            TS != <<t-out,*S,*OUT,1>>],1>>].
;;;
;;;  Discourse -> Discourse S
;;;
   *DS1 : [DS ! DS != <<cat,DS,Discourse,1>>
                DS != <<daughter,DS,*DS,1>>
                DS != <<daughter,DS,*S,1>>
                DS != <<threads,DS,*ThreadDS1 ::
                        [TS ! TS != <<t-need,*DS1,*X,1>>
                              TS != <<t-in,*Y,*OUT,1>>
                              TS != <<t-in,*DS1,*S,1>>
                              TS != <<t-out,*DS1,*OUT1,1>>],1>>]
   ->
     *DS : [DS ! DS != <<cat,DS,Discourse,1>>
                 DS != <<threads,DS,*ThreadDS ::
                         [TS ! TS != <<t-need,*DS,*X,1>>
                               TS != <<t-out,*DS,*OUT,1>>],1>>],
     *S : [S ! S != <<cat,S,sentence,1>>
               S != <<threads,S,*ThreadS ::
                      [TS ! TS != <<t-need,*S,*Y,1>>
                            TS != <<t-out,*S,*OUT1,1>>],1>>].
;;;
;;;  Fulldiscourse -> ds Discourse de
;;;
   [DS ! DS != <<cat,DS,fulldiscourse,1>>
         DS != <<part-of-discourse,DS,DS,1>>
         DS != <<daughter,DS,*S,1>>
         DS != <<daughter,DS,*DS,1>>
         DS != <<threads,DS,
                 *ThreadDS ::
```

```
                 [TS ! TS != <<part-of-discourse,TS,DS,1>>
                       TS != <<t-in,*X,*S,1>>
                       TS != <<t-out,DS,*Y,1>>],1>>]
  ->
    *S : [S ! S != <<cat,S,DisStart,1>>],
    *DS : [DS ! DS != <<cat,DS,discourse,1>>
                DS != <<threads,DS,*ThreadDS ::
                        [TS ! TS != <<t-need,*DS,*X,1>>
                              TS != <<t-out,*DS,*Y,1>>
                        ],1>>],
    [S ! S != <<cat,S,DisEnd,1>>].

Lexical Entries
;;;
;;;  Discourse start and end markers
;;;
ds -
   [DS ! DS != <<cat,DS,DisStart,1>>
         DS != <<DRSOut,DS,AccessStart,[DS !],1>>]
de -
   [DS ! DS != <<cat,DS,DisEnd,1>>]
;;;
;;;  Proper nouns
;;;
Hanako -
   [PN ! PN != <<cat,PN,ProperNoun,1>>
         PN != <<use_of,PN,"Hanako",1>>
         PN != <<type,PN,female,1>>
         PN != <<sem,PN,h,1>>]
Anna -
   [PN ! PN != <<cat,PN,ProperNoun,1>>
         PN != <<use_of,PN,"Anna",1>>
         PN != <<type,PN,female,1>>
         PN != <<sem,PN,a,1>>]
Taro -
   [PN ! PN != <<cat,PN,ProperNoun,1>>
         PN != <<use_of,PN,"Taro",1>>
         PN != <<type,PN,male,1>>
         PN != <<sem,PN,t,1>>]
;;;
;;;  Pronouns
;;;
he -
   [PN ! PN != <<cat,PN,ProNoun,1>>
         PN != <<use_of,PN,"he",1>>
         PN != <<type,PN,male,1>>
```

```
         PN != <<sem,PN,PN1,1>>]
she -
   [PN ! PN != <<cat,PN,ProNoun,1>>
         PN != <<use_of,PN,"she",1>>
         PN != <<type,PN,female,1>>
         PN != <<sem,PN,PN2,1>>]
her -
   [PN ! PN != <<cat,PN,ProNoun,1>>
         PN != <<use_of,PN,"her",1>>
         PN != <<type,PN,female,1>>
         PN != <<sem,PN,PN2,1>>]
it -
   [PN ! PN != <<cat,PN,ProNoun,1>>
         PN != <<use_of,PN,"it",1>>
         PN != <<type,PN,neuter,1>>
         PN != <<sem,PN,PN3,1>>]
;;;
;;;  Common nouns
;;;
man -
   [N ! N != <<cat,N,noun,1>>
        N != <<use_of,N,"man",1>>
        N != <<type,N,male,1>>
        N != <<sem,N,<<R1,MA1,1>>,1>>
        N != <<env,N,ManEnv,1>>]
donkey -
   [N ! N != <<cat,N,noun,1>>
        N != <<use_of,N,"donkey",1>>
        N != <<type,N,neuter,1>>
        N != <<sem,N,<<R1,DA1,1>>,1>>
        N != <<env,N,DonkeyEnv,1>>]
hat -
   [N ! N != <<cat,N,noun,1>>
        N != <<use_of,N,"hat",1>>
        N != <<type,N,neuter,1>>
        N != <<sem,N,<<R1,HA1,1>>,1>>
        N != <<env,N,HatEnv,1>>]
;;;
;;;  Quantifiers
;;;
a -
   [D ! D != <<cat,D,Determiner,1>>
        D != <<use_of,D,"a",1>>
        D != <<sem,D,<<Q1,A1,A2,A3,1>>,1>>
        D != <<env,D,AEnv,1>>]
every -
```

