# PARSING PROLOG TRACES
# USING A DCG GRAMMAR

Ephraim Paz

1986

Cognitive Studies Research Paper

Serial No. CSRP. 055

The University of Sussex,
Cognitive Studies Programme,
School of Social Sciences

Ephraim Paz,
Cognitive Studies programme,
University of Sussex

February 1986

Strictly speaking, a Prolog system, regarded as a theorem produces very succinct answers to queries posed to it : e or NO. This, naturally, is not very illuminating, and ever interpreter produces also the values with which the variab unified in order to reach a positive answer.

In order to make the system's answer even more plausible relatively easy to devise a mechanism for supplying the execut of a program, and to present it to the user.

I am currently building a system that will explanations for results of Prolog program results, w have many advantages over existing explanation mechanisms explanations will give better procedural understanding the program's behaviour and its reasoning process, as well a insight into the declarative meaning of the predicates an that constitute it.

A central element of this system is an analysis and process applied to a Prolog program trace. The parser is a DCG grammar [PEREIRA & WARREN 80]. This paper will desc grammar and the data structure that is generated by the process which utilises this grammar. This data structure is Explanation, which is an intermediate representation of the ex and serves as a blue print for the final product of the syst English explanation.

A brief description of the explanation system will be section 2 of the paper. Section 3 includes the grammar, desc parsing process and the building of the Raw Explanation. Sectio with the conclusions.

## 2. THE EXPLANATION SYSTEM.

The general approach employed includes three major ele extending the expressive power of the language, creati explanation by parsing the execution trace according to a gr traces, and dynamic generation of English explanations from explanation.

These are the steps taken by the system :

```
  PROGRAM    ANNOTATIONS
      |__________|
           |
           V
   ANNOTATED PROGRAM ("rule" facts)
           |
           V (interpreter)
         TRACE
           |
           V (parser)
   RAW EXPLANATION
           |
           V ( English generation)
  ENGLISH EXPLANATION
```

Extending the expressive power of the Prolog s achieved by supplying taxonomies and additional informati different elements of the language, like goals, rules, and proc

This information is stored as annotations attached to the recncate's Key, the type of a rule (the exact nature of relati etween a rule's head and its body), and the role of a rule insid rocedure.

The input for the interpreter is the result of a transformat f the Prolog source, in which clauses are represented ccurrences of the predicate (rule/4) :

```
rule(Type,Role,Head,Body),
```

A Prolog fact has the constant 'true[1] as the last argument, ddition to the regular functions of a Prolog interpreter, xtended interpreter used by the system creates a trace of xecution. The trace includes all the successful invocations Dais, and records of the instantiation status of all the variables goal at the moment of invoking that goal. In fact, any Pn iterpreter could do this and I do not assume anything non-standai ?cause this information is created by the interpreter in any ca: le fact that I do not interpret the original Prolog clauses but )ove mentioned extended "image[11] is a pragmatic way of combining * /pe and role annotations with the clauses and facts.

The trace is a tree-structure. The root of this tree is the tr< P the goal presented as a question to the program. The traces of si >al which had been activated to achieve this goal are sub-trees.

As an example, consider the following simple program :

```
ı(X,Y) :-b(X),c(Y).
)(X) :-d(X).
:(coo).
i(doo).
```

"his program will appear as :

```
*ule(t1,r1,a(X,Y),(b(X),c(Y))).
·ule(t2,r2,b(X),d(X)).
·ule(_,_,c(coo),true).
kule(_,_,d(doo),true)•
```

And here is the trace of the goal a(X,doo) : (Indented i •nvenient reading)

```
Ctrace(C-,+],t1,r1,a(boo,doo)),
    [trace(C-D,t2,r2,b(doo)),
        Ctrace(C-],_,_,d(doo))]],
    Ctrace(OD,_,_,c(coo))]].
```

This trace will be handed over to the next element in t stem- the parser that builds the Raw Explanation, which will scribed in Section 3.

The output of the parsing process is a data-structure which I ca e Raw Explanation. This data structure will be submitted to ecialised natural language generator that will produced an Engli planation according to the structure of the Raw Explanation. Th nerator will use canned templates for the predicates, rhetorical rul II direct the building of sentences out of goals or groups of goal d the system will use different verbs to describe different actio the interpreter.

## 3. THE GRAMMAR.

A central element of the explanation system described in tl seeding section is a program that analyses the execution trace ai

Raw Explanation can be regarded as as a blue print for
explanation, since its contents as well as its structure w
and direct the last phase - that of generating the natura
explanation.

