ProP Documentation

by

Bob Carpenter

October 1989

Report CMU-PHIL-11

Philosophy
Methodology
Logic

Pittsburgh, Pennsylvania 15213-3890

ProP Documentation

Bob Carpenter
Computational Linguistics Program
Carnegie Mellon University

October 11, 1989

1 Introduction

ProP is a Prolog implementation of the PATR-II unification grammar workbench. ProP extends the context-free grammar formalism by allowing partial descriptions of categories in both lexical entries and rules.

Details concerning the original PATR-II system can be found in Shieber *et al* (1983). For an introduction to the linguistic applications of PATR-II and related grammar formalisms, see Shieber (1986). For a more general computational overview of unification, see Knight (1989), which also has an extensive bibliography.

2 Prolog Preliminaries

This section contains all of the information about Prolog necessary to run ProP.

A Prolog *constant* is composed of either a sequence of characters beginning with a lower case letter, a number, or any sequence of characters surrounded by single quotes. So, `abc`, `johnDoe`, `b_17`, `123`, `'JohnDoe'`, `'_65a.'` are constants, and `A19`, `JohnDoe`, `B_19`, `_au8`, `[dd,e]` are not.

In grammar files, blank lines and line breaks are ignored. Any symbols on a line appearing after a `%` symbol are treated as *comments* and ignored.

The Prolog *prompt* looks like

```
| ?-
```

What you type after the prompt is called a *query*. Queries always end with a period. In fact, all of the grammar rules, macros and lexical entries must end with periods.

After the execution of many queries, the words `yes` or `no` will appear before the next prompt. Unless otherwise noted, these responses are irrelevant in the context of ProP.

You can leave Prolog by entering the query `halt` at any prompt (followed by a period, of course).

3 Feature Structure Matrices

In ProP, *feature structure matrices (FSM)* are used to model categories. FSMs can be defined by two cases, one for atomic FSMs and the other for structured ones.

- If σ is a Prolog constant, then $|\ \sigma\ |$ is an *atomic* FSM
- A *structured* FSM provides the following information
  - the values for a finite (possibly empty) set of features, where a feature is a Prolog constant and a feature value is an FSM
  - structure sharing links that indicate that two features in the FSM have the same value

Consider the structured FSM

```
f1: f2: |a1|
    f3: |a2|
f4: f2: f5: |a6|
```

Here the value of the feature `f1` is the FSM

```
f2: |a1|
f3: |a2|
```

The value of the feature `f3` in this FSM is the atomic FSM `|a2|`. Notice that features can occur more than once in an FSM. Note that we write a constant between bars to represent an atomic feature structure.

We can compose the feature value operation to provide values for lists of features, which are also called *paths*. Paths are represented by PROLOG lists of features, that is, as square bracketed sequences of features separated by commas. So we have, in BNF notation,

```
<path> ::= [<feat>,...,<feat>]
```

For instance, `[]`, `[f9]`, `[f7,f2,f1]` are paths, but `f1:f2:f3`, `<f3,f9>`, `f3 f9`, `(f2,f3,f4)` are not. FSMs have values for paths just as they have values for features. In the above FSM, the value of the path `[f4,f2]` is the FSM

```
f5: |a6|
```

Finally, consider the FSM

```
f1: (1) f5: |a1|
        f6: |a2|
        f7: |a3|
f2: f3: (1)
    f4: |a4|
```

The numbers occuring in parentheses in this FSM are called *tags*. Tags are used to indicate *structure sharing*. For instance the tag `(1)` above indicates that the value for the path `[f1]` (or equivalently for the feature `f1`) is the same as for the path `[f2,f3]`. In this case, the additional feature information is not displayed a second time. The value associated with the feature `f2` in the above is thus

```
f3: f5: |a1|
    f6: |a2|
    f7: |a3|
f4: a4
```

The actual number associated with a tag is not important — the number merely serves to identify structure sharing. FSMs which only differ in the numbers of their tags are considered identical.

An FSM is said to contain a *cycle* if there is a path which is tagged with the same label as one of its subpaths. For instance,

```
f1: (1) f2: |a|
        f3: f4: (1)
```

is cyclical since `[f1]` and `[f1,f3,f4]` are shared. Cyclical FSMs are not allowed in PROP. If an attempt is made to create a cycle, PROP will most likely crash.

4 Subsumption and Unification

The collection of categories can be partially ordered by a relation of *subsumption*. We take an FSM C_1 to subsume another FSM C_2 if the information represented by C_1 is *equal or more general* than that given by C_2. We also say that C_2 is more *specific* than C_1. To be more specific than another FSM requires some additional information in terms of additional feature values, more specific feature values or additional shared structure. Subsumption can be formally defined inductively by means of two cases.

