P-PATR: A COMPILER FOR UNIFICATION-BASED GRAMMARS

Technical Note 449

September 15, 1988

By: Susan Beth Hirsh
Artificial Intelligence Center
Computer and Information Sciences Division

This research was supported by the Department of the Navy under Contract N00039-84-C-0524 with the Space and Naval Warfare Systems Command.

333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

P-PATR: A Compiler for Unification-Based Grammars

Susan Beth Hirsh

September 15, 1988

Contents

Preface

I owe a great deal to many people, both for this thesis and for my mental well-being. Of course, my thanks go to Lauri Karttunen. The fact that this document can be understood by anyone other than myself is due to his diligent dissection of the presentation. I am also grateful to Fernando Pereira for his wonderfully responsive answers to my never-ending questions. His enthusiasm was quite infectious and kept me going when things looked bleak. I am deeply indebted to Ivan Sag and Carl Pollard, members of my moral-support team, for keeping me from giving up in times of crisis.

However, the person to whom I owe my most heartfelt thanks is Stuart Shieber, head of my moral-support team. The existence of this document is to a large part due to his encouragement and technical guidance.

Finally, I must thank my family, an important part of my life. The love and support of my parents is a constant comfort. My brother, Haym Hirsh, has also contributed to the completion of this work, from the pictures in the text to his reassuring presence on the other end of the phone when I call.

This research was supported by the Department of the Navy under Contract N00039-84-C-0524 with the Space and Naval Warfare Systems Command.

The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advance Research Projects Agency, or the United States Government.

1 Introduction and Motivations

P-PATR is a compiler for unification-based grammars that is written in Quintus Prolog running on a Sun 2 workstation. P-PATR is based on the PATR-II[1] formalism [14] developed at SRI International. PATR is a simple, unification-based formalism capable of encoding a wide variety of grammars. As a result of this versatility, several parsing systems and development environments based on this formalism have been implemented [18,5]. P-PATR is one such system, designed in response to the slow parse times of most of the other PATR implementations.

Most of the currently running PATR systems operate by *interpreting* a PATR grammar. P-PATR differs from these systems by *compiling* the grammar into a Prolog definite-clause grammar (DCG) [8].

The compilation is done only once for a given grammar; the resulting DCG contains all the information in the original PATR grammar in a form readily conducive to parsing. The advantage of compilation is that less work needs to be done during parsing, as some of the necessary computations have already been performed in the compilation phase.

The use of Prolog as the target language of the compiler is advantageous for two reasons. First, like PATR, Prolog uses unification as its method of operation. By compiling the PATR grammar into Prolog, P-PATR takes advantage of the efficient implementation of Prolog unification. Second, the performance of the resulting DCG can be improved further by compiling it with a Prolog compiler.

This compilation, combined with the use of Prolog, gives P-PATR a speed advantage over the other currently implemented PATR systems.

The rest of this paper is divided into three parts. The first section discusses the basic algorithm used in compiling the PATR grammar into a Prolog DCG. The second section consists of a detailed description of the actual procedure followed during the compilation. The appendix contains a user's manual for the P-PATR system as well as a sample grammar and some selected Prolog code from the system.

[1] Henceforth referred to simply as PATR.

2 Methods

What follows is a detailed explanation of the techniques used in compiling a PATR grammar into a Prolog DCG. First, an explanation of the general mechanisms used in compiling a PATR grammar into a DCG is given. This compilation scheme is then refined so that the DCG produced is equivalent to the original PATR grammar.

2.1 Feature Structures as Prolog Terms

In Prolog, unification operates on terms, not on PATR feature structures. It is therefore necessary to model PATR feature structures as Prolog terms to take advantage of the Prolog unification mechanism.

Prolog terms differ from PATR feature structures in two major ways [14]. First, in a Prolog term a value is identified by its position, while PATR feature structures identify a value by associating it with an attribute. For example, the two Prolog terms

```
head(agreement(number(plural), person(third)))
head(agreement(person(third), number(plural)))
```

do not unify. Because the order of the arguments is different, `number(plural)` is matched against `person(third)` and the unification fails. The second difference is that two Prolog terms unify only if they have the same number of arguments, whereas two PATR feature structures may unify even if they differ in the number of features. For example, the two terms

```
(1) head(agreement(number(plural), person(third)))
(2) head(agreement(number(plural)))
```

do not unify because the arities do not match. Thus, in representing a feature structure as a Prolog term, each structure must be given a fixed order and arity.

There are two methods generally used in modeling feature structures as Prolog terms. They will be referred to as *tailing* and *feature precompilation*.

- *Tailing*

  The first method for converting feature structures to Prolog terms involves the use of tail variables. Each feature structure is encoded as a Prolog term of the form[2]

[2]The Prolog list notation is used to represent a list with an uninstantiated tail variable [3].

```
[feature1:  value1, ...  , featureN: valueN | T],
```

where an uninstantiated tail variable is placed at the end of the list. Then, as this structure is unified with new structures, the features in the new structure are reordered in accordance with the features seen so far, and any new features are unified with the tail variable. For example, feature structures 1 and 2 are represented as the Prolog terms[3]

```
[head:  [agreement:  [number:  plural,
                      person:  third | T1] | T2] | T3]

[head:  [agreement:  [number:  plural | T4] | T5] | T6]
```

and then, when unified, `person:  third` unifies with the tail variable in the `agreement` list, producing the new Prolog term

```
[head:  [agreement:  [number:  plural,
                      person:  third | T1] | T2] | T3]
```

- *Feature Precompilation*

  The second conversion method involves a preliminary pass through the grammar to determine the arity and composition of all complex feature values. On the second pass, every attribute-value pair is placed in the correct position and order with respect to the other features. If a feature is missing from the structure, an uninstantiated variable is inserted in its place. For example, from feature structures 1 and 2 the following information is extracted

  > `head` can be followed by the feature `agreement`, and
  > `agreement` can be followed by the two features,
  > `number` and `person`, in that order.

  These feature structures are converted into the Prolog terms[4]

```
[head:  [agreement:  [number:  plural,
                      person:  third]]]
[head:  [agreement:  [number:  plural,
                      person:  X]]],
```

[3]This is not quite accurate. Throughout this paper feature structures are represented by labeling the values with the attributes they represent, but only the values of the attributes are actually present in the feature structures. The attributes are included for readability only.

[4]Variables are distinguished from atoms by an initial capital letter.

where the missing person value in feature structure 2 is represented by the uninstantiated variable X. The two Prolog terms now unify successfully to

```
[head:  [agreement:  [number:  plural,
                      person:  third]]]
```

P-PATR uses the feature precompilation method described above in encoding the feature structures associated with the PATR grammar entries as Prolog terms. When the unification list of a rule is processed during compilation, this feature information is used in creating the feature structures. For example, given the information extracted from feature structures 1 and 2, the unification

```
<X head agreement person> = <Y head agreement person>
```

produces the following feature structures for X and Y

```
[head:  [agreement:  [person:  A, number:  B]]]
[head:  [agreement:  [person:  A, number:  D]]],
```

where the values of the person attribute are unified and the indeterminate values for number are added to complete the agreement features.

2.2 Basic Compilation

The compilation produces a DCG that has a one-to-one correspondence with the original PATR grammar.

- *Grammar Rules*

  PATR grammar rules consist of a context-free phrase structure (CFPS) rule augmented with a list of unifications. For example

```
S → NP VP:
     <S head> = <VP head>
     <VP head agreement> = <NP head agreement>.
```

  The CFPS part of the rule is

```
S → NP VP
```

and the unifications give the added information that the agreement features of the VP and the NP must be the same.

DCGs are a natural extension of context-free grammars (CFG); a straightforward translation scheme is given by Pereira [7]. The constituents of a DCG rule may be complex symbols, consisting of a functor and a list of arguments. In the translation of a PATR rule to a DCG rule, the CFPS part of the rule provides the functors of the DCG rule, while the feature structure information from the unifications is encoded as the arguments to these functors. For example, the grammar rule just presented is equivalent to the DCG rule[5]

```
s([head:  [agreement:  Y]]) -->
          np([head:  [agreement:  Y]]),
          vp([head:  [agreement:  Y]]).
```

- *Lexical Entries*

  PATR lexical entries consist of a word followed by a list of unifications. For example

  ```
  Word Uther:
       <cat> = NP
       <head agreement person> = third
       <head agreement number> = singular.
  ```

  This entry defines the word "Uther"; the unifications encode the information that "Uther" is a third-person singular NP.

  In translating a PATR lexical entry into a DCG rule, the category of the word becomes the functor for the left-hand side (LHS) of the rule; its argument list is derived from the list of unifications. The right-hand side (RHS) of the rule consists of the word itself. For example, the above lexical entry is equivalent to the DCG rule

  ```
  np([head:  [agreement:  [person:  third,
                           number:  singular]]]) -->
    [uther].
  ```

The example just presented shows a very simple correspondence between the PATR and the DCG formalisms. For reasons explained in the next sections, P-PATR actually uses a more complex compilation technique.

[5]Reentrancy is represented by sharing variables.

2.3 Left-Corner Parsing

The default parsing algorithm for DCGs supplied by Prolog is a top-down, left-to-right, backtracking algorithm. A well-known problem with top-down parsers is that left-recursive grammars can cause them to go into an infinite loop [1]. Because PATR rules are allowed to be left recursive, a compilation technique must be applied that enables the Prolog DCG to handle such rules.

P-PATR compiles a PATR grammar into a DCG that uses a bottom-up parsing algorithm. Bottom-up parsers have no problem with left recursion [1]. The particular parsing technique used is called left-corner parsing [11].

The left corner (LC) of a CFG rule is the first symbol of the right-hand side of the rule. For example, the LC of the rule

```
S → NP VP
```

is the nonterminal NP. In LC parsing, each rule is identified through its LC. The first word in the sentence functions as the initial LC key. The rules whose LC match the key are extracted. The next word in the sentence becomes the new LC key for satisfying the remainder of the right-hand side of these rules. If the right-hand side of the rule is completely satisfied, the left-hand side of the rule is substituted for the LC key and the process is iterated. For example, given the CFG rules

```
(3) S → NP VP
(4) VP → V
(5) NP→ N
(6) V → sleeps
(7) N → Bill,
```

the sentence "Bill sleeps" is parsed as follows:

```
LC key = ''Bill''
matches the LC of Rule 7,
Rule 7 is satisfied

LC key = N
matches the LC of Rule 5,
Rule 5 is satisfied

LC key = NP
matches the LC of Rule 3,
leaving the VP of Rule 3 to be satisfied
```

```
LC key = ''sleeps''
matches the LC of Rule 6,
Rule 6 is satisfied

LC key = V
matches the LC of Rule 4,
Rule 4 is satisfied

Rule 3 is satisfied,
no more input,
parse successful
```

This parsing algorithm avoids the problem of left-recursive rules.[6]

The DCG produced by P-PATR is based on the implementation of the LC algorithm in Matsumoto et al. [6]. Each PATR grammar rule of the type

$$LHS \rightarrow RHS_1 \ldots RHS_n$$

is converted into a DCG rule of the form

```
lc(RHS1, Root) -->
  down(RHS2), ...   , down(RHSN),
  lc(LHS, Root),
```

where `LHS` and `RHS1` through `RHSN` constitute the feature structure information from the unification list of the rule and *Root* is the feature structure of the constituent currently being parsed. In the limit case, *Root* and *LHS* are the same: everything is its own LC.

```
lc(Root, Root) --> [].
```

For example, consider a CFG for noun phrases consisting of a rule

```
NP → Det N
```

and two lexical items: `the:Det` and `girl:N`. The corresponding DCG rule produced by P-PATR is

```
lc(det, Root) -->
  down(n),
  lc(np, Root).
```

[6]Epsilon rules still pose a problem, but they are taken care of separately (Section 2.4).

To understand how this rule is used by the Prolog parser, we first need to define the predicates down and leaf:

```
down(Cat) -->
    leaf(Child),
    lc(Child, Cat).

leaf(Child) -->
    [Word],
    {lex(Word, Child)}.
```

The two words in the grammar are defined by the following Prolog clauses

```
lex(the, det)
lex(girl, n)
```

Let us now see how the string “the girl” is parsed as an NP by using this DCG version of the original CFG. The parse is initiated with the call

```
down(np).
```

This results in the call

```
leaf(Child),
```

which consumes the word “the” and binds the variable *Child* to the word's category det. The next step is to evaluate the call

```
lc(det, np)
```

by finding a match for this clause among the LC rules and satisfying the right-hand side of the LC rule. In this case, we need to satisfy the calls

```
down(n)
lc(np, np)
```

The first clause triggers another call to

```
leaf(Child),
```

which now consumes the word “girl” and binds *Child* to n and the call

```
lc(n, n),
```

which is immediately satisfied because it is an instance of the rule

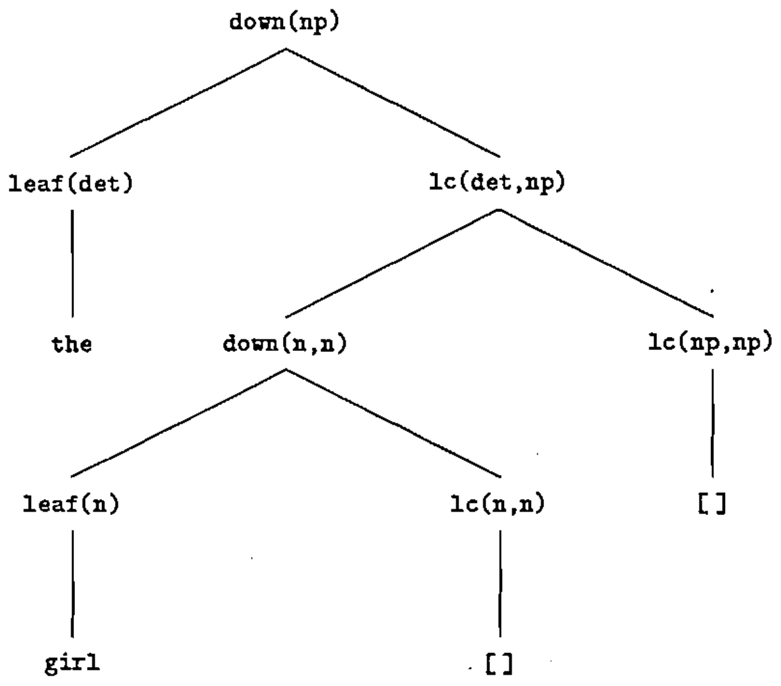
```
        lc(Root, Root) --> [].
```

leaving the call

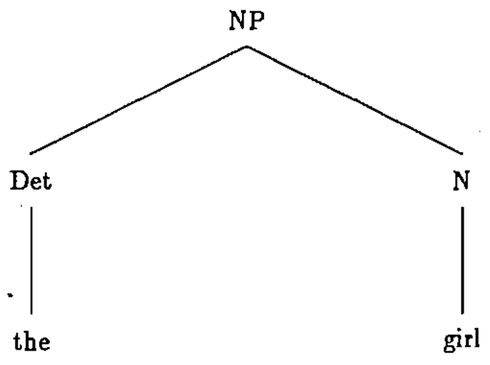
```
        lc(np, np)
```

to be satisfied in the same way.

