Semantic Abstraction and Anaphora

Mark Johnson

Brown University

Martin Kay

Xerox Palo Alto Research Center and Stanford University

Abstract

This paper describes a way of expressing syntactic rules that associate semantic formulae with strings, but in a manner that is independent of the syntactic details of these formulae. In particular we show how the same rules construct predicate argument formulae in the style of Montague grammar[13], representations reminiscent of situation semantics(Barwise and Perry [2]) and of the event logic of Davidson [5], or representations inspired by the discourse representations proposed by Kamp [9]. The idea is that semantic representations are specified indirectly using *semantic construction operators*, which enforce an abstraction barrier between the grammar and the semantic representations themselves. First we present a simple grammar which is compatible with the three different sets of constructors for the three formalisms. We then extend the grammar to provide one treatment that accounts for quantifier raising in the three different semantic formalisms

Introduction

Grammars specifying the relationship between strings and semantic representations often have details of these representations embedded in them. We show how grammar rules can be written in a form which, by abstracting away from details of the semantic representation, acquires greater modularity and hence theoretical perspicuity and practical robustness. In particular, we believe that the approach helps clarify the relationship between apparently disparate theories of semantic representation.[1]. The basis of our proposal is that each grammatical rule should contain, or be paired with, an expression written in terms of semantic construction operators. Different operations can be associated with these operators and, depending on the set in force at a given time, the effect of interpreting the expression will be to construct a representation in one semantic formalism or another. The set of operators contains members corresponding to such notions as *composition*, *conjunction*, etc. The set is small and independent of the semantic formalism. The operations are associated with the operators independently of the grammar and they determine the form of the semantic representation.

We present three different sets of semantic constructors here, which we have dubbed the *predicate-logic*, the *sets-of-infons* and the *discourse-representation* constructors. We begin by introducing the constructors used in this paper: no claims are made for their general sufficiency. Not all of the constructors are relevant to all semantic theories and those not needed for a particular one are given degenerate definitions. The simplest kind of construction operator is the identity function which maps every input i onto just one output, namely i.

The operators are the following:

[1] The kind of separation between the grammar and the details of the semantic representation proposed here also appears in the examples of Pereira and Shieber[12] and in Lexical-Functional Grammar (see [6]). Our use of different sets of semantic constructors with a single grammar is novel, as far as we are aware.

$external(S, EF)$ relates a semantic representation S and an *external form* EF, e.g. a representation that constitutes the parser's output. The internal and external forms are distinguished because the (internal) representation S will, in general, contain information that plays a role in the process of analyzing a sentence (e.g. for anaphora tracking) but that is not part of the logical form (EF) of the sentence as a whole.

$atom(S, Prop)$ specifies that the content of the (internal) semantic representation, S, is the atomic proposition $Prop$. This is used to construct the semantic values for lexical entries, for example.

$conjoin(S1, S2, S12)$ relates three semantic representations. It specifies that the content of $S12$ is constructed by conjoining $S1$ and $S2$. This operator occurs crucially in the semantics of indefinite determiners.

$new_index(S, I)$ specifies that the content of S is I, a *referential index* for a non-anaphoric NP. The form of a referential index is defined by the particular semantic theory.

$accessible_index(S, I)$ specifies that the content of S is a referential index I of some noun phrase that is a potential antecedent of an anaphor. Constraints on accessible indices are defined by the particular semantic theory.

While the primitives discussed in this paper have relatively simple definitions, in other more elaborate theories they may involve non-trivial computation. For example, the *compose* primitive might impose certain discourse-consistency requirements arising from a more restrictive theory of discourse structure than those described here.

A key insight of the Discourse Representation and Situation Semantics accounts, but originating with Karttunen[10], is that anaphoric and quantificational domains coincide. Thus, in (1), *it* can be co-indexed with *a donkey* only if *a donkey* is interpreted as having wide-scope.

(1) Every man kicked a donkey. It developed blue bumps.

The relationship between these sentences is one of (semantic) *precedence*, and we call the operator that relates the corresponding semantic representations *compose*:

$compose(S1, S2, S12)$ specifies that the information in the representation $S12$ is the information in $S1$ followed by the information in $S2$. Compose defines an ordering of semantic operations that particular semantic theories may or may not be sensitive to. (In this paper, the Montague constructors are not sensitive to this ordering, while the other two types of semantic representations are).

