D-PATR: A Development Environment for Unification-Based Grammars

Lauri Karttunen
Artificial Intelligence Center
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025 USA
and
Center for the Study of Language and Information
Stanford University

1 Introduction

D-PATR is a development environment for unification-based grammars on Xerox 1100 series work stations. It is based on the PATR formalism developed at SRI International. This formalism is suitable for encoding a wide variety of grammars. At one end of this range are simple phrase-structure grammars with no feature augmentations. The PATR formalism can also be used to encode grammars that are based on a number of current linguistic theories, such as lexical-functional grammar (Bresnan and Kaplan), head-driven phrase structure grammar (Pollard and Sag), and functional unification grammar (Kay). At the other end of the range covered by D-PATR are unification-based categorial grammars (Klein, Steedman, Uszkoreit, Wittenburg) in which all the syntactic information is incorporated in the lexicon and the remaining few combinatorial rules that build phrases are function application and composition. Definite-clause grammars (Pereira and Warren) can also be encoded in the PATR formalism.

What these approaches have in common is that syntactic rules and lexical entries can be written down as sets of attribute-value pairs. Moreover, because a value at the end of one path of attributes can be shared by another path, the structures that are generated by such grammars can be thought of as directed graphs ("dags"). Unification is the key operation for building these structures. Because unification is associative and commutative, statements in a unification-based grammar formalism are order-independent and bidirectional with respect to parsing and generation. For a comprehensive introduction to unification-based approaches to grammar, see Shieber 1986 (forthcoming).

The idea that led to the present version of D-PATR was to produce a simple compact system for experimenting with unification-based grammars that would run on machines smaller than the Symbolics 3600 for which the original PATR implementation at SRI had been created. The first version of D-PATR, initially called HUG, was written at the Scandinavian Summer Workshop for Computational Linguistics in Helsinki, Finland, at the end of August 1985. Although the actual notation for writing rules in D-PATR in some respects differs from the notation in the original PATR system, essentially both systems implement the same grammar formalism. To emphasize this point, the two implementations are now called Z-PATR (Zeta-LISP PATR) and D-PATR (Interlisp-D PATR). A number of innovations that came in with D-PATR (HUG) have since migrated to Z-PATR. A case in point is the method for minimizing copying in unification that is discussed in the section on parsing and unification. Other implementation differences remain—for example, in the parsing algorithm and in the treatment of gaps—but grammars written for D-PATR are convertible into Z-PATR format, and vice versa.

D-PATR consists of four basic parts:

- A unification package
- Interpreter for rules and lexical items
- Input output routines for directed graphs
- An Earley style chart parser.

These packages are written in simple Interlisp-D for transportability to other dialects of LISP. They do not depend on the features of any particular machine. The only part of D-PATR that is specific to Xerox 1100 series work stations is its **user interface**. This last set of routines takes full advantage of the graphic capabilities of D-machines. It provides good facilities for writing and editing grammars as well as many debugging tools for the grammar writer.

2 Grammar Formalism

2.1 Rules

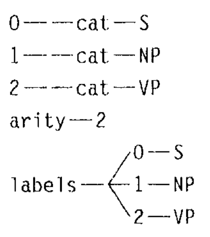
A rule in D-PATR is a list of atomic *constituent labels* that may be followed by *specifications*. Specifications are constraints upon one or more constituents of the rule. In the simplest case, there are no specifications and the labels correspond to symbols in an ordinary phrase structure rule. For example, the rule

```
S → NP VP
```

in D-PATR notation is written as

```
(S NP VP)
```

Before a rule is used by the parser, D-PATR compiles it to a feature set. A feature set can be displayed in different ways—for example, as a matrix or as a directed graph. In this paper, we usually represent feature sets as graphs but the matrix notation will also be used occasionally. In these graphs, the constituents of the rule are represented by labeled arcs, called *attributes*. The nodes in the graph are *values*. A value can be atomic or complex; a complex value is another set of attribute-value pairs. By convention, the symbol on the left-hand side of a phrase structure rule is represented by the numeric attribute 0. Constituents on the right-hand side of the rule are numbered left-to-right, starting with 1. The above rule D-PATR represents as the following feature set, shown here first as a graph and then as the equivalent matrix.