The flow of the computation can be pictured as the tree



This obviously differs from the usual parse tree,



because of the way the LC algorithm uses the rules. The standard phrase-structure tree can easily be produced as a side effect of the parse, if desired.

The above discussion is an oversimplification. In actuality, the values of the variables `Root`, `Cat` and `Child` are feature structures rather than atomic category symbols. For example, the grammar rule presented above becomes the new DCG rule[7]

```
lc([cat:  np, head:  [agreement:  Y]], Root) -->
  down([cat:  vp, head:  [agreement:  Y]]),
  lc([cat:  s, head:  [agreement:  Y]]), Root),
```

and the lexical entry for "Uther" becomes the Prolog clause

```
lex(uther, [cat:  np,
            head:  [agreement:  [person:  third,
                                 number:  singular]]]).
```

A PATR grammar is compiled into a DCG of the form just presented. The compilation technique is revised slightly in the next section to allow for the epsilon rules that produce empty constituents.

2.4 Epsilon Rules

Epsilon rules in a CFG are of the form

$$A \rightarrow \epsilon$$

[7]This occurs after feature information corresponding to the categories of the nonterminals is added to the feature structures (Section 3.1.2).

This type of rule can pose a problem in applying the compilation technique described above. In LC parsing, a rule is keyed by its left corner. If the LC of a rule can be expanded to an empty string, the rule in effect acquires a second left corner.

For example, consider the rules

(8) $\mathtt{A} \rightarrow \mathtt{B\ A}$
(9) $\mathtt{B} \rightarrow \epsilon$

Because B can be expanded by rule 9 to an empty string, rule 8 has two left corners: B and A. For the compilation technique described above to work, each possible LC has to be recognized before a rule is compiled.

The problem is solved in two stages. First, all epsilon rules are extracted from the grammar and put into a separate list. Then all of the remaining rules are examined one by one. If the LC of a rule

$LHS \rightarrow RHS_1 \ldots RHS_n$

can be null, a new rule of the form

$LHS \rightarrow RHS_2 \ldots RHS_n$

is added to the grammar and subjected to the same test. For example, rule 8 above gives rise to the new rule

$\mathtt{A} \rightarrow \mathtt{A}$

by virtue of the possible expansion of B in rule 8 by rule 9.

The technique outlined above is easily extended to PATR grammars. In a PATR grammar, an epsilon rule is of the form

```
A → :
      ⟨Definition⟩.
```

In eliminating the epsilon rules, the unification information must be taken into account. For example, for the PATR grammar rules

```
(10) A → B A:
          <A feature1> = value1
          <B feature2> = <A feature2>

(11) B → :
          <B feature2> = value2,
```

rule 10 gives rise to the new rule

```
A → A:
     <A feature1> = value1
     <A feature2> = value2.
```

2.5 Lexical Organization

We now turn to *lexical templates* and *lexical rules*. Lexical templates are named feature structures and lexical rules are named transformations on feature structures. Both types of entries may include references to templates and rules in their definition. Because templates and rules may be referred to before they are defined, compilation takes place in two stages.

- *Compilation: First Stage*

  Each lexical entry of the PATR grammar is compiled into a temporary DCG rule of the form

  ```
  word(Word, FeatureStructure) :-- ...
  ```

  The right-hand side of a temporary DCG rule typically contains references to the lexical templates and lexical rules that occur in the entry. These references cannot be evaluated, however, until the first stage is completed. The references are of the form

  ```
  template(Name, In, Out)
  ```

  or

  ```
  lex_rule(Name, In, Out),
  ```

  where `In` is the input feature structure to a rule or template, and `Out` is the output feature structure from the rule or template.

  For example, the lexical entry

  ```
  Word Uther:
       noun
  ```

  is compiled into the temporary DCG rule

  ```
  word(uther, FeatureStructure):--
      template(noun, In, FeatureStructure).
  ```

- *Compilation: Second Stage*

  Once the first stage has been completed, the definitions of the lexical rules and lexical templates reside in the Prolog data base (Section 3.2.4). The temporary rules produced in the first stage of compilation could be used by the parser, but this would be inefficient because the lexical templates and rules would be executed each time they are referred to.

  At this point, each lexical entry is executed once by Prolog, evaluating the actions of the rules and templates, and the new feature structure produced is used in converting the entry to its final form.

  For example, the temporary DCG rule

  ```
  word(boy, FeatureStructure):--
      template(noun, In, FeatureStructure)
  ```

  produces the final DCG rule

  ```
  lex(boy, [cat:  n]))
  ```

  once it is executed.

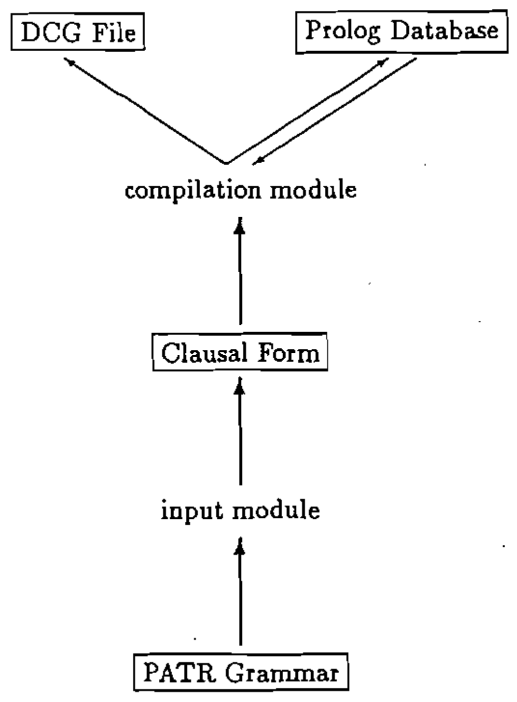


Figure 1: Flow Diagram of P-PATR

3 The P-PATR System

This section provides a step-by-step account of the compilation technique used by P-PATR. An overview of the process is given in Figure 1. As shown in the diagram, compilation is accomplished in two phases: *grammar input* and *grammar compilation*. The grammar input phase produces an intermediate representation of the PATR grammar that is used in the compilation. During compilation, information is both written to a file reserved for the output DCG and asserted into the Prolog data base. The information in the data base is accessed as the compilation proceeds.

3.1 Grammar Input

This phase takes a set of text files containing a PATR grammar and converts it to a Prolog clausal form used by later phases. The grammar is entered in two distinct steps: *tokenization* and *translation*.

3.1.1 Tokenization

Each entry in the PATR grammar is first tokenized and then translated into clausal form. There are six classes of tokens recognized by the P-PATR tokenizer: *identifiers*, *special characters*, *terminators*, *white-space characters*, *comments* and *strings*. Each token type is briefly described below.

- *Identifiers*

  Identifiers are tokens that consist of any alphanumeric characters: a-z, A-Z, and 0-9; and any special intraword characters: underbar (_), asterisk (*), apostrophe ('), questionmark (?), and backquote (`).

- *Special Characters*

  Special characters are tokens consisting of a single character: colon (:), number sign (#), slash (/), arrow (→), square brackets ([,]), angle brackets (<, >), braces ({, }), parentheses ((,)), comma (,), equal sign (=), or dash(-). A sequence of tokens consisting of a dash (-) and a right angle bracket (>) is treated as the single token: arrow (→).

- *Terminators*

  Terminators: period (.) and end_of_file, signal the end of a token stream. Terminators are considered a special case of special characters and are treated as the single tokens: period (.) and end_of_file.

- *White-space Characters*

  White-space characters: space, newline, tab and formfeed are ignored.

- *Comments*

  Comments, which begin when the single-token semicolon (;) is encountered and continue to the end of the line, are ignored.

- *Strings*

  Strings are any list of characters enclosed in double quotes ("). Embedding of double quotes inside a string is done by using a sequence of two double quotes ("").

In all tokens, except for strings, no case distinction is made. All characters are converted to lowercase. Any characters that are not legal in a P-PATR token are ignored and a warning is issued.

3.1.2 Translation

The stream of tokens produced by the tokenization process is now translated into clausal form. Each type of entry in the PATR grammar is translated into a form that will be most appropriate in subsequent compilation (Section 3.2), as follows:

Control Statements

The only type of control statement is the input statement. Input statements are of the form

```
Input ⟨InputFile⟩.
```

When an input statement is encountered during translation, the current input file is temporarily replaced by the file specified in the statement. Once this new file is completely read in, the old input file is restored.

For example, the input

```
Input 'testgram'.
```

causes the current input stream to become the file TESTGRAM.

Parameters

P-PATR recognizes two grammar-dependent parameters: *start symbol* and *attribute order*. These parameters are set by statements that must appear in the grammar before any rules or lexical items are encountered. Other parameters[8] are ignored.

The parameter statements are processed as follows:

- *Start Symbol*

  The start symbol is defined by a statement of the form

  ```
  Parameter:  Start Symbol is ⟨Symbol⟩.
  ```

  The start symbol for the grammar is recorded for use in further compilation as a clause of the form

  ```
  parameter(start(Symbol)).
  ```

[8]There are several other parameters that can be specified in a PATR grammar, but their information is not utilized by this implementation.

- *Attribute Order*

  Attribute order is specified as follows

  ```
  Parameter:  attribute order is ⟨Attributes⟩.
  ```

  This is converted to the Prolog clause

  ```
  parameter(attributes(List)),
  ```

  where `List` corresponds to a list of all attributes in the order specified. For example, the input

  ```
  Parameter:  attribute order is cat head.
  ```

  produces the clause

  ```
  parameter(attributes([cat, head])).
  ```

Grammar Rules

The format for PATR grammar rules is

```
Rule { ⟨Description⟩ }
     ⟨LHS⟩ → ⟨RHS⟩:
     ⟨Definition⟩.
```

In translating the rule into clausal form, all nonterminals are replaced by variables, which are used during compilation. Grammar rules are translated into a clause of the form

```
rule(LHS, RHS, Def)
```

where `LHS` is a variable associated with the left-hand side of a rule, `RHS` is a list of variables associated with the right-hand side of the rule, and `Def` is a list of specifications defining the rule.

In the original PATR grammar, the category information of a nonterminal can be omitted from the list of unifications because it is added automatically during grammar translation. For example, the grammar rule

```
Rule { sentence formation }
     S → NP VP:
          <S head> = <VP head>
          <S head form> = finite
          <VP subcat first> = <NP>
          <VP subcat rest> = end
```

produces the clause

```
rule(S, [NP, VP], [[S, cat] = s,
                   [NP, cat] = np,
                   [VP, cat] = vp,
                   [S, head] = [VP, head],
                   [S, head, form] = finite,
                   [VP, subcat, first] = [NP],
                   [VP, subcat, rest] = end]),
```

where the unification information

```
[S, cat] = s
[NP, cat] = np
[VP, cat] = vp
```

is added to the list of unifications. The only exception is the nonterminal X (with or without a subscript). If this appears in a grammar rule, no category information is added, thus allowing expressions of any category to appear in this position.

P-PATR follows the Z-PATR [18] convention for distinguishing among constituents that have the same category. This is accomplished by means of numeric tags. For example, if two constituents in the same rule are referred to as VP_1 and VP_2, they are both of category VP.

Lexical Items

Each type of lexical item in a PATR grammar is translated accordingly:

- *Lexical Entries*

  The format for lexical entries is

  Word ⟨*Word*⟩:
  ⟨*Definition*⟩.

  In translating a lexical entry into clausal form, the information from the original PATR entry is left unchanged. Thus, lexical entries are translated into clauses of the form

  ```
  lex(Word, Def),
  ```

where Word is a word being defined, and Def a list of specifications defining the word.

The system augments each lexical entry with two new features: lex and sense. It is assumed that the lexical entry does not already contain this information; otherwise it will be duplicated. The lex value for a lexical entry is the word itself. The sense value is the word concatenated with a number that specifies how many previous definitions of this word have occurred in this grammar. For example, given the entry

```
Word Uther:
     <cat> = NP
     <head agreement gender> = masculine
     <head agreement person> = third
     <head agreement number> = singular
     <head trans> = Uther,
```

P-PATR produces the clause

```
lex(uther, [[lex] = uther,
            [sense] = uther,
            [cat] = np,
            [head, agreement, gender] = masculine,
            [head, agreement, person] = third,
            [head, agreement, number] = singular,
            [head, trans] = uther]).
```

If there already exists one previous definition for the word "Uther", the value for the sense feature in the second definition would be uther2.

- *Lexical Templates*

  Lexical templates are of the form

  Let ⟨*Template*⟩ be
       ⟨*Definition*⟩.

  In translating a lexical template into clausal form, the information from the original PATR lexical template is left unchanged. Thus, lexical templates are translated into clauses of the form

  ```
  template(Name, Def),
  ```

  where Name is the name of a lexical template being defined, and Def a list of specifications defining the template.

  For example, the template

```
Let verb be
    <cat> = V.
```

produces the clause

```
template(verb, [[cat] = v]).
```

- *Lexical Rules*

  Lexical rules have the form

  Define ⟨*Rule*⟩ as
          ⟨*Definition*⟩.

  In the clausal-form encoding of the lexical rule, the in and out attributes are replaced by variables, which are used during compilation. Thus, lexical rules are translated into clauses of the form

  ```
  lex_rule(Name, In, Out, Def)
  ```

  where Name is the name of a lexical rule being defined, In is a variable associated the with the input to the rule, Out is a variable associated with the output of the rule, and Def is a list of specifications defining the rule.