- an atomic FSM subsumes only itself
- a structured FSM C_1 subsumes another FSM C_2 just in case
  - every feature f defined in C_1 has an equal or more specific value in C_2
  - every structure sharing that exists in C_1 also exists in C_2

Note that the second clause of this definition requries the set of paths given values in C_1 to be a subset of the paths with values in C_2. Some examples of subsumption follow, with < for the subsumption relation:

```
|a|  <  |a|

f1: |a1|   <   f1: |a1|
                f2: |a2|

f1: (1) f2: |a1|          f1: (1) f2: |a1|
        f3: |a2|     <             f3: |a2|
f2: f3: (1)                    f5: (2)
                               f2: f3: (1)
                               f4: (2)

f1: f2: |a1|    <    f1: (1) f2: |a1|
    f3: |a2|                    f3: |a2|
f3: f2: |a1|          f3: (1)
    f3: |a2|
```

Only the first of these subsumptions holds in the opposite direction, so that only the first pair of categories are *equivalent* in the sense of mutual subsumption. For all intents and purposes, feature structures which subsume each other can be taken to be identical.

The *unification* of two categories can be defined as the smallest category (with respect to subsumption) that is subsumed by both of them. Operationally, unification can be recursively defined by two cases.

- an atom can only unify with itself
- - two FSMs have a unification in which the value of a feature is the unification of the values in the two FSMs if both are defined, the one that's defined if just one is defined and undefined otherwise
  - structure sharings that exist in either feature structure must be preserved, by unifying all of the values of paths that are shared

Of course, unification can fail, as in the case of two distinct atoms, or feature structures with features which fail to unify. FSMs which can not be unified are said to be *incompatible* in the sense of containing contradictory information. Two FSMs can be incompatible if they are distinct atoms or if there is some feature for which they provide incompatible values. Using + to represent unification, we have the following equations

```
|a| + |a| = |a|

f1: |a1|  +  f2: |a2|  =  f1: |a1|
                          f2: |a2|

f1: f2: |a1|       +  f1: f3: f6: |a4|  =  f1: f2: |a1|
    f3: f4: |a2|        f7: |a5|                 f3: f4: |a2|
f5: |a3|                                          f6: |a4|
                                                f5: |a3|
                                                f7: |a5|

f1: (1)      +  f1: f4: |a1|       =  f1: (1) f4: |a1|
f2: f3: (1)     f2: f3: f5: |a2|              f5: |a2|
                                          f2: f3: (1)

f1: (1)       +   f1: |a1|       =   FAILS
f2: f3: (1)       f2: f3: |b|

f1: (1)  +  f1: f3 : (2)   = FAILS
f2: (1)     f2: (2)
```

Note that the last unification fails because the resulting FSM would be cyclic.

Unification and subsumption are defined in such a way that `A < B` if and only if there is some `C` such that `A + C = B`.

5 Descriptions

PROP uses Kasper-Rounds Logic for describing FSMs (Kasper and Rounds 1986). The *well-formed formulas (wff)* of this language can be given using the following Backus-Naur formulation

```
<wff> ::= <atomic-fsm>
        | (<feature> : <wff>)
        | (<path> == <path>)
        | (<wff> , <wff>)
        | (<wff> ; <wff>)
        | _
```

Both features in `<feature>` and atomic FSMs in `<atomic-fsm>` are represented by PROLOG constants. We use the colon for describing the value of a feature, the double equality to indicate structure sharing between paths, the comma to indicate conjunction and the semicolon to indicate disjunction. The underscore in the last line is called the *bottom* and taken to be the unique minimal descripiton which describes every FSM. Note that we have no need for a corresponding *top* description that would correspond to unification failure.

As usual, parentheses may be deleted where not necessary. The operators are ordered according to how tightly they bind. The loosest binding operator is the semicolon, the next is the comma, and everything else binds more tightly. Furthermore, all of the operators are right associative. For instance,

- $\phi, \psi; \gamma = (\phi, \psi); \gamma$
- $f : \phi, \psi = (f : \phi), \psi$
- $f_1 : f_2 : \phi = f_1 : (f_2 : \phi)$

- $\phi, \psi, \gamma = \phi, (\psi, \gamma)$

Formulas are used to describe FSMs. An FSM which is described by a formula is said to *satisfy* that formula. The recursive definition of satisfaction can be presented by cases.

- The FSM $\mid \sigma \mid$ is the only FSM which satisfies the atomic wff σ.
- An FSM satisfies the wff $f : \phi$ just in case its value for the feature f satisfies the wff ϕ.
- An FSM satisfies $\pi_1 == \pi_2$ just in case its value for the path π_1 is shared with its value for the path π_2.
- An FSM satisfies (ϕ, ψ) if it satisfies both ϕ and ψ.
- An FSM satisfies $(\phi; \psi)$ if it satisfies either ϕ or ψ.
- Every FSM satisfies _ .

A description is said to be *satisfiable* if it is satisfied by at least one FSM, and *unsatisfiable* otherwise. Two descriptions are said to be *logically equivalent* if they are satisfied by exactly the same set of FSMs. A description ϕ is said to *logically entail* another description ψ if every FSM which satisfies ϕ also satisfies ψ.

It should be clear from the definition of satisfaction that the order of information within conjunctions and disjunctions, as well as the order of paths in path equalities does not affect the set of solutions to the description.

