Head-Driven Parsing

Martin Kay

Xerox Palo Alto Research Center and Stanford University

There are clear signs of a "Back to Basics" movement in parsing and syntactic generation. Our Latin teachers were apparently right. You should start with the main verb. This will tell you what kinds of subjects and objects to look for and what cases they will be in. When you come to look for these, you should also start by trying to find the main word, because this will tell you most about what else to look for.

In the early days of research on machine translation, Paul Garvin advocated the application of what he called the "Fulcrum" method to the analysis of sentences. If he was the last to heed the injunctions of his Latin teacher, it is doubtless because America followed the tradition of rewriting systems exemplified by context-free grammar and this provided no immediate motivation for the notion of the *head* of a construction. The European tradition, and particularly the tradition of Eastern Europe, where Garvin had his roots, tend more towards dependency grammar, but away from that of mathematical formalization which has been the underpinning of computational linguistics.

But the move now is towards linguistic descriptions that put more information in the lexicon so that grammar rules take on a more schematic quality. Little by little, we moved from rules like

```
(1) VP1 -> VP2 NP
      CaseOf(VP2) = Dative
      CaseOf(NP) = Dative
```

to rules that attain greater abstraction through the use of logical variables (or the equivalent), like

```
(2) VP1 -> VP2 NP
    ObjCase(VP2) = Case
    CaseOf(NP) = Case
```

Where the underlined Case is to be taken as the name of a variable. From there, it was a short step to

```
(3) VP1 -> VP2 X
    ComplementOf(VP2) = X
```

or even

```
(4) VP1 -> VP2 X
    ComplementStringOf(VP2) = X
```

Given rule (2), that parser knows what case the noun must have only after it has encountered the verb. Rules (3) and (4), do not even tell it that the complement must be a noun phrase. In (4) we cannot even tell how many complements ther will be. For most parsers, the problem is masked in these examples by the fact that they apply rules from left to right so that the value of the variable X is known by the time it is needed. In rule (4a), the matter is different.

```
(4a) VP1 -> X VP2
     ComplementStringOf(VP2) = X
```

Needless to say, these things have not gone unnoticed, least of all by the participants in this conference. It has been noted, for example, that definite-clause grammars can be adjusted so as to look for heads before complements and adjuncts. If the head of a sentence is a verb phrase, then it is sufficient to write (6) instead of (5).

```
(5) s(Left/Right) :-
      np(Left/Middle),
      vp(Middle/Right).
(6) s(Left/Right) :-
      vp(Middle/Right),
      np(Left/Middle).
```

A rule that expands the verb phrase would be something like (7).

```
(7) vp(Left/Right) :-
       verb(Left/Middle),
       np(Middle/Right).
```

This time, the order is the usual one because the head is on the left[1].

Of course, all this works if `Left`, `Middle`, and `Right` are something like word numbers that provide random access to the parts of the sentence. To make the system work with difference lists, we need something more, for example, as in (8).

```
(8) s(Left/Right) :-
       append(X, Middle, Left),
       vp(Middle/Right), np(Left/Middle).
```

---

[1] We have now moved to the Prolog convention of using caitalized names for variables.

The reason for the addition is that the parser, embodied here in the set of rules themselves, has no way to tell where the verb phrase will begin. It must therefore consider all possible positions in the string, an end which, against all expectation, is accomplished by the `append` predicate. If `append` is not needed when something like word numbers are used, it is because the inevitable search of the string is being quietly conducted by the Prolog system as it searches its data base, rather than being programmed explcitely.