  For example, the rule

  ```
  Define agentlesspassive as:
         <out cat> = <in cat>
         <out subcat> = <in subcat rest>
         <out head agreement> = <in head agreement>
         <out head aux> = <in head aux>
         <out head trans> = <in head trans>
         <out head form> = passiveparticiple.
  ```

  produces the clause

  ```
  lex_rule(agentlesspassive, In, Out,
          [[Out, cat] = [In, cat],
           [Out, subcat] = [In, subcat, rest],
           [Out, head, agreement] = [In, head, agreement],
           [Out, head, aux] = [In, head, aux],
           [Out, head, trans] = [In, head, trans],
           [Out, head, form] = passiveparticiple]).
  ```

3.2 Grammar Compilation

This phase takes a text file containing a PATR grammar in clausal form and compiles it into a Prolog DCG. Grammar compilation is accomplished in five distinct phases: *parameter processing*, *attribute position generation*, *epsilon precompilation*, *compilation*, and *lexical compilation*.

3.2.1 Parameter Processing

This phase processes the parameter statements specified in the PATR grammar. Parameter statements must occur first in the grammar to ensure their use in the entire compilation.

The two types of parameter statements are treated as follows:

- *Start Symbol*

  A statement of the form

  ```
  parameter(start(Symbol))
  ```

  is asserted into the Prolog data base and written to the DCG file as

  ```
  start(Symbol).
  ```

- *Attribute Order*

  The attribute order is initially represented in clausal form as

  ```
  parameter(attributes(List)),
  ```

  where `List` is a list of attributes with a specified order.

  For each attribute in the list, a clause is asserted into the Prolog data base specifying the order of that attribute. This information is used in maintaining the specified order during output of the feature structures.

  This information is asserted into the Prolog data base as

  ```
  print_order(Attribute, Position),
  ```

  where `Attribute` is an attribute from the list of attributes, and Position is the position of that attribute in the list of attributes.

  For example, given the parameter statement

  ```
  parameter(attributes([cat, head]),
  ```

the clauses

```
print_order(cat, 1)
print_order(head, 2)
```

are asserted into the Prolog data base.

3.2.2 Attribute Position Generation

In PATR, features are pairs of attributes and values. The value of an attribute can be one of three types: indeterminate, atomic, and complex. A complex value is a set of attribute-value pairs. In the following discussion only the complex values contribute information about the attributes; therefore, the other types of values are not discussed.

This phase computes the arity of each complex attribute value and places the features in a fixed order. The information is used in the conversion of PATR feature structures to Prolog terms (Section 2.1).

For each attribute in a PATR grammar, a list is compiled that consists of all the attributes that can follow that attribute in a path specification. For example, given the lexical template

```
template(singular, [[head, agreement, number] = singular]),
```

the information recorded for the attribute `head` is that it can be followed by `agreement` in a path specification. The information that the attribute `agreement` can be followed by `number` is also recorded.

Once all information on the attributes has been compiled, this information is translated into clausal form and is asserted into the Prolog data base and written to the DCG file as

```
feature_order(Attribute, Features, Variables)
```

where `Attribute` is the attribute currently being described, `Features` is a list of pairs consisting of an attribute and a unique variable representing the value of that attribute, and `Variables` is a list of the variables in Features.

For example, from the above template the following clauses are generated and asserted into the Prolog data base:

```
feature_order(main, [head:X], [X])
feature_order(head, [agreement:Y], [Y])
feature_order(agreement, [number:Z], [Z]).
```

A dummy attribute `main` is created to notate those features that can occur as the first feature in a path specification.

Since the list `Features` is used during the output of the feature structures, the order of the attributes must reflect the order specified in the parameter statement. Thus, the list is reordered to reflect the specified order. Any attributes whose order is not determined are just added to the end of the list of features.

3.2.3 Epsilon Precompilation

This pass through the PATR grammar precompiles epsilon rules.

Epsilon rules are represented in clausal form as

```
rule(LHS, [], Def),
```

where the grammar rule has no right-hand side. All other grammar entries are ignored during this pass.

An epsilon rule is compiled into a DCG rule by applying the unification equations attached to the rule, thereby producing a feature structure associated with the rule (Section 2.1). The compiled epsilon rule is then asserted into the Prolog data base and written to the DCG file as

```
null(LHS)
```

where `LHS` is the feature structure associated with the rule.

For example, the epsilon rule

```
rule(Det, [], [[Det, head, agreement, number] = plural])
```

is outputted as

```
null([cat: det, head: [agreement: [number: plural]]]).
```

3.2.4 Compilation

This pass through the PATR grammar uses the information produced in the previous phases to generate a DCG rule for each grammar entry. These DCG rules are written to the DCG file (grammar rules) or recorded in the data base to be further processed during the second compilation stage (lexical items).

Each type of grammar entry is compiled into a DCG rule as follows:

Grammar Rules

All of the unification equations in the grammar rule are applied (Section 2.1), producing the feature structures associated with the rule. For example, the LHS and RHS variables of the rule

```
rule(VP, [V],[[VP, cat] = vp,
              [V, cat] = v,
              [VP, head] = [V, head],
              [VP, subcat] = [V, subcat]])
```

become

```
VP becomes [cat:  vp
            head:  X
            subcat:  Y]

V becomes  [cat:  v
            head:  X
            subcat:  Y].
```

These feature structures, together with the rule itself, are now compiled into a DCG rule in left-corner format (Section 2.3).

At this point, the solution to the problem caused by epsilon rules is applied (Section 2.4). As a result, one rule may expand to a set of rules. These rules are written to the DCG file in a form that is slightly more complex than that presented in Section 2.3

```
lc(RHS1, Parent, Branch1, Tree) -->
         down(RHS2, Branch2), ...  , down(RHSN, BranchN),
         lc(LHS, Parent, NewTree, Tree)
```

where RHS1 through RHSN are the feature structures associated with the right-hand side of the rule, Parent is the feature structure associated with the left-hand side of the rule, Branch1 through BranchN are the parse trees associated with the right-hand side of the rule, Tree is the parse tree associated with the left-hand side of the rule, and NewTree is the parse tree associated with the entire rule.

For example, the rule presented above becomes the DCG rule

```
lc([cat:  v,
    head:  X,
    subcat:  Y],
   Parent, Branch1, Tree) -->
lc([cat:  vp,
    head:  X,
    subcat:  Y],
   Parent, vp(Branch1), Tree).
```

Lexical Items

Each lexical item in the grammar is compiled into a DCG rule. These rules, unlike grammar rules, are not written directly to the DCG file. They are asserted into the Prolog data base to be compiled and written to the DCG file in a later stage.

Each type of lexical item is asserted into the Prolog data base with a different functor but they are processed in the same way. First, all of the specifications in the definition are processed. If a specification is a unification, it is applied (Section 2.1); if it is a reference to a lexical rule or lexical template, the reference is put into the form

```
template(Name, In, Out)
```

or

```
lex_rule(Name, In, Out),
```

where `In` is the input feature structure to a rule or template, and `Out` is the output feature structure from the rule or template. These references are expanded in the second compilation phase.

- *Lexical Entries*

  Lexical entries are asserted into the Prolog data base in the form

  ```
  word(Word, FeatureStructure):--
      Def.
  ```

  where `Word` is the name of a lexical entry, `FeatureStructure` is the feature structure associated with the lexical entry, and `Def` includes references to rules and templates producing `FeatureStructure`.

  For example, the lexical entry

  ```
  lex(uther, [[lex = uther], [sense = uther1], noun])
  ```

is compiled into

```
        word(uther, FeatureStructure): --
           template(noun, [lex:  uther, sense:  uther1],
                     FeatureStructure).
```

- *Lexical Templates*

  Lexical templates are asserted into the Prolog data base in the form

```
        template(Name, FeatureStructure):--
               Def.
```

  where Name is the name of a lexical template, FeatureStructure is the feature structure associated with the template, and Def includes references to rules and templates producing FeatureStructure.

  For example, the lexical template

```
        template(mainverb, [[head, aux = false], verb])
```

  is compiled into

```
        template(mainverb, FeatureStructure):--
               template(verb, [head:  [aux:  false]],
                      FeatureStructure).
```

- *Lexical Rules*

  Unlike lexical entries and lexical templates, lexical rules are not allowed to contain references to rules or templates in their definition. Thus, lexical rules are asserted into the Prolog data base in the form

```
lex_rule(Name, In, Out).
```

  where Name is the name of a lexical rule, In is the feature structure associated with the input to the rule, and Out is the feature structure associated with the output from the rule.

  For example, the lexical rule

```
        lex_rule(nom, [[Out, head] = [In, head], [Out, cat] = n])
```

  is compiled into

```
        lex_rule(nom, [cat:  v, head:  X], [cat:  n, head:  X]).
```

3.2.5 Lexical Compilation

Lexical entries are initially compiled into DCG rules with explicit calls to the templates and lexical rules they utilize. Because these calls are re-executed each time they are encountered, the system would be inefficient to use. At the second stage of compilation, these references are eliminated by merging the corresponding feature structures with the rest of the definition.

Once this process is completed, the DCG rules for the lexical entries no longer contain any references to lexical templates or rules; therefore, the rules and templates need not be recorded in the DCG file.

The new lexical entries are written to the DCG file as

```
lex(Word, FeatureStructure).
```

For example, the initial DCG rules

```
word(boy, Y):—
     template(noun, X, Y).
template(noun, X, Y):--
         Y = [cat:  n].
```

produce the new DCG rule

```
lex(boy, [cat:  n]).
```

Sentence	Parse time (in seconds)
Uther sleeps	0.066
Uther storms Cornwall	0.067
Knights sleep	0.084
Cornwall is stormed	0.1
A knight storms Cornwall	0.1

Table 1: Execution Statistics

4 Conclusions

To test whether the P-PATR system lives up to the expectations that motivated its development, it will be necessary to compare it with the two other currently running PATR systems: D-PATR [5] and Z-PATR [18]. Because of disparities in the versions of the PATR formalism assumed by each system, accurate statistics are not presently available, but the preliminary results seem promising.

Sample execution statistics can be seen in Table 1. These are the execution results from the DCG produced by P-PATR, using as input the grammar in Section B. It is easy to see from these statistics that a DCG produced by P-PATR is a speedy parsing tool.

4.1 Further Work

P-PATR is far from complete. Changes are being made to improve the system's performance and expand its capabilities. These enhancements include the following:

- *Improved Parser Performance*

  Because Prolog uses a depth-first control strategy, a DCG generates the first parse for a sentence quickly, but when all parses must be produced, the necessary backtracking slows the parse down significantly. To solve this problem, predictive [9] capabilities will be added to P-PATR to eliminate some of the superfluous backtracking so that all parses can be found faster.

- *Compatibility with the Other PATR Systems*

  To allow better comparisons of performance, it would be desirable to be able to run the same grammar on P-PATR as on the other two systems discussed above. Some work is currently being done [16] on developing a standard specifying a single version of the PATR formalism to which all

PATR systems would conform. Once this is done, the same grammar can be used with equal ease on all PATR systems.

- *Morphological Analysis*

  P-PATR does not currently perform morphological analysis. For each form of a lexical entry in a PATR grammar, a separate entry in the grammar must be present. By encorporating the work being done on morphological analysis in the PATR framework [2] into P-PATR, only the stem forms of the lexical entries need be entered into the lexicon.

A User's Manual for the P-PATR System

A.1 Starting Up the System

To start P-PATR, load the file LOADPATR.PL into the Prolog data base.[9] This file loads the rest of the system and initializes all execution flags.

A.1.1 Loading Necessary Files

The P-PATR system consists of three basic modules: READPATR.PL, COMPILEPATR.PL and PATRLIBRARY.PL. Each of these modules is in turn divided further into submodules, which are loaded by their parent module. A complete list of all files that must reside in the Prolog data base for compilation to proceed is given below.

READPATR.PL

This module includes all files necessary for translating a PATR grammar into clausal form. The files are:

- READTOKENS.PL: Reads in a PATR rule and returns it as a list of tokens.
- READPATR.PL: Takes a list of tokens and translates it into clausal form.

COMPILEPATR.PL

This module includes all the files that are necessary in converting a clausal representation of a PATR grammar to a Prolog DCG. The files are

- COMPILEPATR.PL: Compiles a clausal form into a DCG.
- READRULES.PL: Reads in a list of PATR rules.
- PARAMETERS.PL: Records the information contained in the parameter statements.
- PATHS.PL: Compiles all information on the position and order of the features.

[9] Loading a file into Prolog involves either compiling or interpreting that file. The current implementation compiles these files, but the system could easily be modified to interpret them, if desired. The difference is that it takes longer to compile than to interpret a Prolog file, but a compiled file executes much faster.

- EPSILONS.PL: Preprocesses all epsilon rules.
- COMPILEGRAMMAR.PL: Performs the actual compilation of the grammar entries.
- UNIFY.PL: Applies the unification equations constraining a rule.
- COMPILELEX.PL: Compiles all lexical entries.

PATRLIBRARY.PL

This module consists of a single file that contains predicates common to all of the modules. The predicates included perform basic operations needed by the entire system.

A.1.2 Trace Flags

In LOADPATR.PL, there are four execution flags that can be toggled by the user:

- **`trace_input`** (default `no`): `Yes` prints out the clausal representation of each PATR rule as it is processed in the grammar input module.
- **`trace_paths`** (default `no`): `Yes` prints the feature information compiled during execution of the attribute position generation module.
- **`trace_rules`** (default `no`): `Yes` prints out each DCG rule as it is processed in the compilation module.
- **`load_parser`** (default `yes`): `No` suppresses the loading of the compiled DCG after compilation.

To change the values of any of the execution flags, the user must modify the values in LOADPATR.PL.[10]

A.2 Compiling a PATR Grammar

Once all of the necessary files reside in the Prolog data base, the system is ready for use.

[10]The values can also be changed later by means of the Prolog predicates **`abolish`** and **`assert`** [3].

A.2.1 Grammar Input

The file to be compiled must first be translated into clausal form by a call to the grammar input module. The calling sequence is

```
grammar(File),
```

where the name of the file to be compiled can be any Prolog atom or string [3].

The grammar input module then translates the file into clausal form and puts the output into a new file whose name is that of the initial file with the new file type extension PTRP. When the input module is invoked, it displays the message

```
''Reading ...''
```

and, once input is completed, the execution time (in seconds) of the input module is printed.

For example, the file DEMOGRAM.PATR is translated into clausal form through the call

```
grammar('demogram.patr'),
```

producing the new file DEMOGRAM.PTRP.