$$\begin{bmatrix} 0 & \begin{bmatrix}\text{cat} & \text{S}\end{bmatrix} \\ 1 & \begin{bmatrix}\text{cat} & \text{NP}\end{bmatrix} \\ 2 & \begin{bmatrix}\text{cat} & \text{VP}\end{bmatrix} \\ \text{arity} & 2 \\ \text{labels} & \begin{bmatrix} 0 & \text{S} \\ 1 & \text{NP} \\ 2 & \text{VP}\end{bmatrix} \end{bmatrix}$$

2.2 Specifications

In the above rule, the *cat* feature is interpreted by D-PATR as a constraint on the manner in which the constituent can be instantiated. More constraints can be added by annotating the rule with specifications. A specification is a two-item list of the form

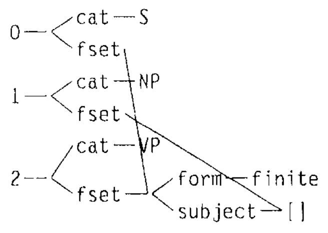
```
( { attribute | path } { path | value } ).
```

Here *attribute* is an atom, *path* is a list, and *value* is either an atomic symbol, a list of specifications, or an abbreviation for such a list. The last case is distinguished from the first by prefixing the value symbol with @ when it has an abbreviatory role. Ignoring the @-cases, this gives four different kinds of specifications: (*attribute value*), (*path value*), (*attribute path*), and (*path path*). The same feature set can often be specified in several different ways; in choosing one, we generally try to minimize the number of parentheses.

Below is a simple example of a phrase structure rule augmented with specifications.

```
(S NP VP
   ((0 fset)(2 fset))
   ((1 fset) (2 fset subject))
   ((2 fset form) finite))
```

The first specification is an analogue of an LFG ↑ = ↓ annotation; the S node inherits its feature set from the VP. In addition, NP is VP's subject and VP's feature set contains the feature [form: finite]. D-PATR compiles this rule to the following graph. (From now on, we shall omit the two non-constituent attributes, *arity* and *labels*, from our display.)



As the graph shows, the feature set of the S-node is the same as VP's feature set. The NP constituent has been unified with the subject in the feature set of the S and the VP.

It is permissible in D-PATR to write rules that do not assign their constituents to any specific syntactic category, such as S, NP, VP, and the like. The default assumption is that the constituent labels also serve as values of *cat*. By declaring them to be *DummyCategories*, the grammar writer can override this convention. The default dummy symbols are X, Y, and Z. Instead of (S NP VP), one could just as well write

```
(X Y Z
   ((0 cat) S)
   ((1 cat) NP)
   ((2 cat) VP)
```

It is also legal to leave *cat* entirely unspecified. This option is useful for expressing rules of function application and composition in lexically-based categorial grammars.

2.3 Words and Stems

In its present form, D-PATR does not have a morphological analyzer to relate inflected or derived forms of words to entries in a morpheme lexicon. All lexemes must be entered individually. In anticipation of having a better solution available in the future, D-PATR presently splits the lexicon into two parts: *words* and *stems*. The format of the two lexicons is the same, but entries in the word lexicon may contain a reference to an entry in the stem lexicon. For example, the entries for *am*, *are*, *is*, *was*, *were*, etc. in the word lexicon can refer to the entry for *be* in the stem lexicon. Consequently, what is common to all forms of the auxiliary can be stated in a single place.