When *a donkey* is interpreted as having narrow scope with respect to *every man* in (1), the reference marker introduced by *a donkey* is located in a context subordinate to the sentence as a whole, and hence not accessible to anaphors in the following discourse. To provide for this, we introduce the following operator:

$subordinate(S, SubName, Sub)$ specifies that S contains an anaphorically and quantificationally subordinate representation Sub, which has the "name" $SubName$. The $SubName$ would be distinguished from Sub in non-extensional theories of meaning, where a meaning is distinguished from its propositional content (say), as in the sets-of-infons representation described below.

We turn now to the grammar without quantifier-raising. We formulate both the grammar and the semantic constructors in pure Prolog (exploiting the syntactic sugar of *Definite Clause Grammar* (Pereira and Shieber [12], pp. 70–79)) because it is expressive enough for our purposes and is widely used in work of this kind(see, *inter alia* Colmerauer [3], Abramson and Dahl [1] and [11]).

A Grammar using Semantic Constructors

The grammar generates simple transitive clauses and subject-relative clauses that do not

involve long-distance dependencies. It is based on the Montague-style grammars presented in Chapter 4 of Pereira and Shieber[12], and the treatments of agreement, Wh-dependencies, etc., presented there could also be incorporated without difficulty.

```
/**********************************
* Operators:                      *
*   ^    for lambda abstracts,    *
*   ==>  for implication.         *
**********************************/
:- op(950, xfy, ^).

:- op(300, xfx, ==>).

parse(String,ExtSem) :-
  external(IntSem,ExtSem),
  s(IntSem,String,[]).

/**********************************
* The grammar                     *
**********************************/

s(S) --> np(VP^S), vp(VP).
np(NP) --> det(N1^NP), n1(N1).
n1(N) --> n(N).
n1(X^S) --> n(X^S1), rc(X^S2),
  {conjoin(S1,S2,S)}.
vp(X^S) --> v(X^VP), np(VP^S).
rc(VP) --> [that], vp(VP).

v(X^Y^S) --> [Verb],
  {verb(Verb,X^Y^Pred),
   atom(Pred,S)}.
n(X^S) --> [Noun],
 {noun(Noun,X^Pred),
  new_index(X,S1),
  atom(Pred,S2),
  compose(S1,S2,S)}.
det((X^Res)^(X^Scope)^S) --> [Det],
 {determiner(Det, Res^Scope^S)}.
np((X^S1)^S) --> [Pronoun],
 {pronoun(Pronoun),
  accessible_index(X,S2),
  compose(S1,S2,S)}.

/**********************************
* The lexicon                     *
**********************************/

pronoun(he).
pronoun(she).
pronoun(him).
pronoun(her).
pronoun(it).

verb(likes, X^Y^likes(X,Y)).
verb(saw, X^Y^see(X,Y)).
verb(beats, X^Y^beat(X,Y)).
verb(owns, X^Y^own(X,Y).

noun(woman, X^woman(X)).
noun(man, X^man(X)).
noun(donkey, X^donkey(X)).

determiner(a, Res^Scope^S ):-
  conjoin(Res,Scope,S).
determiner(every,
          Res0^Scope^S ):-
  compose(S1, S2, S),
  subordinate(Res, ResName, S1),
  compose(Res0, Res1, Res),
  subordinate(Scope, ScopeName, Res1),
  atom(ResName ==> ScopeName, S2).
```

Most of the grammar should be familiar, even if it is somewhat more pedantically expressed than is usual. Following Pereira and Shieber (who were in turn inspired by Montague), VP and N meanings are represented by terms of the form `X^S`, where `X` represents a referential index and `S` represents an S meaning. NP meanings are represented by terms of the form `VP^S` (or equivalently, `(X^S0)^S`), where `VP` represents a VP meaning, `X` a referential index, and `S` and `S0` represent S meanings.

All manipulation of semantic values is performed by constructor primitives, rather than by explicit construction of terms. For example, the `N1` production that introduces relative-clauses invokes `conjoin` explicitly to conjoin the semantic values of the `N` and the relative clause to yield the semantic value of the `N1`. The sharing of the referential index `X` between the N and the VP is performed in the grammar alone, since it is a syntactic rather than semantic property of the construction.

The semantic component of the production that introduces lexical nouns has two parts. `S0` represents the atomic predicate `Pred` associated with the lexical meaning of the noun. `S2` represents the fact that `X` is a (possibly new) referential index. The component `S` of the semantic value associated with the noun contains all of the information in `S0` and `S2`.

