DATR: A Language for Lexical Knowledge Representation[*]

Roger Evans[†]
University of Brighton

Gerald Gazdar[‡]
University of Sussex

Much recent research on the design of natural language lexicons has made use of non-monotonic inheritance networks as originally developed for general knowledge representation purposes in Artificial Intelligence. DATR *is a simple, spartan language for defining*

Path	Value
`syn cat`	`verb`
`syn type`	`main`
`syn form`	`present participle`
`mor form`	`love ing`

Table 1
Path/value pairs for present participle of *love*

elements.

This problem is overcome in DATR in the following way: such exhaustively listed path/value statements are indeed present in a description, but typically only **implicitly** present. Their presence is a logical consequence of a second set of statements, which have the concise, generalisation-capturing properties we expect. To make the distinction sharp, we call the first type of statement **extensional** and the second type **definitional**. Syntactically, the distinction is made with the equality operator: for extensional statements (as above), we use =, while for definitional statements we use ==. And, although our first example of DATR consisted entirely of extensional statements, almost all the remaining examples will be definitional. The semantics of the DATR language binds the two together in a declarative fashion, allowing us to concentrate on concise definitions of the network structure from which the extensional "results" can be read off.

Our first step towards a more concise account of Word1 and Word2 is simply to change the extensional statements to definitional ones:

```
Word1:
    <syn cat>  == verb
    <syn type> == main
    <syn form> == present participle
    <mor form> == love ing.
Word2:
    <syn cat>  == verb
    <syn type> == main
    <syn form> == passive participle
    <mor form> == love ed.
```

This is possible because DATR respects the unsurprising condition that if at some node a value is specifically **defined** for a path with a definitional statement, then the corresponding extensional statement also holds. So the statements we previously made concerning Word1 and Word2 remain true, but now only **implicitly** true.

Although this change does not itself make the description more concise, it allows us to introduce other ways of describing values in definitional statements, in addition to simply specifying them. Such value **descriptors** will include inheritance specifications which allow us to gather together the properties that Word1 and Word2 have solely by virtue of being verbs. We start by introducing a VERB node:

```
VERB:
    <syn cat>  == verb
    <syn type> == main.
```

and then redefine Word1 and Word2 to inherit their verb properties from it. A direct encoding for this is as follows:

```
Word1:
    <syn cat>  == VERB:<syn cat>
    <syn type> == VERB:<syn type>
    <syn form> == present participle
    <mor form> == love ing.
Word2:
    <syn cat>  == VERB:<syn cat>
    <syn type> == VERB:<syn type>
    <syn form> == passive participle
    <mor form> == love ed.
```

In these revised definitions the right hand side of the <syn cat> statement is not a direct value specification, but instead an inheritance descriptor. This is the simplest form of

DATR inheritance, it just specifies a new node and path from which to obtain the required value. It can be glossed roughly as "the value associated with `<syn cat>` at `Word1` is the same as the value associated with `<syn cat>` at `VERB`". Thus from `VERB:<syn cat> == verb` it now follows that `Word1:<syn cat> == verb`[6].

However, this modification to our analysis seems to make it less rather than more concise. It can be improved in two ways. The first is really just a syntactic trick: if the path on the right hand side is the same as the path on the statements concerning `Word1`. If we add these back in, the complete definition looks like this:

```
    Word1:
        <syn> == VERB
        <syn form> == present
```

`root>`. This descriptor is equivalent to `Word1:<mor root>` and, since `<mor root>` is not defined at `Word1`, the empty path definition applies, causing it to inherit from `Love:<mor root>`, and thereby return the expected value, `love`. Notice here that each element of a value can be defined entirely independently of the others; for `<mor form>` we now have an inheritance descriptor for the

This specifies inheritance of <`mor root`> from the query node, which in this case is `Word1`. The path <`mor root`> is not defined at `Word1` but inherits the value `love` from `Love`. Finally, the definition of <`mor form`> at `VERB` adds an explicit `ing`, resulting in a value of `love ing` for `Word1:`<`mor form`>. However, had we begun evaluation at, say, a daughter of the lexeme `Eat`, we would have been directed from `VERB:`<`mor form`> back to the original daughter of `Eat` to determine its <`mor root`>, which would be inherited from `Eat` itself. So we would have ended up with the value `eat ing`.

