An Efficient Implementation of PATR for Categorial Unification Grammar

Todd Yampol
Stanford University

Lauri Karttunen
Xerox PARC and CSLI

1 Introduction

This paper describes C-PATR, a new C implementation of the PATR-II formalism [4, 5] for unification-based grammars. It includes innovations that originate with a project for developing an efficient translator from English to first-order logic [2], most notably the extension of the standard unification algorithm for list values [section 3]. In addition the unifier and a chart parser tuned for categorial grammars, the system (C-PATR) contains a set of tools necessary for grammar development. These tools include facilities for hierarchical lexicon design and interactive grammar debugging [section 4].

2 Grammar Formalism

2.1 PATR-II as implemented in C-PATR

PATR-II is a formalism for describing grammars in terms of feature structures. C-PATR supports two equivalent notational systems for representing feature structures, *path equations* and *attribute-value matrices.* Path equations can be used to define a hierarchical system of *templates* [section 4] that encode linguistic generalizations. Internally, feature structures as are represented as directed graphs (DGs). PATR-style feature structures are capable of describing a wide variety of unification-based grammars. The present version of C-PATR is designed to support only pure categorial grammars. It does not support the use of explicit phrase structure rules, thus C- PATR is not an exhaustive implementation of PATR.

2.2 Categorial grammars as feature structures

A categorial grammar represents syntactic relations in a completely lexical fashion, i.e. without explicit phrase structure rules. Lexical items belong to *basic* or *functor* categories. A basic category is inert, in that it does not seek to combine with other categories. Functor categories perform the bulk of the work by actively seeking to combine with other categories. A functor category specifies the category of its *argument*, a *direction* in which to search for the argument, and the category of the *result* that is produced by applying the functor to its argument. With only this simple machinery, it is possible to describe a wide range of syntactic phenomena.

In C-PATR, basic categories are those with *NONE* as the value of the *argument* attribute. (*NONE* is a regular atomic value that is given special status by the parser.) Functor categories must have values specified for the *argument*, *direction*, and *result* attributes (see Figure 1).

The parsing algorithm manages the formation of constituents through the application of functors to their arguments [see section 3]. The *argument* and *result* attributes can contain information other than simple category designations. For example, the sample grammar in the appendix uses these slots to place constraints on the argument, to pass information from the argument to the functor, and to construct a semantic representation.

Figure 1: Traditional categorial descriptions of Noun (basic) and V-intrans (a functor)

3 Unification and Parsing Algorithms

C-PATR offers two varieties of unification. A *standard unification* algorithm (adapted from D-PATR [1]) is used in creating the internal representation of a grammar, while a more complex algorithm featuring *list unification* [see below] is employed by the parser. The parser itself is a fairly standard *active chart parser* (also adapted from D-PATR).

3.1 Optimizing parsing and unification

Function application is the only compositional technique used by C-PATRs parser. More powerful techniques such as functional composition and type-raising are not used. In parsing a non-trivial sentence, hundreds of unifications are attempted, hence the data types and algorithms that C-PATR employs during unification must be optimized in order to achieve efficient parsing. In order to perform quick comparisons while keeping symbol names readily available, a symbol in C-PATR is designated to be the location in memory of its print name, maintained on a letter tree, where each unique symbol-name has only one entry.

3.2 List unification

Merging partial information by unification is not sufficient for the description of all the correspondences between syntactic and semantic representation. A case in point is the semantics of conjoined noun phrases [2]. An appropriate semantic representation for a sentence like *b and c are small* is aconjoined formula, *small(b)* $\wedge$ *small(c)*. Such representations cannot be derived by pure unification because two instances of the logical predicate *small* with different arguments must be produced from a single instance of the word *small*. The same difficulty arises with reciprocal pronouns (*each other*) and numeral determiners. C-PATR solves this problem by extending unification to *list values*, with an effect that is similar to abstraction and lambda conversion in logic. For example, a conjoined noun phrase, such as *b and c*, may require that the verb phrase it combines with has a list-valued semantic representation. If the verb phrase, such as *are small*, is not of that type, the unifier simply coerces the argument to a list value thereby producing two copies of its semantic translation.

The algorithm for list unification is quite straightforward. (1) Two lists can be unified if they have the same number of elements, and if each corresponding pair of elements is unifiable. (2) Two lists of unequal lengths are not unifiable. (3) To unify a list of length **n** with a simple DG (non-list), coerce the non-list into a list by making **n** copies of the non-list, unifying each instance the non-list with a successive element of the list. (4) If any single sub-unification fails, then the whole unification fails. In our system, list values are represented as feature structures using the special attributes *first* and *rest* (analogous to CAR and CDR in Lisp).