```
   [D ! D != <<cat,D,Determiner,1>>
        D != <<use_of,D,"every",1>>
        D != <<sem,D,<<Q1,A1,A2,A3,1>>,1>>
        D != <<env,D,EveryEnv,1>>]
;;;
;;;  Verbs
;;;
sings -
   [VP ! VP != <<cat,VP,Verb,1>>
         VP != <<use_of,VP,"sings",1>>
         VP != <<sem,VP,<<R1,A1,1>>,1>>
         VP != <<env,VP,SingEnv,1>>]
smiles -
   [VP ! VP != <<cat,VP,Verb,1>>
         VP != <<use_of,VP,"smiles",1>>
         VP != <<sem,VP,<<R1,A1,1>>,1>>
         VP != <<env,VP,SmileEnv,1>>]
walks -
   [VP ! VP != <<cat,VP,Verb,1>>
         VP != <<use_of,VP,"walks",1>>
         VP != <<sem,VP, <<R1,WA1,1>>,1>>
         VP != <<env,VP,WalkEnv,1>>]
talks -
   [VP ! VP != <<cat,VP,Verb,1>>
         VP != <<use_of,VP,"talks",1>>
         VP != <<sem,VP,<<R1,TA1,1>>,1>>
         VP != <<env,VP,TalkEnv,1>>]
likes -
   [V ! V != <<cat,V,Verb,1>>
        V != <<use_of,V,"likes",1>>
        V != <<sem,V,<<R1,A1,A2,1>>,1>>
        V != <<env,V,LikeEnv,1>>]
;;;
;;;  Prepositions
;;;
with -
   [PREP ! PREP != <<cat,PREP,Preposition,1>>
           PREP != <<use_of,PREP,"with",1>>
           PREP != <<sem,PREP,<<P1,A1,A2,1>>,1>>
           PREP != <<env,PREP,WithEnv,1>>]
```

A.5 DPL-NL description

This is an ASTL description which gives a DPL like treatment to the Rooth fragment—details are given in Chapter 6. Utterance situations are related to input and output assignments which contain actual assignments and conditions on what they assign.

This is based on the DRT description. Basically the whole DRT description can be used and only the constraints regarding the relations between DRSs need to be re-written. The threading relations of DRSs and DPL assignments are exactly the same.

```
Individuals
    {}
Relations
    ( use_of/2 cat/2
      sem/2 env/2 type/2 threads/2
      NounPhrase/1 Sentence/1 VerbPhrase/1 Discourse/1
      ProperNoun/1 ProNoun/1 PrepPhrase/1 Preposition/1
      FullDiscourse/1 DisStart/1 DisEnd/1
      subj/1 pred/1
      label/2 anchor/2
      exists/1 forall/2
      sing/1 like/2 smile/1
      donkey/1 man/1 hat/1 with/2
      named/2 male/1 female/1 neuter/1
      AssignIn/2 AssignOut/2
      t-in/2 t-out/2 t-feed/2 t-need/2 )
Hush  ;; relations not to be displayed on output (by default)
    (daughter threads)
Parameters
    {R1,A1,A2, A3, S,S1,S2,TS,NP,VP,PN,V,Env,DS,Res,
     PN1, PN2, PN3, A, PREP, PP, E1, H, T, P,
     DA1, MA1, HA1, WA1, TA1, T1, H1, P1 }
Variables
    {*X, *Y, *Z, *S, *U, *I, *K, *G, *Fact, *VPEnv, *SEnv,
     *VEnvType, *Env, *PPEnv, *PEnv, *EnvType, *Nenv, *NPEnv
     *R1, *A1, *A2, *A3, *VR1, *VA1, *VA2, *DS, *DS1, *DS2, *PN,
     *pred, *VPEnvType, *VEnv,
     *AssignIn, *AssignIn, *AssignOut,
     *ThreadS, *ThreadNP, *ThreadVP, *ThreadV, *Thread, *ThreadDS,
     *ThreadDS1, *TYPE, *OUT, *TD,
     *A, *Acc, *OUT1, *M1, *M2,
     *QEXPR, *Range, *Body, *BodyAssign, *RangeAssign,
     *QUANT, *Q, *PVAR, *PRANGE, *PBODY, *Name
     *T1, *T2, *T3, *S1, *S2, *P1, *P2, *NPSem, *BodyOut,
     *NP, *VP, *V, *N, *DET, *PP, *PREP, *N1, *D}
Situations
    (SingEnv :: [Env ! Env != <<label,R1,pred,1>>
                       Env != <<anchor,R1,sing,1>>
                       Env != <<label,A1,subj,1>>]
     SmileEnv :: [Env ! Env != <<label,R1,pred,1>>
```