A lexical entry is a list consisting of a *form* and a list of *subentries*. Each subentry in turn is a list headed by a morphological *category* and any number of *specifications*. A specification can be a two-item list of the type discussed in the previous section or a *template*. A template is an abbreviation for a list of specifications. For example, the entry for *kisses* in the word lexicon might look as follows:

```
(kisses (V kiss PresTense Sg3)
        (N kiss Pl)).
```

Here N and V are used as names of morphological categories; *kiss* refers to an entry in the stem lexicon; *PresTense*, *Sg3* and *Pl* are templates. The fact that *kiss* is a stem and *Sg3* a template is not marked; it is rather determined by where their definitions are found. The entry for *kiss* in the stem lexicon could be, for example,

```
(kiss (V VMain TakesNP Dyadic)
      (N)).
```

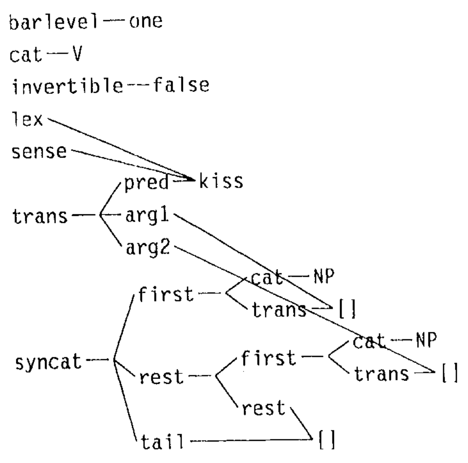
When the definitions for *kisses* and *kiss* are interpreted, the templates and other specifications that occur in their subentries are processed sequentially from left to right. Each item is compiled to a directed graph and superimposed on the graph previously compiled. This overwriting operation differs from standard unification in that it never fails; if two specifications give conflicting values to some path, the later specification overrules the earlier one. The lexicon writer can take advantage of this convention to set up a hierarchical feature system in which initial default assignments can be overridden by later specifications.

2.4 Templates

Definitions for templates have the same format as the entries in the word and stem lexicons except that there are no multiple subentries; templates are assumed to be unambiguous. A template definition is simply a list consisting of a *template name* and a number of specifications. For example, the template names that appear in the entries for *kiss* might be expanded as follows: (Note that a specification may be either a two-item list of the form discussed in section 2.2 or a name of another template.)

```
(V OneBar)
(OneBar (barlevel one))
(VMain Predicate (invertible false))
(Predicate ((trans pred) (sense)))
(TakesNP ((syncat first cat) NP)
         ((syncat rest first cat) NP)
         ((syncat rest rest)(syncat tail)))
(Dyadic  ((trans arg1)
          (syncat first trans))
         ((trans arg2)
         (syncat rest first trans)))
```

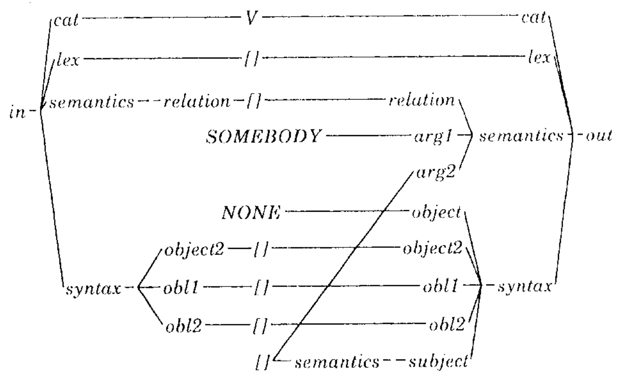
With these definitions, the verb entry for *kiss* in the stem lexicon compiles to the graph shown below.



The role of the template *TakesNP* in this entry is to state that the verb kiss requires two NP's as its syntactic arguments. The first element of the list is the value of the path <syncat first>, the second is <syncat rest first>. The template *Dyadic* links the two arguments in the semantic translation of *kiss* to the translations of its syntactic arguments.

2.5 Lexical Rules

The expansion of morpheme definitions by means of templates is a straightforward matter: an initial graph acquires additional features and, perhaps, new values for features it already has. D-PATR also allows a more radical transformation by means of *lexical rules*. A lexical rule is a special kind of template with two attributes: *in* and *out*. In applying a lexical rule to a graph, the latter is first unified with the value of *in*. If the operation succeeds, the value of *out* is passed on as the result. Because the values under *out* can be linked selectively to the corresponding values under *in*, the usual result of applying a lexical rule is a metamorphosis of the input graph. As an example, let us consider the graph corresponding to a possible lexical rule for *Passive* in English. To make it easier to see the effect of the rule, the graph is turned aroud so that the *out* values are opposite to the corresponding *in* values; the indeterminates in the middle are unified with their counterparts in the word to which the rule is applied.