The analysis is now almost the way we would like it to be. However, by moving <`mor form`> from `Word1` to `VERB`, we have introduced a new problem: we have frozen in the present participle as the (default) value of <`mor form`> for all verbs. Clearly, if we want

```
        <mor form> == "<mor "<syn form>">".
```

This statement employs a DATR construct, the **evaluable path**, which we have not encountered before. The right hand side consists of a (global) path specification, one of whose component attributes is itself a descriptor, to be evaluated before the outer path can be. The effect of the above statement is to say that <`mor form`> globally inherits from the path given by the atom `mor` followed by the global value of <`syn form`>. For `Word1`, <`syn form`> is `present participle`, so <`mor form`> inherits from <`mor present participle`>. But for `Word2`, <`mor form`> inherits from <`mor passive participle`>. Effectively, the <`syn form`> is being used as a parameter to control which specific form should be considered **the** morphological form. Evaluable paths may themselves be global (as in our example) or local and their evaluable components may also involve global or local reference.

Our analysis now looks like this:

```
    VERB:
        <syn cat> == verb
        <syn type> == main
        <mor form> == "<mor "<syn form>">"
        <mor past> == "<mor root>" ed
        <mor passive> == "<mor past>"
        <mor present> == "<mor root>"
        <mor present participle> == "<mor root>" ing
        <mor present tense sing three> == "<mor root>" s.
    Love:
        <> == VERB
        <mor root> == love.
    Word1:
        <> == Love
        <syn form> == present participle.
    Word2:
        <> == Love
        <syn form> == passive participle.
```

The entire analysis is somewhat larger than the original, but it encodes all the past and present tense forms as well as all three participial forms. More importantly, almost all the information is in the `VERB` node and is common to many verb lexemes[8]. Indeed, the other nodes are as small as they reasonably could be: `Love` simply states that it is a verb with morphological root `love` and `Word1` simply states that it is a present participle instance of `Love`.

Of course, `Love` is a completely regular verb. But DATRr0ve v.75(Eff5(man)-)]TJ-R4090.06.12Td(global)Tj30-

_

_

```
        <mor present tense plur> == are
        <mor past tense sing one> == <mor past tense sing three>
        <mor past tense sing three> == was
        <mor past tense plur> == were.
```

In this section we have moved from simple attribute/value listings to a compact, generalisation-capturing representation for a fragment of English verbal morphology. In so doing, we have seen examples of most of the important ingredients of DATR: local and global descriptors, definition by default, and evaluable paths.

3. The

are defined recursively, and come in seven kinds. The simplest descriptor is just an atom or variable:

```
atom1
$var1
```

Then there are three kinds of **local inheritance descriptor**: a node, an (evaluable) path, and a node/path pair. Nodes are primitive tokens, paths are descriptor sequences (defined below) enclosed in angle brackets and node/path pairs consist of a node and a path separated by a colon:

```
Node1
<desc1 desc2 desc3 ...>
Node1:<desc1 desc2 desc3 ...>
```

Finally there are three kinds of **global inheritance descriptor**, which are quoted variants of the three local types just described:

```
"Node1"
"<desc1 desc2 desc3 ...>"
"Node1:<desc1 desc2 desc3 ...>"
```

A descriptor sequence is a (possibly empty) sequence of descriptors. The recursive definition of evaluable paths in terms of descriptor sequences allows arbitrarily complex expressions to be constructed, such as[15]:

```
"Node1:<"<atom1>" Node2:<atom2>>"
"<"<<Node1:<atom1 atom2> atom3>" Node2 "<atom4 atom5>" <> >">"
```

But the value sequences determined by such definitions are **flat**: they have no structure beyond the simple sequence and in particular do not reflect the structure of the descriptors that define them.

We shall sometimes refer to descriptor sequences containing only atoms as **simple** values, and similarly (unquoted) path expressions containing only atoms as simple paths.

3.1.3 Sentences. DATR sentences represent the statements which make up a description. As we have already seen, there are two basic statement types, extensional and definitional, and these correspond directly to simple extensional and definitional sentences, which are made up from the components introduced in the preceding section.

Simple extensional sentences take the form

```
Node:Path = Ext
```

where `Node` is a node, `Path` is a simple path, and `Ext` is a simple value. Extensional sentences derivable from the examples given in Section 2 include:

```
Do:<mor past participle> = done.
Mow:<mor past tense sing one> = mow ed.
Love:<mor present tense sing three> = love s.
```

Simple definitional sentences take the form

```
Node:Path == Def.
```

where `Node` and `Path` are as above and `Def` is an arbitrary descriptor sequence. Definitional sentences already seen in Section 2 include:

```
Do:<mor past> == did.
VERB:<mor form> == "<mor "<syn form>">".
EN_VERB:<mor past participle> == "<mor root>" en.
```

15 A descriptor containing an evaluable path may include nested descriptors which are either local or global. Our use of the local/global terminology always refers to the outermost descriptor of an expression.