The bottom description _ will not affect a solution if added in as a conjunct, so that the descriptions $(\_, \phi)$ and $(\phi, \_)$ and ϕ are all logically equivalent. Note that the minimal FSM which satisfies _ does not contain any specific information and will thus not show up in any of the displays except as empty space. Of course, this is the only FSM which will be displayed as empty space, so no confusion should arise.

It is important to note that there are two ways in which a description can be satisfied by more than one FSM. The first is through *vagueness*. For instance, the feature structures

```
f1: f2: (1) f4: |a1|
f3: (1)
```

and

```
f1: f2: (1) f4: |a1|
            f5: (2) |a2|
f3: (1)
f6: (2)
```

both satisfy the description `([f1,f2] == [f3], f3:f4:a1)`. But, the first has the important property of being the (subsumption) minimal category that satisfies the description, while the second is an FSM which is subsumed by the first. Such a descripiton is said to be *vague*. Every FSM which satisfies this description is more specific than the first FSM above. Conversely, every FSM more specific than the first FSM above satisfies the description. In this sense, every description is vague. Extra information can always be added about new features.

The second route to multiple satisfaction is through ambiguity. For instance, both

```
f1: f2: |a|
```

and

```
f3: f1: (1)
    f2: (1)
```

satisfy the description (`f1:f2:a ; f3:([f1] == [f2])`). But, neither of these solutions subsumes the other. Such a description is said to be *ambiguous* or *non-deterministic*. This kind of ambiguity can only arise from disjunctions in descriptions.

Kasper and Rounds (1986) have proved a normal form theorem stating that for every description, there is a recursively decidable finite set of FSMs such that every FSM that satisfies the description is more specific than some FSM in the set. The proof of this result lies in the construction of a sound and complete unification algorithm.

This means that a description will be ambiguous if and only if there is more than one such minimal FSM in the set of minimal satisfiers of the description. As a general rule of unification grammar writing, ambiguity should be traded for vagueness wherever possible.

Due to the fact that there are possibly an exponential number of minimal solutions for a disjunctive description, the satisfaction problem for descriptions is NP-hard. This result stems solely from the ability of the description language to express disjunction, which leads to ambiguity. The satisfaction problem for unambiguous descriptions without disjunction can be solved in polynomial time.

6 Lexical Entries

Lexical entries take the following form

```
<lex> ::= <word> ---> <wff>.
```

A word in `<word>` is just an arbitrary PROLOG constant. The idea here is that the word can have any lexical category which satisfies the formula it is associated with.

For instance, the lexical entry

```
mukesh ---> syn:head:maj:(nom:yes,
                          verb:no),
            syn:(head:agr:pers:3,
                 foot:bar:2),
            syn:head:(agr:num:sing,
                      agr:nform:norm),
            sem:mukesh.
```

would mean that john has the category

```
syn : head : maj : nom : |yes|
                   verb : |no|
             agr : pers : |3|
                   num : |sing|
                   nform : |norm|
      foot : bar : |2|
sem : |mukesh|
```

which is the minimal FSM satisfying its lexical description.

One or more entries may be given for each word. For instance, we could have

```
bank --->  syn:noun, sem:riverbank.
bank --->  syn:noun, sem:moneybank.
bank --->  syn:verb, sem:turn.
```

or the equivalent disjunctive

```
bank ---> (syn:noun, sem:(riverbank ; moneybank))
          ; (syn:verb, sem:turn).
```

7 Grammar Rules

Grammar rules in PROP are written with a bottom-up context-free grammar orientation. The BNF specification of a rule is

```
<rule> ::= <catnamelist> ===> <catname> if <form>.
```

Elements of `<catname>` are simply PROLOG constants and elements of `<catnamelist>` are PROLOG sequences of such constants. A PROLOG sequence contains elements separated by commas, such as `(x1,x2,...,xN)`. Note that it is *not* possible to have an empty sequence `()`, thus there are no null productions allowable in the grammar. The formula after the `if` serves to restrict the categories named in the rules. Keep in mind that the names given to categories in a rule have no significance in the grammar, but simply serve to uniquely identify the information associated with each node in a local tree. For instance, consider the simple rule

```
np, vp ===> s if
  np:maj:np,
  vp:maj:vp,
  [np,agr] == [vp,agr].
  s:maj:s,
  [s,sem] == [vp,sem],
  [s,sem,subj] == [np,sem].
```

The minimal FSM which satisfies the formula given by this rule is

```
np: maj: |np|
    agr: (1)
    sem: (3)
vp: maj: |vp|
    agr: (1)
    sem: (2) subj: (3)
s: maj: |s|
   sem: (2)
```

The interpretation of such a rule is that `FSM1` and `FSM2` can rewrite to `FSM3` if we unify

```
np: FSM1
vp: FSM2
```

with the solution to the rule to get an FSM whose value for `s` is `FSM3`. This allows information to flow up from the daughter categories to the mothers via unification. For instance, this rule would allow us to carry out the following reduction

```
maj: |np|       maj: |vp|                      maj: |s|
agr: |sing|     sem: rel: |feast|     ===>     sem: rel: |feast|
sem: |uther|                                        subj: |uther|
```

since unifying

```
np: maj: |np|
    agr: |sing|
    sem: |uther|
vp: maj: |vp|
    sem: rel: |feast|
```

with the rule yields

```
np: maj: |np|
    agr: (1) |sing|
    sem: (3) |uther|
vp: maj: |vp|
    agr: (1) |sing|
    sem: (2) subj: (3)
             rel: |feast|
s: maj: |s|
   sem: (2)
```

which has the appropriate value for the feature s.