The trace analyser is expressed as a Definite Claus
[Pereira and Warren 80], and the notation is the Prolog Gr
Notation as it appears in chapter 9 of [CLOCKSIN & MELLI
Because the input is not a linear list but can be a nested l
is a special rule that indicates that if an element is a list
be treated recursively, element by element.

Each rule is combined of left hand side (LHS) and right
(RHS), and looks like this : LHS --> RHS. LHS is a
of the form C(a1,a2,...an) :T, and RHS is one or more such
possibly followed by Prolog goals surrounded by curly brackets

The connector ":" is declared as a Prolog
op(10,XFX,:). C(a1,a2,...an) is a grammatical category,
arguments a1...an are additional bits of information that
helpful for the explanation- like the type, the role and the
the element described by the category. Examples of categori
trace, head, body, goal, predicate, system predicate, i/o
and specific predicates like cut, write, or user defined
of the executed program.

The second argument of the ":" term is the resulti
tree. It usually starts with the category name and so
arguments, followed by the parse trees of the elements that
this category.

Example :

```
trace_rule(Type):[rule(Type),H,B]-->trace_head(Type):H,tra
```

This rule expresses the fact that the trace of a
composed of a trace of a head and a trace of a body, and the
parse tree will be a list starting with the word "rule", with
followed by the lists that will result from parsing the he
body.

Here is a fragment from the grammar, which invo
following main categories:

- ltr - legal trace.
- t_g - trace of a goal. 1 argument - type(fact or head)
- t_r - trace of a rule. 4 arguments:
  - Instantiation Status of the rule's head.
  - Rule Type (e.i. recursive, cut-type rule)
  - Rule Role in procedure (e.i. exception, catch-a
  - Rule Head.
- t_h - trace of a rule head.
- t_b - trace of a rule body.
- t_f - trace of a fact. 4 arguments:
  - type of fact
  - Action Type (Generate, Retrieve, or Test)
  - Role in procedure (e.i. catch-all).
  - Role in clause (e.i. condition).

The main rules are:

```
ltr :[ltr, G] --> t_g(Goal) : G.

t_g(F) :F --> t_f(fact, ActionType,Role1,Role2) :F.
t_g(Head) :R --> t_r(Is,RuleType,Rulerole,Head) :R.

t_r(Is,RuleType,Rulerole,Head) :[rule(RuleType),H,B] -->
```

```
                    t_b :B.

_h(Head) :[head,Head] --> t_f(head,_,_,head) :P.

_b : [body,G|B] --> t_g(Goal) : G ,t_b : [body|B].
_b : [] --> [].

_f(head,_,_,head) :[head,Head] --> [Head],!,{is_head(Head)}.
_f(fact, ActionType,_,_) :[fact(ActionType),FACT] -->
                   [trace(IS,_,_,FACT)],!,
                   {is_fact(FACT)},
                   {find_action_type(FACT,IS,ActionType)}.

s_head(Head) :-rule(_,_,Head,Body),Body = true.
s_fact(Fact) :-rule(_,_,Fact,true).
_cut-->[!].
```

Here are some examples of rules that identify and process speci tterns that appear in the trace:

```
==== CATCH ALL CASE ======
_g(F) :[F,Auxtree] --> t_f(fact, ActionType,catch_all,_) :F,!,
                             {buildaux(F,Auxtree)}.
```

Catch-all rule : The attribute of role in procedure w ntioned in Sec.2. This rule demonstrates one of the justificatic r including this information as additional annotation.

When explaining how the program had come to its conclusion, it m und quite unclear why this last clause had been chosen. The resu using this rule is the addition of a sub-tree which consists of t bel "chosen because the following cases failed" and the list l the other cases in the procedure. A possible refinement of th le may include an investigation into the deeper reasons f fference between clauses in the procedures - i.e. if all t ads look alike with just one different argume (X,a),p(X,b),..) the difference can be traced down to the specif gument and explained accordingly.

```
==== CUT RULE ============
_r(Is,DecType,Rulerole,cut_rule,Head):[rule(RuleType),H,Bcond,Bmain]
                        -->
                    t_h(Is,T,R,Head) :H,
                    t_b :Bcond,
                    t_cut,
                    t_b :Bmain.
```

The cut rule : If one of the goals in a Rule's body is a cu many cases, the goals which appear before the cut serve nditions for choosing this rule,and the cut is used for making su at once the rule has been chosen, if and when it fails no oth le will be tried. This is the case when the rules represe tually exclusive options, (as opposed to the case when they a dered in increasing generality order)

The cut itself as a goal is not instrumental in the explanatic nce its function is more of a control or "punctuation symbol d its trace is dropped from the explanation. The goals th ecede it get the attribute "condition" and a possible renderi to English of a typical cut rule will be something like :