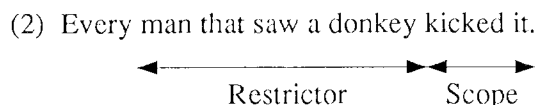
The production introducing (lexical) pronouns requires that the referential index `X` of the pronoun be accessible in `S0`, and specifies that the `S` component is the composition of `S0` and the `S0` component of the VP meaning. (Recall that the semantic representations of pronouns, like all NP's, are terms of the

form `VP^S`, so the `S0` is a component of the meaning of the VP or V phrase that this pronoun is an argument of).

Undoubtedly the most complex component of the grammar is the lexical entry for *every*. Because the structure of the lexical entries for all anaphoric scope-inducing quantifiers will be similar to the entry for *every*, we explain it in some detail.

The quantification induced by the determiner *every* is described in terms of the determiner's *restriction*, which defines the entities that the quantification ranges over, and its *scope*, the component of the expression quantified over. (2) indicates the components of the utterance corresponding to the restriction and the scope of the quantifier *every* in the absence of quantifier-raising.

(2) Every man that saw a donkey kicked it.



The grammars presented here identify the restrictor and the scope of a determiner in the syntax; e.g. quantifier-raising arises from the grammar permitting multiple assignments of components of the utterance to the restrictions and scopes of the determiners of that utterance.

The semantic value associated with lexical entry for a determiner in the grammars presented here is a term of the form *Res^Scope^Sentence*, where *Res* is the semantic value associated with the restrictor and *Scope* is the semantic value associated with the scope. A grammar directly constructing predicate-logic style semantic representations would assign the lexical entry in (3) to the determiner *every*, where '==>' is interpreted as the implication operator in semantic representations (see Pereira and Shieber [12]).

(3) *determiner(every,*
Res^Scope^(Res==>Scope)).

This lexical entry does not suffice for our purposes, since it provides no information about the relative anaphoric scope relationships between the restrictor, the scope, and that portion of the utterance external to the quantificational expression as a whole.

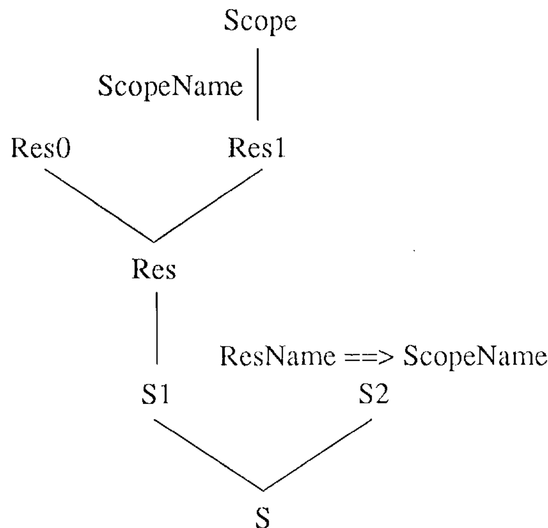
Anaphors in opaque quantificational expressions can refer to entities superordinate to the quantificational expression, but in general anaphors outside of an opaque quantificational expression cannot refer to entities introduced in either the restriction or scope of the quantificational expression[2]. Anaphors in the scope of an opaque quantificational expression can refer to entities introduced in the restriction of that expression (e.g. as in (3) above), but anaphors in the restriction cannot refer to entities introduced in the scope.

The `compose` and `subordinate` predicates in the lexical entry for *every* in the grammar presented above express subordination relationships that describe the behavior of opaque determiners. The semantic representation `S` is the composition of `S1` and `S2`, where `S2` is the semantic atom `ResName ==> ScopeName`. `Res` is subordinate to `S1`, and is itself the composition of `Res0` and `Res1`, where `Res0` is the semantic representation of the restrictor. `Scope` is subordinate to `Res1`, and is the semantic representation of the scope. The diagram on the following page sketches the relationship between the various semantic entities mentioned in the lexical entry for *every*. Subordination relationships are depicted by vertical lines (the name of the subordinate space is written alongside the line), and composition relationships are indicated by V-shaped diagonals.

The *Predicate-Logic* Constructors

These constructors build a *predicate-logic* type of semantic representation in a fairly transparent fashion. Pronouns are treated as free variables, there are no constraints on their distribution, and anaphoric binding is not

[2] There are exceptions to this: for example, anaphors can refer to proper names introduced in the restrictor or scope of opaque determiners. Within the framework described below, this can be treated by adding a new semantic construction operator `add_top_level`, which adds a referential index to the most superordinate level



treated. Thus the definitions of the constructors `new_index`, `accessible_index`, `compose` and `subordinate` have degenerate definitions.