The formal generative power of PATR-II is such that for any recursively enumerable language, there is a PATR-II grammar which accepts it. In particular, it is possible to write grammars which are undecidable. As we mention below in the section on parsing, undecidable grammars all have the characteristic of allowing some string to have an infinite number of subsumption incomparable analyses.

8 Macros

Since the construction of grammars often requires the use of the same categorial information over and over again in different categories, it is helpful to have an efficient means of coping with such redundancy. For this purpose, there are *macros* patterned after the templates of PATR-II. A macro can be thought of as an abbreviation for a formula. Macros are written into the grammar in the form

```
<macro> ::= <macro-name> macro <wff>.
```

where a macro name in <macro-name> is some PROLOG constant.

We add the additonal clause

```
<wff> ::= @ <macro-name>
```

to the definition of well-formed formulas, so that a marked macro name may stand in for formulas in any part of the grammar (including in the definition of other macros). The space between the @ symbol and the macro name is not significant.

For instance, consider the two macros

```
np          macro  maj:np.
third_sing  macro  agr:(num:sing,person:third).
```

With these macros, the following lexical entries would produce identical results

```
uther ---> maj:np, agr:num:sing, agr:person:third.
uther ---> @np, @third_sing.
```

Recursive macro definitions will cause chaos. Such definitions lead to an infinite regress which will cause the gramamr to hang at compile time. For instance, the macro definitions

```
a macro b.
b macro a,d.
```

are not acceptable, since trying to expand a leads us to try and expand b which leads us back to expanding a.

9 Grammar Compilation

The query `new_grammar('filename')` will load a file residing in `filename`. For instance, a grammar in the file `'gram.pl'` is loaded by

```
| ?- new_grammar('gram.pl').
```

This will load the lexical entries, rules and macros from the specified file and compile the lexicon and rule set from them.

There are also specialized predicates `new_macros`, `new_rules` and `new_lex` to read and compile just the appropriate grammar components from a file. These can also be used to extract the lexicon from one complete grammar file and the rules from another.

Compiling a grammar consists of finding all of the minimal solutions to the rules and lexical entries and recording them before parsing ever begins. This is similar to converting the entire grammar into disjunctive normal form. Since the grammar is compiled, the substitution of a formula for another formula with the same set of minimal solutions will have no effect on run-time speed.

10 Grammar Inspection

There are a number of utilities in PROP which can be used to display parts of the grammar. From here on out, all examples will be drawn from the sample grammar in the appendix.

To query a lexical entry, use the following format

```
| ?- lex(feasted).

WORD: feasted
ENTRY:
syn : head : maj : verb : |yes|
                  nom : |no|
             agr : nform : |norm|
             vform : |fin|
      foot : bar : |0|
      subcat : |1|
sem : tim : |past|
      rel : |feast|

Another?  y.
WORD: feasted
ENTRY:
syn : head : maj : verb : |yes|
                  nom : |no|
             agr : nform : |norm|
             vform : |past_part|
      foot : bar : |0|
      subcat : |1|
sem : aspect : |perfect|
      rel : |feast|

Another?  y.
no
```

Notice that after the first entry, the program prompts the user, who may enter `y.` or `n.` to tell the program to either provide another entry or stop looking for entries. Note that this query will not

rest until you've given it a period. In this context, the answer of **no** is significant in that it signals the lack of additional solutions to the query. This user query system is used in all of the interface modules.

`lex` and the other grammar inspection predicates `rule` and `macro` can be used with or without arguments. When used without arguments, they will iterate through the solutions in the relevant part of the grammar one by one.

To inspect a rule in the grammar, use

```
| ?- rule((np,vp ===> s)).

MOTHER:
syn : head : (2) maj : (0) verb : |yes|
                           nom : |no|
                  vform : |fin|
                  agr : (1)
      foot : bar : |2|
sem : (6) subj : (5)
DAUGHTERS:
 * syn : head : maj : nom : |yes|
                       verb : |no|
                case : |subj|
                agr : (1)
         foot : bar : |2|
   sem : (5)
 * syn : head : (2)
         foot : bar : |1|
   sem : (6)
```

Note that for the predicate rule it is neccessary to enclose the rule in two sets of parentheses so that PROLOG does not get confused about the commas. Rules are printed out with the mother first, followed by a sequence of daughters. The beginning of each daughter category is marked with an asterisk. In this example and from now on, we will eliminate the queries for multiple solutions from our examples. Such a rule can be thought of in terms of local tree templates. For a local tree to be acceptable it must be able to unify with a local tree generated by one of the rules in the grammar.