```
                     Env != <<anchor,R1,smile,1>>
                     Env != <<label,A1,subj,1>>]
     WalkEnv :: [Env ! Env != <<label,R1,pred,1>>
                       Env != <<anchor,R1,walk,1>>
                       Env != <<label,WA1,subj,1>>]
     TalkEnv :: [Env ! Env != <<label,R1,pred,1>>
                       Env != <<anchor,R1,talk,1>>
                       Env != <<label,TA1,subj,1>>]
     LikeEnv :: [Env ! Env != <<label,R1,pred,1>>
                       Env != <<anchor,R1,like,1>>
                       Env != <<label,A1,subj,1>>
                       Env != <<label,A2,obj,1>>]
     ManEnv :: [Env ! Env != <<label,R1,pred,1>>
                      Env != <<anchor,R1,man,1>>
                      Env != <<label,MA1,arg1,1>>]
     DonkeyEnv :: [Env ! Env != <<label,R1,pred,1>>
                         Env != <<anchor,R1,donkey,1>>
                         Env != <<label,DA1,arg1,1>>]
     HatEnv :: [Env ! Env != <<label,R1,pred,1>>
                      Env != <<anchor,R1,hat,1>>
                      Env != <<label,HA1,arg1,1>>]
     AEnv :: [Env ! Env != <<label,Q1,quantifier,1>>
                    Env != <<anchor,Q1,some,1>>
                    Env != <<label,A1,var,1>>
                    Env != <<label,A2,range,1>>
                    Env != <<label,A3,body,1>>]
     EveryEnv :: [Env ! Env != <<label,Q1,quantifier,1>>
                        Env != <<anchor,Q1,every,1>>
                        Env != <<label,A1,var,1>>
                        Env != <<label,A2,range,1>>
                        Env != <<label,A3,body,1>>]
     WithEnv :: [Env ! Env != <<label,P1,prep,1>>
                       Env != <<anchor,P1,with,1>>
                       Env != <<label,A1,arg1,1>>
                       Env != <<label,A2,arg2,1>>] )
GoalProp
   *S : [S ! S != <<cat,S,fulldiscourse,1>>
             S != <<AssignOut,S,*AssignOut,1>>]
Constraints
```

These first set of constraints define the relationship between the incoming assignments and the outgoing assignment in the various types of node. The only interesting ones are sentences where a new condition is added, nouns where type information male/female is added and pronouns where the current `AssignIn` is checked for a object of the right type that has already been mentioned. Determiners also do interesting things. The other utterance types simply "copy" the `AssignIn` to `AssignOut`.

```
   *S : [S ! S != <<AssignOut,S,
                    *AssignIn &
                    [DS ! DS != <<*VR1,*VA1,1>>],1>>]
   <=
      *S : [S ! S != <<cat,S,sentence,1>>
                S != <<AssignIn,S,*AssignIn,1>>
                S != <<sem,S,<<*R1,*A1,1>>,1>>
                S != <<env,S,*SEnv ::
                       [Env ! Env != <<anchor,*R1,*VR1,1>>
                              Env != <<anchor,*A1,*VA1,1>>],1>>].

   *S : [S ! S != <<AssignOut,S,
                   *AssignIn &
                    [DS ! DS != <<*VR1,*VA1,*VA2,1>>],1>>]
   <=
      *S : [S ! S != <<cat,S,sentence,1>>
                S != <<AssignIn,S,*AssignIn,1>>
                S != <<sem,S,<<*R1,*A1,*A2,1>>,1>>
                S != <<env,S,*SEnv ::
                       [Env ! Env != <<anchor,*R1,*VR1,1>>
                              Env != <<anchor,*A1,*VA1,1>>
                              Env != <<anchor,*A2,*VA2,1>>],1>>].

*S : [S ! S != <<Assignout,S,
                *AssignIn  &
                 [A ! A != <<named,*X,*Name,1>>
                      A != <<type,*X,*TYPE,1>>
                      A != <<assigned,*X,1>>],1>>]
  <=
      *S : [S ! S != <<cat,S,ProperNoun,1>>
                S != <<use_of,S,*Name,1>>
                S != <<type,S,*TYPE,1>>
                S != <<sem,S,*X,1>>
                S != <<AssignIn,S,*AssignIn,1>>].

*S : [S ! S != <<Assignout,S,*AssignIn &
                       [A ! A != <<assigned,*X,1>>
                            A != <<is,*X,*Z,1>>],1>>]
  <=
      *S : [S ! S != <<cat,S,Pronoun,1>>
                S != <<type,S,*TYPE,1>>
                S != <<sem,S,*X,1>>
                S != <<AssignIn,S,*AssignIn,1>>
                S != <<Accessible,S,
                        *Acc :: [A ! A != <<type,*Z,*TYPE,1>>
                                     A != <<accessible,*Z,1>>],
```

```
                    1>>].

*S : [S ! S != <<AssignOut,S,
                 *AssignIn &
                 [DS ! DS != <<forall,*RangeAssign,
                                     *BodyAssign,1>>],1>>]
  <=
      *S : [S ! S != <<cat,S,Determiner,1>>
                S != <<part-of-discourse,S,*TD,1>>
                S != <<sem,S,<<*Q,*X,*Y,*Z,1>>,1>>
                S != <<AssignIn,S,*AssignIn,1>>
                S != <<env,S,*SEnv ::
                       [Env ! Env != <<anchor,*Q,every,1>>],1>>],
      *T1 : [TS ! TS != <<part-of-discourse,TS,*TD,1>>
                  TS != <<t-body,*S,*Body  ::
                          [S ! S != <<part-of-discourse,S,*TD,1>>
                               S != <<AssignOut,S,
                                      *BodyAssign,1>>],1>>],
      *T2 : [TS ! TS != <<part-of-discourse,TS,*TD,1>>
                  TS != <<t-range,*S,*Range ::
                          [S ! S != <<part-of-discourse,S,*TD,1>>
                               S != <<AssignOut,S,
                                      *RangeAssign,1>>],1>>].

*S : [S ! S != <<AssignOut,S,*BodyOut,1>>]
  <=
      *S : [S ! S != <<cat,S,Determiner,1>>
                S != <<part-of-discourse,S,*TD,1>>
                S != <<sem,S,<<*Q,*X,*Y,*Z,1>>,1>>
                S != <<env,S,*SEnv ::
                       [Env ! Env != <<anchor,*Q,some,1>>],1>>],
      *T2 : [TS ! TS != <<part-of-discourse,TS,*TD,1>>
                  TS != <<t-body,*S,*Body ::
                          [S ! S != <<part-of-discourse,S,*TD,1>>
                               S != <<AssignOut,S,
                                      *BodyOut,
                                      1>>],1>>].