A.2.2 Grammar Compilation

Once the PATR grammar is in clausal form, it is compiled into a Prolog DCG by a call to the grammar compilation module. The calling sequence is

```
compilepatr(File),
```

where the file-type extension of the file name may be omitted, as it is assumed to have the extension PTRP.

The grammar compilation module then compiles that file into a DCG and puts the output into a new file whose name is that of the initial file with the new file-type extension DCG. When the compilation module is invoked it displays the following

```
''Compiling ...''
```

and, once compilation is completed, the execution time (in seconds) of the compilation module[11] is displayed.

For example, the file DEMOGRAM.PTRP is compiled through the call

```
compilepatr(demogram),
```

producing the new file DEMOGRAM.DCG.

A.3 Parsing a Sentence

A.3.1 Loading the DCG

Once the PATR grammar is compiled, the DCG file is loaded into the Prolog data base.[12] When loaded, the DCG file itself loads a support module PATRSUPPORT.PL containing additional predicates that are needed in parsing. PATRSUPPORT.PL also loads a submodule PP.PL that contains a feature structure pretty printer, as well as submodule READIN.PL that includes a sentence reader. In all, the files that must reside in the Prolog data base before parsing can proceed are

- File.DCG: DCG file produced by compilation module.
- PATRSUPPORT.PL: Support module for the parser.
- PP.PL: Feature structure pretty printer.
- READIN.PL: Sentence reader.

A.3.2 Sentence Parsing

Once all necessary files are loaded, sentences can be parsed by entering the statement

```
patr.
```

The parser is now ready for input.

[11]This is not completely accurate. The execution time of the compilation module is displayed to two steps. First, the execution time of the compilation itself is displayed and if the `load_parser` execution flag has been toggled on, a second execution time is displayed that corresponds to the loading time.

[12]This can be done by toggling an execution flag or by loading it manually into the Prolog data base.

Sentence Input

The input environment consists of an input loop for the sentences. Each sentence entered at the prompt ":" is parsed. End of input is signaled by the command "control-d" entered at the input prompt.

Parser Output

Once a sentence is parsed, four pieces of information are returned by the parser:

- *Number of parses*

  The number of parses for the sentence.

- *Execution time*

  The time (in seconds) that it took to parse the sentence.

- *Parse tree*

  A parse tree is displayed for each of the parses for the sentence. The parse tree is represented as a parenthesized list.

  For example, for the sentence "Uther sleeps" the parse tree might be

  ```
  s(np(n(uther)), vp(v(sleeps)))
  ```

- *Feature structure corresponding to the sentence*

  A feature structure for each of the parses for a sentence is displayed as an attribute-value matrix.

  For example, a possible feature structure associated with the sentence "Uther sleeps" is represented as

  ```
  [cat:  s
   head:  [form:  finite
           trans:  [pred:  sleep
                    arg1:  uther]
           aux:  false]]
  ```

A.4 Sample Session with the P-PATR System

The following is a transcript of a session with P-PATR, using the grammar in Section B.

```
Quintus Prolog Release 1.6 (Sun)
Copyright (C) 1986, Quintus Computer Systems, Inc.
               All rights reserved.

| ?- compile(loadpatr).
[pp.pl compiled (7.350 sec 1848 bytes)]
[readin.pl compiled (2.450 sec 964 bytes)]
[patrsupport.pl compiled (18.017 sec 5552 bytes)]
[patrlibrary.pl compiled (2.100 sec 728 bytes)]
[readtokens.pl compiled (9.634 sec 2968 bytes)]
[readpatr.pl compiled (28.716 sec 9948 bytes)]
[readrules.pl compiled (1.067 sec 432 bytes)]
[paths.pl compiled (12.717 sec 3700 bytes)]
[epsilons.pl compiled (1.317 sec 620 bytes)]
[parameters.pl compiled (1.850 sec 496 bytes)]
[compilegrammar.pl compiled (5.483 sec 1520 bytes)]
[compilelex.pl compiled (0.634 sec 244 bytes)]
[unify.pl compiled (3.950 sec 900 bytes)]
[compilepatr.pl compiled (29.833 sec 9388 bytes)]
[loadpatr.pl compiled (79.850 sec 26588 bytes)]

yes
| ?- grammar('sample.patr').

Reading ...

Runtime = 11.899994

yes
| ?- compilepatr(sample).

Compiling ...

Runtime = 5.633995

Loading ...
[sample.dcg compiled (20.633 sec 3728 bytes)]
```

```
yes
| ?- patr.
|: Uther sleeps.

Runtime = 0.066000

Analysis # 1:

Parse Tree = s(np(uther),vp(v(sleeps)))

[cat: s
 head: [form: finite
        trans: [pred: sleep
                arg1: uther]
        aux: false]]


Number of Parses = 1
|: Cornwall is stormed.

Runtime = 0.100000

Analysis # 1:

Parse Tree = s(np(cornwall),vp(vp(v(is)),vp(v(stormed))))

[cat: s
 head: [form: finite
        trans: [pred: storm
                arg2: cornwall]]]


Number of Parses = 1
|: Knights sleep.

Runtime = 0.084000

Analysis # 1:

Parse Tree = s(np(nom(knights)),vp(v(sleep)))

[cat: s
 head: [form: finite
        trans: [pred: sleep
```

```
                 arg1: knights]
         aux: false]]


Number of Parses = 1
|: A knight storms Cornwall.

Runtime = 0.100000

Analysis # 1:

Parse Tree = s(np(det(a),nom(knight)),
               vp(vp(v(storms)),np(cornwall)))

[cat: s
 head: [form: finite
        trans: [pred: storm
                arg1: knight
                arg2: cornwall]
        aux: false]]


Number of Parses = 1
|: Uther storms Cornwall.

Runtime = 0.067000

Analysis # 1:

Parse Tree = s(np(uther),vp(vp(v(storms)),np(cornwall)))

[cat: s
 head: [form: finite
        trans: [pred: storm
                arg1: uther
                arg2: cornwall]
        aux: false]]


Number of Parses = 1
|: Uther sleep.

Runtime = 0.050000
```

```
*** Cannot parse [uther,sleep]
|: A knights storm Cornwall.

Runtime = 0.050000

*** Cannot parse [a,knights,storm,cornwall]
|: ^D
yes
| ?- halt.

[ End of Prolog execution ]
```

B Sample Grammar and Prolog DCG

```
;;;==========================================================================
;;;               Demonstration Grammar
;;; (adapted from Sample Grammar 4 in Shieber's book on unification [14])
;;;
;;; Includes    subject-verb agreement
;;;             complex subcategorization
;;;             logical-form construction
;;;             lexical organization by templates
;;;               and lexical rules
;;;==========================================================================

Parameter:  Start Symbol is S.

Parameter:  Attribute order is  cat lex sense head
                                subcat first rest
                                form agreement person
                                       number gender
                                trans pred arg1 arg2.

;;;=======================================================
;;;                  Grammar Rules
;;;=======================================================

Rule {sentence formation}

        S -> NP VP:

                <S head> = <VP head>
                <S head form> = finite
                <VP subcat first> = <NP>
                <VP subcat rest> = end.

Rule {np formation}

        NP -> Det Nom:

                <NP head> = <Det head>
                <NP head> = <Nom Head>.

Rule {plural nouns}

        Det -> :

                <Det head agreement number> = plural.
```

```
Rule {trivial verb phrase}

       VP -> V:

              <VP head> = <V head>
              <VP subcat> = <V subcat>.

Rule {complements}

       VP_1 -> VP_2 X:

              <VP_1 head> = <VP_2 head>
              <VP_2 subcat first> = <VP_1 subcat first>
              <VP_2 subcat rest first> = <X>
              <VP_2 subcat rest rest> = <VP_1 subcat rest>.

;;;==================================================
;;;                  Definitions
;;;==================================================

Let Verb be <cat> = v.

Let Finite be Verb
             <head form> = finite.

Let Nonfinite be Verb
                <head form> = nonfinite.

Let ThirdPerson be <subcat first head agreement person> = third.

Let Singular be <subcat first head agreement number> = singular.

Let Plural be <subcat first head agreement number> = plural.

Let ThirdSing be Finite
                ThirdPerson
                Singular.

Let MainVerb be Verb
                <head aux> = false.

Let Transitive be MainVerb
                  <subcat first cat> = NP
                  <subcat rest first cat> = NP
                  <subcat rest rest> = end
                  <head trans arg1> = <subcat first head trans>
                  <head trans arg2> = <subcat rest first head trans>.

Let Intransitive be MainVerb
                   <subcat first cat> = NP
                   <subcat rest> = end
                   <head trans arg1> = <subcat first head trans>.

Let Raising be  <subcat first cat> = NP
                <subcat rest first cat> = VP
                <subcat rest first subcat rest> = end
                <subcat rest first subcat first> = <subcat first>
                <subcat rest rest> = end.

Define AgentlessPassive as <out cat> = <in cat>
                           <out subcat> = <in subcat rest>
                           <out head agreement> = <in head agreement>
                           <out head aux> = <in head aux>
                           <out head trans> = <in head trans>
                           <out head form> = passiveparticiple.

;;;=========================================================
;;;                   Lexicon
;;;=========================================================

Word uther:

           <cat> = np
           <head agreement gender> = masculine
           <head agreement person> = third
           <head agreement number> = singular
           <head trans> = uther.

Word cornwall:

           <cat> = np
           <head agreement person> = third
           <head agreement number> = singular
           <head trans> = cornwall.

Word knights:

           <cat> = nom
           <head agreement gender> = masculine
           <head agreement person> = third
           <head agreement number> = plural
           <head trans> = knights.
```

```
Word knight:

          <cat> = nom
          <head agreement gender> = masculine
          <head agreement person> = third
          <head agreement number> = singular
          <head trans> = knight.

Word a:

          <cat> = det
          <head agreement number> = singular.

Word sleeps:   Intransitive ThirdSing
               <head trans pred> = sleep.

Word sleep:    Intransitive Plural
               <head trans pred> = sleep.

Word storms:   Transitive ThirdSing
               <head trans pred> = storm.

Word stormed:  Transitive AgentlessPassive
               <head trans pred> = storm.

Word is:       Raising ThirdSing
               <subcat rest first head form> = passiveparticiple
               <head trans> = <subcat rest first head trans>.
```

The following is the DCG produced by P-PATR for the foregoing grammar:

```
ensure_loaded(patrsupport).

start(s).

feature_order(main,[cat:_6688,lex:_6681,sense:_6674,head:_6660,
                    subcat:_6667],
                   [_6688,_6681,_6674,_6660,_6667]).
feature_order(head,[form:_6873,agreement:_6880,trans:_6866,aux:_6887],
                   [_6866,_6873,_6880,_6887]).
feature_order(subcat,[first:_7024,rest:_7031],
                    [_7024,_7031]).
feature_order(first,[cat:_7148,lex:_7141,sense:_7134,head:_7120,
                    subcat:_7127],
                   [_7148,_7141,_7134,_7120,_7127]).
feature_order(rest,[first:_7328,rest:_7335],
                   [_7328,_7335]).
feature_order(agreement,[person:_7431,number:_7424,gender:_7438],
                        [_7424,_7431,_7438]).
feature_order(trans,[pred:_7558,arg1:_7565,arg2:_7572],
                   [_7558,_7565,_7572]).
feature_order(arg1,[pred:_7690,arg1:_7697,arg2:_7704],
                   [_7690,_7697,_7704]).
feature_order(arg2,[pred:_7822,arg1:_7829,arg2:_7836],
                   [_7822,_7829,_7836]).

null([cat: det,
     lex: _7973,
     sense: _7978,
     head: [form: _8072,
           agreement: [person: _8117,
                      number: plural,
                      gender: _8127],
           trans: _8082,
           aux: _8087],
     subcat: _7988]).

lc([cat: np,
    lex: _8231,
    sense: _8236,
    head: _8241,
    subcat: _8246],
    _8691,_8746,_8693)-->
down([cat: vp,
      lex: _8179,
      sense: _8184,
      head: [form: finite,
             agreement: _8491,
             trans: _8496,
             aux: _8501],
     subcat: [first: [cat: np,
                      lex: _8231,
                      sense: _8236,
                      head: _8241,
                      subcat: _8246],
              rest: end]],
     _8686),
lc([cat: s,
    lex: _8283,
    sense: _8288,
    head: [form: finite,
           agreement: _8491,
           trans: _8496,
           aux: _8501],
    subcat: _8298],
    _8691,s(_8746,_8686),_8693).

lc([cat: det,
    lex: _8855,
    sense: _8860,
    head: _8813,
    subcat: _8870],
    _9156,_9211,_9158)-->
down([cat: nom,
      lex: _8803,
      sense: _8808,
      head: _8813,
      subcat: _8818],
      _9151),
lc([cat: np,
    lex: _8907,
    sense: _8912,
    head: _8813,
    subcat: _8922],
    _9156,np(_9211,_9151),_9158).

lc([cat: nom,
    lex: _8803,
    sense: _8808,
    head: [form: _9261,
           agreement: [person: _9267,
                       number: plural,
                       gender: _9269],
            trans: _9259,
            aux: _9271],
    subcat: _8818],
    _9283,_9338,_9285)-->
lc([cat: np,
    lex: _8907,
    sense: _8912,
    head: [form: _9261,
           agreement: [person: _9267,
                       number: plural,
                       gender: _9269],
           trans: _9259,
           aux: _9271],
    subcat: _8922],
    _9283,np(_9338),_9285).

lc([cat: v,
    lex: _9394,
    sense: _9399,
    head: _9404,
    subcat: _9409],
    _9687,_9742,_9689)-->
lc([cat: vp,
    lex: _9446,
    sense: _9451,
    head: _9404,
    subcat: _9409],
    _9687,vp(_9742),_9689).

lc([cat: vp,
    lex: _9798,
    sense: _9803,
    head: _9808,
    subcat: [first: _10053,
             rest: [first: _286,
                    rest: _10141]]],
    _10472,_10527,_10474)-->
down(_286,_10467),
lc([cat: vp,
    lex: _9850,
    sense: _9855,
    head: _9808,
    subcat: [first: _10053,
             rest: _10141]],
    _10472,vp(_10527,_10467),_10474).

lex(uther,[cat: np,
            lex: uther,
            sense: uther1,
            head: [form: _10838,
                   agreement: [person: third,
                               number: singular,
                               gender: masculine],
                   trans: uther,
                   aux: _10848],
            subcat: _10850]).

lex(cornwall,[cat: np,
              lex: cornwall,
              sense: cornwall1,
              head: [form: _10838,
                     agreement: [person: third,
                                 number: singular,
                                 gender: _10846],
                     trans: cornwall,
                     aux: _10848],
              subcat: _10850]).

lex(knights,[cat: nom,
             lex: knights,
             sense: knights1,
             head: [form: _10838,
                    agreement: [person: third,
                                number: plural,
                                gender: masculine],
                    trans: knights,
                    aux: _10848],
             subcat: _10850]).

lex(knight,[cat: nom,
            lex: knight,
            sense: knight1,
            head: [form: _10838,
                   agreement: [person: third,
                               number: singular,
                               gender: masculine],
                   trans: knight,
                   aux: _10848],
            subcat: _10850]).

lex(a,[cat: det,
       lex: a,
       sense: a1,
       head: [form: _10838,
                agreement: [person: _10844,
                            number: singular,
                            gender: _10846],
                trans: _10836,
                aux: _10848],
         subcat: _10850]).

lex(sleeps,[cat: v,
            lex: sleeps,
            sense: sleeps1,
            head: [form: finite,
                   agreement: _10846,
                   trans: [pred: sleep,
                           arg1: _10840,
                           arg2: _10842],
                   aux: false],
            subcat: [first: [cat: np,
                             lex: _10966,
                             sense: _10964,
                             head: [form: _10956,
                                    agreement: [person: third,
                                                number: singular,
                                                gender: _11322],
                                    trans: _10840,
                                    aux: _10960],
                             subcat: _10962],
                     rest: end]]).

lex(sleep,[cat: v,
           lex: sleep,
           sense: sleep1,
           head: [form: _10844,
                  agreement: _10846,
                  trans: [pred: sleep,
                          arg1: _10840,
                          arg2: _10842],
                  aux: false],
           subcat: [first: [cat: np,
                            lex: _10966,
                            sense: _10964,
                            head: [form: _10956,
                                   agreement: [person: _11170,
                                               number: plural,
                                               gender: _11172],
                                   trans: _10840,
                                   aux: _10960],
                            subcat: _10962],
                  rest: end]]).

lex(storms,[cat: v,
            lex: storms,
            sense: storms1,
            head: [form: finite,
                   agreement: _10846,
                   trans: [pred: storm,
                           arg1: _10840,
                           arg2: _10842],
                   aux: false],
            subcat: [first: [cat: np,
                             lex: _10966,
                             sense: _10964,
                             head: [form: _10956,
                                    agreement: [person: third,
                                                number: singular,
                                                gender: _11366],
                                    trans: _10840,
                                    aux: _10960],
                             subcat: _10962],
                     rest: [first: [cat: np,
                                    lex: _10988,
                                    sense: _10986,
                                    head: [form: _10978,
                                           agreement: _10980,
                                           trans: _10842,
                                           aux: _10982],
                                    subcat: _10984],
                            rest: end]]]).

lex(stormed,[cat: v,
             lex: _10854,
             sense: _10852,
             head: [form: passiveparticiple,
                    agreement: _10846,
                    trans: [pred: storm,
                            arg1: _10840,
                            arg2: _10842],
                    aux: false],
             subcat: [first: [cat: np,
                              lex: _10988,
                              sense: _10986,
                              head: [form: _10978,
                                     agreement: _10980,
                                     trans: _10842,
                                     aux: _10982],
                         subcat: _10984],
                 rest: end]]).

lex(is,[cat: v,
        lex: is,
        sense: is1,
        head:
         [form: finite,
          agreement: _10840,
          trans: _10836,
          aux: _10842],
        subcat:
         [first:
           [cat: np,
            lex: _10984,
            sense: _10982,
            head:
             [form: _11256,
              agreement:
               [person: third,
                number: singular,
                gender: _11264],
               trans: _11254,
               aux: _11266],
            subcat: _10980],
          rest:
           [first:
             [cat: vp,
              lex: _10866,
              sense: _10864,
              head:
               [form: passiveparticiple,
                agreement: _10858,
                trans: _10836,
                aux: _10860],
              subcat:
               [first:
                 [cat: np,
                  lex: _10984,
                  sense: _10982,
                  head:
                   [form: _11256,
                    agreement:
                     [person:third,
                      number: singular,
                      gender: _11264],
                    trans: _11254,
          aux: _11266],
        subcat: _10980],
      rest: end]],
  rest: end]]]).
```

