DETERMINISTIC AND BOTTOM-UP PARSING IN PROLOG

Edward P. Stabler, Jr.

University of Western Ontario
London, Canada

ABSTRACT

It is well known that top-down backtracking context free parsers are easy to write in Prolog, and that these parsers can be extended to give them the power of ATN's. This report shows that a number of other familiar parser designs can be very naturally implemented in Prolog. The top-down parsers can easily be constrained to do deterministic parsing of LL(k) languages. Bottom-up backtrack parsers can also be elegantly implemented and similarly constrained to do deterministic LR(k) parsing. Very natural extensions of these LR(k) parser designs suffice for deterministic parsing of natural languages of the sort carried out by the Marcus(1980) parser.

I INTRODUCTION

Pereira and Warren(1980) have shown that Prolog provides facilities for writing top-down backtracking context free parsers.* For example, a simple context free grammar like (CFG1) is easily converted into the Prolog code for the corresponding parser (CFP1):

```
(CFG1)
s --> np vp               det --> your
np --> det n              n --> claim
np --> det n rel          comp --> that
rel --> comp s            v --> is
vp --> v adj              adj --> funny
```

```
(CFP1)
s([s,NP,VP],P0,P):-np(NP,P0,P1),vp(VP,P1,P).
np([np,Det,N],P0,P):-det(Det,P0,P1),n(N,P1,P).
np([np,Det,N,Rel],P0,P):-det(Det,P0,P1),n(N,P1,P2),
                         rel(Rel,P2,P).
rel([rel,Comp,S],P0,P):-comp(Comp,P0,P1),s(S,P1,P).
vp([vp,V,Adj],P0,P):-v(V,P0,P1),adj(Adj,P1,P).
det([det,your],[your|T],T).
n([n,claim],[claim|T],T).
comp([comp,that],[that|T],T).
v([v,is],[is|T],T).
adj([adj,funny],[funny|T],T).
```

---

* The slight acquaintance with Prolog required for a complete understanding of this report will just be presumed. Prolog is coming to be fairly well known, and there are good introductions to the language (Clocksin and Mellish, 1981).

In the parser, the category symbols _s_, _np_, _vp_ and so on, are treated as three place predicates, where the first argument holds a labelled bracketting representing the derivation tree dominated by that category symbol, and where the string to be parsed is given in what remains of the list in the second argument when the list in the third argument place is taken off its tail. So given (CFP1), the Prolog query,

```
s(P,[your,claim,is,funny],[]).
```

is a request to parse a string generated by the grammar. The query will succeed and return the parse tree,

```
P = [s,[np,[det,your],[n,claim]],
       [vp,[v,is],[adj,funny]]]
```

Since Prolog tries the rules of the grammar in order, it will first try to get a successful parse with the first rule for _np_, and if this attempt fails (as it will for any _np_ containing a _rel_), the parser will backtrack and the second rule will be tried.

This kind of parser has all of the problems of standard context free top-down backtrack parsers: left recursive rules can cause looping; the language generated by a context free grammar will not generally be the language that is parsable by the corresponding parser (even when the parser does not loop); and, like any backtracking parser, this kind of parser can be quite inefficient (Aho and Ullman, 1972). Pereira and Warren(1980) have pointed out that we can increase the power of such a Prolog parser as much as we like by elaborating the parser rules. It is not hard, for example, to elaborate the rules of such a parser to produce a parser that is equivalent to an ATN. A number of projects have used elaborations of Prolog top-down context free parsing capabilities with some success (Pereira and Warren, 1980), but since other parser designs have particular advantages, it is of interest to consider whether Prolog is a good implementation language for them. Deterministic parsers can be substantially more efficient than backtrack parsers, and there is some reason to think that deterministic bottom-up parser designs are particularly well suited for natural languages (Marcus, 1980). This report will show how these parser designs can be implemented very naturally in Prolog.