*S : [S ! S != <<AssignMid,S,
                 *G &
                 [A ! A != <<assigned,*X,1>>],1>>]
  <=
      *S : [S ! S != <<cat,S,Determiner,1>>
                S != <<sem,S,<<*Q,*X,*Y,*Z,1>>,1>>
                S != <<env,S,*SEnv ::
                       [Env ! Env != <<anchor,*Q,some,1>>],1>>
```

```
                S != <<AssignIn,S,*G,1>>
                S != <<ind,S,*I,1>>].


*S : [S ! S != <<AssignMid,S,
                  [A ! A != <<assigned,*X,1>>
                       A != <<of-type,A,*G,1>>],1>>]
  <=
      *S : [S ! S != <<cat,S,Determiner,1>>
                S != <<sem,S,<<*Q,*X,*Y,*Z,1>>,1>>
                S != <<env,S,*SEnv ::
                       [Env ! Env != <<anchor,*Q,every,1>>],1>>
                S != <<AssignIn,S,*G,1>>
                S != <<ind,S,*I,1>>].

*S : [S ! S != <<AssignOut,S,
                  *AssignIn &
                  [DS ! DS != <<*VR1,*VA1,1>>
                        DS != <<type,*VA1,*TYPE,1>>],1>>]
  <=
      *S : [S ! S != <<cat,S,NounPhrase,1>>
                S != <<daughter,S,
                       *DS1 :: [DS !
                                DS != <<cat,DS,determiner,1>>],1>>
                S != <<daughter,S,
                       *DS2 :: [DS !
                                DS != <<cat,DS,noun,1>>
                                DS != <<type,DS,*TYPE,1>>
                                DS != <<sem,DS,<<*R1,*A1,1>>,1>>],1>>
                S != <<env,S,*SEnv ::
                      [Env ! Env != <<anchor,*R1,*VR1,1>>
                             Env != <<anchor,*A1,*VA1,1>>],1>>
                S != <<AssignIn,S,*AssignIn,1>>].

*S : [S ! S != <<AssignOut,S,*BodyAssign,1>>]
  <=
      *S : [S ! S != <<cat,S,NounPhrase,1>>
                S != <<part-of-discourse,S,*TD,1>>
                S != <<daughter,S,
                       *DS :: [DS !
                               DS != <<cat,DS,ProperNoun,1>>],1>>],
      *T2 : [TS ! TS != <<part-of-discourse,TS,*TD,1>>
                  TS != <<t-body,*S,*Body ::
                          [S ! S != <<part-of-discourse,S,*TD,1>>
                               S != <<AssignOut,S,
                                      *BodyAssign,1>>],1>>].
```

```
*S : [S ! S != <<AssignOut,S,*BodyAssign,1>>]
  <=
      *S : [S ! S != <<cat,S,NounPhrase,1>>
                S != <<daughter,S,
                       *DS :: [DS !
                               DS != <<cat,DS,ProNoun,1>>],1>>],
      *T2 : [TS ! TS != <<part-of-discourse,TS,*TD,1>>
                  TS != <<t-body,*S,*Body ::
                          [S ! S != <<part-of-discourse,S,*TD,1>>
                               S != <<AssignOut,S,
                                      *BodyAssign,1>>],1>>].

*S : [S ! S != <<AssignOut,S,*AssignOut,1>>]
  <=
      *S : [S ! S != <<cat,S,Noun,1>>
                S != <<use_of,S,*X,1>>
                S != <<AssignIn,S,*AssignOut,1>>].

*S : [S ! S != <<AssignOut,S,
                 *AssignIn &
                 [DS ! DS != <<*VR1,*VA1,*VA2,1>>],1>>]
  <=
      *S : [S ! S != <<cat,S,Noun,1>>
                S != <<env,S,*SEnv ::
                       [Env ! Env != <<label,*VA1,arg1,1>>],1>>
                 S != <<daughter,S,*PP ::
                        [PP ! PP != <<cat,PP,PrepPhrase,1>>
                              PP != <<sem,PP,<<*R1,*A1,*A2,1>>,1>>
                              PP != <<env,PP,*Env ::
                                      [Env !
                                       Env != <<anchor,*R1,
                                                *VR1,1>>
                                       Env != <<anchor,*A2,
                                                 *VA2,1>>],1>>],1>>
                 S != <<AssignIn,S,*AssignIn,1>>].

*S : [S ! S != <<AssignOut,S,*AssignOut,1>>]
  <=
      *S : [S ! S != <<cat,S,VerbPhrase,1>>
                S != <<AssignIn,S,*AssignOut,1>>].

*S : [S ! S != <<AssignOut,S,*BodyAssign,1>>]
  <=
      *S : [S ! S != <<cat,S,Verb,1>>
                S != <<part-of-discourse,S,*TD,1>>
                S != <<sem,S,<<*R1,*A1,1>>,1>>],
```

```
      *T2 : [TS ! TS != <<part-of-discourse,TS,*TD,1>>
                  TS != <<t-body,*S,*Body ::
                          [S ! S != <<part-of-discourse,S,*TD,1>>
                               S != <<AssignOut,S,
                                      *BodyAssign,1>>],1>>].

*S : [S ! S != <<AssignOut,S,*AssignOut,1>>]
  <=
      *S : [S ! S != <<cat,S,Verb,1>>
                S != <<sem,S,<<*R1,*A1,*A2,1>>,1>>
                S != <<AssignIn,S,*AssignOut,1>>].

*S : [S ! S != <<AssignOut,S,*AssignOut,1>>]
  <=
      *S : [S ! S != <<cat,S,Preposition,1>>
                S != <<AssignIn,S,*AssignOut,1>>].