Each of these sentences corresponds directly to a DATR statement. However we extend the notion of a sentence to include an abbreviatory

There is a natural procedural interpretation of this kind of inheritance, in which the value associated with the definitional expression is determined by "following" the inheritance specification and looking for the value at the new site. So given a DATR description (i.e., a set of definitional statements) and an initial node/path query, we look for the node and path as the left hand side of a definitional statement. If the definitional statement for this pair provides a local descriptor, then we follow it, by changing one or both of the node or path, and then repeat the process with the resulting node/path pair. We continue until some node/path pair specifies an explicit value. In the case of multiple expressions on the right hand side of a statement, we pursue each of them entirely independently of the others. This operation is local in the sense that each step is carried out without reference to any context wider than the immediate definitional statement at hand.

Declaratively speaking, local descriptors simply express equality constraints between definitional values for node/path pairs. The statement:

```
    Node1:Path1 == Node2:Path2.
```

can be read approximately as "if the value for `Node2:Path2` is defined, then the value of `Node1:Path1` is defined and equal to it". There are several points to notice here. First, if `Node2:Path2` is **not** defined, then `Node1:Path1` is unconstrained, so this is a weak directional equality constraint. However, in practice this has no useful consequences, due to interactions with the default mechanism – see Section 5.1 below. Second, "defined" here means "defined by a definitional statement", that is a "`==`" statement: local inheritance operates entirely with definitional statements, implicitly introducing new ones for `Node1:Path1` on the basis of those defined for `Node2:Path2`. Finally, as we shall discuss

For example, when a global path is specified, it effectively "returns control" to the current global node (often the original query node) but with the newly given path. Thus in Section 2, above, we saw that the node `VERB` defines the default morphology of present forms using global inheritance from the path for the morphological root:

```
    VERB:<mor present> == "<mor root>".
```

The node from which inheritance occurs is that stored in the global context. So a query of `Love:<mor present>` will result in inheritance from `Love:<mor root>` (via `VERB:<mor present>`), while a query of `Do:<mor present>` will inherit from `Do:<mor root>`.

Similarly, a quoted **node** form accesses the globally stored **path** value, as in the following example:

```
    Declension1:
        <vocative> == -a
        <accusative> == -am.
    Declension2:
        <vocative> == "Declension1"
        <accusative> == -um.
    Declension3:
        <vocative> == -e
        <accusative> == Declension2:<vocative>.
```

Here, the value of `Declension3:<accusative>` inherits from `Declension2:<vocative>` and then from `Declension1:< accusative>`, using the global path (in this case the query path), rather than the local path (`<vocative>`) to fill out the specification. So the resulting value is `-am` and not `-a` as it would have been if the descriptor in `Declension2` had been local rather than global.

We observed above that when inheritance through a global descriptor occurs, the global context is altered to reflect the new node/path pair. Thus after `Love:<mor present>` has inherited through `"VERB:<mor root>"`, the global path will be `<mor root>` rather than `<mor present>`. When we consider quoted node/path pairs, it turns out that this is the only property that makes them useful. Since a quoted node/path pair completely respecifies both node and path, its immediate inheritance characteristics are the same as the unquoted node/path pair. However, because it also alters the global context, its effect on any **subsequent** global descriptors (in the evaluation of the same query) will be different:

```
    Declension1:
        <vocative> == "<nominative>"
        <nominative> == -a.
    Declension2:
        <vocative> == Declension1
        <nominative> == -u.
    Declension3:
        <nominative> == -i
        <accusative> == "Declension2:<vocative>".
```

In this example, the value of `Declension3:<accusative>` inherits from `Declension2:<vocative>` and then from `Declension1:<vocative>` and then from `Declension2:<nominative>` (because the global node has changed from `Declension3` to `Declension2`) giving a value of `-u` and not `-i` as it would have been if the descriptor in `Declension3` had been local rather than global.

There are a number of ways of understanding this global inheritance mechanism. The description we have given above amounts to a "global memory" model, in which a DATR query evaluator is a machine equipped with two memories: one containing the current local node and path, and another containing the current global node and path.

–

The declarative interpretation of global inheritance suggests an alternative procedural characterisation to the one already discussed, which we outline as follows. Starting

when one

4. DATR techniques

The DATR fragments introduced above illustrate the basic descriptive resources provided by the language. We now present some further examples, showing how these basic components combine to provide a powerful representation tool.