The effect of the rule is to make a transitive verb lose the object slot in its syntactic frame, even though semantically it remains a two-place predicate. The semantic effect of the rule is to unify *arg2* with the subject's semantics and to assign to *arg1* the value SOMEBODY. This is similar to the analysis of passives in some LFG grammars.

2.6 Fillers and Gaps

Constructions such as the following contain constituents that, semantically and syntactically, fill a vacant slot—a *gap*—somewhere in the adjacent structure.

That paper I don't intend to read — .
Good avocados are hard to find — .
The neighbor *whose car* you asked to borrow — called.
Is this the company *the histogram of whose production* she wants to display — ?

From a parser's point of view, there are two main problems to be solved. For the parse to succeed, the filler needs to be available when the incomplete structure is encountered. There must also be a way to ensure that a designated filler will be consumed by a gap somewhere. A third problem is that, in relative clauses, the filler must contain a relative pronoun.

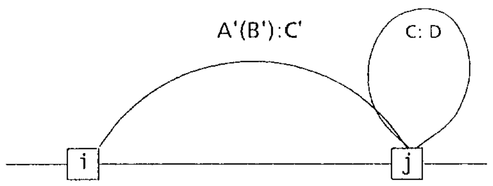
Many solutions to these problems have been proposed and could be implemented in D-PATR. As a convenience, D-PATR also makes available to the grammar writer a built-in default mechanism for distributing the information about fillers, gaps, and relative pronouns in an appropriate way. The original idea, conceived by Fernando Pereira, was implemented for gaps in Z-PATR by Stuart Shieber. The scheme in D-PATR is an improvement in that it also handles sentences with nested filler-gap dependencies.

The default mechanism uses four special features: *gapIn*, *gapOut*, *relIn* and *relOut*. These features need to be mentioned explicitly only in rules that introduce fillers, such as the relative-clause rule, and in the lexical entries of relative and interrogative pronouns. Other rules are automatically augmented by D-PATR in the appropriate manner when they are compiled to feature sets used by the parser. By deactivating this facility, the grammar writer can also take care of fillers and gaps in a manner of his own choosing.

3 Parsing and Unification

D-PATR uses an active chart parser that proceeds in a top-down, breadth-first manner. Because the constituents in a rule are feature sets rather than atomic symbols, the task is a bit more complicated than in standard implementations of Earley's algorithm. We consider two cases here.

Let us assume that the parser is in the process of trying to build an instance of the rule A → B C and that it has successfully instantiated B as B'. At this point, it will enter a partial instantiation of the rule on the chart. We designate this active edge as A'(B'): C'. Here the colon marks the line between daughter constituents that have been found and daughters that still need to be instantiated. When an active edge is added to the chart, the parser needs to find all the rules that match the first uninstantiated constituent to the right of the colon. In the case at hand, it needs to match C' against the left-hand sides of all rules to determine what rules it should now try to instantiate. For example, if there is a rule C → D in the grammar and C is compatible with C', a looping C: D or C': D' edge should be added to the chart.

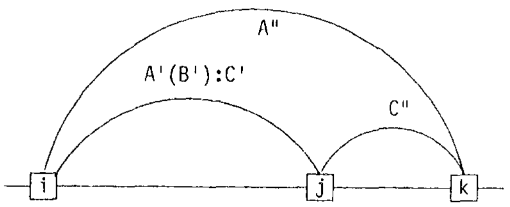


In the case of an ordinary phrase-structure grammar, this matching task is simple because constituents are represented by atomic category labels. Furthermore, A = A', B = B', and C = C'. For D-PATR, the situation is more complicated. First of all, the constituents are feature sets; second, the constituents in a partially instantiated rule are generally not equal to the corresponding constituents in an uninstantiated rule. Because of the links among constituents in a unification-based grammar, instantiating B as B' in the rule A → B C may also have an effect on the feature sets of A and C. This is why we label the resulting edge A'(B'): C'. Using the feature set C' to find the rules that could instantiate it is no more difficult than using the original C, but it is less efficient because the result cannot be saved and reused when another instance of C must be built later.

