16/4/82

Introduction to Unification

1. Term Unification

In the fragments of grammars that we have encountered so far, we have often found it necessary or desirable to add features to categories. We have been fairly ad hoc about doing this, including features or leaving them off as appropriate to the point under discussion. Now we must be more precise about what is going on in a grammar augmented with features.

We have already encountered a precise definition of how we expect a grammar with features to behave, when number agreement was discussed in the context of Definite Clause Grammars. In a rule of the form:

s --> np(Num),vp(Num).

we expect the rule to be used if and only if the same value is bound to both occurences of the variable Num. In particular, it does not matter whether the final value of Num originated in the np, the vp, or both, as long as it has only a single value. And of course, this rule will still be applicable if Num never gets a value, as in parsing a sentence like:

the sheep can eat the grass.

So what we intend by writing this rule is that the values of Num on subject and predicate unify. In DCGs, unification is not only the mechanism which binds features, but also what drives the parse. That is, a rule is determined to be applicable if its left-hand side unifies with a goal in the right-hand side of another rule that is applicable. In general, therefore, the objects that we are interested in unifying are **terms.** A term may be an atom, a variable, or a compound term, the latter being a functor and some fixed number of arguments each of which is a term. Intuitively, we want two terms to unify if they contain compatible information. More precisely, two terms unify if there exists an assignment of values to variables, called a **substitution,** that when applied to the two terms makes them identical. A substitution can be thought of as a function that applies to a term and returns a term with each of the variables replaced by its value as given in the substitution. So for instance the terms

p(X,f(Y,b),c)
p(g(Z,c),f(a,W),V)

may be made identical by the substitution:

{g(Z,c)/X, a/Y, b/W, c/V}

Here is a function that takes two terms represented in LISP list notation,

e.g. p(a,f(b,c)) ≡ (p a (f b c))

and returns such a substitution if there is one, or fail if there is not.

```
function unify(T1,T2) -> substitution

if T1 is a variable then
    if T1 occurs in T2 then
          return fail
```

16/4/87

```
            else return T2/T1
    else if T2 is a variable then
            if T2 occurs in T1 then
                return fail
            else return T1/T2
        else if T1 = T2 then
                return nil
    %       else

    begin
    H1 <- first(T1)
    H2 <- first(T2)
    T1 <- rest(T1)
    T2 <- rest(T2)
    S1 <- unify(T1,T2)
    if S1 = fail return fail
    G1 <- apply(S1,T1)
    G2 <- apply(S1,T2)
    S2 <- unify(G1,G2)
    if S2 = fail return fail
    return compose(S1,S2)
    end
```

What is going on here can be simply glossed as:

A variable unifies with any term it does not occur in.

An atom unifies only with an identical atom.

Compound terms unify if their functors are identical and their arguments unify pairwise, and the substitutions obtained as a result of each of these unifications are compatible.

The Prolog interpreter embodies an algorithm very similar to this one, except that the check for occurence of a variable in the term that substitutes for it is not made, as a concession to efficiency. To see the import of this, try typing X = f(X) to the Prolog interpreter.

2. Graph Unification

So far, we have used features to describe the following phenomena:

number: {sing,plur}
subcat: {1,2,3,...} or {intrans,trans,ditrans,...}
vform: {fin,bse,inf,...}
gap: {gap,nogap}

As we attempt to extend the coverage of our grammar, we will soon come up against the limitations of term unification. In a term, the number of arguments that a functor may have (its **arity**) is fixed, and each argument is identified solely by its position within the term. This encoding of features gives rise to the following problems. First, we must remember the correspondance between positions and features. Secondly, if we want to add a new feature, we must change our grammar at every point that feature is relevant. Finally, if we want to refer to the value of a feature in a term, we must specify the rest of

16/11/87

the elements in the term, perhaps by marking them as anonymous variables. To illustrate this, consider the handling of agreement. Suppose we had a term with functor 'agr', whose arguments were the values of various agreement features. Initially, we might just include number, so the term would have the form:

agr(Num)

If we then wanted to add agreement for person, we would have to change all instances of this term throughout our grammar to be of the form:

agr(Num,Per)

When we specify values for agreement features, e.g. on lexical entries, we must remember the order - the two terms

agr(sing,3)
agr(3,sing)

are not equivalent. And if we wished to specify a value only for person, not number, as in the lexical entry for 'fish', we must write:

agr(_,3)

not merely agr(3).