The old-fashioned parser had no trouble finding the beginnings of things because they were always immediately adjacent, either to the boundaries of the sentence, or to another phrase whose position was already known. Given the sentence

I sold my car to a student of African languages whom I met at a party

and given appropriate rules, the head-driven parser will correctly identify "my car" as the direct object of "sold". But it will also consider for this role at least the following:

```
(8)   a student
      a student of African
      a student of African languages
      a student of African languages whom I met
      a student of African languages whom I met at a party
      African
      African languages
      African languages whom I met
      African languages whom I met at a party
      languages
      languages whom I met
      languages whom I met at a party
      a party
```

It will reject them only when it fails to extend them far enough to the left to meet the right-hand edge of the word "sold". Likewise, the last four entries on the list will be constructed again as possible objects for the preposition "of". As we shall see, this problem is not easy to put to set aside.

Of course, definite-clause grammars have other problems, when interpreted directly by a standard Prolog processor. The most notorious of these is that, in their classical form, they cycle indefinitely when provided with a grammar that involves left recursion. However this can be overcome by using a more appropriate interpreter such as the one given in Appendix A of this paper. It

does not touch the question of the additional work that has to be done in parsing a sentence.

Two solutions to the problem suggest themselves immediately. One is to use an undirected bottom-up parsing strategy, and the other is to seek an appropriate adaptation of chart parsing to a directed, head-driven, strategy. The first solution works for the simple reason that the problem we are facing simply does not arise in undirected bottom-up processing. There is no question of finding phrases that are adjacent to, or otherwise positioned relative to, other phrases. The strategy is a purely opportunistic one which finds phrases wherever, and whenever, its control strategy dictates. A simple chart parser with these properties is given in Appendix B. It accepts only unary and binary rules, but this is not a real restriction because these binary rules can function as meta-rules that interpret the more general of the actual grammar according to something like the following scheme. Real rules have a similar format to that used in the program in Appendix A, namely

```
rr(Mother, [L1, L2 ... Ln], Head, [R1, R2 ... Rm])
```

L_1 ... L_n are the non-head (complement) daughters of 'Mother' to the left of the head, and R_1 ... R_m are those to the right. For convenience, we give the ones on the left in the reverse of the order in which they actually appear so that the one nearest to the head is written first. We define the binary rule predicate referred to in the algorithm somewhat as follows;

```
rule(p(Mother, L, Rest), Head, Next) :-
   rr(Mother, L, Head, [Next|Rest]).
rule(p(Mother, Rest, []), Next, Head) :-
   rr(Mother, [Next|Rest], Head, []).
rule(p(Mother, L, T), p(Mother, L, [H|T]), H).
rule(p(Mother), H, p(Mother, [H|T], []).
```

One special unary rule is required, namely

```
rule(Mother, p(Mother, [], [])).
```

The scheme is reminiscent of categorial grammar. `p(Category, Left, Right)` is a partially formed phrase belonging to the given `Category` which can be completed by adding the items sepecified by the `Left` list on the left, and the `Right` list on the right.

This scheme has a certain elegance in that the parser itself is simple and does not reflect any peculiarities of head-driven grammar. Only the simple meta-rules given above are in any way special. Furthermore, the performance properties

of the chart parser are not compromised. On the other hand, this inactive chart parser cannot be extended to make it into an active chart parser in a straightforward manner as our second solution requires. This is the crux of the matter that this paper addresses.

Suppose that the verb has been located that will be the head of a verb phrase, but that it remains to identify one or two objects for it on the right. A standard active chart parser does this by introducing active edges at the vertex to the right of the verb which will build the first object if the material necessary for its construction is available, or comes to be available. As the construction procedes, active edges stretch further and further to the right intil the construction is complete and the corresponding inactive edge is introduced. This works only because the phrase can be built incrementally starting from the left, that is, starting next to the phrase to which it must be adjacent. But this strategy is not open to the head-driven parser which must begin by locating, or constructing the head of the new phrase. The rest of the phrase must then be constructed outwards from the head. We are therefore forced to modify the standard approach.