A property is identical with the term representing it:

```
atom(Prop, Prop).
```

The conjunction of `P` and `Q` is represented by the term `P&Q`.

```
conjoin(P,P,P&Q).
```

There are no constraints on new indices.

```
new_index(_,_).
```

There are no constraints on accessible indices.

```
accessible_index(_,_).
```

Sequencing is unimportant.

```
compose(P,P,P).
```

A Subordinate space can be introduced freely.

```
subordinate(_,Sub,Sub).
```

Internal and external forms are identical.

```
external(P,P).
```

The grammar described above and the predicate-logic constructors yield analyses such as the following:

```
?- p([a,man,owns,a,donkey],S).
S = man(X)&donkey(Y)&own(X,Y)

?- p([every,man,that,owns,a,donkey,
      beats,it],S).
S = (man(X)&donkey(Y)&own(X,Y))
    ==>beat(X,Z)
```

Roughly this latter form might be interpreted as: if X is a man and Y is a donkey and X owns Y, then there is a Z such that X beats Z.

The *Sets-of-Infons* Constructors

The constructors for the *sets-of-infons* and the *discourse-representation* both constrain anaphora by requiring that the referential indices provided by the `accessible_index` constructor be indices that were introduced by `new_index` in some earlier representation (where precedence is defined by the `compose` constructor). This entails that the internal form of these semantic representations encode information about preceding representations. Both constructors thread this information using the difference-list technique described in [8].

The primitive element of the *sets-of-infons* representation is inspired by the *infons* of Situation Semantics [2]. We represent an infon as a term of the form `Sit:P`, which means that `P` is true in the situation `Sit`. For example, Kim's sleeping in situation `s0` is represented by `s0:sleep(kim)`. For simplicity arbitrarily named constants (like the *gensyms* of Lisp) are used as the names of situations in this representation: this has the disadvantage that the definitions of the external and subordinate constructors are not declaratively specified.[3]

The internal form of a *sets of infons* representation has three components. We represent them in Prolog with a term of the form `@(Sits, InfonsIn, InfonsOut)`. The first is a stack whose top element is the situation currently being defined, and whose other elements are the situations superordinate to this one (as defined by the `subordinate` constructor). The second component is the set of all infons introduced in representations preceding this one. The infons in this list associated with the current or a superordinate situation provide the information needed for the `accessible_index` constructor. The third component of the representation is the set

[3] All that is required is that there is an infinite stock of situation names, so e.g. integers could have been used as situation names at the expense of a slight complication of the representation's data structures.

of infons introduced in preceding representations with the addition any infons added to the representation by the semantic representation constructor. In describing the term `@(Sits, InfonIn, InfonsOut)`, we use the names `InfonsIn` and `InfonsOut` to stress the fact that they constitute a difference list.

```
:- op( 900, xfx, : ).
atom(P, @([Sit|_],Is,[(Sit:P)|Is])).
compose(@(Ss,I0s,I1s),
        @(Ss,I1s,I2s),
        @(Ss,I0s,I2s)).
conjoin(I1, I2, I12) :-
  compose(I1, I2, I12).
subordinate(@([Sit|Sits],I0s,I1s),
            Sit, @(Sits,I0s,I1s)):-
  gensym(Sit).
new_index(Index,S) :-
  atom(i(Index),S).
accessible_index(Index,@(Ss,Is,Is)):-
  member(Sit:i(Index),Is),
  member(Sit,Ss).
external(@([Sit], [], Is), Sit:Is):-
  gensym(Sit).
```

The `atom` constructor introduces a new atomic proposition `P` as an infon `Sit:P`, where `Sit` is the situation currently being constructed. Notice that `InfonsOut` is the same as `InfonsIn` but for the addition of `(Sit:P)`.

The `compose` constructor threads the difference list of infons through both of the representations, so the composed representation contains all of the infons added to the sets of infons composed. The `conjoin` constructor is equivalent to the `compose` constructor.

The `subordinate` constructor introduces a new subordinate representation by pushing a new situation name `Sit` on to the list of (now superordinate) situations. The difference list of infons is threaded through the subordinate representation so that any infons added to it will appear in the superordinate representation as well.

The `new_index` constructor adds an atom of the form `i(Index)` to the representation S: no constraints are placed on `Index`.