3.3 Chart Parser

C-PATRs chart parser is a simplified version of general chart parsing algorithm. In a categorial grammar, all constituents are formed from two pieces (a functor and an argument), thus the parser need only consider binary rules.

The parser includes a subsumption filter [1]. Just before an edge is added to the

chart, the filter checks if there are any identical edges spanning the same nodes as the candidate edge. If there are any such edges, then the duplicate edge is not placed on the chart. Subsumption checking eliminates redundant analyses, and improves parsing efficiency for grammars that have many different ways to reach the same analysis. When a more complete parsing record is desired, the subsumption filter can be toggled off.

4 Special Features

4.1 Hierarchical lexicon design

C-PATR allows the user to specify a grammar in terms of a hierarchical system of templates. The grammar is divided into two parts, a set of templates and a set of lexical entries. Each template consists of a name (designated by an @-sign) followed by a set of explicit path equations and references to other templates [see Appendix A]. The path equations are compiled into directed graphs. When a template is referred to within another template definition the latter inherits the path equations of the former. The sample grammar makes use of template inheritance in the entries for **@Vtrans**, **@Ga**, and **@O** [see Appendix]. A template can also be used in a path equation (as in the sample grammar's entries for **@V\Vstem** and **@Particle**) to define a complex value.

The format of the lexicon file is identical to that of the template file except that the labels for lexical entries do not begin with @-signs. While a number of path equations usually constitute the body of a template, a typical lexical entry contains few explicit path equations. If a set of templates is well constructed, the list of template names mentioned in a lexical entry constitutes a meaningful high-level description of the word. [see Appendix B]. Path equations mentioned in a lexical entry should describe only the idiosyncratic properties of the word. The form of the entry is automatically assigned to the attribute *lex* unless specified otherwise.

4.2 Interactive grammar debugging and lexicon compiling

In designing a grammar, the user specifies templates or expanded lexical entries within a text file. C-PATR then compiles the text into an internal representation for the parser. This compilation task has been optimized to allow for reasonable interactive grammar development and debugging on small personal computers. On a Sun- 4, a 100K source grammar compiles into a 140K binary form in 5 seconds. On a Mac-II, the same task takes 30 seconds. To improve the grammar loading efficiency on the Macintosh, C-PATR provides a facility for pre-compiling the grammar. The Mac resource file created by pre-compilation loads in less than 2 seconds.

4.3 Services provided by C-PATR

C-PATR is driven by single character commands. These are summarized in Figure 2:

Type a sentence to parse or:
- *n* to see contents of edge number *n*
- b to run a batch test
- f to toggle subsumption filter
- l to view lexical entries for a word
- m to view a micro-dump of chart
- l to load a new lexicon
- o to specify an output file
- p to review phrase that was parsed
- q to quit
- t to toggle result print format
- s to view a short dump of chart
- t to view logical translation(s)
- u to unify two arbitrary edges
- v to toggle variable style
- w to list words
- x to view extra long chart dump
- z to zap expanded lexicon to a file

Figure 2: C-PATR command summary

5 Conclusion

C-PATR has advantages in size, speed, and portability over its predecessors. By choosing C as our implementation language, we gained in all three areas. Earlier PATR implementations, written in Lisp and Prolog, require the high overhead of an interpreter. C- PATRs 135k of source code compiles into a 58k stand-alone application on the Mac, and an 82k stand-alone on the Sun-4. C-PATR is an order of magnitude faster than D-PATR. C-PATR has been compiled on the Macintosh and on various Unix systems.

There are currently plans to enhance C-PATRs existing syntactic component with a two-level morphological analyzer [3]. The sample grammars treatment of *yonda* [see Appendix] is an example of how one might make use of morphologically analyzed forms.

C-PATR is available through the Center for the Study of Language and Information at Stanford.

Acknowledgements

Thanks to the Center for the Study of Language and Information and the Symbolic Systems Program for their generous support of this project. Also, thanks to Dorit Benshalom for offering many valuable suggestions that directly influenced the design of C-PATR.

Bibliography

[1] Karttunen, Lauri, D-PATR, *A development environment for unification-based grammars*, Report No. CSLI-86-81, Center for the Study of Language and Information, Stanford, California, 1986.