4.1 Case constructs and parameters

Evaluable paths allow the value of one path to be determined by the value of another. More generally, the values of an arbitrary number of descriptors can be invoked as parameters in an evaluable path and thus determine the value of a particular node/path pair. The familiar **case** construct of procedural programming languages is readily implemented, as the following example describing English plural suffixes shows:

```
NOUN:
    <plural> == <case of "<origin>">

    <case of latin masculine> == -i
    <case of latin neuter>    == -a
    <case of>                 == -s

    <origin> == norman.

Cat:
    <> == NOUN.
Datum:
    <> == NOUN
    <origin> == latin neuter.
Alumnus:
    <> == NOUN
    <origin> == latin masculine.
```

Here the value of the `<origin>` attribute of a noun (denoting its etymological source) is used to determine the value of its `<plural>`

we saw in Section 2 above, the passive participle form of *sew* is fully described by the node definition for `Word3`.

```
    Word3:
        <> == Sew
        <syn form> == passive participle.
```

For finite forms, we could use a similar technique. From this,

```
    Word4:
        <> == Sew
        <syn form> == present sing third.
```

we would want to be able to infer this:

```
    Word4:
        <mor form> = sew s
```

However, the components of `<syn form>`, `present`, `sing`, `third` are themselves values of features we probably want to represent independently. One way to achieve this is to define a value for `<syn form>` which is itself parameterised from the values of these other features. And the appropriate place to do this is in the `VERB` node, thus:

```
    VERB:
        <syn form> == "<syn tense>" "<syn number>" "<syn person>".
```

This says that the default value for the syntactic form of a verb is a finite form, but exactly which finite form depends on the settings of three other paths, `<syn tense>`, `<syn number>` and `<syn person>`. So now we can express `Word4` as:

```
    Word4:
        <> == Sew
        <syn tense> == present
        <syn number> == sing
        <syn person> == third.
```

This approach has the advantage that the attribute ordering used in the `<mor...>` paths is handled internally: the leaf nodes need not know or care about it[23].

4.2 Boolean logic

We can, if we wish, use parameters in evaluable paths that resolve to `true` or `false`. We can then define standard truth tables over DATR paths:

```
    Boolean:
        <> == false
        <or> == true
        <if> == true
        <not false> == true
        <and true true> == true
        <if true false> == false
        <or false false> == false.
```

This node defines the standard truth tables for all the familiar operators and connectives of the propositional calculus expressed in Polish rather than infix order[24]. Notice, in particular, how the DATR default that is not so. Consider a hypothetical language in which personal proper names have one of two genders, masculine or feminine. Instead of the gender being wholly determined by the sex of the referent, the gender is determined partly by sex and partly by the phonology. Examples of this general type are quite common in the world's languages[25]. In our hypothetical example, the proper name will have feminine gender either if it ends in a consonant and denotes a female or if it ends in a stop consonant but does not denote a female. We can encode this situation in DATR as follows[26]:

```
Personal_name:
    <> == Boolean
    <ends_in_consonant> == "<ends_in_stop>"
    <gender_is_feminine> ==
              <or <and "<female_referent>" "<ends_in_consonant>">
                  <and <not "<female_referent>"> "<ends_in_stop>">>.
```

We can then list some example lexical entries for personal proper names[27]:

```
Taruz:
    <> == Personal_name
    <female_referent> == true
    <ends_in_consonant> == true.
Turat:
    <> == Personal_name
    <female_referent> == true
    <ends_in_stop> == true.
Tarud:
    <> == Personal_name
    <ends_in_stop> == true.
Turas:
    <> == Personal_name
    <ends_in_consonant> == true.
```

Note that both `Turas` and `Tarud` turn out not to denote females, given the general `false` default in `Boolean`[28]. The genders of all four names can now be obtained as theorems:

```
Taruz: <gender_is_feminine> = true.
Turat: <gender_is_feminine> = true.
Tarud: <gender_is_feminine> = true.
Turas: <gender_is_feminine> = false.
```

25 For example, Fraser & Corbett (1995) use DATR to capture a range of phonology/morphology/semantics interdependencies in Russian. And Brown & Hippisley (1994) do the same for a Russian segmental phonology/prosody/morphology interdependency. But one can find such interdependencies in English also: see Ostler & Atkins (1992: 96-98).

26 Note that complex expressions require path embedding. Thus, for example, the well-formed negation of a conditional is <`not` <`if` .. ..>> rather than <`not` `not`

4.3 Finite state transduction

Perhaps surprisingly, DATR turns out to be an excellent language for defining finite state transducers (FSTs)[29] A

```
    <l o v e>           = l o v e
    <l o v e + s>       = l o v e s
    <l o v e + e d>     = l o v e d
    <l o v e + e r>     = l o v e r
    <l o v e + l y>     = l o v e l y
    <l o v e + i n g>   = l o v i n g
    <l o v e + a b l e> = l o v a b l e.
```

4.4 Representing lists

DATR's foundation in path/value specifications means that many of the representational idioms of unification formalisms transfer fairly directly. A good example is the use of `first` and `rest` attributes to represent list-structured features, such as syntactic arguments and subcategorised complements. The following definitions could be used to extend our verb fragment by introducing the path <`syn args`>, which determines a list of syntactic argument specifications.

```
NIL:
    <> == nil
    <rest> == UNDEF
    <first> == UNDEF.
VERB:
    <syn cat> == verb
    <syn args first syn cat> == np
    <syn args first syn case> == nominative
    <syn args rest> == NIL:<>.
```