Finally, macros can be inspected in the same way, with

```
| ?- macro(np).

MACRO: np
syn : head : maj : nom : |yes|
                   verb : |no|
      foot : bar : |2|
```

11 Parsing and Debugging

PROP uses an active bottom-up chart parser enhanced to cope with the partial nature of unification grammars. Details on chart parsing can be found in Allen (1988, pages 60–72). The most significant difference between a unification chart parser and a standard chart parser is that the unification version must check a proposed edge to make sure it does not stand in a subsumption relationship either way to an existing edge. If a proposed edge is subsumed by something on the chart, then it is not added. If it subsumes an edge on the chart, the existing edge is removed and the new edge added. This can lead to unexpected behaviour if some expected edges don't show up on the chart.

It is important to keep in mind that the bottom-up chart parser in PROP will construct the entire chart before informing the user of any complete analyses. A grammar which produces an infinite set of subsumption incomparable categories will thus cause the parser to hang.

A list of words to be parsed must be given as a PROLOG list to the predicate **rec**. For instance

```
| ?- rec([uther,feasted]).

STRING:
0 uther 1 feasted 2

CATEGORY:
syn : head : maj : verb : |yes|
                  nom : |no|
             vform : |fin|
             agr : pers : |3|
                   num : |sing|
                   nform : |norm|
      foot : bar : |2|
sem : subj : |uther|
      tim : |past|
      rel : |feast|
```

The string is first displayed for convenience, since the numbers are necessary to inspect particular zones of the chart later.

To look at parse trees of the most recently parsed string, use

```
| ?- tree.

* np,vp===>s *
syn : head : (2) maj : (0) verb : |yes|
                           nom : |no|
                 vform : |fin|
                 agr : (1) pers : |3|
                           num : |sing|
                           nform : |norm|
      foot : bar : |2|
sem : (5) subj : |uther|
          tim : |past|
          rel : |feast|
    * lexical entry *
    syn : head : maj : nom : |yes|
                       verb : |no|
                 case : |subj|
                 agr : (1)
          foot : bar : |2|
    sem : |uther|
        <uther>
    * iv===>vp *
    syn : head : (2)
          foot : bar : |1|
    sem : (5)
        * lexical entry *
```

```
        syn : head : (2)
              foot : bar : |0|
              subcat : |1|
        sem : (5)
            <feasted>
```

Note that the trees are printed out in the standard computer-oriented indented list format. That is, the mother category is aligned at some column, and all of the daughters are indented. Each category is also preceeded by the name of the rule that was successfully fired, including lexical entries. The actual lexical token follows its lexical entry in the standard format. This is where mnemonic labels for categories in rules come in handy. Again, in trees, we do not print out redundant information at more than one node.

There is also a parameterized version of `tree` in which the span from left to right can be specified. For instance, `tree(0,1)` would print out just the part of the tree spanning `uther`.

12 Inspecting the Chart

Edges in the chart may be inspected in the following way

```
| ?- edge(0,1).

SPANS: < uther >

INACTIVE EDGES:
---------------
SPAN 0 TO 1
syn : head : maj : nom : |yes|
                   verb : |no|
             agr : pers : |3|
                   num : |sing|
                   nform : |norm|
      foot : bar : |2|
sem : |uther|

ACTIVE EDGES:
-------------
SPAN 0 TO 1
syn : head : (2) maj : (0) verb : |yes|
                           nom : |no|
                 vform : |fin|
                 agr : (1) pers : |3|
                           num : |sing|
                           nform : |norm|
      foot : bar : |2|
sem : (5) subj : |uther|
AFTER GETTING:
 * syn : head : (2)
         foot : bar : |1|
   sem : (5)
```

Note that the inactive edges are printed out followed by the active edges. The active edges include information on the categories that they need to get before becoming inactive. These categories are

printed out in the same manner as daughters in rules.

There is a facility to trace the parser as it adds edges to the chart. At any prompt, use the query

```
| ?- edge_trace.
```

to turn the edge tracer on and the query

```
| ?- no_edge_trace.
```

to turn the tracer off. When the tracer is on, it will display the edges as they are being produced, as well as any result category that is found after the chart is built. After every edge that is displayed, the user will be queried as to whether or not to proceed. A `n.` answer will turn off the tracer, but not abort the parse. Any other input will continue tracing with the next edge (remember to follow any input with a period).

13 Off-Line Input and Output

There is a mechanism for allowing a corpus of test sentences to be entered into a file and re-used. The file containing the test sentences should contain queries of the form `test(<words>)`, as in

```
test([uther,feasted]).
test([uther,feasted,yesterday]).
test([uther,probably,feasted,yesterday]).
```

To test the sentences in the file `filename`, use

```
| ?- recfile(filename).
```

This will cause all of the categories generated for each string to be printed out in order.

A Running C-Prolog under Unix

Change directories to where the grammar file(s) reside.