C Selected Code

```
% Module: COMPILEPATR.PL
% Author: Susan B. Hirsh
% Purpose: Compile a clausal form of a PATR-II grammar into a
%          DCG.

% load all supplemental modules

:- ensure_loaded( readrules ).      % read in PATR-II rules
:- ensure_loaded( parameters ).     % handle parameter statements
:- ensure_loaded( paths ).          % generate feature information
:- ensure_loaded( epsilons ).       % precompile epsilon rules
:- ensure_loaded( compilegrammar ). % compile the PATR-II grammar
:- ensure_loaded( unify ).          % unify PATR-II equations
:- ensure_loaded( compilelex ).     % precompile lexical entries



% External predicates :
%
%
% Module COMPILEGRAMMAR.PL -
%
% compile_grammar/3 -
%    compile PATR-II grammar into a DCG.
%
%
% Module COMPILELEX.PL
%
% compile_lex/1 -
%   execute each lexical entry in the database.
%
%
% Module EPSILONS.PL
%
% epsilons/2 -
%    precompile epsilon rules.
%
%
% Module PARAMETERS.PL
%
% parameter/3 -
%    process all parameter statements.
%
%
```

```
% Module PATHS.PL
%
% paths/2 -
%    generate all feature information.
%
%
% Module PATRLIBRARY.PL
%
% file_name/3 -
%    create a new file name with a new ending.
% write_clause/2 -
%    write clause to output stream in Prolog clause format.
%
%
% Module PATRSUPPORT.PL
%
% format_stats/0 -
%    output statistics on runtime.
% set_timer/0 -
%    reset runtime timer.
%
%
% Module READRULES.PL
%
% input_rules/2 -
%    Read in PATR-II rules from .PTRP file.



%-----------------------------------------------------------
%
% compilepatr( File )
%
% Input :
%   File - name of input file (must have .PTRP extension)
%
%
% Take a list of PATR-II rules produced by READPATR.PL and
%  convert them into a definite-clause grammar (DCG).


compilepatr( File ) :-
   format( '~nCompiling ...~n', [] ),   % output current status
   input_rules( File, Rules ),          % read in grammar rules
   output_rules( File, Rules ).         % convert rules to DCG
```

```
%--------------------------------------------------------------
%
% output_rules( File, Rules )
%
% Input :
%   File - name of input file
%   Rules - list of PATR-II rules
%
%
% Convert PATR-II rules into a DCG and output the DCG.


output_rules( File, Rules ) :-
   file_name( File, ".dcg" , Output ),    % output file is File.dcg
   open( Output, write, OutStream ),      % open output file
   % insert line into DCG to include runtime support
   write_clause( ( :- ensure_loaded( patrsupport ) ),
                 OutStream ),
   compile_rules( Rules, OutStream ),     % compile PATR-II rules
   close( OutStream ),                    % close output file
   ( load_parser( yes ) ->                % is DCG to be loaded
   load_dcg( Output )                     % load the DCG
   | true ).                              % do nothing



%--------------------------------------------------------------
%
% compile_rules( Rules, OutStream )
%
% Input :
%   Rules - list of PATR-II rules
%   OutStream - current output stream
%
%
% Compile PATR-II rules into a DCG.


compile_rules( Rules, OutStream ) :-
   set_timer,                                % set runtime timer
   parameter( Rules, OnlyRules, OutStream ), % handle parameters
   paths( OnlyRules, OutStream   ),          % get feature information
   % precompile epsilon rules
   epsilons( OnlyRules, OutStream ),
   compile_grammar( OnlyRules, Rules, OutStream ),% make DCG
   % execute lexical entries
   compile_lex( OutStream ),
   format_stats.                              % output compile statistics


%------------------------------------------------------------
%
% load_dcg( Output )
%
% Input :
%   Output - name of output file
%
%
% Load DCG into Prolog database.


load_dcg( Output ) :-
   format( '~nLoading ...~n',[] ),% output current status
   ensure_loaded( Output ).
```

```
% Module: COMPILEPATR.PL
% Submodule: READRULES.PL
% Author: Susan B. Hirsh
% Purpose: Read in a list of PATR rules.


% External predicates :
%
%
% Module PATRLIBRARY.PL
%
% file_name/3 -
%    create a new file name with a new ending.



%---------------------------------------------------------
%
% input_rules( File, Rules )
%
% Input :
%   File - input file name
%
% Output :
%   Rules - list of all PATR-II rules from input file
%
%
% Read in PATR-II rules from input file and put into a list.


input_rules( File, Rules ) :-
   seeing( Infile ),                      % save current input file
   file_name( File, ".ptrp", Input ),     % input file is File.ptrp
   see( Input ),                          % open input file
   read_rules( Rules ),                   % read in the rules
   seen,                                  % close input file
   see( Infile ).                         % restore input file



%---------------------------------------------------------
%
% read_rules( Rules )
%
% Output :
%   Rules - list of PATR-II rules
```

```
%
% Read in a list of PATR-II rules.


read_rules( Rules ) :-
   read( Rule ),                      % read in the first rule
   read_more_rules( Rule, Rules ).   % read in the rest



%------------------------------------------------------------
%
% read_more_rules( PreviousRules, NewRules )
%
% Input :
%   PreviousRules - list of PATR-II rules as it is being
%                    built up
%
% Output :
%   NewRules - list of PATR-II rules
%
% Read in a list of PATR-II rules.


%  stop at the end of the file
read_more_rules( end_of_file, [] ) :- !.

%  keep reading until the end of the file
read_more_rules( Rule,[ Rule | Rules ] ) :-
   read( NewRule ),                   % read in a PATR-II rule
   read_more_rules( NewRule, Rules ). % read in the rest
```

```
% Module: COMPILEPATR.PL
% Submodule: PARAMETERS.PL
% Author: Susan B. Hirsh
% Purpose: Record the information from the parameter statements.


% External predicates :
%
%
% Module PATRLIBRARY.PL
%
% write_clause/2 -
%    write clause to output stream in Prolog clause format.



%-------------------------------------------------------------
%
% parameter( Rules, NewRules )
%
% Input :
%  Rules - list of PATR-II rules
%
% Output:
%  NewRules - list of PATR-II rules minus parameter statements
%
%
% Handle parameter statements first, as they must appear only at
%  the top of the file.


% handle start symbol
parameter( [ parameter( start( Symbol ) ) | Rules ], NewRules,
          OutStream ):-
   assert( start(Symbol) ),         % assert start symbol
   write_clause( ( start(Symbol) ), OutStream ),   % write to output
   parameter( Rules, NewRules, OutStream ).        % handle others

% keep track of attribute order
parameter( [ parameter( attributes( List ) ) | Rules ], NewRules,
          OutStream ):-
   % record the correct order
   record_order( List, 1 ),
   % handle other parameter stmnts
   parameter( Rules, NewRules, OutStream).
```

```
% ignore restrictor
parameter( [ parameter( restrictor( _List ) ) | Rules ], NewRules,
           OutStream ) :-
   parameter( Rules, NewRules, OutStream ).

% ignore translation
parameter( [ parameter( translation( _List ) ) | Rules ], NewRules,
           OutStream ) :-
   parameter( Rules, NewRules, OutStream ).

% no more parameter statements - return list minus parameters
parameter( Rules, Rules, _OutStream ).



%------------------------------------------------------------
% record_order( Attributes, Place )
%
% Input :
%   Attributes - list of attributes in the order in which they
%                are to appear
%   Place - position in the list of the current attribute
%
%
% Record the print order of each attribute.


% record the position of each attribute
record_order( [ Attribute | Attributes ], Place ) :-
   % assert for use in printing
   assert( print_order(Attribute,Place) ),
   % increment position
   NewPlace is Place + 1,
   % go on to the next attribute
   record_order( Attributes, NewPlace ).

% no more attributes
record_order( [], _Place ).
```

```
%  Module: COMPILEPATR.PL
%  Submodule: PATHS.PL
%  Author: Susan B. Hirsh
%  Purpose: Compile all information on position and order of
%           the features.


% External predicates :
%
%
% Module PATRLIBRARY.PL
%
% write_clause/2 -
%    write clause to output stream in Prolog clause format.
%
%
% Module PATRSUPPORT.PL
%
% print_order/2 -
%    the printing order of this feature in the feature structure.



%---------------------------------------------------------
%
% paths( Rules, OutStream )
%
% Input :
%   Rules - list of PATR-II rules
%   OutStream - current output stream
%
%
% Generate for each attribute a list of the features that can
%  follow it and assert this information into the data base
%  and output into output file.
%
% For example :
%
%  The rule
%   rule(NP,[N],[[NP,cat]=np,[N,cat]=n,[NP,body]=[N,body]])
%
%  would produce the list :
%     feature_order(main,[cat:X,body:Y],[X,Y])
%
%  where the attribute 'main' is a dummy attribute used to designate
%  that a feature following it was the first feature in a path
% specification.

paths( Rules, OutStream ) :-
   % create lists of Vars, Bindings, and Pairs
   type_info( Rules, [ Main ], Types,[ main=Main ], Bindings, [],
             Pairs ),
   calc_types( Pairs ),      % make pairs into paths
   tails( Types ),           % get rid of tail variables
   % assert paths into database and write into output file
   output_paths( Bindings, OutStream ).



%---------------------------------------------------------
%
% type_info( Rules, OldTypes, Types, OldBindings, Bindings,
%            OldPaths, Paths )
%
% Input :
%   Rules - list of rules in PATR-II format
%   OldTypes - types found so far
%   OldBindings - bindings found so far
%   OldPaths - paths found so far
%
% Output :
%   Types - list of types
%   Bindings - list of bindings
%   Paths - list of paths
%
%
% Extract from each rule the features used in that rule.  From
%  this feature information compile three different lists :
%
%  Types : a list of variables associated with the features
%  Bindings : a list containing information as to which attributes
%   are bound to which variables.
%  Pairs : a list specifying which features can follow which others


% no more rules
type_info( [], Types, Types, Bindings, Bindings, Pairs, Pairs ).

% extract info from each rule
type_info( [ Rule | Rules ], Types, Rtypes, Bindings, Rbindings,
          Pairs, Rpairs):-
   % features are contained in the unification equations of a rule
   unifs( Rule, Unifs, Type ),   % get feature information
   % process the feature information
   info( Type, Unifs, Types, Ntypes, Bindings, Nbindings, Pairs,
         Npairs ),
   % do the rest of the rules
   type_info( Rules, Ntypes, Rtypes, Nbindings, Rbindings, Npairs,
              Rpairs).


%------------------------------------------
%
% unifs( Rule, Unifs, Type )
%
% Input :
%   Rule - current PATR-II rule
%   Type  - what kind of rule this is
%
% Output :
%   Unifs - list of unifications for that rule
%
%
% Extract the unification equations from the rule.


% grammar rule
unifs( rule(_Lhs,_Rhs,Unifs), Unifs, rule ).

% lexical entry
unifs( lex(_Word,Unifs), Unifs, lex ).

% lexical template
unifs( template(_Name,Unifs), Unifs, lex ).

% lexical rule
unifs( lex_rule(_Name,_InFS,_OutFS,Unifs), Unifs, rule ).



%-----------------------------------------------------------
% info( Type, Unifs, OldTypes, Types, OldBindings, Bindings,
%       OldPaths, Paths )
%
% Input :
%   Type - the type of rule it is
%   Unifs - list of unifications for that rule
%   OldTypes - list of types so far
%   OldBindings - list of bindings so far
%   OldPaths - list of paths so far
%
% Output :
%   Types - list of types
%   Bindings - list of bindings
%   Paths - list of paths
%
%
% Extract feature information from the unification equations.


% no more unifications in this rule
info( _All, [], Types, Types, Bindings, Bindings, Pairs,
      Pairs ).

% ignore template and lexical rule names, as these features are
%  handled in template or rule definitions
info( Kind, [ Template| T ], Types, Rtypes, Bindings, Rbindings,
      Pairs, Rpairs) :-
   atomic( Template ), !,   % this is a template or lexical rule
   % go on to the next unification equation
   info( Kind, T, Types, Rtypes, Bindings, Rbindings, Pairs,
         Rpairs ).

% for rules :

%   handle unifications of the form : Path1 = Path2
%    E.G.,
%      <S head> = <VP head>
info( rule, [ [ _Var1 | Features1 ] =
              [ _Var2 | Features2 ] | T ],
   Types, Rtypes, Bindings, Rbindings, Pairs, Rpairs ) :-
   % unify the final feature values so that paths can unify
   add_paths( Features1, Types, Ntypes, Bindings, Nbindings, Pairs,
              Npairs, main, Last),
   add_paths( Features2, Ntypes, Mtypes, Nbindings, Mbindings,
              Npairs, Mpairs, main, Last ),
   % go on to the next unification equation
   info( rule, T, Mtypes, Rtypes, Mbindings, Rbindings, Mpairs,
         Rpairs ).

%   handle unifications of the form : Path = val
%    E.G.,
%      <X cat> = np
info( rule, [ [ _Var | Features ]=Atom | T ], Types, Rtypes,
      Bindings, Rbindings, Pairs, Rpairs ) :-
   atomic( Atom ),
   % add feature information
   add_paths( Features, Types, Ntypes, Bindings, Nbindings, Pairs,
              Npairs, main, _Last),
   % go on to next unification
   info( rule, T, Ntypes, Rtypes, Nbindings, Rbindings, Npairs,
         Rpairs ).

% for lexical entries or templates :

%   handle unifications of the form : Path = val
%    E.G.,
%      <cat> = np

info( lex, [ Features=Atom |T ], Types, Rtypes, Bindings,
      Rbindings, Pairs, Rpairs) :-
   atomic( Atom ),!,
   % add feature information
   add_paths( Features, Types, Ntypes, Bindings, Nbindings, Pairs,
              Npairs, main, _Last ),
   % go on to next unification
   info( lex, T, Ntypes, Rtypes, Nbindings, Rbindings, Npairs,
         Rpairs ).

%   handle unifications of the form : Path1 = Path2
%    E.G.,
%      <head> = <head>
info( lex, [ Features1=Features2 | T ], Types, Rtypes, Bindings,
      Rbindings, Pairs, Rpairs ) :-
   % unify the final feature values so that paths can unify
   add_paths( Features1, Types, Ntypes, Bindings, Nbindings, Pairs,
              Npairs, main, Last ),
   add_paths( Features2, Ntypes, Mtypes, Nbindings, Mbindings,
              Npairs, Mpairs, main, Last ),
   info( lex, T, Mtypes, Rtypes, Mbindings, Rbindings, Mpairs,
         Rpairs).



%-----------------------------------------------------------
%
% add_paths( Features, OldTpes, Types, OldBindings, Bindings,
%            Oldpairs, Pairs, Place, Last )
%
% Input :
%   Features - list of features in one unification equation
%   OldTypes - list of types so far
%   OldBindings - list of bindings so far
%   OldPairs - list of pairs so far
%   Place - previous feature
%   Last - Var value of last feature on the list
%
% Output :
%   Types - list of types
%   Bindings - list of bindings
%   Pairs - list of pairs
%
% Create the three list of Types, Bindings and Pairs as described.


% last feature, just return variable for later unifications
add_paths( [], Types, Types, Bindings, Bindings, Pairs, Pairs,
           Place, Last ):-
   % get variable equivalence of this attribute
   search( Place, Bindings, Last ).

% add on the Types, Bindings and Pairs
add_paths( [ Feature | Features ], Types, Rtypes, Bindings,
           Rbindings, Pairs, Rpairs, Place, Last ) :-
   % get variable value
   search( Place, Bindings, Var ),
   % get Type and Binding information
   checkpaths( Feature, Types, Ntypes, Bindings, Nbindings ),
   % get pair information
   add_pairs( Var, Feature, Pairs, Npairs ),
   % handle next attribute
   add_paths( Features, Ntypes, Rtypes, Nbindings, Rbindings, Npairs,
              Rpairs, Feature, Last ).



%-----------------------------------------------------------
%
% search( Place, Bindings, Var )
%
% Input :
%   Place - current attribute to look up
%   Bindings - list of bindings
%
% Output :
%   Var - Prolog Var value of the attribute
%
%
% Look up the Var value of the current attribute on the Bindings
%  list.
% stop when you find the attribute
search( Place, [ Place=Var | _Bindings ], Var ) :- !.

% keep searching until you find it
search( Place, [ _Binding | Bindings ], Var ) :-
   search( Place, Bindings, Var ).



%----------------------------------------------------------
%
% checkpaths( Feature, Oldtypes, Types, Oldbindings, Bindings )
%
% Input :
%   Feature - current attribute
%   OldTypes - list of types
%   OldBindings - list of bindings
%
% Output :
%   Types - new list of types if attribute was added