*S : [S ! S != <<AssignOut,S,*AssignOut,1>>]
  <=
      *S : [S ! S != <<cat,S,PrepPhrase,1>>
                S != <<AssignIn,S,*AssignOut,1>>].

*S : [S ! S != <<AssignOut,S,*AssignOut,1>>]
  <=
      *S : [S ! S != <<cat,S,Discourse,1>>
                S != <<AssignIn,S,*AssignOut,1>>].

*S : [S ! S != <<AssignOut,S,*AssignOut,1>>
      ]
  <=
      *S : [S ! S != <<cat,S,FullDiscourse,1>>
                S != <<threads,S,*T1 ::
                          [TS ! TS != <<t-out,*S,*X ::
                            [DS ! DS != <<AssignOut,DS,*AssignOut,1>>]
                            ,1>>],1>>].
```

In this description accessibility is a direct function of the assigned DPL variables in the incoming assignment. We need merely to state the type of the accessibility situation to be that of the incoming assignment and that assigned variables are accessible ones.

```
*S : [S ! S != <<Accessible,S,*Acc :: *AssignIn,1>>]
   <=
      *S : [S ! S != <<AssignIn,S,*AssignIn,1>>].

*S : [S ! S != <<accessible,*X,1>>]
   <=
```

```
      *S : [S ! S != <<assigned,*X,1>>].

*S : [S ! S != <<Accessible,S,*Acc :: *G,1>>]
   <=
      *S : [S ! S != <<Accessible,S,
                       *Acc :: [T ! T != <<of-type,T,*G,1>>],1>>].
```

The relationship between the AssignIn and the previous AssignOut is done by threading. All the situations which support the t-in are checked for one where information about the threading of that situation is found.

```
*S : [S ! S != <<AssignIn,S,*AssignIn,1>>]
<=
  *T1 : [TS ! TS != <<part-of-discourse,TS,*TD,1>>
              TS != <<t-in,*S,
                      *S1 ::
                       [S1 !
                        S1 != <<AssignOut,S1,
                                *AssignIn,1>>],1>>].

*S : [S ! S != <<AssignIn,S,*K,1>>]
<=
  *T1 : [TS ! TS != <<part-of-discourse,TS,*TD,1>>
              TS != <<t-note,
                      *DET ::
                       [S1 ! S1 != <<AssignMid,S1,*K,1>>],
                      *S,1>>].
```

This is basically a lexical rule to add the base case of the threads. Lexical items thread through themselves as they have no daughters.

```
*S : [S ! S != <<threads,S,*Thread ::
                 [TS ! TS != <<t-need,*S,*S,1>>],1>>]
<=
   *S : [S ! S != <<use_of,S,*Y,1>>].

*Thread : [TS ! TS != <<t-out,*S,*S,1>>]
<=
   *S : [S ! S != <<cat,S,noun,1>>
             S != <<use_of,S,*Y,1>>
             S != <<threads,S,*Thread,1>>].

*Thread : [TS ! TS != <<t-out,*S,*S,1>>]
<=
   *S : [S ! S != <<cat,S,verbphrase,1>>
```

```
             S != <<use_of,S,*Y,1>>
            S != <<threads,S,*Thread,1>>].
```

Spurious partial analyses can occur so it is necessary to identify which utterance situations are part of which which discourse. Thus each utterance situation is related to the full discourse situation it is part of. Also each situation containing threading information is marked likewise.

```
*D : [D ! D != <<part-of-discourse,D,*S,1>>]
<=
   *S : [S ! S != <<cat,S,FullDiscourse,1>>
             S != <<daughter,S,*D,1>>].

*D : [D ! D != <<part-of-discourse,D,*X,1>>]
<=
   *S : [S ! S != <<daughter,S,*D,1>>
             S != <<part-of-discourse,S,*X,1>>].

*T1 : [D ! D != <<part-of-discourse,D,*X,1>>]
<=
   *S : [S ! S != <<threads,S,*T1,1>>
             S != <<part-of-discourse,S,*X,1>>].
Grammar Rules
;;;
;;;  S -> NP VP
;;;
   *S : [S ! S != <<cat,S,Sentence,1>>
             S != <<sem,S,*Fact,1>>
             S != <<env,S,*SEnv ::
                    *VPEnvType &
                    [Env ! Env != <<anchor,*P1,*NPSem,1>>],1>>
             S != <<daughter,S,*NP,1>>
             S != <<daughter,S,*VP,1>>
             S != <<threads,S,*ThreadS ::
                    [TS ! TS != <<t-need,*S,*Y,1>>
                          TS != <<t-in,*X,*Z,1>>
                          TS != <<t-body,*OUT,*OUT1,1>>
                          TS != <<t-body,*OUT1,*S,1>>
                          TS != <<t-in,*S,*VP,1>>
                          TS != <<t-out,*S,*OUT,1>>],1>>]
   ->
    *NP : [NP ! NP != <<cat,NP,NounPhrase,1>>
                NP != <<sem,NP,*NPSem,1>>
                NP != <<threads,NP,*ThreadNP ::
                       [TS ! TS != <<t-need,*NP,*Y,1>>
                             TS != <<t-out,*NP,*OUT,1>>
```