Enter the shell command

```
%  prolog
```

at which point you will see

```
C-Prolog version 1.5
| ?-
```

/prolog/ can be called with additional heap, local stack, global stack, atom area and auxiliary stack, with the flags

```
-h  heap
-g  global stack
-l  local stack
-t  trail
-a  atom area
-x  auxiliary stack
```

For instance, if you run out of heap space or global or local stack, use

```
%  prolog -h 5000 -l 5000 -g 5000
```

If PROP is in the file `propfilename`, use the command

```
| ?- reconsult('propfilename').
```

to load PROP.

For the LCL Bobcats, PROP can be found in

```
/net/prospero/usr/local/prolog/ProP/prop.pl
```

and the sample grammar in

```
/net/prospero/usr/local/prolog/ProP/gram.pl
```

B Context-Free and Definite Clause Grammars

Context-free grammars (CFG) are just a special case unification grammars in which every category is atomic. To encode a context-free grammar, just use constants for the description of every category in every rule and every lexical entry.

PROP is general enough to encode *definite clause grammars (DCG)* of the sort often found in PROLOG implementations. This is because general PROLOG terms may be used in place of constants in atomic FSMs. The only thing to keep in mind here is that the subsumption check is not implemented with DCGs in mind and will assume a subsumption holds if two categories can be unified.

For example, a simple PROP DCG encoding number agreement between subjects and verbs would be:

```
john ---> subj(sing).       parliament ---> subj(plu).
votes ---> pred(sing).      vote ---> pred(sing).
c1,c2 ===> c3 if c1:subj(X), c2:pred(X), c3:s.
```

It should be noted that when printing out a category which is merely a variable nothing will be displayed just as with the minimal satisfier of the bottom description.

Of course, PROP uses the same parser for these grammars and thus provides a bottom-up interpretation of DCGs and CFGs.