%   Bindings - new list of bindings if attribute was added
%
%
% Check if an attribute is bound in the Bindings list and add it
%  if it isn't already there.


% add attribute if it is not there
checkpaths( Feature, Types, [ Var | Types ], [],
            [ Feature=Var ] ).

% if it is there, do nothing
checkpaths( Feature, Types, Types, [ Feature=Var | Bindings ],
            [ Feature=Var | Bindings ] ) :- !.

% if it is not there, keep trying the rest of the list
checkpaths( Feature, Types, Rtypes, [ Binding | Bindings ],
            [ Binding | Rbindings ] ) :-
   checkpaths( Feature, Types, Rtypes, Bindings, Rbindings ).



%----------------------------------------------------------
% add_pairs( Var, Feature, OldPairs, Pairs )
%
```

```
% Input :
%   Var - var to add
%   Feature - attribute to add
%   OldPairs - previous list of pairs
%
% Output :
%   Pairs - new list of pairs
%
% Add a Var and a Feature to Pairs list.


add_pairs( Var, Feature, Pairs, [ Var : Feature | Pairs ] ).



%-----------------------------------------------------------
%
% calc_types( Pairs )
%
%  Input :
%   Pairs - list of pairs
%
%
% Once all of the Pairs have been done, go throught the Pairs list
%  and add all pairs to the one preceding them.
%
% For  example :
%  Pairs will look like [[A:head],[A,cat]]
%   and now A will look like [head,cat]


% add pair to list
calc_types( [ Type : Label | Pairs ] ):-
   insert( Label, Type ),   % unify it into the Prolog variable
   calc_types( Pairs ).     % go to next pair

% no more pairs
calc_types( [] ).



%-----------------------------------------------------------
% insert( Feature, Variable )
%
% Input :
%   Feature - current attribute
%   Variable - variable to insert value into
```

```
%
%
% Unify a feature into the Prolog variable if it is not already
%  there.


% it is already there, do nothing
insert( Label, [ Label | _ ] ) :- !.

% unify feature into variable
insert( Label, [ _ | Labels ] ) :-
   insert( Label, Labels ).



%-----------------------------------------------------------
%
% tails( Types )
%
% Input :
%   Types - list of types
%
% Change all tail variables tbat are sideeffects of Insert
%  to [].


%  change tail variable to []
tails( Var ) :-
   var( Var ), !,      % this is a tail variable
   Var = [].

%  not a list, do nothing
tails( Atom ) :-
   atomic( Atom ), !.

%  check all internal lists
tails( [ Head | Tail ] ) :-
   tails( Head ),      % process first list
   tails( Tail ).      % process the rest of the list

tails( [] ).



%-----------------------------------------------------------
%
% output_paths( Bindings )
```

```
%
% Input :
%   Bindings - list that now has a feature and all the features
%              that can follow it
%
%
% Go through the list of bindings that now have all features in
%  the variable and create paths.
% For example :
%  Bindings will be main=[head,cat]
%  Path is [main,[head:A,cat:B],[A,B]]


% no more in bindings list
output_paths( [], _OutStream ).

% make paths for all bindings
output_paths( [ Binding | Bindings ], OutStream ) :-
   make_path( Binding, OutStream),        % make the path
   output_paths( Bindings, OutStream ).   % go to next binding



%------------------------------------------------------------
%
% make_path( Feature, OutStream )
%
% Input :
%   Feature - a feature and list of features that can follow it
%   OutStream - current output stream
%
%
% Change path into Prolog variables and then assert and output.


% feature cannot be followed
make_path( _Head=[], _OutStream ).

% process this path
make_path( Head=Features, OutStream ) :-
   % change path into variables
   change( Head, Features, LabelList, VarList ),
   % assert and output
   write_path( Head, LabelList, VarList, OutStream ).
```

```
%-----------------------------------------------------------
%
% change( Head, Features, LabelList, VarList )
%
% Input :
%   Head - starting attribute
%   Features - list of features that can follow Head
%
% Output :
%   LabelList - list of Prolog variables and features for the path
%   VarList - same as LabelList with no attributes
%
%
% Put path into Prolog variables
% Take binding
%    main = [cat,head]
%  and make the lists :
%    LabelList - [cat:Cat,head:Head]
%    VarList - [Cat,Head]


% no more paths
change( _Head, [], [], [] ).

% change each path
change( Head, [ Feature | Features ],[ Feature : Var | Labels ],
        [ Var | Vars ] ) :-
   change( Head, Features, Labels, Vars ).



%-----------------------------------------------------------
%
% write_path( MainFeature, LabelList, VarList, OutStream )
%
%  Input :
%   MainFeature - feature that all other features follow
%   LabelList - list of attributes and variables
%   VarList - same as LabelList with no attributes
%   OutStream - current output stream
%
%
% Output path as:
%
%   feature_order(MainFeature, LabelList, VarList)
```

```
write_path( Main, LabelList, VarList,OutStream) :-
   % reorder the list into printing order
   reorder( LabelList, OrderLabelList ),
   % output to screen if trace flag is on
   (trace_paths( yes ) ->
   format( 'Path is ~w~n',
           [ feature_order(Main,OrderLabelList,VarList) ] )
   | true ),
   % assert into database for use during compilation
   assert( feature_order(Main,OrderLabelList,VarList) ),!,
   % write into output file for use during parse
   write_clause((feature_order(Main,OrderLabelList,VarList)),
                OutStream ).



%-------------------------------------------------------
%
% reorder( FS, NewFS )
%
%  Input :
%   FS - feature structure to be reordered
%
%  Output :
%   NewFS - feature structure with features as specified
%
%
% Reorder the features in the FS according to the order given
%  in the parameter statement.


reorder( Pairs, NewPairs ) :-
   % attach the order in which they should appear
   number( Pairs, NumberedPairs ),
   keysort( NumberedPairs, SortedPairs ),   % sort by position
   % get rid of position numbers
   clean( SortedPairs, NewPairs ).



%-------------------------------------------------------
%
% number( FS, NewFS )
%
%  Input :
%   FS - feature structure to be reordered
%
```

```
%  Output :
%   NewFS - feature structure with features labeled with
%           their position number
%
%
% Attach onto each feature in the FS the position number specified.


% number each feature
number( [ Label : Value | Rest ],
        [ Position-(Label:Value) | NRest ] ):-
   find_order( Label, Position ),   % get position number
   number( Rest, NRest ).           % do the rest

% no more features
number( [], [] ).



%----------------------------------------------------------
%
% find_order( Feature, Position )
%
%  Input :
%   Feature - feature to get position of
%
%  Output :
%   Position - position number of the feature
%
%
% Get the position number of the feature


% order was specified
find_order( Label, Position ) :-
   print_order( Label, Position ), !.   % specified order

% no order given, affix to the end
find_order( _Label, 9999 ).



%----------------------------------------------------------
%
% clean( FS, NewFS )
%
%  Input :
```

```
%   FS - feature structure with feature positions attached
%
%  Output :
%   NewFS - feature structure without feature positions
%
%
% Remove position numbers attached to the features.


% get rid of position number
clean( [ _N-(Label:Value) | Nrest ],
       [ Label : Value | Rest ] ) :-
   clean( Nrest, Rest ).

% no more features
clean( [], [] ).
```

```
% Module: COMPILEPATR.PL
% Submodule: EPSILONS.PL
% Author: Susan B. Hirsh
% Purpose: Preprocess all epsilon rules.


% External predicates :
%
%
% Module UNIFY.PL
%
% apply_rule_unifs/1 -
%    apply the unification equations for a grammar or lexical rule.
%
%
% Module PATRLIBRARY.PL
%
% write_clause/2 -
%    write clause to output stream in Prolog clause format.



%-----------------------------------------------------------
%
% epsilons( Rules, NewRules, OutStream )
%
% Input :
%   Rules - List of PATR-II rules
%   OutStream - current output stream
%
% Output :
%   NewRules - List of PATR-II rules minus epsilon rules
%
%
% Precompile epsilon rules for use in compilation.


% no more rules
epsilons( [], _OutStream ).

% go through the rules
epsilons( [ Rule | Rules ] , OutStream ) :-
   e_rule( Rule, OutStream ),      % is it an epsilon rule?
   epsilons( Rules, OutStream ).   % go to next rule
```

```
%-----------------------------------------------------------------
%
% e_rule( Rule, OutStream )
%
% Input :
%   Rule -  PATR-II rule
%   OutStream - current output stream
%
%
% Compile the epsilon rule into a Prolog clause.
%  The clause is of the form :
%   null( FS )
%  where FS is the feature structure associated with the rule.


%  this is an epsilon rule
e_rule( rule(Lhs,[],Unifs), OutStream ) :-
   apply_rule_unifs( Unifs ), !,    % unify equations
   % output to screen if trace flag is on
   ( trace_rules( yes ) ->
   format( 'EPSILON Rule is ~w~n', [ null(Lhs) ] )
   | true ),
   % assert into the database for use during compilation
   assert( null(Lhs) ), !,
   % output to file.dcg
   write_clause( ( null(Lhs) ), OutStream ).

% error - cannot compile this rule
e_rule( rule(Lhs,[],Unifs), _OutStream ) :-
   format('~n*** Cannot compile rule: ~w~n',[rule(Lhs,[],Unifs)]).

%  not an epsilon rule
e_rule( _Rule, _OutStream ).
```

```
% Module: COMPILEPATR.PL
% Submodule: COMPILEGRAMMAR.PL
% Author: Susan B. Hirsh
% Purpose: Perform the actual compilation of the grammar entries.


% External predicates :
%
%
% Module UNIFY.PL -
%
% apply_lex_unifs/5 -
%    apply the unification equations for a template or lexical
%    entry.
% apply_rule_unifs/1 -
%    apply the unification equations for a grammar or lexical rule.
%
%
% Module PATRLIBRARY.PL
%
% clausify/3 -
%    create Prolog clause from a head and a list of clauses.
% reverse/3 -
%    reverse a list.
% write_clause/2 -
%    write clause to output stream in Prolog clause format.
%
%
% Module PATRSUPPORT.PL
%
% find_category/2 -
%    find the value of the category attribute in a feature structure.
% null/1 -
%    precompiled epsilon rule.



%-------------------------------------------------------------
%
% compile_grammar( Rules, RuleList, OutStream )
%
% Input :
%   Rules - list of rules to be made into DCG
%   RuleList - list of rules in current PATR-II grammar
%   OutStream - current output stream
%
```

```
%
% Take each PATR-II rule and convert it into a DCG rule.


% no more rules to compile
compile_grammar( [], _RuleList, _OutStream ).

% compile each rule
compile_grammar( [ Rule | Rules ], RuleList, OutStream ) :-
   % compile the rule
   compile_rule( Rule, RuleList, OutStream ),
   % do next rule
   compile_grammar( Rules, RuleList, OutStream ).



%-----------------------------------------------------------
%
% compile_rule( Rule, RuleList, OutStream )
%
% Input :
%   Rule - current PATR-II rule
%   RuleList - list of all rules in current PATR-II grammar
%   OutStream - current output stream
%
%
% Convert each PATR-II rule into a DCG rule or a Prolog clause.
%  Grammar rules become DCG rules and lexical items become
%  directly executable Prolog clauses that are executed when
%  compilation is completed, resulting in full lexical entries.


% ignore epsilon rules, because they were precompiled
compile_rule( rule( _Lhs, [], _Unifs ), _Rules, _OutStream ).

% error - parameter statements must be at start of the file
compile_rule( parameter(_Statement), _Rules, _OutStream ) :-
   format('~n*** Parameter statements must occur at start of grammar file!~n',[]).

% handle grammar rules
% grammar rules are compiled into DCG rules of the form :
%
%   lc( Rhs1, Parent, OldBranch, NewBranch ) -->
%      down( Rhs2, Branch2 ),...down( RhsN, BranchN ),
%      lc( Lhs, Parent, Tree, NewBranch ).
%
% where:
```

```
%   Rhs1..RhsN - element of the right-hand side of the rule
%   Parent - variable associated with the parent of the rule
%   OldBranch - parse tree so far
%   NewBranch - parse tree after application of this rule
%   Branch2..BranchN - parse trees for each node
%   Tree - parse tree for that rule
%
compile_rule( rule( Lhs, Rhs, Unifs ), _Rules, OutStream ):-
   apply_rule_unifs( Unifs ),      % unify equations
   % create the DCG rule for the initial rule
   grammar_rule( Lhs, [ Rhs ], OutStream ),
   % return new list of rules with epsilon expansions
   epsilon( Rhs, AllRhs ),
   % create the DCG rule for the grammar rules
   grammar_rule( Lhs, AllRhs, OutStream ).

% handle lexical rules
% lexical rules are compiled into Prolog clauses of the form :
%
%   lex_rule( Name, InFS, OutFS ).
%
% where:
%   Name - name of this lexical rule
%   InFS - input feature structure to this rule application
%   OutFS - output feature structure after rule application
%
compile_rule( lex_rule( Name, InFS, OutFS, Unifs ), _Rules,
              _OutStream ):-
   apply_rule_unifs( Unifs ),             % unify equations
   assert( lex_rule(Name,InFS,OutFS) ).   % assert into database

% handle lexical entries
% lexical entries are compiled into clauses of the form:
%
%   word( Name, FS ) -->
%     applications of lexical rules and templates into
%     FS1..FSN, where last application puts result
%     into FS.
%
compile_rule( lex( Word, Unifs ), Rules, _OutStream ) :-
   % unify equations
   apply_lex_unifs( Unifs, Rules, List, HeadFS, _FS ),
   % put into clause form
   clausify( word(Word,HeadFS), List, Clause ),
   assert( Clause ).

% handle lexical templates
% lexical templates are compiled into clauses of the form :
%
%   template( Name, InFS, OutFS ) -->
%     applications of lexical rules and templates into
%     FS1..FSN, where last application puts result
%     into OutFS.
%
compile_rule( template( Name, Unifs ), Rules, _OutStream ) :-
   % unify equations
   apply_lex_unifs( Unifs, Rules, List, OutFS, InFS ),
   % put rule into clause form
   clausify( template(Name,InFS,OutFS), List, Clause ),
   assert( Clause ).

% rule could not be compiled - error
compile_rule( Rule, _Rules, _OutStream ) :-
   format('~n*** Cannot compile rule: ~w~n',[Rule]).



%-----------------------------------------------------------
%
% grammar_rule( Lhs, Rhs, OutStream )
%
%  Input :
%   Lhs - left-hand side of the rule
%   Rhs - all possible right-hand sides for this rule
%   OutStream - current output stream
%
%
% Take each possible Rhs for the rule and create a DCG rule for
%  it.


% make a DCG from each Rhs
grammar_rule(Lhs,[ [ Rhs1 | RhsN ] | MoreRhs ],OutStream) :-
   % create right-hand side
   rhs( Lhs, RhsN, Parent, [], Clauses, Branch, NewBranch),
   start_symbol( Lhs, OutStream ),      % find grammar start symbol
   % output new DCG rules to the screen if trace flag is set
   ( trace_rules( yes ) ->
   format( 'GRAMMAR RULE is ~w~n',
         [(lc(Rhs1,Parent,Branch,NewBranch) --> Clauses)])
   | true ),
   % output new DCG rule to file.dcg
   write_clause( (lc(Rhs1,Parent,Branch,NewBranch)-->Clauses),
                 OutStream ),
   % go to next right-hand side
   grammar_rule( Lhs, MoreRhs, OutStream ).

% no more right-hand sides to make rules from
grammar_rule( _Lhs, [], _OutStream ).



%-----------------------------------------------------------
%
% rhs( Lhs, Rhs, Parent, Branch, List, OldBranch, NewBranch )
%
% Input :
%   Lhs - left hand side of the rule
%   Rhs - All but first of right-hand side of the rule
%   Parent - parent of this rule
%   OldBranch - branch variable for the left-hand side of the rule
%   NewBranch - branch variable for the left-hand side of the rule
%
% Output :
%   List - list of Rhs elements in the form for an LC rule
%   Branches - list of branches as variables in the parse tree
%
%
% Create the clauses for the right-hand side of the DCG rule


% keep track of branches and build up right-hand side
rhs( Lhs, [ Rhs1 | Rhs ], Parent, Branches, (down(Rhs1,Branch),NewRhs),
     OldBranch, NewBranch ) :-
   rhs( Lhs, Rhs, Parent, [ Branch | Branches ], NewRhs, OldBranch,
        NewBranch ).

% no more branches, create left-hand side
rhs( Lhs, [], Parent, Branches, lc(Lhs, Parent, Tree, NewBranch),
     OldBranch, NewBranch ) :-
   % get category for parse tree
   find_category( Lhs, Cat ),
   % put constituents in proper order for tree
   reverse( Branches, [], Constituents ),
   % put into form Cat(OldBranch,Constituents)
   Tree =..[Cat|[OldBranch|Constituents]].



%-----------------------------------------------------------
%
```

```
% epsilon( Rule, Newrules )
%
%  Input :
%   Rule - current rule
%
% Output :
%   Newrule - a list of all possible right-hand sides for this