```
                            TS != <<t-feed,*NP,*Z,1>>],1>>
          ],
    *VP : [VP ! VP != <<cat,VP,VerbPhrase,1>>
               VP != <<sem,VP, *Fact ,1>>
               VP != <<env,VP,*VPEnv :: *VPEnvType &
                       [Env ! Env != <<label,*P1,subj,1>>],1>>
               VP != <<threads,VP,*ThreadVP ::
                       [TS ! TS != <<t-need,*VP,*X,1>>
                             TS != <<t-out,*VP,*OUT1,1>>],1>>].
;;;
;;;  VP -> V NP
;;;
   *VP : [VP ! VP != <<cat,VP,VerbPhrase,1>>
              VP != <<sem,VP,*Fact,1>>
              VP != <<env,VP,
                      *VPEnv ::
                       *VEnvType &
                       [Env ! Env != <<anchor,*P1,*NPSem,1>>],1>>
              VP != <<daughter,VP,*V,1>>
              VP != <<daughter,VP,*NP,1>>
              VP != <<threads,VP,*ThreadVP ::
                      [TS ! TS != <<t-need,*VP,*Y,1>>
                            TS != <<t-in,*V,*Z,1>>
                            TS != <<t-out,*VP,*OUT,1>>
                            TS != <<t-in,*VP,*V,1>>],1>>]
   ->
    *V : [V ! V != <<cat,V,Verb,1>>
             V != <<sem,V,*Fact,1>>
             V != <<env,V,*VEnv :: *VEnvType &
                     [Env ! Env != <<label,*P1,obj,1>>],1>>
             V != <<threads,V,*ThreadV ::
                     [TS ! TS != <<t-need,*V,*X,1>>],1>>],
    *NP : [NP ! NP != <<cat,NP,NounPhrase,1>>
               NP != <<sem,NP,*NPSem,1>>
               NP != <<threads,NP,*ThreadNP ::
                       [TS ! TS != <<t-need,*NP,*Y,1>>
                             TS != <<t-feed,*NP,*Z,1>>
                             TS != <<t-out,*NP,*OUT,1>>],1>>].
;;;
;;;  VP -> Vintrans
;;;
   *VP : [VP ! VP != <<cat,VP,VerbPhrase,1>>
              VP != <<sem,VP,<<*R1,*A1,1>>,1>>
              VP != <<env,VP,*VPEnv,1>>
              VP != <<daughter,VP,*V,1>>
              VP != <<threads,VP,*ThreadVP ::
```

```
                     [TS ! TS != <<t-need,*VP,*VP,1>>
                           TS != <<t-out,*VP,*V,1>>],1>>]
   ->
    *V : [V ! V != <<cat,V,Verb,1>>
              V != <<sem,V,<<*R1,*A1,1>>,1>>
              V != <<env,V,*VPEnv,1>>].
;;;
;;;  NP -> Det Noun
;;;
    *NP : [NP ! NP != <<cat,NP,NounPhrase,1>>
                NP != <<sem,NP,*A1,1>>
                NP != <<env,NP,*NPEnv ::
                        *EnvType &
                        [Env ! Env != <<anchor,*A,*A1,1>>],1>>
                NP != <<daughter,NP,*DET,1>>
                NP != <<daughter,NP,*N,1>>
                NP != <<threads,NP,*T2 ::
                        [TS ! TS != <<t-need,*NP,*X,1>>
                              TS != <<t-in,*NP,*OUT,1>>
                              TS != <<t-note,*DET,*Y,1>>
                              TS != <<t-range,*DET,*NP,1>>
                              TS != <<t-feed,*NP,*NP,1>>
                              TS != <<t-out,*NP,*DET,1>>],1>>]
    ->
      *DET : [DET ! DET != <<cat,DET,Determiner,1>>
                    DET != <<sem,DET,<<*QUANT,*A1,*A2,*A3,1>>,1>>
                    DET != <<threads,DET,*T1 ::
                             [TS ! TS != <<t-need,*DET,*X,1>>],1>>],
      *N : [N ! N != <<cat,N,noun,1>>
                N != <<env,N,*Env :: *EnvType &
                      [Env ! Env != <<label,*A,arg1,1>>],1>>
                N != <<sem,N,<<*R1,*A,1>>,1>>
                N != <<threads,N,*T3 ::
                      [TS ! TS != <<t-need,*N,*Y,1>>
                            TS != <<t-out,*N,*OUT,1>>],1>>].
;;;
;;;  NP -> Pronoun
;;;
   *NP : [NP ! NP != <<cat,NP,NounPhrase,1>>
               NP != <<sem,NP,*X,1>>
               NP != <<daughter,NP,*PN,1>>
               NP != <<threads,NP,*ThreadNP ::
                       [TS ! TS != <<t-need,*NP,*Y,1>>
                             TS != <<t-out,*NP,*NP,1>>
                             TS != <<t-feed,*NP,*PN,1>>],1>>]
   ->
```