[2] Karttunen, Lauri, *Translating from English to Logic in Tarski's World*, In the Proceedings of of ROCLING-II, September 22-24, Sun- Moon Lake, Taiwan, 1989, pp 43-72.

[3] Koskenniemi, Kimmo, *Two-Level Morphology: A General Computational Model for Word-Form Recognition and Production*, Publications No. 11, Department of General Linguistics, University of Helsinki, Helsinki, Finland, 1983.

[4] Shieber, Stuart, *An Introduction to Unification-based Approaches to Grammar*, CSLI Lecture Note Series, Volume 4, Chicago University Press, Chicago, Illinois, 1986.

[5] Shieber, Stuart, *Parsing and Type Inference for Natural and Computer Languages*, Technical Note 460, Stanford Research International, Menlo Park, California, 1989.

Appendix: Grammar for a fragment of Japanese created in C-PATR

A Templates for Japanese

```
@Basic
    <argument> = NONE.

@Functor-left
    <direction> = left.

@Functor-right
    <direction> = right.
@V
    @Basic
    <cat> = Vstem
    <semantics pred> = <lex> .

@Vtrans
    @V
    <syntax ga> = <semantics agent>
    <syntax o> = <semantics theme> .
```

```
@V\Vstem
    @Functor-left
    <cat> = V\Vstem
    <argument cat> = Vstem
    <result> = @Basic
    <result cat> = V
    <result morphology> = <morphology>
    <result syntax> = <argument syntax>
    <result semantics> =
        <argument semantics> .

@Past <morphology tense> = past.

@Informal <morphology level> = informal.

@Noun
    @Basic
    <cat> = N
    <semantics ind> = <lex> .

@Particle
    @Functor-left
    <cat> = Particle
    <argument cat> = N
    <result cat> = NP
    <result> = @Functor-right
    <result argument cat> = V
    <result result> = @Basic
    <result result cat> =
        <result argument cat>
    <result result semantics> =
        <result argument semantics>
    <result result morphology> =
        <result argument morphology> .

@Ga
    @Particle
    <result argument syntax ga> =
        < argument semantics ind>
    <result result syntax ga> = filled
    <result result syntax o> =
        <result argument syntax o> .
@O
    @Particle
    <result argument syntax o> =
        <argument semantics ind>
    <result result syntax ga> =
        <result argument syntax ga>
    <result result syntax o> = filled.
```

B Unexpanded lexical entries

```
john @Noun.
hon @Noun.
ga @Ga.
o @O.

yom
    @Vtrans
    <lex> = yomu.

-ta
    @V\Vstem
    @Past
    @Informal.
```

C Sample expanded entry for the particle *ga*

```
[
cat:Particle
argument:[ cat:N
           semantics:[ ind:#1 ] ]
direction:left
result:  [ cat:NP
           argument:[ cat:V
                      morphology:#2
                      syntax:[ ga:#1
                               o:#3 ]
                      semantics:#4 ]
           direction:right
           result:  [ cat:V
                      morphology:#2
                      syntax:[ ga:filled
                               o:#3 ]
                      semantics:#4 ]
           argument:NONE ]
lex:ga
]
```

D Sample C-PATR session

Welcome to C-PATR!

lexicon type:
1. templates (.tem file)
2. expanded lexicon (.xlx file)
-->1
What is the template file? **coling.tem**
What is the lexicon file? **coling.lex**

Loading attribute ranking.....done

– templates –
@Basic
@Functor-left
@Functor-right
@V
@Vtrans
@V\Vstem
@Past
@Informal
@Noun
@Particle
@Ga
@O

– lexical items –
john
hon
ga
o
yom
-ta

>john ga hon o yom -ta
[*john read a book.* Note that yonda has been morphologically analyzed.]
john ga hon o yom -ta
number of parses: 1
0.100 seconds
11 edges, 31 dgs, 79 avs

>m
[C-PATR command to list the span of each edge]
0. john
1. ga
2. john ga
3. hon
4. o
5. hon o
6. yom
7. -ta
8. yom -ta
9. hon o yom -ta
10. john ga hon o yom -ta

>10
[C-PATR command to display edge #10, which contains the parse]

content:
[cat:V
 morphology:[level:informal
             tense:past]
 syntax:[ga:filled
         o:filled]
 semantics:[pred:yomu
            agent:john
            theme:hon]
 argument:NONE]

parse tree:
V[NP[N<john>
     Particle<ga>]
  V[NP[N<hon>
       Particle<o>]
    V[Vstem<yom>
      V\Vstem<-ta>]]]

>q

bye!