II LL(K) PARSING

As was noted, the simple parser presented above will have to backtrack to parse any np containing a rel, but it is clear that the parser could always avoid this backtracking if it could look ahead 3 symbols: All np's will begin with a det and a n, and only np's containing a rel will have a comp after the n. It is easy to elaborate (CFP1) so that it performs this lookahead to deterministically parse the language generated by (CFG1) as an LL(3) language. We need only reverse the two rules for np so that the rule for the np complements will be considered first, and change that rule so that it will apply only if the comp "that" is the third word waiting to be parsed:

```
    (CFP2)
    np([np,Det,N,Rel],[W1,W2,that|T],P):-
          !,det(Det,[W1,W2,that|T],P1),
          n(N,P1,P2),
          rel(Rel,P2,P).
    np([np,Det,N],P0,P):-!,det(Det,P0,P1),
                        n(N,P1,P).*
```

Obviously, we could use this technique to perform any test we like on the first k symbols of the string remaining to be parsed, so this sort of implementation will always suffice for LL(k) parsing. Notice that the Prolog representation of the function from the first k symbols of the string to the parser rule to be used is quite perspicuous.

III BOTTOM-UP PARSING

Bottom-up parsers for context free languages are quite popular. The simplest and best-known are the "shift-reduce" parsers which "shift" a symbol from the input stream into a stack and then try to "reduce" the structures at the top of the stack to higher level structures using the grammar rules; when no more reductions of the symbols at the top of the stack are possible, another symbol is shifted from the input stream, and so on until the input stream is empty (or until some final punctuation is reached) and the stack contains only a sentence structure. The following Prolog parser for (CFG1) neatly captures this design:

```
(CFP3)
start([I|X],P):-parse(X,[I],P).

parse([I|X],Stack,P):-reduce(Stack,Newstack),
                      parse(X,[I|Newstack],P).
parse([],Stack,[s|S]):-reduce(Stack,[[s|S]]).

reduce([X|Y],[X2|Y2]):-
     reduce1([X|Y],[X1|Y1]),
     reduce([X1|Y1],[X2|Y2]).
reduce(X,X).

reduce1([your|X],[[det,your]|X]).
reduce1([claim|X],[[n,claim]|X]).
reduce1([[n,N],[det,Det]|X],
        [[np,[det,Det],[n,N]]|X]).
reduce1([that|X],[[comp,that]|X]).
reduce1([[s|S],[comp,Comp]|X],
        [[rel,[comp,Comp],[s|S]]|X]).
reduce1([[rel|Rel],[n,N],[det,Det]|X],
        [[np,[det,Det],[n,N],[rel|Rel]]|X]).
reduce1([[vp|VP],[np|NP]|X],
        [[s,[np|NP],[vp|VP]]|X]).
reduce1([[adj,Adj],[v,V]|X],
        [[vp,[v,V],[adj,Adj]]|X]).
reduce1([is|X],[[v,is]|X]).
reduce1([funny|X],[[adj,funny]|X]).
```

The predicate start takes a list containing the string to be parsed (the input buffer) as its first argument, and will return the labelled bracketting for that string as its second argument. The predicate parse keeps the remaining input as its first argument, the active node stack as its second argument, and will return the finished labelled bracketting as its third argument. The parsing begins when start puts the first symbol of the input string into the active node stack and calls the procedure parse. Parse reduces the stack and then shifts the next input symbol onto the stack, until the input buffer is empty and the active node stack contains just one structure dominated by the sentence category s. The predicate reduce simply calls the reduce1 rules until they no longer apply, and the reduce1 rules simply match their first arguments with structures in the stack corresponding to the right-hand sides of (CFG1) grammar rules and return as their second argument a structure dominated by the category on the left hand side of the grammar rule. Thus, given the query,

```
        start([your,claim,is,funny],P).
```

this parser will shift "your" onto the stack, and the first reduce1 rule will apply to reduce "your" to the structure "[det,your]". Another symbol is then shifted from the input onto the stack, and so on until the stack contains the same labelled bracketting that (CFP1) produces for the string.