```
      *PN : [PN ! PN != <<cat,PN,Pronoun,1>>
                  PN != <<sem,PN,*X,1>>
                  PN != <<threads,PN,*Thread ::
                          [TS ! TS != <<t-need,*PN,*Y,1>>],1>>].
;;;
;;;  NP -> Propernoun
;;;
   *NP : [NP ! NP != <<cat,NP,NounPhrase,1>>
               NP != <<sem,NP,*X,1>>
               NP != <<daughter,NP,*PN,1>>
               NP != <<threads,NP,*ThreadNP ::
                       [TS ! TS != <<t-need,*NP,*Y,1>>
                             TS != <<t-out,*NP,*NP,1>>
                             TS != <<t-feed,*NP,*PN,1>>],1>>]
   ->
      *PN : [PN ! PN != <<cat,PN,ProperNoun,1>>
                  PN != <<sem,PN,*X,1>>
                  PN != <<threads,PN,*Thread ::
                          [TS ! TS != <<t-need,*PN,*Y,1>>],1>>].
;;;
;;;  N -> N PP
;;;
  *N1 :  [N ! N != <<cat,N,noun,1>>
              N != <<daughter,N,*N,1>>
              N != <<daughter,N,*PP,1>>
              N != <<sem,N,<<*R1,*A1,1>>,1>>
              N != <<type,N,*TYPE,1>>
              N != <<env,N,*NEnv :: *Envtype, 1>>
              N != <<threads,N,*T2 ::
                     [TS ! TS != <<t-need,*N1,*X,1>>
                           TS != <<t-in,*N1,*PP,1>>
                           TS != <<t-body,*OUT1,*N1,1>>
                           TS != <<t-in,*Y,*OUT,1>>
                           TS != <<t-out,*N1,*OUT1,1>>],1>>]
    ->
      *N : [N ! N != <<cat,N,noun,1>>
                N != <<env,N,*Env :: *Envtype &
                       [Env ! Env != <<label,*A1,arg1,1>>],1>>
                N != <<sem,N,<<*R1,*A1,1>>,1>>
                N != <<type,N,*TYPE,1>>
                N != <<threads,N,*T1 ::
                       [TS ! TS != <<t-need,*N,*X,1>>
                             TS != <<t-out,*N,*OUT,1>>],1>>],
      *PP : [PP ! PP != <<cat,PP,PrepPhrase,1>>
                  PP != <<sem,PP,*Fact,1>>
                  PP != <<threads,PP,*T3 ::
```

```
                         [TS ! TS != <<t-need,*PP,*Y,1>>
                               TS != <<t-out,*PP,*OUT1,1>>],1>>].
;;;
;;;  PP -> P NP
;;;
    *PP : [PP ! PP != <<cat,PP,PrepPhrase,1>>
                PP != <<daughter,PP,*PREP,1>>
                PP != <<daughter,PP,*NP,1>>
                PP != <<sem,PP,*Fact,1>>
                PP != <<env,PP,*PPEnv :: *EnvType &
                        [Env ! Env != <<anchor,*P2,*NPSem,1>>],1>>
                PP != <<threads,PP,*T1 ::
                        [TS ! TS != <<t-need,*PP,*Y,1>>
                              TS != <<t-in,*X,*Z,1>>
                              TS != <<t-in,*PP,*PREP,1>>
                              TS != <<t-out,*PP,*OUT,1>>],1>>]
    ->
      *PREP : [PREP !
               PREP != <<cat,PREP,Preposition,1>>
               PREP != <<sem,PREP,*Fact,1>>
               PREP != <<env,PREP,*PEnv :: *Envtype &
                        [Env ! Env != <<label,*P2,arg2,1>>],1>>
               PREP != <<threads,PREP,*T2 ::
                         [TS ! TS != <<t-need,*PREP,*X,1>>],1>>
               ],
      *NP :  [NP ! NP != <<cat,NP,NounPhrase,1>>
                   NP != <<sem,NP,*NPSem,1>>
                   NP != <<threads,NP,*T3 ::
                           [TS ! TS != <<t-need,*NP,*Y,1>>
                                 TS != <<t-feed,*NP,*Z,1>>
                                 TS != <<t-out,*NP,*OUT,1>>],1>>].
;;;
;;;  Discourse -> S
;;;
   *DS : [DS ! DS != <<cat,DS,Discourse,1>>
               DS != <<daughter,DS,*S,1>>
               DS != <<threads,DS,*ThreadDS ::
                       [TS ! TS != <<t-need,*DS,*X,1>>
                             TS != <<t-in,*DS,*S,1>>
                             TS != <<t-out,*DS,*OUT,1>>],1>>]
   ->
     *S : [S ! S != <<cat,S,sentence,1>>
               S != <<threads,S,*ThreadS ::
                      [TS ! TS != <<t-need,*S,*X,1>>
                            TS != <<t-out,*S,*OUT,1>>],1>>].
;;;
```

```
;;;  Discourse -> Discourse S
;;;
   *DS1 : [DS ! DS != <<cat,DS,Discourse,1>>
                DS != <<daughter,DS,*DS,1>>
                DS != <<daughter,DS,*S,1>>
                DS != <<threads,DS,*ThreadDS1 ::
                        [TS ! TS != <<t-need,*DS1,*X,1>>
                              TS != <<t-in,*Y,*OUT,1>>
                              TS != <<t-in,*DS1,*S,1>>
                              TS != <<t-out,*DS1,*OUT1,1>>],1>>]
   ->
     *DS : [DS ! DS != <<cat,DS,Discourse,1>>
                 DS != <<threads,DS,*ThreadDS ::
                         [TS ! TS != <<t-need,*DS,*X,1>>
                               TS != <<t-out,*DS,*OUT,1>>],1>>],
     *S : [S ! S != <<cat,S,sentence,1>>
               S != <<threads,S,*ThreadS ::
                      [TS ! TS != <<t-need,*S,*Y,1>>
                            TS != <<t-out,*S,*OUT1,1>>],1>>].
;;;
;;;  Fulldiscourse -> ds Discourse de
;;;
   [DS ! DS != <<cat,DS,fulldiscourse,1>>
         DS != <<part-of-discourse,DS,DS,1>>
         DS != <<daughter,DS,*DS,1>>
         DS != <<daughter,DS,*D,1>>
         DS != <<threads,DS,
                 *ThreadDS ::
                 [TS ! TS != <<part-of-discourse,TS,DS,1>>
                       TS != <<t-in,*X,*D,1>>
                       TS != <<t-out,DS,*Y,1>>],1>>]
  ->
    *D : [S ! S != <<cat,S,DisStart,1>>],
    *DS : [DS ! DS != <<cat,DS,discourse,1>>
                DS != <<threads,DS,*ThreadDS ::
                        [TS ! TS != <<t-need,*DS,*X,1>>
                              TS != <<t-out,*DS,*Y,1>>]
                        ,1>>],
    [S ! S != <<cat,S,DisEnd,1>>].