C Sample Grammar

```
% RULES
% -----

np, vp ===> s if
   np: @subj_np,
   vp:(@vp,@finite),
   s: @s,
   [np,syn,head,agr] == [vp,syn,head,agr],
   [s,syn,head] == [vp,syn,head],
   [s,sem] == [vp,sem],
   [s,sem,subj] == [np,sem].

det, n ===> np if
  det: @det,
  n: @n,
  np: @np,
```

```
   [det,syn,head,agr] == [n,syn,head,agr],
   [np,syn,head] == [n,syn,head],
   [np,sem,quant] == [det,sem],
   [np,sem,restr] == [n,sem].

iv ===> vp if
   iv:(@verb,
       syn:subcat:1),
   vp: @vp,
   [vp,syn,head] == [iv,syn,head],
   [vp,sem] == [iv,sem].

tv, np ===> vp if
   tv:(@verb,
       syn:subcat:2),
   np: @obj_np,
   vp: @vp,
   [vp,syn,head] == [tv,syn,head],
   [vp,sem] == [tv,sem],
   [vp,sem,obj] == [np,sem].

bv, np1, np2 ===> vp if
   bv: (@verb,
        syn:subcat:3),
   np1: @obj_np,
   np2: @obj_np,
   vp: @vp,
   [vp,syn,head] == [bv,syn,head],
   [vp,sem] == [bv,sem],
   [vp,sem,obj] == [np2,sem],
   [vp,sem,obj2] == [np1,sem].

aux, vp1 ===> vp2 if
   aux: (@verb,
         syn:subcat:4),
   vp1: @vp,
   vp2: @vp,
   [aux,syn,aux] == [vp1,syn,head,vform],
   [vp2,syn,head] == [aux,syn,head],
   [vp2,syn,head,agr] == [vp1,syn,head,agr],
   [vp2,sem,xcomp] == [vp1,sem],
   [vp2,sem] == [aux,sem].

cv, inf ===> vp if
   cv: (@verb,
        syn:subcat:5),
   inf: (@vp,
         syn:head:vform:inf),
   vp: @vp,
   [vp,syn,head] == [cv,syn,head],
   [vp,syn,head,agr] == [inf,syn,head,agr],
```

```
   [vp,sem] == [cv,sem],
   [vp,sem,xcomp] == [inf,sem],
   [vp,sem,subj] == [inf,sem,subj].

adv,vp1 ===> vp2 if
   adv: (@adv,
         syn:order:pre),
   vp1: @vp,
   vp2: @vp,
   [vp2,syn,head] == [vp1,syn,head],
   [vp2,sem,rel] == [adv,sem],
   [vp2,sem,xcomp] == [vp1,sem].

vp1,adv ===> vp2 if
   vp1: @vp,
   adv: (@adv,
         syn:order:post),
   vp2: @vp,
   [vp2,syn,head] == [vp1,syn,head],
   [vp2,sem,rel] == [adv,sem],
   [vp2,sem,xcomp] == [vp1,sem].

adj, n1 ===> n2 if
   adj: @adj,
   n1: @n,
   n2: @n,
   [n2,syn,head] == [n1,syn,head],
   [n2,sem,rel] == [adj,sem],
   [n2,sem,xcomp] == [n1,sem].



% LEXICON
% -------

% Proper Names
% ------------
uther --->
   @name,
   sem:uther.
lancelot --->
   @name,
   sem:lancelot.

% Pronouns
% --------
it --->
   @np, @third_sing, @norm, @pro.
he --->
   @subj_np, @third_sing, @norm, @pro.
him --->
```

```
   @obj_np, @third_sing, @norm, @pro.

% Indexicals
% ----------
i --->
   @subj_np, @first, @sing, @norm,
   sem:speaker.
me --->
   @obj_np, @first, @sing, @norm,
   sem:speaker.
you --->
   @np, @second, @norm,
   sem:hearer.
we --->
   @subj_np, @first, @plu, @norm,
   sem:speaker_and_hearer.
us --->
   @obj_np,  @first, @plu, @norm,
   sem:speaker_and_hearer.

% Expletives
% ----------
it --->
   @subj_np, @it,
   sem:dummy.

% Determiners
% -----------
the --->
   @det, @third, @norm,
   sem:the.
every --->
   @det, @third_sing, @norm,
   sem:every.
all --->
   @det, @third, @plu, @norm,
   sem:all.

% Nouns
% -----
sheep --->
   @n,
   sem:sheep.
knight --->
   @n, @sing,
   sem:knight.
knights --->
   @n, @plu,
   sem:knights.

% Adjectives
```

```
% ----------
tall --->
  @adj,
  sem:tall.

% Intransitive Verbs
% ------------------
feast --->
  @verb, @norm, @base,
  syn:subcat:1,
  sem:rel:feast.
feasted --->
  @verb, @norm, @past,
  syn:subcat:1,
  sem:rel:feast.
feasts --->
  @verb, @norm, @third_sing, @present,
  syn:subcat:1,
  sem:rel:feast.
feast --->
  @verb, @norm, @non_third_sing, @present,
  syn:subcat:1,
  sem:rel:feast.
feasting --->
  @verb, @norm, @present_participle,
  syn:subcat:1,
  sem:rel:feast.
feasted --->
  @verb, @norm, @past_participle,
  syn:subcat:1,
  sem:rel:feast.

% Expletive Intransitive Verbs
% ----------------------------
rain --->
  @verb, @it, @base,
  syn:subcat:1,
  sem:rel:rain.
rained --->
  @verb, @it, @past,
  syn:subcat:1,
  sem:rel:rain.
rains --->
  @verb, @it, @present,
  syn:subcat:1,
  sem:rel:rain.
raining --->
  @verb, @it, @present_participle,
  syn:subcat:1,
  sem:rel:rain.
rained --->
```

```
   @verb, @it, @past_participle,
   syn:subcat:1,
   sem:rel:rain.

% Transitive Verbs
% ----------------
fight --->
   @verb, @norm, @base,
   syn:subcat:2,
   sem:rel:fight.
fought --->
   @verb, @norm, @past,
   syn:subcat:2,
   sem:rel:fight.
fights --->
   @verb, @norm, @third_sing, @present,
   syn:subcat:2,
   sem:rel:fight.
fight --->
   @verb, @norm, @non_third_sing, @present,
   syn:subcat:2,
   sem:rel:fight.
fighting --->
   @verb, @norm, @present_participle,
   syn:subcat:2,
   sem:rel:fight.
fought --->
   @verb, @norm, @past_participle,
   syn:subcat:2,
   sem:rel:fight.

% Bintransitive Verbs
% -------------------
give --->
   @verb, @norm, @base,
   syn:subcat:3,
   sem:rel:give.
gave --->
   @verb, @norm, @past,
   syn:subcat:3,
   sem:rel:give.
gives --->
   @verb, @norm, @third_sing, @present,
   syn:subcat:3,
   sem:rel:give.