We propose to enrich the notion of a chart so that instead of simply active and inactive edges, it contains five different types of object. Edges can be *active* and *inactive*, but they can also be *pending* or *current*. This gives four of the five kinds. The fifth we shall refer to simply as a *seek*. It is a record of the fact that phrases with a given label are being sought in a given region of the chart. A seek contains a label and also identifies a pair of vertices in the chart. It is irrelevant at the level of generality of this discussion whether we think of the seek as actually being located in, or on, one of the vertices, or being representable as a transition between them. A condition that the chart is required to maintain is that edges with the same label as that of a seek, both of whose end points lie within the region of the seek, must be current. Edges which are not so situated must be pending. The standard chart regime never calls for information in a chart to change, but that is not the case here. When a new seek is introduced, pending edges are modified to become current as necessary to maintain the above invariant.

The fundamental rule (Henry Thompson's term) of chart parsing is that an action is taken, possibly resulting in the introduction of new edges, whenever the introduction of a particular new edge brings the operative end of an active edge together, at the same vertex, with an end of an inactive edge. If the label on the inactive edge is of the kind that the active edge can consume, a new

edge is introduced, possibly provoking new applications of the fundamental rule. The fundamental rule also applies in our enriched charts, but only to current edges–pending edges are ignored by it.

Suppose once again that a verb has been identified and that we are now concerned to find its sisters to the right. The verb can have been found only because there was a seek in existance for verbs covering the region in which it was found, and this, in its turn, will have come about because seeks were extant in that region for higher-level phrases, notably verb phrases. The objects we are now interested to locate must lie entirely in a region bounded on the left by the verb itself and, on the right, by the furthest right-hnd end of a VP seek that includes the verb. Accordingly, a new seek is established for NP's in this region. The immediate effect of this will be to make current any pending edges in that region that are inactive and labeled NP, or active and labeled with a rule that forms NP's.

It remains to discuss how active edges, whether current or pending, are introduced in the first place. The simplest solution seems to be to do this just as it would be in an undirected, bottom-up, parser. Whenever a current inactive edge is introduced, or a pending one becomes current, active edges are introduced, one for each rule that could accept the new item as head. However, these do not become current until a need for them emerges higher in the structure, and this is signaled by the introduction of a seek.

Consider, for example, the sentence *the dog saw the cat* and assume that *dog*, *saw*, and *cat* are nouns, *saw* is also a transitive verb, and that the grammar contains the following rules:

```
rule(s(s(NP, VP)), [np(NP)], vp(VP), []).
rule(vp(vp(V, NP)), [], v(V), [np(NP)]).
rule(np(np(D, N)), [det(D)], n(N), []).
```

The sequence of events involved in parsing the sentence with a parser that follows a simple shift reduce regime, would be as follows:

```
1. Add pending for det(det(the)) from 0 to 1,
    Left = [], Right = []
2. Add pending for n(n(dog)) from 1 to 2, Left = [],
   Right = []
3 .Add edge for v(v(saw)) from 2 to 3, Left = [],
   Right = []
4. Add edge for vp(vp(v(saw),_653)) from 2 to 3,
   Left = [], Right = [np(_653)]
5. Add edge for vp(vp(v(saw),_653)) from 2 to 3,
```

```
   Left = [], Right = [s(_653)]
6. Add pending for n(n(saw)) from 2 to 3, Left = [],
   Right = []
7. Add pending for det(det(the)) from 3 to 4,
   Left = [], Right = []
8. Add edge for n(n(cat)) from 4 to 5   Rule = 0 / 0, Left = [],
   Right = []
9. Add edge for np(np(_690,n(cat))) from 4 to 5,
   Left = [det(_690)], Right = []
10. Add edge for det(det(the)) from 3 to 4,
   Left = [], Right = []
11. Add edge for np(np(det(the),n(cat))) from 3 to 5
    Rule = r4 / 1, Left = [], Right = []
12. Add edge for vp(vp(v(saw),np(det(the),n(cat)))) from 2 to 5,
    Left = [], Right = []
13. Add edge for s(s(_1507,vp(v(saw),np(det(the),n(cat))))))
    from 2 to 5, Left = [np(_1507)], Right = []
14. Add edge for n(n(dog)) from 1 to 2, Left = [],
    Right = []
15. Add edge for np(np(_2014,n(dog))) from 1 to 2,
    Left = [det(_2014)], Right = []
16. Add edge for det(det(the)) from 0 to 1,
    Left = [], Right = []
17. Add edge for np(np(det(the),n(dog))) from 0 to 2,
    Left = [], Right = []
18. Add edge for s(s(np(det(the),n(dog)),vp(v(saw),
    np(det(the),n(cat))))) from 0 to 5, Left = [],
    Right = []

Result = [s(s(np(det(the),n(dog)),vp(v(saw),np(det(the),n(cat)))))],
```