D-PATR solves this problem by carrying the original rule along with its partially instantiated form on active edges. The matching task for the prediction step of Earley's algorithm is performed using the constituent from the original rule rather than its current instance.

A similar problem arises when an inactive edge is entered on the chart. When the parser has instantiated C as C" and entered it on the chart, it has to find all the incoming active edges at the starting vertex of C" that could be extended with the newly found constituent. If C" were an atomic symbol, this task would be simple because it would involve only simple equality checks; because C" is a feature set, we would have to use unification, which is a more time-consuming operation. D-PATR avoids the problem entirely by keeping track, as part of the prediction step, of what edges C" could be used to extend. When an active edge is entered on the chart, one piece of information in the edge label is a pointer to the edges that could be extended with it. Initially, the list contains only the edge that generated the new edge; other edges may be added later. This information is passed along on whenever an existing edge is extended to a new one. At the point at which C" is added to the chart, no checks are necessary because the new edge already has a pointer to every incoming edge at the starting vertex that can now perhaps be extended.

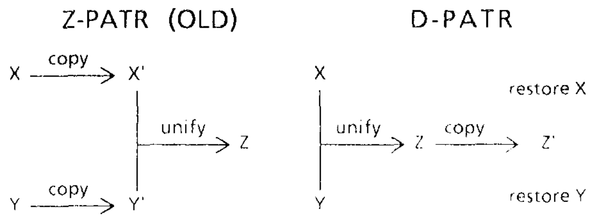
Let us now consider a situation, in which the chart contains two adjacent edges A'(B'):C' and C".



In the course of trying to extend the the active edge with C" to build A", the parser has to unify it with the C" constituent of the active edge. The nature of chart parsing is such that, whether or not this unification succeeds, it must not alter the contents of the two operand edges. Both A'(B'):C' and C" must remain on the chart because they may be needed later on for some other unification.

Because unification is a destructive operation, some of the earlier implementations of unification-based chart parsing, e.g. Z-PATR, never apply it directly. Instead, the feature sets are first copied and unification is then applied to the copies. In this way, the operands are left untouched, as the parsing algorithm requires, but the method is computationally inefficient because it involves a great deal of copying. D-PATR solves the problem in a novel way. In D-PATR, unification is implemented so that the original state of the input structures can be restored after the operation has been completed. Whenever a destructive change is about to be made in the value of an attribute, the cell and its contents are saved in an array. After unification, all the effects of the operation can be undone by restoring the saved values. D-PATR takes advantage of this option in the following way.

When the parser tries to extend A'(B'): C' to A" by unifying C' with C", the operation is applied directly to the two feature sets without them being copied in advance. If the unification fails, its effects are simply cancelled by restoring the original feature sets from the save array. If the operation succeeds, the resulting structure is copied and then the original feature sets are then restored. The copied result remains of course unaffected by the cancellation. The following sketch summarizes the difference between D-PATR and earlier versions of Z-PATR with respect to copying and unification. Here X and Y stand for the original feature sets, Z for the result, and the copied structures are identified with primes.



As the illustration shows, the new method entails making only one copy, not two, when the operation succeeds. In the event of failure, D-PATR simply restores the original structures without copying anything; the old method always copies both input structures.

In the case of Z-PATR, the new method has shortened parsing times by a factor of three. It is expected that this technique can be further improved by implementing some form of structure sharing [Karttunen and Kay 1985; Pereira 1985] to minimize the need for copying.

4 Conclusion

Unlike some other grammar development systems—for example, Ronald Kaplan's LFG Grammar Writer's Workbench [Kiparsky 84]—D-PATR is not an implementation of a particular linguistic theory. It is designed to be an efficient generic tool for exploring a range of grammar formalisms in which unification plays a central role. Because of its friendly interface and display facilities, D-PATR can also be used for educational purposes, in particular, to demonstrate chart parsing and unification.