The `accessible_index`[4] constructor is satisfied for a referential index `Index` if `Index` was introduced by `new_index` to a preceding non-subordinate representation, i.e. if the context contains an infon `Sit:i(Index)`, where `Sit` is the current or a superordinate situation name.

The `external(Internal, External)` predicate initializes `Internal` to have no superordinate situations and no preceding context, and returns the list of infons associated with this Internal representation as its external form.

When these constructors are used with the grammar defined above, the following analyses are obtained:

```
?- p([a,man,owns,a,donkey],S).
S = s0:[S0:own(X,Y),s0:i(Y),
    s0:donkey(Y),S0:i(X),s0:man(X)]
```

This can be paraphrased as: Situation $S0$ contains individuals X and Y; in $s0$ X is a man, Y is a donkey and X owns Y.

```
?- p([every,man,owns,a,donkey],S).
 S = s0:[S0:s1==>s2,S2:own(X,Y),
     s2:i(Y),s2:donkey(Y),
     s1:i(X),S1:man(X)]
```

This can be paraphrased as: Situation $s0$ contains the fact that all situations of type $s1$ are also situations of type $s2$. A situation is of type $s1$ if it contains individuals X and Y, and X is a man and Y is a donkey. A situation is of type $s2$ if X owns Y.

```
?- p([every,man,that,owns,a,donkey,
      beats,it],S).
S = s0:[s0:s1==>s2,s2:beat(X,Y),
    s1:own(X,Y),s1:i(Y),s1:donkey(Y),
    s1:i(X),s1:man(X)]
```

This can be paraphrased as: Situation $s0$ contains the fact that all situations of type $s1$ are also situations of type $s2$. A situation is of type $s1$ if it contains individuals X and Y, X

[4] The predicate `member` used here, and elsewhere in this paper, has its standard logical definition: viz: `member(X, [X | _]).` `member(X, [_, L]) :- member(X, L).` If this definition is used with the grammars and constructors given in this paper, the SLD selection rule of Prolog may lead to non-termination. It is in general necessary to delay the evaluation of the `member` predicate until its second argument is instantiated, which can be done using the `freeze` primitive of Prolog II.

is a man, Y is a donkey and X owns Y. A situation is of type $s2$ if X beats Y.[5]

The Discourse-Representation Constructors

The representations built by these constructors are inspired by the "box representations" of Kamp's (1981) Discourse Representation Theory [9]. A discourse representation "box" is represented by the list of items that constitute its contents. A representation is a difference-pair of the lists of the representations of the currently open boxes (i.e. the current box and all superordinate boxes), as in Johnson and Klein [8]. In Prolog, we use the binary '–' operator to separate the two members os the pair.

```
atom(P, [B|Bs]-[[P|B]|Bs]).
compose(B0s-B1s, B1s-B2s, B0s-B2s).
conjoin(P1, P2, P) :-
  compose(P1, P2, P).
subordinate([[]|B0s]-[B|B1s],
            B, B0s-B1s).
new_index(Index, C) :-
  atom(i(Index),C).
accessible_index(Index, Bs-Bs) :-
  member(B,Bs), member(i(Index),B).
external([[]]-[S], S).
```

The atom constructor introduces a new atomic proposition P. by adding it to the current box, i.e. the first element of the list of open boxes.

The compose constructor threads the difference list representing the open boxes through both compose representations of the items being composed in the same way that the compose constructor of the sets-of-infons representations does. The conjoin constructor is equivalent to the compose constructor.

The subordinate constructor introduces an empty subordinate box onto the list of currently open boxes. The "name" B of the subordinate box is the list of atoms it contains.

The new_index constructor adds an atom of the form i(Index) to the semantic representation: no constraints are placed on Index (as in the sets-of-infons representation).

The accessible_index constructor is satisfied by a referential index Index if Index is introduced by new_index in a preceding non-subordinate representation, i.e. if one of the superordinate boxes contains i(Index).

The external(Internal,External) predicate initializes Internal to have exactly one open box (empty), and returns the contents of that box as its external form.