%             rule
%
%
% As long as the first element of the right-hand side of the rule
%  can be expanded by an epsilon rule, return the rule minus that
%  element.
%
% For example :
%   The rules   S -> NP VP
%               NP -> e
%
% Will produce the list [ VP ], since the NP can be expanded by
%  the epsilon rule.  When returned, there are understood
%  to be two rules now instead of the one.  The rules are:
%
%      S -> NP VP
%      S -> VP


% check the first element of the right-hand side
epsilon( [ Rhs1 | Rhs ], [ Rhs | NewRhs ]) :-
   null( Rhs1 ), !,         % can be expanded by an epsilon rule
   epsilon( Rhs, NewRhs ). % check if next can be

% no more nonterminals can be expanded by epsilon rule
epsilon( _Rhs, [] ).


%------------------------------------------------------------
%
% start_symbol( Lhs, OutStream )
%
% Input :
%   Lhs - left-hand side of the rule
%   OutStream - current output stream
%
%
% If no start symbol for the grammar has been specified, it is
%  the nonterminal on the left-hand side of the first rule.
% start symbol is already specified
start_symbol( _Lhs, _OutStream ) :-
   start( _Cat ), !.      % start symbol is in database

% no start symbol, need to add one
start_symbol( Lhs, OutStream ) :-
   find_category( Lhs, Cat ),   % get Cat of start symbol
   assert( start(Cat) ),        % assert as start symbol
   % start symbol is needed in parsing, so output it to parser file
   write_clause( ( start(Cat) ) , OutStream ).
```

```
% Module: COMPILEPATR.PL
% Submodule: UNIFY.PL
% Author: Susan B. Hirsh
% Purpose: Apply the unification equations constraining a rule.


% External predicates :
%
%
% Module PATRSUPPORT.PL
%
% thepath/4 -
%   extract the value for a particular feature from a feature structure.



%--------------------------------------------------------
%
% apply_rule_unifs( Unifs )
%
% Input :
%   Unifs - list of unifications for the rule
%
%
% Unify the values in each equation in a grammar or lexical rule.


% handle unifications of the form : Path1 = Path2
%  E.G.,
%    <S head> = <VP head>
apply_rule_unifs( [ [ Var1 | Features1 ]=
                    [ Var2 | Features2 ] | T ] ) :-
   % find paths from these features and unify values
   find_path( main, Features1, Val, Var1 ),
   find_path( main, Features2, Val, Var2),
   apply_rule_unifs( T ).

% handle unifications of the form : Path = val
%  E.G.,
%    <X cat> = np
apply_rule_unifs( [ [ Var | Features ]=Atom | T ] ):-
   atomic( Atom ),
   % find location of this feature
   find_path( main, Features, Atom, Var ),
   apply_rule_unifs( T ).
```

```
% no more unification equations
apply_rule_unifs( [] ).



%--------------------------------------------------------------
%
% apply_lex_unifs( Unifs, Rules, List, HeadFS, FS )
%
% Input :
%   Unifs - list of unifications for the rule
%   Rules - list of rule in the grammar
%
%  Output :
%   List - list of rules to assert
%   HeadFS - feature structure for head of clause
%   FS - initial input feature structure
%
%
%  Unify the values in each equation in a lexical entry or template.


% no more unifications to do
apply_lex_unifs( [], _Rules, [], FS, FS ).

%  handle unifications of the form : Path = val
%   E.G.,
%    <X cat> = np
apply_lex_unifs( [ Features=Atom | T ], Rules, List, HeadFS,
                 FS ):-
   atomic( Atom ),!,
   % get position of this feature
   find_path( main, Features, Atom, FS ),
   apply_lex_unifs( T, Rules, List, HeadFS, FS ).

% add application of a lexical template to the new DCG rule
apply_lex_unifs( [ Atom | T ], Rules,
                 [ template(Atom,FS,TempFS) | List ],
                 HeadFS, FS ) :-
   atomic( Atom ),
   % make sure this is a template
   find_type( Atom, Rules, template ),!,
   apply_lex_unifs( T, Rules, List, HeadFS, TempFS ).

% add application of a lexical rule to the new DCG rule
apply_lex_unifs( [ Atom | T ] , Rules,
                 [ lex_rule(Atom,FS,TempFS) | List ],
                HeadFS, FS) :-
   atomic( Atom ),
   apply_lex_unifs( T, Rules, List, HeadFS, TempFS ).

%  handle unifications of the form : Path1 = Path2
%   E.G.,
%    <head> = <head>
apply_lex_unifs( [ Features1=Features2 | T ] , Rules, List, HeadFS,
                FS) :-
   find_path( main, Features1, Val, FS ),
   find_path( main, Features2, Val, FS ),
   apply_lex_unifs( T, Rules, List, HeadFS, FS ).



%-------------------------------------------------------
%
% find_path( PrevFeature, Features, Value, Var )
%
% Input :
%   PrevFeature - previous feature in the path
%   Features - list of features in this equation
%   Value - postion of feature in feature structure
%   Var - feature structure unifications are acting on
%
%
% Follow a path of features and return the value at the end.


%  do for each feature in list
find_path( Place, [ Head | Rest ] , Atom, Var ) :-
  % search in path information
   thepath( Place, Head, Var, Path ),
   find_path( Head, Rest, Atom, Path ).

%  stop at end of feature list
find_path( _Place, [], Atom, Atom ).



%-------------------------------------------------------
%
% find_type( Atom, Rule, Type )
%
% Input :
%   Atom - name of current template or lexical rule
%   Rule - top value on rule list
```

```
%
% Output :
%   Type - template or rule, depending on type of rule
%
%
% Return whether a lexical item is a template or lexical rule.


% lexical template
find_type( Atom, [ template(Atom,_Unifs) |  _Tail ] ,
           template ) :- !.

% lexical rule
find_type( Atom, [ lex_rule(Atom,_InFS,_OutFS,_Unifs) | _Tail ],
           rule ) :- !.

% keep searching
find_type( Atom, [ _Head | Tail ], Type ) :-
   find_type( Atom, Tail, Type ).
```

```
% Module: COMPILEPATR.PL
% Submodule: COMPILELEX.PL
% Author: Susan B. Hirsh
% Purpose: Compile all lexical entries.


% External predicates :
%
%
% Module PATRLIBRARY.PL
%
% write_clause/2 -
%    write clause to output stream in Prolog clause format.
%
%
% Module PATRSUPPORT.PL
%
% word/2 -
%    lexical entry from the database.



%----------------------------------------------------------
%
% compile_lex( OutStream )
%
% Input :
%   OutStream - current output stream
%
%
% Precompile each lexical entry.  This involves actually
%  executing each one and then outputting the new entry as
%  lex( Word, FS ).


% execute the lexical entry and output it
compile_lex( OutStream ) :-
   word( Word, FS ),        % execute lexical entry
   % output to screen if trace flag is set
   ( trace_rules( yes ) ->
   format( 'LEXICAL ENTRY is ~w~n', [ (lex(Word,FS) ) ] )
   | true ),
   % write lexical entry to output file
   write_clause( ( lex(Word,FS) ), OutStream ),
   fail.                    % go to next lexical entry
```

```
% no more lexical entries
compile_lex( _OutStream ).
```

```
%  Module: PATRSUPPORT.PL
%  Author: Susan B. Hirsh
%  Purpose: Support module for the parser.


% Predicates necessary at runtime :
%
%
% LC Parser -
%
% patr/0 -
%    input loop using start symbol.
% parse/2 -
%    get all parses for a sentence
% print_parses/2 -
%    print parses and parse trees for a sentence.
% misc. LC predicates - parse/2, down/2, lc/4, and leaf/2
%
%
% Utility predicates -
%
% alphanumeric/1 -
%    character is alphanumeric.
% append/3-
%    concatenate two lists.
% case_shift/2 -
%    convert a list to lower case.
% concat/3 -
%    concatenate two atoms.
% digit/1 -
%    character is a digit.
% end_file/1 -
%    end of file character.
% find_category/2 -
%    find value of category attribute in a feature structure.
% format_stats/0 -
%    print runtime statistics.
% full_stop/1 -
%    '.'
% member/2 -
%    check membership in a list.
% new_line/1 -
%    new-line character.
% string_size/2 -
%    get length of an atom.
% set_timer/0 -
%    set runtime timer.
% thepath/4 -
%     return the value of a path.
% upper/1 -
%     character is upper case.


% set up the system :


% declare type of predicates

:- dynamic(null/1).     % epsilon rules
:- dynamic(start/1).   % start symbol
:- dynamic(feature_order/3).   % path information

% operator definition
:- op(500,xfx,:).


% LC rules appear in two files
:- multifile lc/6.


:- no_style_check( all ).   % suppress warnings


:- ensure_loaded( pp ).      % load prettyprinter
:- ensure_loaded( readin ).  % load sentence reader



% predicates necessary for the LC parser to run


% initial calling sequence
parse( Cat, Tree ) --> down( Cat, Tree ).



% pick up a new left corner when one has been processed

% epsilon rules
down( Cat, [] ) --> { null( Cat ) }.

% get next word and find whose left corner it is
```

```
down( Cat, Tree ) -->
   leaf( Child, OldTree ),
   lc( Child, Cat, OldTree, Tree ).


% every phrase is the left corner of itself
lc( Type, Type, Tree, Tree ) --> [].


% this is a word
%  get the category information for parse tree
leaf( FS, Tree ) -->                   % handle lexical entries
   [ Word ],                           % get the word
   { lex( Word, FS ) },                % get word's feature structure
   { find_category( FS, Cat ) },       % get category of feature structure
   { Tree =..[Cat,Word] }.             % make parse tree


%-----------------------------------------------------------
%
% patr
%
%
% Read in a sentence and parse it with the start symbol.  By starting
%  the parse with a feature structure with the 'cat' feature
%  specified as the start symbol, the only good parses are those
%  that result in a parse whose 'cat' feature is that start symbol.


patr :-
   read_in( Sentence ),              % read in the sentence
   start( Symbol ),                  % get the start symbol
   find_category( S, Symbol ),       % create filter for sentence parse
   parse( S, Sentence ).             % parse the sentence


%-----------------------------------------------------------
%
% parse( Structure, Sentence )
%
% Input :
%   Structure - feature structure whose structure the parse must
%               match
%   Sentence - sentence to parse
%
%
% Get all parses for a sentence and print them out.  Read in a
%  new sentence and parse it.


% no more sentences
parse( S, end_of_file ):- !.

% parse sentences
parse( S, [] ) :- !,
   read_in( NewSentence ),
   parse( S, NewSentence ).

parse( S, Sentence ) :-
   set_timer,                  % set runtime timer
   % get all parses and parse trees for a sentence
   bagof( S-Tree, parse(S,Tree,Sentence,[]), Parses ),!,
   format_stats,               % print runtime statistics
   print_parses( Parses,0 ),   % print parses
   read_in( NewSentence ),     % read in a new sentence
   parse( S, NewSentence ).    % parse that sentence

% error - couldn't parse
parse( S, NoParse ) :-
   format_stats,             % print runtime statistics
   % print error message
   format('~n*** Cannot parse ~w~n',[NoParse]),
   read_in( NewSentence ),   % read a new sentence
   parse( S, NewSentence ).  % parse that sentence



%-----------------------------------------------------------
%
% print_parses( Parses, NumParses )
%
% Input :
%   Parses - list of parses and parse trees for a given sentence
%   NumParses - number of parses for the sentence
%
%
% Print the parses and parse trees for a sentence.


% print analysis on each parse
```

```
print_parses( [ Parse-Tree | Parses ], Count ) :-
   % increment count of number of parses
   NewCount is Count + 1,
   % print analysis
   format('~nAnalysis # ~d:~n~nParse Tree = ~w~n',[NewCount,Tree]),
   format('~p~n',[Parse]),
   print_parses( Parses, NewCount ).   % next parse

% no more parses
print_parses( [], Count ) :-
   % print count of number of parses
   format('~nNumber of Parses = ~d~n',[Count]).



%----------------------------------------------------------
%
% concat( Atom1, Atom2, Atom )
%
% Input :
%   Atom1 - first atom
%   Atom2 - second atom
%
% Output :
%   Atom - concatenation of Atom1 and Atom2
%
%
% Concatenate two atoms.


concat( Atom1, Atom2, Result ) :-
   name( Atom1, List1 ),            % put in list form
   name( Atom2, List2 ),            % put in list form
   append( List1, List2, List3 ),   % concatenate as lists
   name( Result, List3 ).           % make into an atom



%----------------------------------------------------------
%
% append( List1, List2, List )
%
% Input :
%   List1 - first list
%   List2 - second list
%
% Output :
```

```
%   List - concatenation of List1 and List2
%
%
% Concatenate two lists.


% a list appended to the empty list is that list
append( [], List, List ).

% a list appended to another adds the head to a new list
append( [ Head | List1 ], List2, [ Head | List3 ] ) :-
   append( List1, List2, List3 ).



%-----------------------------------------------------------
%
% find_category( FS, Cat )
%
% Input :
%   FS - a feature structure to get the category value from
%
% Output :
%   Cat - category value of the feature structure
%
%
% Get the value of the category attribute in the FS.

find_category( Lhs, Cat ) :-
   thepath( main, cat, Lhs, Cat ),    % get category of nonterminal
   atomic( Cat ), !.

% if Cat value isn't an atom, make it X
find_category( Lhs, x ).



%-----------------------------------------------------------
%
% member( Element, List )
%
% Input :
%   Element - element to check membership of
%   List - list to check membership in
%
%
% Check whether an element is a member of a list.
% element is head of the other list
member( Element, [ Element | _Rest ] ) :- !.

% keep searching the list
member( Element, [ _Head | Rest ] ) :-
   member( Element, Rest ).



%-----------------------------------------------------------
%
% set_timer
%
%
% Reset runtime timer.

set_timer :-
   statistics( runtime, [ _, _RunTime ] ).



%-----------------------------------------------------------
%
% format_stats
%
%
% Print runtime information


format_stats :-
   statistics( runtime, [ _, Stats ] ),     % get runtime
   Time is Stats/1000,                      % convert to seconds
   format( '~nRuntime = ~f~n', [ Time ] ). % print runtime.



%-----------------------------------------------------------
%
% string_size( Atom, Size )
%
% Input :
%   Atom - atom to get length of
%
% Output :
%   Size - number of characters in the atom
%
```

```
%
% Get the number of characters in an atom.


string_size( String, Size ) :-
   name( String, List ),   % make into a list
   length( List, Size ).   % get the length of the list



%-----------------------------------------------------------
%
% thepath( Node, Label, Term, Value )
%
% Input :
%   Node - start feature
%   Label - current feature
%
% Output :
%   Term - feature structure
%   Value - value of Label attribute
%
%
% Return of the value of the path <Node,Label>.


thepath( Node, Label, Term, Value ) :-
   % get the structure of the FS
   feature_order( Node, FS, Term ),
   member( Label:Value, FS ).         % get the value



%-----------------------------------------------------------
%
% case_shift( Token, NewToken )
%
% Input :
%   Token - current input token
%
% Output :
%   NewToken - Token in all lower case.
%
%
% Convert token to all lower case.
% if upper case, convert to lower
case_shift( [Upper | Mixed ], [ Letter | Lower ]) :-
   upper( Upper ), !,
   Letter is Upper+32,       % make lower case
   case_shift( Mixed, Lower ).

% if not upper case, ignore
case_shift( [ Other | Mixed ], [ Other | Lower ]) :-
   case_shift( Mixed, Lower ).

% no more to lower case
case_shift( [], [] ).



%----------------------------------------------------------
%
% alpha_numeric( Char )
%
% Input :
%   Char - character to check
%
%
% Check whether a character is alphanumeric.


alpha_numeric(Ch) :-
   ( upper( Ch )            %  A..Z
   ; Ch >= 97, Ch =< 122    %  a..z
   ; digit( Ch )            %  0..9
   ; Ch = 95                %  '_'
   ; Ch = 63                %  '?'
   ; Ch = 42                %  '*'
   ; Ch = 39                %  nonstandard "'"
   ; Ch = 96                %  nonstandard "`"
   ).



%----------------------------------------------------------
%
% digit( Char )
%
% Input :
%   Char - character to check
%
%
```

```
% Check whether a character is a digit.


digit( Ch ) :-          %  0..9
   ( Ch >= 48
   , Ch =< 57
   ).



%-----------------------------------------------------------
%
% upper( Char )
%
% Input :
%   Char - character to check
%
%
% Check whether a character is an upper case letter.


upper( Ch ) :-          %  A..Z
   ( Ch >= 65
   , Ch =< 90
   ).


% input delimiters

full_stop( 46 ).         %  '.'
end_file( -1 ).          % end of file
new_line( 10 ).          % new line
```