D-PATR is not a commercial product. It is made available to users outside SRI who might wish to develop unification-based grammars. D-PATR is currently being used for grammar development at SRI International, CSLI, and Xerox PARC. For a more comprehensive discussion of D-PATR and its features, see Karttunen (forthcoming).

Acknowledgments

D-PATR is a close relative of Z-PATR, the first PATR implementation, whose main architect is Stuart Shieber. Many of the ideas in D-PATR come from the work of Ronald Kaplan, Martin Kay, and Fernando Pereira. The matrix display package for feature sets in D-PATR was written by Per-Kristian Halvorsen. I would also like to thank Annie Zaenen and Hans Uszkoreit for their help in designing the system.

This research, made possible in part by a gift from the System Development Foundation, was also supported by the Defense Advanced Research Projects Agency under Contract N00039-84-C-0524 with the Naval Electronics Systems Command.

References

Kaplan, R. and J. Bresnan, "Lexical-functional grammar: A Formal System for Grammatical Representation," *The Mental Representation of Grammatical Relations*, J. Bresnan, ed., MIT Press, Cambdridge, Massachusetts, 1983.

Karttunen, L. and M. Kay, "Structure Sharing with Binary Trees," *Proceedings of the 23rd Annual Meeting of the ACL*, Association for Computational Linguistics, 1985.

Karttunen, L. *D-PATR: A Development Environment for Unification-Based Grammars*, CSLI Report, Center for the Study of Language and Information, Stanford, California (forthcoming in 1986).

Kay, M., "Parsing in Functional Unification Grammar," *Natural Language Parsing*, D. Dowty, L. Karttunen, and A. Zwicky, eds., Cambridge University Press, Cambridge, England, 1985.

Kiparsky, C. "LFG Manual," manuscript, Xerox Palo Alto Research Center, Palo Alto, California (1985).

Pereira, F. C. N., "A Structure-Sharing Representation for Unification-Based Grammar Formalisms," *Proceedings of the 23rd Annual Meeting of the ACL*, Association for Computational Linguistics, 1985.

Pereira, F. C. N. and D. H. D. Warren, "Definite-Clause Grammars for Language Analysis—a Survey of the Formalism and a Comparison with Augmented Transition Networks," *Artificial Intelligence*, 13:231-278, 1980.

Pollard, C., *Generalized Phrase Structure Grammars, Head Grammars, and Natural Languages*, Ph.D. dissertation, Stanford University, Stanford, California (1984).

Pollard, C., Lecture notes on head-driven phrase-structure grammar, Center for the Study of Language and Information, unpublished (February 1985).

Shieber, S. M., H. Uszkoreit, F. C. N. Pereira, J. J. Robinson, and M. Tyson, "The Formalism and Implementation of PATR-II," *Research on Interactive Acquisition and Use of Knowledge*, B. Grosz and M. Stickel, eds., SRI Final Report 1894, SRI International, Menlo Park, California, 1983.

Shieber, S. M., L. Karttunen, and F. C. N. Pereira, *Notes from the Unification Underground: A Compilation of Papers on Unification-Based Grammar Formalisms.* Technical Report 327, Artificial Intelligence Center, SRI International, Menlo Park, California (June 1984).

Shieber, S. M., *An Introduction to Unification-Based Approaches to Grammar*, CSLI Lecture Notes Series, (University of Chicago Press, Chicago Illinois, forthcoming in 1986).

Steedman, M., "Combinators, Categorial Grammars, and Parasitic Gaps," paper presented at the Tucson Conference on Categorial Grammar (June 1985).

Uszkoreit, H., "On Categorial Unification Grammars," in this volume.

Wittenburg, K., *Some Properties of Combinatory Categorial Grammars of Relevance to Parsing.* Technical Report HI-012-86, Microelectronics and Computer Technology Corporation, Austin, Texas, (January 1986).