With these constructors, the parser yields the following semantic values for the test sentences.

```
?- p([a,man,owns,a,donkey],S).
S = [own(X,Y),donkey(Y),i(Y),
    man(X),i(X)]
```

This representation is true just in case there are two individuals X and Y, X is a man and Y is a donkey, and X owns Y.

```
?- p([every,man,owns,a,donkey],S).
S = [[man(X),i(X)]==>[own(X,Y),
    donkey(Y),i(Y)]]
```

This representation is true just in case for all individuals X such that X is a man there is an individual Y such that Y is a donkey and X owns Y.

```
?- p([every,man,that,owns,a,donkey,
      beats,it],S).
S = [[own(X,Y),donkey(Y),i(Y),man(X),
    i(X)]==>[beat(X,Y)]]
```

This representation is true just in case for all individuals X and Y such that X is a man and Y a donkey and X owns Y, it is also true that X beats Y.

Extending the Grammar to handle Quantifier-Raising

In this section we sketch a syntactic account of quantifier-raising inspired by the implementation of Cooper-storage (Cooper [4])

[5] The grammar and the sets of infons constructors also generate an additional reading in which the man that owns the donkey beats himself; i.e. *it* is taken as anaphorically dependent on *every man*. Simple extensions to the grammar (e.g. requiring the index of a pronoun to differ from the index of all c-commanding NPs) or the semantics (e.g. requiring the gender of the pronoun to agree with its antecedent's gender) would rule out this spurious analysis.

presented in Pereira and Shieber [12], to which we refer the reader for details. Each syntactic constituent is associated with a list of quantifiers that are "in storage" (this corresponds in an LF-movement account of quantifier-scope to being raised out of this constituent). Quantificational determiners add items to the quantifier store, and at S nodes, quantifiers are removed from the store and applied to the semantic representation. The quantifier-store of nodes at which quantifiers are neither added nor removed is the shuffle of the quantifier-stores of its children.[6]. The grammar presented below is simply the grammar presented above with the addition of quantifier-storage. The lexical entries for this grammar are the same as the above, and so are not listed here.

```
q(String, Analysis) :-
  external(S, Analysis),
  s(S, [], String, []).
s(S,Qs) --> np(VP^S1,Qnp), vp(VP,Qvp),
  {shuffle(Qnp,Qvp,Q1s),
   apply_some(Q1s,S1,Qs,S) }.
np(NP,Qnp) --> det(N1^NP,Qdet),
               n1(N1,Qn1),
  {append(Qdet,Qn1,Qnp) }.
n1(N,Qn) --> n(N,Qn).
n1(X^S,Qn1) --> n(X^S1,Qn),
                rc(X^S2,Qrc),
  {conjoin(S1,S2,S),
   shuffle(Qn,Qrc,Qn1)}.
vp(X^S,Qvp) --> v(X^VP,Qv),
                np(VP^S,Qnp),
  {shuffle(Qv,Qnp,Qvp)}.
rc(X^S2,Qrc) --> [that], vp(X^S1,Qvp),
  {apply_some(Qvp,S1,Qrc,S2) }.
np((X^S1)^S,[]) --> [Pronoun],
  {pronoun(Pronoun),
   accessible_index(X,S2),
   compose(S1,S2,S) }.
v(X^Y^S,[]) --> [Verb],
  {verb(Verb,X^Y^Pred),
   atom(Pred,S)}.
n(X^S,[]) --> [Noun],
  {noun(Noun,X^Pred),
   compose(S1,S2,S),
   atom(Pred,S1),
   new_index(X,S2)}.
det((X^Res)^(X^Scope)^Scope,
    [Quant]) --> [Det],
  {determiner(Det, Res^Quant)}.
```

[6] Treating the quantifier-store as a syntactic feature can express many properties of LF-movement accounts, such as quantificational islandhood, etc., without the explicit construction of additional representations

The proposition `shuffle(L1, L2, L3)` is true just in case `L3` is a list that can be seen as having been constructed in a sequence of steps in each of which the next available item is taken from either `L1` or `L2` and added to the end. So long as items remain on both `L1` and `L2`, it is immaterial which of them supplies the next member of `L3`. What is important is that the members of `L1` and `L2` should all be on `L3`, and in their original order. This relationship is assured by the following Prolog clauses:

```
shuffle([],[],[]).
shuffle([Q|Q1s],Q2s,[Q|Q3s]) :-
  shuffle(Q1s,Q2s,Q3s).
shuffle(Q1s,[Q|Q2s],[Q|Q3s]) :-
  shuffle(Q1s,Q2s,Q3s).
```

The first clause asserts that the proposition is true of three empty lists, and serves to terminate the recursion implicit in the other two. The second clause says that, if `Q2s` and `Q3s` are suffixes of a pair of lists to be shuffled, and that shuffling them gives `Q1s`, then the item that precedes `Q1s` in the final result can come from the first list, that is, it can be the item preceding `Q1s`. The third clause says that, alternatively, the item preceding `Q1s` can come from the second list.