We write `add edge...` when the edge being added is current. Notice that the edge for the word *saw*, construed as a verb, is initially introduced as current, because the goal is to find a sentence and a seek is therefore extant for S, VP, and V, covering the whole string. The N edge for *saw*, however, is pending. In step 4, the active adge is introduced that will consume the object of *saw* when it is found. This introduces a seek for NP and N between vertex 3and the end of the sentence. For this reason, when *cat* is introduced in step 8, it is as a current edge. Notice, however, that *the*, in step 7, is introduced as pending, because it is not the head of a NP. However, the introduction of the active NP edge in step 9 causes the edge for *the* to be made current, and this is what happens in step 10. The active S edge in step 13 activates the search for an NP before the verb so

that all the remaining edges are introduced as current. At the end of the process all pending edges have been made current except the one corresponding to the nominal interpretation of `saw`.

The Prolog code that implements this strategy is considerably more compicated that that for the techniques discussed earlier, and I have therefore not included it.

I believe that the strategy I have outlined is the natural one for anyone to adopt who is determined to work with a head-driven active chart parser. However, it is entirely unclear that the advantages that it offers over the simple undirected chart parser are worth its considerable added expense in complexity. Notice that, if one of the other nouns in the sentence just considered also had a verbal interpretation, the search for noun phrases would have been active everywhere. The longer the sentence, and therefore the more pressing the need for high performance, the more active regions there would be in the string and the more nearly the process as a whole would approximate that of the undirected technique. This should not, of course, be taken as an indictment of head-driven parsing, which is interesting for reasons having nothing to do with performance. It does, however, suggest that the temptation to claim that it is also a natural source of efficiency should be resisted.

Appendix A – A PARSER-GENERATOR FOR HEAD-DRIVEN GRAMMAR.