Here extensions of <`syn args first`> specify properties of the first syntactic argument, while extensions of <`syn args rest`> specify the others (as a first/rest list). `UNDEF` is the name of a node that is not defined in the fragment, thus ensuring that <`syn args rest first`>, <`syn args rest rest`>, and so forth are all undefined. The fragment above provides a default specification for <`syn args`> for verbs consisting of just one argument, the subject NP. Subclasses of verb may, of course, override any part of this default; for instance, transitive verbs add a second syntactic argument for their direct object:

```
TR_VERB:
    <> == VERB
    <syn args rest first syn cat> == np
    <syn args rest first syn case> == accusative
    <syn args rest rest> == NIL:<>.
```

The description can be improved by using a separate node, `NP_ARG`, to represent the (default) properties of noun-phrase arguments:

```
NP_ARG:
    <first syn cat> == np
    <first syn case> == accusative
    <rest> == NIL:<>.
VERB:
    <syn cat> == v
    <syn args> == NP_ARG:<>
    <syn args first syn case> == nominative.
TR_VERB:
    <> == VERB
    <syn args rest> == NP_ARG:<>.
```

`TR_VERB` accepts the `NP_ARG` default unconditionally for the direct object argument, but `VERB` overrides the default `case` for its subject argument. The effect of the empty path (<>) specification in the `NP_ARG` inheritances is to "strip off" the leading subpath from the path whose value is inherited. The default mechanism adds the same path extension to both sides, giving rise to statements such as the following:

```
    VERB:<syn args first syn cat> == NP_ARG:<first syn cat>.
    TR_VERB:<syn args rest first syn cat> == NP_ARG:<first syn cat>.
    TR_VERB:<syn args rest first syn case> == NP_ARG:<first syn case>.
```

Three element argument lists, such as that needed for ditransitive verbs, are constructed in the obvious way (where `PP_ARG` is assumed to be like `NP_ARG` but for prepositional-phrase complements):

```
    DI_VERB:
        <> == TR_VERB
        <syn args rest rest> == PP_ARG:<>.
```

4.5 Lexical rules

A lexical representation language needs to be able to express the relations that are now widely thought to be in the domain of lexical rules[32]. Canonically, such rules deal with the phenomena that used to be described by the "cyclic rules" of late 1960s transformational grammar. Characteristically, they pertain to rather specific classes of lexical items (e.g., transitive verbs, or tensed auxiliary verbs) and they are subject to exceptions of various kinds[33]. It is these characteristics that have led many linguists to consign them to the lexicon. They usually involve a difference in argument structure and this is sometimes accompanied by a morphological difference. The combination of evaluable paths with a standard encoding of argument lists make it rather easy to define lexical rules in DATR[34].

Here, by way of illustration, is a partial analysis of verbs that implements a lexical rule for syntax of the (agentless) passive construction[35]:

```
    VERB:
        <mor past> == "<mor root>" edyn _A"
```

Evans et al. (1995)

path. Instead, DATR allows their relatedness of meaning to be captured by using the definition of one in the definition of another.

A very few words in English have alternative morphological forms for the same syntactic specification. An example noted by Fraser & Hudson (1990, 62) is the plural of *hoof* which, for many English speakers, can appear as both *hoofs* and *hooves*[39]. DATR does not permit a theorem set such as the following to be derived from a consistent description:

```
Word7:
    <syn number> = plural
    <mor form> = hoof s
    <mor form> = hoove s.
```

But it is quite straightforward to define a description that will lead to the following theorem set:

```
Word7:
    <syn number> = plural
    <mor form> = hoof s
    <mor form alternant> = hoove s.
```

Or something like this:

```
Word7:
    <syn number> = plural
    <mor forms> = hoof s | hoove s .
```

Or this:

```
Word7:
    <syn number> = plural
    <mor forms> = { hoof s , hoove s }.
```

Of course, as far as DATR is concerned `{ hoof s , hoove s }` is just a sequence of seven atoms. It is up to some component external to DATR which makes use of such complex values to interpret it as a two member set of alternative forms. Likewise, if we have some good reason for wanting to put together the various senses of *cherry* into a value returned by a single path, then we can write something like this:

```
Cherry:
    ...
    <sem glosses> == { <sem gloss 1> , <sem gloss 2> , <sem gloss 3> }.
```

which will then provide this theorem:

```
Cherry:
    <sem glosses> = { sweet red berry with pip ,
                      tree bearing sweet red berry with pip ,
                      wood from tree bearing sweet red berry with pip }.
```

Also relevant here are the various techniques for reducing lexical disjunction discussed in Pulman (forthcoming).