IV LR(K) PARSING

Notice that (CFP3) is a backtracking shift-reduce parser; it will have to backtrack whenever it parses a string containing a np complement. This is so because whenever it gets a noun structure and a determiner structure at the top of the stack, it will reduce them to a nounphrase structure regardless of whether a complement is waiting in the input buffer to be parsed next. This backtracking can be eliminated, though, with one symbol of lookahead. We simply need to constrain the reduce1 rule that performs the np reduction so that it will not apply if there is a comp at the head of the input buffer. This requires that the reduce1 rules apply not only according to what is at the top of the stack but also according to what is in the input buffer; the new reduce1 rules accordingly have the input buffer

---

* The Prolog cut symbol, "!", blocks bactracking. Since this parser is deterministic, we want to begin every rule with a cut. This will have the desired effect of causing the parser to abandon any parse which tries to use a rule and fails.

as their first argument. Since every det n sequence allowed by (CFG1) is followed either by the comp "that" or the v "is", the easiest way to avoid backtracking is simply to allow the simple noun phrase reduction (i.e., the application of the third reduce1 rule) only when the symbol "is" is at the head of the input buffer:

```
(CFP4)
start([I|X],P):-parse(X,[I],P).

parse([I|X],Stack,P):-reduce([I|X],Stack,Newstack),
                       !,parse(X,[I|Newstack],P).
parse([],Stack,[s|S]):-reduce(Input,Stack,[[s|S]]).

reduce(Input,[X|Y],[X2|Y2]):-
    reduce1(Input,[X|Y],[X1|Y1]),!,
    reduce(Input,[X1|Y1],[X2|Y2]).
reduce(Input,X,X).

reduce1(Input,[your|X],[[det,your]|X]).
reduce1(Input,[claim|X],[[n,claim]|X]).
reduce1([is|I],[[n,N],[det,Det]|X],
        [[np,[det,Det],[n,N]]|X]).
reduce1(Input,[that|X],[[comp,that]|X]).
reduce1(Input,[[s|S],[comp,Comp]|X],
        [[rel,[comp,Comp],[s|S]]|X]).
reduce1(Input,[[rel|Rel],[n,N],[det,Det]|X],
        [[np,[det,Det],[n,N],[rel|Rel]]|X]).
reduce1(Input,[[vp|VP],[np|NP]|X],
        [[s,[np|NP],[vp|VP]]|X]).
reduce1(Input,[[adj,Adj],[v,V]|X],
        [[vp,[v,V],[adj,Adj]]|X]).
reduce1(Input,[is|X],[[v,is]|X]).
reduce1(Input,[funny|X],[[adj,funny]|X]).
```

This is a deterministic LR(1) parser for (CFG1). Obviously, we could allow our reduce1 rules to perform whatever computations we wanted on the first k symbols of the Input and on the Stack, so this style of implementation will suffice for any LR(k) parsing. Again, the representation of the function from the first k symbols of the input and the active node stack to the parser rule to be used is perspicuous; it is equivalent to the standard LR table representations.

V THE MARCUS PARSER

Marcus(1980) showed how the basic design of an LR(k) parser can be extended to give it sufficient power to parse English in a very natural way. Most importantly: we allow arbitrary tests on the first k cells of the input buffer (in Marcus, 1980, k=5); we allow the parser to put parse structures into the cells of the input buffer as well as into the active node stack; we allow the parser to look arbitrarily deep into the active node stack; and we group the parser rules into packets in such a way that only rules from active packets can be executed, and these packets are activated and deactivated by the parser rules.

The rules of the Marcus parser are written in a language (Pidgin) which is compiled into Lisp. Each grammar rule has the form:

```
[test1][test2][test3][**c;test4] --> action
```

where test1 is a test performed on the structure in the first buffer cell, similarly for test2 and test3, and test4 is a test on the active node stack. In our Prolog implementation, corresponding to each of Marcus's grammar rules there is a parser rule of the form:*

```
reduce([First,Second,Third|Restbuffer],
      [[Packet,Activenode]|Reststack],Counter,P):-
    member(name,Packet),
    test1(First),
    test2(Second),
    test3(Third),
    test4([[Packet,Activenode]|Reststack]),
    reduce(Newbuffer,Newstack,Newcounter,P).
```

This form is appropriate since the active packets are associated with each node in the active node stack, so the member clause checks to make sure that the rule name is in the active packet; the tests on the first three buffer cells and the stack are done next, and if they succeed, the action consisting of appropriately modifying the input buffer and the stack is carried out, and reduce is called again. The counter is used simply to number the nodes as they are created and pushed onto the active node stack. The last argument of reduce is the completed parse which will be in the active node stack when final punctuation at the end of a grammatical string is picked up from the buffer.