The grammar also makes use of the predicate `apply_some(Quants, OldSemanticValue, UnappliedQuants, NewSemanticValue)`

which is true if applying zero or more quantifiers from the beginning of the list *Quants* to a given *OldSemanticValue* yields *NewSemanticValue* and leaves a suffix of that list of quantifiers, namely *UnappliedQuants* still unapplied. It can be defined with the following pair of clauses, the first of which terminates the sequence of applications and the second of which applies the next quantifier in sequence.

```
apply_some(Qs,P,Qs,P).
apply_some([P^Qp|Qs],P,Q1s,P1):-
  apply_some(Qs,Qp,Q1s,P1).
```

The new grammar can be used with the three different semantic constructors presented above. Using the Predicate-Logic constructors, it yields results like the following:

```
?- q([a,man,owns,a,donkey],S).
S = donkey(Y)&man(X)&own(X,Y);
S = man(X)&donkey(Y)&own(X,Y)
```

This example has two (semantically-equivalent) representations corresponding to the two scope possibilities for the two existentially quantified NPs.

```
?- q([every,man,owns,a,donkey],S).
S = donkey(Y) & man(X)==>own(X,Y);
S = man(X)==>(donkey(Y)&own(X,Y))
```

In this example the two non-equivalent representations correspond to the two different scope possibilities for the quantified NPs. These readings paraphrase as: There is a donkey Y and for each man X, X owns Y and For each man X there is a donkey Y and X owns Y".

```
?- q([every,man,that,owns,
      a,donkey,beats,it],S).
S = donkey(Y)&(man(X)&own(X,Y))
    ==>beat(X,Z);
S = (man(X)&donkey(Y)&own(X,Y))
    ==>beat(X,Z);
```

In this example the two non-equivalent representations correspond to the two different scope possibilities for the quantified NPs. These readings paraphrase as: There is a donkey Y and for each man X such that X owns Y it is the case that X beats Y, and "For each man X and donkey Y such that X owns Y, it is the case that X beats Y.

Using the sets-of-infons constructors, we get the following results:

```
?- q([every,man,owns,a,donkey],S).
S = s0:[s0:s1==>s2,s2:own(X,Y),
    s1:i(X),s1:man(X),S0:i(Y),
    s0:donkey(Y)];
S = s0:[s0:s1==>s2,s2:own(X,Y),
    s2:i(Y),s2:donkey(Y),s1:i(X),
    s1:man(X)]
```

The scope possibilities are indicated here by the situation in which the noun phrases are interpreted. The first reading displayed corresponds to the quantifier-raised interpretation, which paraphrases as: Situation $s0$ contains the individual Y, the fact that Y is a donkey, and the fact that for all ways of making $s1$ true, $s2$ is also true, where $s1$ contains the individual X and the fact that X is a man, and $s2$ contains the fact that X owns Y. Since Y is in $s0$, under this reading it is a potential antecedent for anaphors in for following sentences.

The second reading differs from the first in that the NP *a donkey* is interpreted in the subordinate situation $s1$ instead of $S0$. As well as causing *a donkey* to be quantificationally subordinate to *every man*, this also makes *a donkey* unavailable as a potential antecedent for anaphors in following sentences.

We can therefore account for the fact that under normal intonation *a donkey* is interpreted as having wide scope over *every man* in the following discourse fragment (3).

(3) Every man saw a donkey. It had a bushy tail

We now consider one of the famous "donkey" sentences:

```
?- q([every,man,that,owns,a,donkey,
     beats,it],S).
S = s0:[s0:s1==>s2,s2:beat(X,Y),
     s1:own(X,Y),s1:i(X),s1:man(X),
     s0:i(Y),s0:donkey(Y)];
S = s0:[s0:s1==>s2,s2:beat(X,Y),
     s1:own(X,Y),s1:i(Y),s1:donkey(Y),
     s1:i(X),s1:man(X)]
```

The first reading displayed again corresponds to the quantifier-raised interpretation, which paraphrases as: Situation $s0$ contains an individual Y, and the facts that Y is a donkey and that every way of making $S1$ true also makes $S2$ true, where $S1$ contains the individual X and the facts that X is a man and X owns Y, and $S2$ contains the fact that X beats Y.