give --->
   @verb, @norm, @non_third_sing, @present,
   syn:subcat:3,
   sem:rel:give.
giving --->
   @verb, @norm, @present_participle,
```

```
   syn:subcat:3,
   sem:rel:give.
given --->
   @verb, @norm, @past_participle,
   syn:subcat:3,
   sem:rel:give.

% Subject Raising Verbs
% --------------------
seem --->
   @verb, @base,
   syn:subcat:5,
   sem:rel:seem.
seemed --->
   @verb, @past,
   syn:subcat:5,
   sem:rel:seem.
seems --->
   @verb, @present,
   ((@third_sing, @norm)
   ; @it),
   syn:subcat:5,
   sem:rel:seem.
seem --->
   @verb, @norm, @non_third_sing, @present,
   syn:subcat:5,
   sem:rel:seem.
seeming --->
   @verb, @present_participle,
   syn:subcat:5,
   sem:rel:seem.
seemed --->
   @verb, @past_participle,
   syn:subcat:5,
   sem:rel:seem.

% Auxiliaries
% -----------
did --->
   @verb, @past,
   syn:subcat:4,
   syn:aux:base.
do --->
   @verb, @non_third_sing, @present,
   syn:subcat:4,
   syn:aux:base.
does --->
   @verb, @third_sing, @present,
   syn:subcat:4,
   syn:aux:base.
```

```
have --->
  @verb, @base,
  syn:subcat:4,
  syn:aux:past_part.
had --->
  @verb, @past,
  syn:subcat:4,
  syn:aux:past_part.
have --->
  @verb, @non_third_sing, @present,
  syn:subcat:4,
  syn:aux:past_part.
has --->
  @verb, @present,
  ((@third_sing, @norm)
  ; @it),
  syn:subcat:4,
  syn:aux:past_part.

to --->
  @verb, @inf,
  syn:subcat:4,
  syn:aux:base.

be --->
  @verb, @base,
  syn:subcat:4,
  syn:aux:pres_part.
was --->
  @verb, @past, @sing, @norm,
  syn:subcat:4,
  syn:aux:pres_part.
was --->
  @verb, @past, @it,
  syn:subcat:4,
  syn:aux:pres_part.
were --->
  @verb, @past, @norm,
  (@second
  ;@plu),
  syn:subcat:4,
  syn:aux:pres_part.
am --->
  @verb, @present, @first, @sing, @norm,
  syn:subcat:4,
  syn:aux:pres_part.
is --->
  @verb, @present, @third_sing, @norm,
  syn:subcat:4,
  syn:aux:pres_part.
are --->
```

```
   @verb, @present, @norm,
   (@second
   ;@plu),
   syn:subcat:4,
   syn:aux:pres_part.
 being --->
   @verb, @present_participle,
   syn:subcat:4,
   syn:aux:pres_part.
 been --->
   @verb, @past_participle,
   syn:subcat:4,
   syn:aux:pres_part.

% Adverbs
% -------
slowly --->
   @adv,
   sem:slowly.
probably --->
   @adv,
   syn:order:pre,
   sem:probably.
yesterday --->
   @adv,
   syn:order:post,
   sem:yesterday.



% MACROS
% ------

bar0  macro  syn:foot:bar:0.
bar1  macro  syn:foot:bar:1.
bar2  macro  syn:foot:bar:2.

pro  macro  sem:pro.

first   macro  syn:head:agr:pers:1.
second  macro  syn:head:agr:pers:2.
third   macro  syn:head:agr:pers:3.

sing  macro  syn:head:agr:num:sing.
plu   macro  syn:head:agr:num:plu.

third_sing      macro  @third, @sing.
non_third_sing  macro  @plu ; @second ; @first.

finite  macro  syn:head:vform:fin.
base    macro  syn:head:vform:base.
```

```
inf     macro  syn:head:vform:inf.

present  macro  @finite,  sem:tim:present.
past     macro  @finite,  sem:tim:past.

present_participle  macro
  syn:head:vform:pres_part,
  sem:aspect:progressive.
past_participle macro
  syn:head:vform:past_part,
  sem:aspect:perfect.

norm  macro  syn:head:agr:nform:norm.
it    macro  syn:head:agr:nform:it.

nominal      macro  syn:head:maj:nom:yes.
non_nominal  macro  syn:head:maj:nom:no.

verbal      macro  syn:head:maj:verb:yes.
non_verbal  macro  syn:head:maj:verb:no.

n   macro  @nominal, @non_verbal, @bar1.
np  macro  @nominal,  @non_verbal,   @bar2.

name  macro  @np,  @third_sing,  @norm.

subj_np  macro  @np,  syn:head:case:subj.
obj_np   macro  @np,  syn:head:case:obj.

verb  macro  @verbal, @non_nominal, @bar0.
vp    macro  @verbal, @non_nominal, @bar1.
s     macro  @verbal, @non_nominal, @bar2.

det  macro  syn:head:maj:det.
adv  macro  syn:head:maj:adv.
adj  macro  syn:head:maj:adj.
```

Acknowledgements

I would like to thank Alex Franz for comments on two earlier drafts. The fact that this piece of software exists in usable form is due to Lori Levin, who wanted a PATR-II parser for her Natural Language Processing course at Carnegie Mellon. PROP was developed on a Hewlett-Packard Bobcat 350 workstation using C-Prolog.

References

Allen, J, 1987. *Natural Language Understanding.* Benjamin Cummings, Menlo Park.

Kasper, R. T. and W. C. Rounds, 1986. A logical semantics for feature structures. In *Proceedings of the 24th Annual Conference of the Association for Computational Linguistics*, pages 235–

242.

Knight, K, 1989. Unification: A Multidisciplinary Survey. *ACM Computing Surveys*, Vol. 21, No. 1, pages 93–124.

Shieber, S. M., 1986. *An Introduction to Unification-Based Approaches to Grammar.* CSLI Lecture Notes, Vol. 4, Chicago University Press, Chicago.

Shieber, S. M., H. Uszkoreit, F. C. N. Pereira, J. Robinson, and M. Tyson, 1983. The formalism and implementation of PATR-II. In *Research on Interactive Acquisition and Use of Knowledge*, SRI International, Menlo Park, California.