4.7 Encoding DAGs

As a feature-based formalism with a syntax modelled on PATR, it would be reasonable to expect that DATR can be used to describe directed acyclic graphs (DAGs) in a PATR-like fashion. Consider an example such as the following:

```
DAG1:
    <vp agr> == <v agr>
```

39 See also the *dreamt/dreamed* verb class discussed by Russell et al. (1992, 330-331).

```
        <v agr per> == 3
        <vp agr gen> == masc.
```

This looks like simple reentrancy from which we would expect to be able to infer:

```
    DAG1:
        <vp agr per> = 3.
```

And, indeed, this turns out to be valid. But matters are not as simple as the example makes them appear: if `DAG1` was really the DAG it purports to be, then we would also expect to be able to infer:

```
    DAG1:
        <v agr gen> = masc.
```

But this is not valid, in fact <`v agr gen`> is undefined. It might be tempting to conclude from this that the equality operator in DATR is very different from the corresponding operator in PATR, but this would be to misunderstand what has happened in this example. In fact, the semantics of the statement

```
    DAG1:
        <vp agr> == <v agr>.
```

taken in isolation is very similar to the semantics of the corresponding PATR statement: both assert equality of values associated with the two paths. The DATR statement is slightly weaker in that it allows the left-hand-side to be defined when the right-hand-side is undefined. But, even in DATR, if both sides are defined they must be the same, so, in principle, the value of the left-hand-side does semantically constrain the value of the right-hand-side. However, in a DATR description, specifying explicit values for extensions of the left-hand-side of such an equality constraint **overrides** its effect, and thus does not influence the values on its right-hand-side.

Another difference lies in the fact that DATR subpaths and superpaths can have values of their own:

```
    DAG2:
        <v agr> == sing
        <v agr per> == 3.
```

From this little description we can derive the following statements, inter alia:

```
    DAG2:
        <v agr> = sing
        <v agr num> = sing
        <v agr per> = 3
        <v agr per num> = 3.
```

From the perspective of a standard untyped DAG-encoding language like PATR, this is strange. In PATR, if <`v agr per`> has value `3`, then neither <`v agr`> nor <`v agr per num`> can have (atomic) values.

As these examples clearly show, DATR descriptions do not map **trivially** into (sets of) standard DAGs (although neither are they entirely dissimilar). But that does not mean that DATR descriptions cannot **describe** standard DAGs. Indeed, there are a variety of ways in which this can be done. An especially simple approach is possible when the DAGs one is interested are all built out of a set of paths whose identity is known in advance (Kilbury et al. 1991). In this case, we can use DATR paths as DAG paths, more or less directly:

```
    PRONOUN2:
        <referent> == '<' 'NP' referent '>'.
    She2:
        <> == PRONOUN2
        <case> == nominative
        <person> == third
        <number> == singular.
```

From this description, we can derive the following theorems:

```
    She2:
        <case> = nominative
        <person> = third
        <number> = singular
        <referent> = < NP
```

The sequence of atoms on the right

with a node/path pair, but at most one of these defines a value or global with that inherited from another. Because of this, the handling of multiple inheritance is an issue which is central to the design of any formalism for representing inheritance networks.

	Theory	Query	Value
Conventional inference	*given*	*given*	*unknown*
Reverse query	*given*	*unknown*	*given*
Theory induction	*unknown*	*given*	*given*

with bit-vectors for speed and compactness. At the other extreme, Duda & Gebhardi (1994) present an interface between a PATR-based parser and a DATR map strings of atomic phonemes to strings of atomic phones. But it also allows one to encode full-blown feature and syllable-tree based prosodic analyses.

Unlike the formalisms typically proposed by linguists, DATR does not attempt to embody in its design any substantive and restrictive universal claims about the lexicons of natural language. That does not distinguish it from most NLP formalisms, of course. However, we have also sought to ensure that its design does not embody features that

Human-Computer Studies **41.1/2,** 149-177.

Walter Daelemans, Koenraad De Smedt & Gerald Gazdar (1992) Inheritance in natural language processing . *Computational Linguistics* **18.2,** 205-218.

Walter Daelemans & Gerald Gazdar, eds. (1992) *Computational Linguistics* **18.2** & **18.3**, special issues on inheritance.

Walter Daelemans & Erik-Jan van der Linden (1992) Evaluation of lexical representation formalisms. In Jan van Eijck & Wilfried Meyer, eds. *Computational Linguistics in the Netherlands: Papers from the Second CLIN Meeting.* Utrecht: OTS, 54-67.

Marc Domenig & Pius ten Hacken (1992) *Word Manager: A System for Morphological Dictionaries.* Hidesheim: Georg Olms Verlag.