"Since conditions A and B were satisfied, The rule for ca was activated, and the tax was calculated according to t explanation, and this analysis can not be used when a declara explanation is required. The basic problem is the lack of declara* semantics for the cut, and these rules are not recommendat for the way Prolog should be used, but rather dealing with prog as they are written in reality),

```
===== RECURSIVE RULE =====
t_r (Is,DecType,RuleroIe,recursive,Head):Crule(RuleType),H,Brecursiv<
                              ->
                          t_h(Is,T,R,Head) :H,
                          t_b :Cbody|List_of_goals3,
                          •Crecursive_rel(Head,List_of_goals)>,
                          {recursive_treatment(B,Brecursive)>.
```

Recursive Rules : The parser can identify parts of the execu trace which had been created by recursive rules. The importance this fact is two-fold : On the one hand it is illuminating just add a remark to the effect that a predicate was used recursively the other hand, a recursive trace of more than, say, two levels recursion may call for a special handling procedure. In many c< a whole chain of recursive steps can be "telescoped[11] into one gem statement. Or one could just drop the recursant part of the rule leave the operand.

```
===== PROLOG FACT ======
t_f(fact, ActionType,__,_) : [fact (ActionType),FACT] ->
                         [trace(IS,_,_,FACT)D,!,
                         {is_fact(FACT)>,
                         {find_action_type(FACT,IS,ActionType)>.
```

One of the functions of this rule is to classify the predicate, first distinction is between user and system predicates, for syj predicated will have their ready-made templates and explana* routines. System predicates are then sorted into further sub tyj and each type will get a special treatment in the final explanat As an example of a system predicate group is input-output predict like "write[11] or "nl". Such predicates can be omitted in cer cases (e.i. when a strictly declarative explanation is given).

"is" is a special system predicates which will cause explanation to produce a clause like "X was calculated as 7 * 3 " the trace X is 7 * 3.

There are also predicates used for comparison like < or >= they will drive the explanation system to use the verb "compare".

Another function embedded in this rule is that of creating action type of a predicate by combining information about instantiation status with knowledge about its key. The key is argument whose value determines a unique occurrence of the predict e.g, if the meaning of father(X,Y) is that X is the father of Y, th< will be the key). There are three action types: GENERATE, RETRII and TEST. When a goal is evoked with an uninstantiated key, (as father(jacob,Y)), one can assume that the function of this goal ii generate a possible candidate for a solution. When a goal is ev< with the key already instantiated, but one or more of the o arguments uninstantiated, the function of the goal is usually retrieve the missing information and so to "complete" predicate (as is the case in father(X,benjamin)• If the goal fully instantiated when called (as in father(jacob,benjamin)), aim of this action is to test or confirm some fact.

## 4. CONCLUSION.

DCG rules are a convenient representation for many kinds
ıformation. This grammar is used here in a parser that identifi
ructures in a Prolog extended trace.

This paper describes how this representation is used
ıpture information and "meta-knowledge" about Prolog program
ıe grammar is integrated in a program that analyses a trace of Prol
ogram which had been interpreted by an extended interpreter. T
ogram produces a blue-print for an explanation for the behaviour
ıat program. This blue print will be the input for a natural langua
nerator which will produce the final product of the system.

The core of the program is a DCG parser. Once a structure
lentified, the Raw Explanation building mechanism takes t
ıpropriate action. This action may be simply an omission
rtain parts of the trace, or more complicated actions li
trieving related information and adding additional parts to t
planation.
Many implicit features, become explicit and can be integrat
the explanation, thanks to the knowledge embedded in t
ammar. Some of the examples mentioned in this paper are the usa
instantiation status to infer the exact nature of action done whi
ifying facts with facts, and the "catch-all" role of a rule, whi
monstrates implicit procedural knowledge utilised by the programme.
The modularity of the grammar representation is ve
nvenient, and additional structures can be added simply by addi
re rules to the grammar. There is also "vertical" modularity - th
rt of the system generates an intermediate data structure th
ll be processed by the natural language generator, somewhat alo
e lines recommended in [McDonald,82].

REFERENCES


CLOCKSIN & MELLISH, 81
Programming in Prolog
Springer-Verlag

McDONALD,82 D.D. McDonald
'Natural Language Generation as a Computational Problem: an Introduction'
in Brady (ed.): Computational Theories of Discour
MIT Press.

PAZ,85 E.PAZ
A System for Generating Explanations of Prolog Program Results.
Cognitive Studies Research Paper, University of Su

PEREIRA & WARREN, 80
Definite Clause Grammar Compared with Augmented Transition Networks
Artificial Intelligence 13(3).