Finally, the discourse-representation constructors yield the following:

```
?- q([every,man,owns,a,donkey],S).
S = [[i(X),man(X)]==>[own(X,Y)],i(Y),
    donkey(Y)]  S = [[i(X),man(X)]
    ==>[own(X,Y),i(Y),donkey(Y)]]
```

These representations are direct notational variants of the two set-of-infons representations of this sentence given above. The truth conditions of the first reading correspond to the wide-scope interpretation of *a donkey*, and can be paraphrased as: There is a donkey Y, and for every man X, X owns Y.

```
?- q([every,man,that,owns,a,donkey,
      beats,it],S).
S = [[own(X,Y),i(X),man(X)]
    ==>[beat(X,Y)],i(Y),donkey(Y)];
S = [[own(X,Y),i(Y),donkey(Y),i(X),
    man(X)]==>[beat(X,Y)]]
```

Again, these representations are direct notational variants of the two sets-of-infons representations of this sentence given above. The truth conditions of the first reading correspond to the wide-scope interpretation of *a donkey*, and can be paraphrased as: There is a donkey Y, and for every man X such that X owns Y, X beats Y.

The same correlation between quantificational scope and anaphoric scope holds with these constructors, as expected.

Conclusion

We have worked out a scheme for computing the logical forms of sentences incrementally in the course of parsing them which we believe achieves an unprecedented level of abstraction of the semantic from the syntactic parts of the grammar. The very incrementality of the scheme might be used to argue against it. Given the prevalence of scope ambiguities, the interests of computational efficiency may be best served by a scheme that delays all semantic computation until the parsing is complete so as not to work unnecessarily on phrases that turn out not to be capable of incorporation in a complete analysis of the sentence. Hobbs and Shieber [7] adopt such a scheme apparently on the grounds of greater perspicuity. In any case, the modifications that need to be made to our scheme are entirely trivial, requiring only the introduction of a modest amount of symbolic computation. Basically, the idea is to use operations which, instead of returning pieces of the final logical form incrementally and nondeterministically, return expression that will exhibit this nondeterministic behavior when evaluated later. The later evaluation will, of course, be as specified be the definitions we have given. In short, we believe that the abstractions we have created effectively isolate the syntactic rules both from the corresponding semantic formalism and from the architecture of the system by which both of them will be interpreted.

Bibliography

[1] Abramson, H., and Dahl, V. Logic Grammars. Springer Verlag, New York, 1989.

[2] Barwise, J., and Perry, J. Situations and Attitudes. Bradford Books/MIT Press, Cambridge, Massachusetts, 1983.

[3] Colmerauer, A. An interesting subset of natural language. In Logic Programming, K. L. Clark and S.-A. Tarnlund, Eds. Academic Press, New York, 1982.

[4] Cooper, R. Quantification and Syntactic Theory, vol. 21 of Synthese Language Library. D. Reidel, Dordrecht, 1983.

[5] Davidson, D. The logical form of action sentences. In The Logic of Decision and Action, N. Rescher, Ed. University of Pittsburgh Press, Pittsburgh. Pennsylvania, 1967.

[6] Fenstad, J. E., et al. Situations, Language and Logic. Reidel, Dordrecht, 1987.

[7] Hobbs, J. R., and Shieber, S. M. An algorithm for generating quantifier scopings. Computational Linguistics 13, 1-2, 47–63.

[8] Johnson, M., and Klein, E. Discourse, anaphora and parsing. In Coling 88 (Bonn, West Germany, 1986).

[9] Kamp, H. A theory of truth and semantic representation. In Formal Methods in the Study of Language, J. A. G. Groenendijk, T. M. V. Janssen, and M. B. J. Stokhof, Eds., vol. 136. Mathematical Centre Tracts, Amsterdam, 1981, pp. 277–322.

[10]Karttunen, L. Discourse referents. In Syntax and Semantics, 7, J. McCawley, Ed. Academic Press, New York, 1976, pp. 363–385.

[11]McCord, M. C. Focalizers, the scoping problem, and semantic interpretation rules in logic grammars. In Logic Programming and Its Applications. Ablex, New Jersey, 1986.

[12]Pereira, F. C. N., and Shieber, S. M. Prolog and Natural Language Analysis, vol. 10 of C.S.L.I. Lecture Notes Series. Chicago University Press, Chicago, 1987.

[13]Thomason, R. Formal Philosophy. Selected Papers of Richard Montague. Yale University Press, New Haven, Connecticut, 1974.