References

[1] Alfred Aho, Ravi Sethi, and Jeffrey D. Ullman. *Principles of Compiler Design.* Addison-Wesley Publishing Co., 1986.

[2] John Bear. A morphological recognizer with syntactic and phonological rules. In *Proceedings of the Eleventh International Conference on Computational Linguistics*, University of Bonn, Bonn, German Federal Republic, 25-29 August 1986.

[3] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, 1981.

[4] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley Publishing Co., 1979.

[5] Lauri Karttunen. D-PATR: a development environment for unification-based grammars. CSLI Report No. CSLI-86-61, CSLI, Stanford, California, 1986.

[6] Yuji Matsumoto, Hozumi Tanaka, Hideki Hirakawa, Hideo Miyoshi, and Hideki Yasukawa. BUP: a bottom-up parser embedded in Prolog. *New Generation Computing*, 1:145–158, 1983.

[7] Fernando C. N. Pereira. Logic for natural language analysis. Technical Note 275, Artificial Intelligence Center, SRI International, Menlo Park, California, 1983.

[8] Fernando C. N. Pereira and David H. D. Warren. Parsing as deduction. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*, Massachusetts Institute of Technology, Cambridge, Massachusetts, 15-17 June 1983.

[9] Fernando C.N. Pereira. Deductive computation of grammar properties. Forthcoming.

[10] Fernando C.N. Pereira and Stuart M. Shieber. *Prolog and Natural Language Analysis.* CSLI, Stanford, California, 1987.

[11] D.J. Rosenkrantz and P.M. Lewis II. Deterministic left corner parsing. In *IEEE Conference Record 11th Annual Symposium on Switching and Automata Theory*, 1968.

[12] Stuart M. Shieber. Criteria for designing computer facilities for linguistic analysis. *Linguistics*, 23:189–211, 1985.

[13] Stuart M. Shieber. The design of a computer language for linguistic information. In *Proceedings of the Tenth International Conference on Computational Linguistics*, Stanford University, Stanford, California, 2-7 July 1984.

[14] Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar*. CSLI, Stanford, California, 1986.

[15] Stuart M. Shieber. The PATR-II experimental system. 1984. Stanford University, Stanford, California.

[16] Stuart M. Shieber. Standard for the PATR computer language. Forthcoming.

[17] Stuart M. Shieber, Lauri Karttunen, and Fernando 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.

[18] Stuart M. Shieber, Hans Uszkoreit, Fernando C. N. Pereira, Jane J. Robinson, and Mabry Tyson. The formalism and implementation of PATR-II. In *Research on Interactive Acquisition and Use of Knowledge*, Artificial Intelligence Center, SRI International, Menlo Park, California, 1983.

[19] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques*. The MIT Press, Cambridge, Massachusetts, 1986.

[20] David Warren, Luis Pereira, and Fernando Pereira. Prolog - the language and its implementation compared with Lisp. In *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, University of Rochester, Rochester, New York, 15-17 August 1977.