All these problems are solved simultaneously by the adoption of an alternative representation for features. In this representation, the name of a feature is made explicit. Once this step is taken, features may be referred to by name, so that position in a term is no longer the means by which a feature is located. Thus we would notate a specification for agreement features corresponding to

agr(sing,3)

as

$$\begin{bmatrix} \text{num:} & \text{sing} \\ \text{pers:} & 3 \end{bmatrix} \qquad (D_{3sg})$$

This is identical to the specification

$$\begin{bmatrix} \text{pers:} & 3 \\ \text{num:} & \text{sing} \end{bmatrix}$$

and if we wish to talk only about the feature person, then we can write

$$\begin{bmatrix} \text{pers:} & 3 \end{bmatrix}$$

without loss of compatability with other more fully specified feature specifications.

Representations of features in this form are called several different names in the linguistics literature. The name **feature structures** is self-explanatory. The name **functional structures** can be understood by considering these structures as partial functions from feature names to feature values. That is, the feature structure D_{3sg} above is a function that takes the feature name 'pers' to the value '3', the feature name 'num' to the value 'sing', and all other names to undefined. In set-theoretic terms, the set of name:value pairs comprising a feature structure may contain at most one pair with a given name, thus conforming to the definition of a function.

16/11/87

Finally, the name DAG for such structures refers to the possibility of drawing them as Directed Acyclic Graphs. Such a graph has nodes corresponding to feature structures, edges labelled with feature names, and leaves labelled with atomic feature values. These graphs would actually be trees, were it not for the fact that they can be reentrant. That is, exactly the same sub-structure can be reached by following different paths through the graph. In the notation we have met so far, this reentrancy is represented as follows:

$$\begin{bmatrix} f: & \boxed{1} & [h: & a] \\ g: & \boxed{1} & & \end{bmatrix}$$

Note that this a different structure from:

$$\begin{bmatrix} f: & [h: & a] \\ g: & [h: & a] \end{bmatrix}$$

since if we unify them with

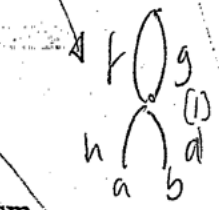
$$[f: \quad [d: \ b]]$$

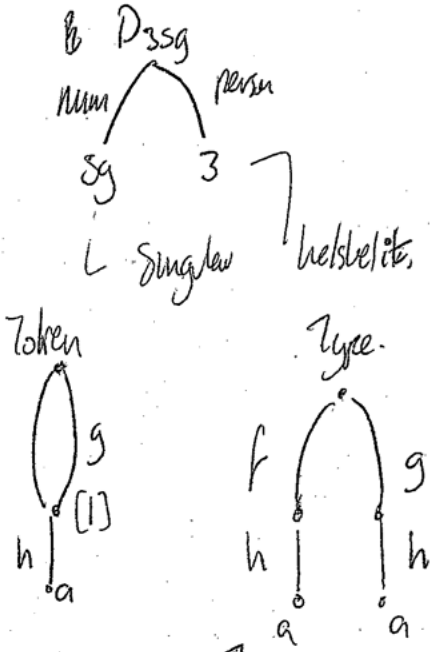
the first will now be

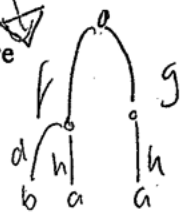
$$\begin{bmatrix} f: & 1 \begin{bmatrix} h: & a \\ d: & b \end{bmatrix} \\ g: & 1 \end{bmatrix}$$

and the second will be

$$\begin{bmatrix} f: & \begin{bmatrix} h: & a \\ d: & b \end{bmatrix} \\ g: & [h: \quad a] \end{bmatrix}$$







3. The PATR-II formalism

First note that once we have this general view of features, we can represent entire categories as feature structures. The context-free rule

s -> np vp

can be considered as a shorthand for a rule plus feature specifications of the form:

X_0 -> X_1 X_2
<X_0 cat> = s
<X_1 cat> = np
<X_2 cat> = vp

This rule is written in the notation of a grammar formalism called PATR-II. The feature specifications enclosed in angle brackets are called path equations. The rule

s -> np(Agr) vp(Agr)

translates into:

X_0 -> X_1 X_2
<X_0 cat> = s
<X_1 cat> = np
<X_2 cat> = vp
<X_1 agr> = <X_2 agr>

number agreement in PATR-II