This is a simple head-driven recursive-descent parser. There is a distinction between the top level `parse` predicate and the `syntax` predicate to eliminate inessential arguments to the top level call, and also because the program can, with only minor modifications in `syntax`, be used as a generator. The predicate `head` is assumed to be defined as part of the grammar. It is true of a pair of grammatical labels if the second can be the head (of the head, of the head ...) of the first. Having hypothesized the label of the eventual lexical head of a phrase that will satisfy the current goal, `syntax`calls `range` to find a word in the string with that label. If such a word is found, its position in the string will be given by the `HRange` (head range) difference list and it must, in any case, lie within the range of the string given by `Maxl` and `Maxr`. The `build` predicate constructs phrases with the given putative head so long as their labels stand in the `head` relation to the goal.

```
/*************************************************************************
* parse(String, Result)                                                  *
*                                                                        *
* String is a list of words                                              *
* Struct is the structure (nondeterministically) returned if the parse   *
* succedes                                                               *
*************************************************************************/
parse(String, Struct):-
   syntax(String/[]/Struct, String/[]).
/*************************************************************************
* syntax((L/R)/Goal, Maxl/Maxr)                                          *
*                                                                        *
*   G is the Goal for the parsen.                                        *
*   L/R is a DL giving the bounds of the phrase satisfying the goal      *
*   Maxl/Maxr gives the string bounds for the current search.            *
*                                                                        *
*************************************************************************/

syntax(Range/Goal, Max) :-
   head(Goal, Head),                    % Find lexical head for Goal
   range(HRange/Head, Max),             % Associate Head with actual
                                        % word and string position.
   build(Range/Goal, HRange/Head, Max).  % Build bottom up based on Head.

/*************************************************************************
* range((L/R)/Head, MaxL/MaxR)                                           *
*                                                                        *
* True of (1) position L/R in the string                                 *
*         (3) with grammatical description Head                          *
*         (4) somewhere in the string range MaxL/MaxR (parsing)          *
*************************************************************************/
%
% Whole maximum range explored.
% ============================
range(_, X/X) :-   !, fail.
%
% Next word in maximum range is the required head.
% ===============================================

range(L/R/Head, L/_) :-   dict(L/R, Head).
%
% Try again one place to the right.
% ================================
range(Head, [H|T]/MaxR) :-
   range(Head, T/MaxR).

/*************************************************************************
* build((GL/GR)/Goal, (HL/HR)/Head, MaxL/MaxR)                           *
*                                                                        *
* Build phrases bottom up based on the Head located in the string at     *
* HL/HR.  The location of the phrase found will be GL/GR and it must     *
* fall in the range MaxL/MaxR.                                           *
*************************************************************************/
build(X, X, _).                         % Current head is result.
build(GL/GR/Goal, HL/HR/Head, MaxL/MaxR) :- % Find rule matching Head
  rr(Lhs, Left, Head, Right), head(Goal, Lhs),
  build_left(Left, LL/HL, MaxL/HL),         % Check left daughters
  build_right(Right, HR/RR, HR/MaxR),       % and right daughters.
  build(GL/GR/Goal, LL/RR/Lhs, MaxL/MaxR).  % Try building further on that.
```

```
build_left([], X/X, _). build_left([H|T], L/R, MaxL/MaxR) :-
  syntax(HL/R/H, MaxL/MaxR),
  build_left(T, L/HL, MaxL/HL).build_right([], X/X, _).
build_right([H|T], L/R, MaxL/MaxR) :-
  syntax(L/HR/H, MaxL/MaxR),
  build_right(T, HR/R, HR/MaxR).
```

Appendix B – A SIMPLE INACTIVE CHART PARSER

This is a chart version of a nondeterminisitc shift-reduce parser. Vertices of the chart are constructed from left to right, one on each recursive call to `parse/3`. A vertex is a list of edges headed by a number which is provided for convenience in printing. An edge takes the form `[label, next-vertex]`. The predicate `build_edge` is given a word and its successor vertex and returns a completed vertex. It succeeds once for each entry that the word has in the dictionary and, for each one, calls `build_edge1`. This can succeed in three ways, all of which are collected into the list of edges contributing to the current vertex by virtue of the `setof` construction. The three possbilities are (1) The word's lexical entry itself labels an edge; (2) A unary rule applies to the entry, and its left-hand side labels an edge, and (3) A binary rule matches the entry and an entry in the next vertex (`member([Label, Next1], Next)`). Each new label is passed to `build-edge1` to be processed in the same manner as the original lexical entry.

```
parse(String, Result) :-
   parse(String, [0], Result).

parse([], V, V).
parse([Word|Rest], [N|Next], Vertex) :-
   setof(Edge, build_edge(Word, [N|Next], Edge), V),
   M is N+1,                                  % Next vertex number
   parse(Rest, [M|V], Vertex).                % {M|V] is the vertex

build_edge(Word, Next, Edge) :-
   dict(Word, Entry),                         % Dictionary lookup
   build_edge1(Entry, Next, Edge).

build_edge1(Entry, Next, [Entry, Next]).      % Shift.
build_edge1(Entry, Next, Edge) :-             % Reduce one item
   rule(Lhs, Entry),
   build_edge1(Lhs, Next, Edge).
build_edge1(Entry, [N|Next], Edge) :-         % Reduce two items
```

```
member([Label, Next1], Next),
rule(Lhs, Label, Entry),
build_edge1(Lhs, Next1, Edge).
```