The similarity between Marcus's grammar rules and our parser rules is easy to see. Consider for example, the following rules from Marcus(1980) and our corresponding Prolog rules:

```
{rule MAJOR-DECL-S in ss-start
[=np][=verb]-->
Label c s,decl,major.
Deactivate ss-start.Activate parse-subj.}

{rule YES-NO-Q in ss-start
[=auxverb][=np]-->
Label c s,quest,ynquest,major.
Deactivate ss-start.Activate parse-subj.}


/*MAJOR_DECL IN SS_START*/
reduce([[First|Tree1],[Second|Tree2]|Restbuffer],
      [[Packet,[Cat|Tree]]|Reststack],Counter,P):-
    remove_member(ss_start,Packet,Newpacket),
    member(np,First),
    member(verb,Second),!,
    append(Cat,[s,decl,maj],Newcat),
    reduce([[First|Tree1],[Second|Tree2]|
            Restbuffer],
           [[[parse_subj|Newpacket],[Newcat]]|
            Reststack],
           Counter,P).
```

---

* Actually, in our implementation we use additional argument places in the reduce predicate to handle Marcus's "attention shift" rules, which are used in nounphrase parsing, and also for holding case structures. These rules are beyond the scope of this discussion, but they do not affect the basic ideas presented here.

```
/*YES_NO_Q IN SS_START*/
reduce([[First|Tree1],[Second|Tree2]|Restbuffer],
       [[Packet,[Cat|Tree]]|Reststack],Counter,P):-
     remove_member(ss_start,Packet,Newpacket),
     member(auxverb,First),
     member(np,Second),!,
     append(Cat,[s,quest,ynquest,maj],Newcat),
     reduce([[First|Tree1],[Second|Tree2]|
              Restbuffer],
             [[[parse_subj|Newpacket],[Newcat]]|
              Reststack],
             Counter,P).
```

Our parser, like Marcus's, associates a list of features with each node in the tree structures, so that the tests on the buffer cells are simple list membership tests. If they succeed, the rule is run and the new state of the parser is defined in the call of the reduce procedure. No rule that runs will fail unless the input string is unacceptable, so backtracking over calls to reduce can be blocked. The parser operation is easy to follow, and the rules are easy to write.

Our current Prolog implementation of the Marcus parser is not particularly fast, but it should be extendable to get greater coverage of the language without the substantial increases in parsing time that would be expected with extensions of a strictly top-down parser. The following strings were parsed by our compiled DEC-10 Prolog implementation in the times indicated:

```
[john,has,scheduled,a,meeting,for,wednesday]
P = [s(1),[np(2),john],[aux(3),[perf(4),has]],[vp(5
),[verb(6),scheduled],[np(7),a meeting],[pp(8),for
wednesday]],[finalpunc,.]]
time=70ms

[the,meeting,has,been,scheduled,for,wednesday]
P = [s(1),[np(2),the meeting],[aux(3),[perf(4),has]
,[passive(5),been]],[vp(6),[verb(7),scheduled],[np(
8),trace boundto([the,meeting])],[pp(9),for wednesd
ay]],[finalpunc,.]]
time=84ms

[the,meeting,seems,to,have,been,scheduled,for,wedne
sday]
P = [s(1),[np(2),the meeting],[aux(3)],[vp(4),[verb
(5),seems],[np(16),[s(6),[np(7),trace boundto([the,
meeting])],[aux(8),[to(9),to],[perf(10),have],[pass
ive(11),been]],[vp(12),[verb(13),scheduled],[np(14)
,trace boundto([boundto([the,meeting])])],[pp(15),f
or wednesday]]]],[finalpunc,.]]
time=145ms
```

REFERENCES

[1] Aho, A.V. and J.D. Ullman (1972) The Theory of Parsing, Translation, and Compiling, Volume 1: Parsing. Englewood Cliffs, NJ: Prentice-Hall.

[2] Clocksin, W.F. and C.S. Mellish (1981) Programming in Prolog. New York: Springer-Verlag.

[3] Marcus, M. (1980) A Theory of Syntactic Recognition for Natural Language. Cambridge, MA: MIT Press.

[4] Pereira, F.C.N. and D.H.D. Warren (1980) Definite clause grammars for natural languages. Artificial Intelligence, 13, pp.231-278.