Lexical Entries
;;;
;;;  Discourse start and end markers
;;;
ds -
   [DS ! DS != <<cat,DS,DisStart,1>>
```

```
         DS != <<AssignOut,DS,[T !],1>>]
de -
   [DS ! DS != <<cat,DS,DisEnd,1>>]
;;;
;;;  Proper nouns
;;;
Hanako -
   [PN ! PN != <<cat,PN,ProperNoun,1>>
         PN != <<use_of,PN,"Hanako",1>>
         PN != <<type,PN,female,1>>
         PN != <<sem,PN,h,1>>]
Anna -
   [PN ! PN != <<cat,PN,ProperNoun,1>>
         PN != <<use_of,PN,"Anna",1>>
         PN != <<type,PN,female,1>>
         PN != <<sem,PN,a,1>>]
Taro -
   [PN ! PN != <<cat,PN,ProperNoun,1>>
         PN != <<use_of,PN,"Taro",1>>
         PN != <<type,PN,male,1>>
         PN != <<sem,PN,t,1>>]
;;;
;;;  Pronouns
;;;
he -
   [PN ! PN != <<cat,PN,ProNoun,1>>
         PN != <<use_of,PN,"he",1>>
         PN != <<type,PN,male,1>>
         PN != <<sem,PN,PN1,1>>]
she -
   [PN ! PN != <<cat,PN,ProNoun,1>>
         PN != <<use_of,PN,"she",1>>
         PN != <<type,PN,female,1>>
         PN != <<sem,PN,PN2,1>>]
her -
   [PN ! PN != <<cat,PN,ProNoun,1>>
         PN != <<use_of,PN,"her",1>>
         PN != <<type,PN,female,1>>
         PN != <<sem,PN,PN2,1>>]
it -
   [PN ! PN != <<cat,PN,ProNoun,1>>
         PN != <<use_of,PN,"it",1>>
         PN != <<type,PN,neuter,1>>
         PN != <<sem,PN,PN3,1>>]
;;;
;;;  Common nouns
```

```
;;;
man -
   [N ! N != <<cat,N,noun,1>>
        N != <<use_of,N,"man",1>>
        N != <<type,N,male,1>>
        N != <<sem,N,<<R1,MA1,1>>,1>>
        N != <<env,N,ManEnv,1>>]
donkey -
   [N ! N != <<cat,N,noun,1>>
        N != <<use_of,N,"donkey",1>>
        N != <<type,N,neuter,1>>
        N != <<sem,N,<<R1,DA1,1>>,1>>
        N != <<env,N,DonkeyEnv,1>>]
hat -
   [N ! N != <<cat,N,noun,1>>
        N != <<use_of,N,"hat",1>>
        N != <<type,N,neuter,1>>
        N != <<sem,N,<<R1,HA1,1>>,1>>
        N != <<env,N,HatEnv,1>>]
;;;
;;;  Quantifiers
;;;
a -
   [D ! D != <<cat,D,Determiner,1>>
        D != <<use_of,D,"a",1>>
        D != <<sem,D,<<Q1,E1,A2,A3,1>>,1>>
        D != <<ind,D,I1,1>>
        D != <<env,D,AEnv,1>>]
every -
   [D ! D != <<cat,D,Determiner,1>>
        D != <<use_of,D,"every",1>>
        D != <<sem,D,<<Q1,A1,A2,A3,1>>,1>>
        D != <<ind,D,I2,1>>
        D != <<env,D,EveryEnv,1>>]
;;;
;;;  Verbs
;;;
sings -
   [VP ! VP != <<cat,VP,Verb,1>>
         VP != <<use_of,VP,"sings",1>>
         VP != <<sem,VP,<<R1,A1,1>>,1>>
         VP != <<env,VP,SingEnv,1>>]
smiles -
   [VP ! VP != <<cat,VP,Verb,1>>
         VP != <<use_of,VP,"smiles",1>>
         VP != <<sem,VP,<<R1,A1,1>>,1>>
```

```
         VP != <<env,VP,SmileEnv,1>>]
walks -
   [VP ! VP != <<cat,VP,Verb,1>>
         VP != <<use_of,VP,"walks",1>>
         VP != <<sem,VP, <<R1,WA1,1>>,1>>
         VP != <<env,VP,WalkEnv,1>>]
talks -
   [VP ! VP != <<cat,VP,Verb,1>>
         VP != <<use_of,VP,"talks",1>>
         VP != <<sem,VP,<<R1,TA1,1>>,1>>
         VP != <<env,VP,TalkEnv,1>>]
likes -
   [V ! V != <<cat,V,Verb,1>>
        V != <<use_of,V,"likes",1>>
        V != <<sem,V,<<R1,A1,A2,1>>,1>>
        V != <<env,V,LikeEnv,1>>]
;;;
;;;  Prepositions
;;;
with -
   [PREP ! PREP != <<cat,PREP,Preposition,1>>
           PREP != <<use_of,PREP,"with",1>>
           PREP != <<sem,PREP,<<P1,A1,A2,1>>,1>>
           PREP != <<env,PREP,WithEnv,1>>]
```