Markus Duda & Gunter Gebhardi (1994) DUTR – A DATR-PATR interface formalism. In Harald Trost, ed., *Proceedings of KONVENS-94,* Vienna: Oesterreichische Gesellschaft fuer Artificial Intelligence, 411-414.

for Computational Linguistics, 137-142.
James Kilbury, Petra [Barg] Naerger & Ingrid Renz (1994) Simulation lexicalischen Erwerbs In Sascha W. Felix, Christopher Habel & Gert Rickheit *Kognitive Linguistik: Repraesentation und Prozesse.* Opladen: Westdeutscher Verlag, 251-271.
Adam Kilgarriff (1993) Inheriting verb alternations. *Sixth Conference of the European Chapter of the Association for Computational Linguistics,* 213-221.
Adam Kilgarriff (1995) Inheriting polysemy. In Patrick Saint-Dizier & Evelyne Viegas, eds. *Computational Lexical Semantics.* Cambridge: Cambridge: Cambridge University Press, 00-00.
Adam Kilgarriff & Gerald Gazdar (1995) Polysemous relations. In F.R. Palmer, ed. *Grammar and Meaning: Essays in Honour of Sir John Lyons.* Cambridge: Cambridge University Press, 1-25.
Hans-Ulrich Krieger (1994) Derivation without lexical rules. In C.J. Rupp, M.A. Rosner & R.L. Johnson, eds. *Constraints, Language and Computation.* London: Academic Press, 277-313.
Hans-Ulrich Krieger & John Nerbonne (1993) Feature-based inheritance networks for computational lexicons. In Ted Briscoe, Valeria de Paiva and Ann Copestake, eds. *Inheritance, defaults, and the lexicon.* Cambridge, Cambridge University Press, 90-136.
Hans-Ulrich Krieger, Hannes Pirker & John Nerbonne (1993) Feature-based allomorphy. *31st Annual Meeting of the Association for Computational Linguistics,* 140-147.
Hagen Langer (1994) Reverse queries in DATR. *COLING-94,* 1089-1095.
Hagen Langer & Dafydd Gibbon (1992) DATR as a graph representation language for ILEX speech oriented lexica. Technical Report **ASL-TR-43-92/UBI,** University of Bielefeld.
Alex Lascarides, Nicholas Asher, Ted Briscoe & Ann Copestake (forthcoming) Order independent and persistent typed default unification. To appear in *Linguistics & Philosophy.*
Marc Light (1994) Classification in feature-based default inheritance hierarchies. In Harald Trost, ed., *Proceedings of KONVENS-94,* Vienna: Oesterreichische Gesellschaft fuer Artificial Intelligence, 220-229.
Marc Light, Sabine Reinhard & Marie Boyle-Hinrichs (1993) INSYST: an automatic inserter system for hierarchical lexica. *Sixth Conference of the European Chapter of the Association for Computational Linguistics,* 471.
Paul McFetridge & Aline Villavicencio (1995) A hierarchical description of the Portuguese verb. *Proceedings of the XIIth Brazilian Symposium on Artificial Intelligence,* 00-00.
Chris Mellish & Ehud Reiter (1993) Using classification as a programming language. *IJCAI-93,* 696-701.
Teruko Mitamura & Eric H. Nyberg III (1992) Hierarchical lexical structure and interpretive mapping in machine translation. *COLING-92* Vol. IV, 1254-1258.
John Nerbonne (1992) Feature-based lexicons – an example and a comparison to DATR. In Dorothee Reimann, ed. *Beitrage des ASL-Lexicon-Workshops.* Wandtlitz, 36-49.
Nicholas Ostler & B.T.S. Atkins (1992) Predictable meaning shift: some linguistic
10.4(Computational342.-10.001Td(409-44.1-9.11992-1.4023Td(Nomes-124.2(KPutemo
CMrph Umlautbestc290.2(hreiburg-1202.3(ri-TJ10.001TLT(iinhm-1243.(heerarc13.
207rNerbtis
Cexical Semantics.-14043(and-13123.(lKno-12.4(lie44.9(4(geTJ-0.001T
Pabine Reinhard & Mafydd (19921
110.0
740Td(230:)-j-120.74010.0801Td[(As-496.0612(nual)-11286.83(Me)1028.54(eting)-12649.5(of)-13635.7(the)]1401239(Asso)-72.6412(iation)-11325.3(for)-TJ9
Jerald Gaen Mickhmod NThmatsn
PDfault Rfinter-12.(2sati-1301(mac22.41(hine &nd afinter-TJT(Isati-1321((2phon

Alan W. Black and Stephen G. Pulman (1992) *Computational Morphology.* Cambridge, Ma.: MIT Press.

Graham Russell (1993) Review of Marc Domenig & Pius ten Hacken (1992) *Word Manager: A System for Morphological Dictionaries.* Hidesheim: Georg Olms Verlag. *Computational Linguistics* **19.4,** 699-700.

Graham Russell, Afzal Ballim, John Carroll & Susan Warwick-Armstrong (1992) A practical approach to multiple default inheritance for unification-based lexicons. *Computational Linguistics* **183,** 311-337.

Harvey Sacks (1973) On some puns with some intimations. In Roger W. Shuy. ed. *Report of the 23rd Annual Roundtable Meeting on Linguistics and Language Studies.* Washington D.C.: Georgetown University Press, 135-144.

Stuart M. Shieber (1986) *An Introduction to Unification Approaches to Grammar.* Stanford: CSLI/Chicago University Press.

Greg Stump (1992) On the theoretical status of position class restrictions on inflectional affixes. In Geert Booij & Jaap van Marle, eds. *Year Book of Morphology 1991.* Dordrecht: Kluwer, 211-241.

David S. Touretzky (1986) *The Mathematics of Inheritance Systems.* London/Los Altos: Pitman/Morgan Kaufmann.

Mark A. Young (1992) Nonmonotonic sorts for feature structures. *AAAI-92,* 596-601.

Mark A. Young & Bill Rounds (1993) A logical semantics for nonmonotonic sorts. *Proceedings of the 31st Annual Meeting of the ACL,* 209-215.

APPENDIX: The critical literature on DATR reviewed

Since DATR has been in the public domain for the last half dozen years and been widely used in Europe during that period (by the standards of lexical knowledge representation

Finally, Domenig & ten Hacken contend that lexical inheritance formalisms (and thus DATR) are unusable for the purpose for which they were designed because the humans who have to work with them for lexicon ever, we agree with their earlier comment "that orthogonal multiple default inheritance is at this stage the best solution for conflicts" (p61) and can see no computational linguistic motivation for equipping DATR with a further primitive inheritance mechanism[53].

Their fourth objection consists of the claim that "it is not possible in DATR to have complex structured objects as values" (p64). In one sense this is true since DATR values are simply sequences of atoms. But although true, it does not provide support for a sound objection. DATR can construct those sequences of atoms on the basis of a complex recursive description, and the atom sequences themselves can **represent** complex recursive objects so far as NLP system components outside the lexicon are concerned. The sequences of atoms that DATR provides as values simply constitute an interface for the lexicon that is entirely neutral with respect to the representational requirements of external components. For what is intended to be a general purpose lexical knowledge representation language, not tied to any particular conceptions of linguistic structure or NLP formalism, this neutrality seems to us to be a feature, not a bug.

In a fifth objection, they note correctly that the semantics of paths in DATR and PATR is different but then go on to claim that DATR paths "could be better described as atomic attributes" that "do not correspond with a recursive structure" and whose "only function is to support prefix matching" (p64). None of these latter claims are true. If DATR paths were atomic attributes then our Section 4.3, above, on finite state introduction of defaults is meant to eliminate[56].

At the root of Krieger & Nerbonne's (1993) critique of DATR is a complaint that it fails to provide all the resources of a modern fully equipped unification grammar the fact that an atom may mean one thing in the semantics of DATR and something quite different in the semantics of a feature formalism will lead to "massive redundancy" (p47) in lexical specifications (the phrase gets repeated in Bouma & Nerbonne 1994). Again, no argument in support of this conclusion is offered. And we cannot see how semantic overloading of atoms gives rise, of itself, to any kind of redundancy[58]. Indeed, those who design programming languages normally introduce semantic overloading in order to achieve economy of expression.

Finally, Bouma & Nerbonne (1994) comment that "in spite of Kilgarriff's (1993) interesting work on modelling some derivational relations in the pure inheritance machinery of DATR, we know of no work attempting to model potentially recursive derivational relations, and we remain sceptical about relying on inheritance alone for this". We are not sure what they mean by "the pure inheritance machinery of DATR" or why they think that someone attempting an analysis of recursive derivation in DATR would want to do so using "pure inheritance" alone. Here is a trivial (and linguistically rather pointless) DATR analysis of the more complex of their two examples:

```
    Word:
        <v> == "<>"
        <a from n> == <n> + al
        <v from a> == <a> + ize
        <n from v> == <v> + tion.
    Institute:
        <> == Word
        <root> == institute.
```

From this description we can derive theorems like these:

```
    Institute:
        <root> = institute
        <n from v root> = institute + tion
        <a from n from v root> =
                          institute + tion + al
        <v from a from n from v root> =
                          institute + tion + al + ize
        <n from v from a from n from v root> =
                          institute + tion + al + ize + tion.
```

Note the recursive reintroduction of the *tion*