Basic Parsing Techniques: an introductory survey

Stephen G. Pulman
University of Cambridge Computer Laboratory,
and SRI International, Cambridge

April 1991: *To appear in Encyclopedia of Linguistics, Pergamon Press and Aberdeen University Press*

'Parsing' is the term used to describe the process of automatically building syntactic analyses of a sentence in terms of a given grammar and lexicon. The resulting syntactic analyses may be used as input to a process of semantic interpretation, (or perhaps phonological interpretation, where aspects of this, like prosody, are sensitive to syntactic structure). Occasionally, 'parsing' is also used to include both syntactic and semantic analysis. We use it in the more conservative sense here, however.

In most contemporary grammatical formalisms, the output of parsing is something logically equivalent to a tree, displaying dominance and precedence relations between constituents of a sentence, perhaps with further annotations in the form of attribute-value equations ('features') capturing other aspects of linguistic description. However, there are many different possible linguistic formalisms, and many ways of representing each of them, and hence many different ways of representing the results of parsing. We shall assume here a simple tree representation, and an underlying context-free grammatical (CFG) formalism. However, all of the algorithms decribed here can usually be used for more powerful unification based formalisms, provided these retain a context-free 'backbone', although in these cases their complexity and termination properties may be different.

Parsing algorithms are usually designed for classes of grammar rather than tailored towards individual grammars. There are several important properties that a parsing algorithm should have if it is to be practically useful. It should be 'sound' with respect to a given grammar and lexicon; that is, it should not assign to an input sentence analyses which cannot arise from the grammar in question. It should also be 'complete'; that it, it should assign to an input sentence all the analyses it can have with respect to the current grammar and lexicon. Ideally, the algorithm should also be 'efficient', entailing the minimum of computational work consistent with fulfilling the first two requirements, and 'robust': behaving in a reasonably sensible way when presented with a sentence that it is unable to fully analyse successfully. In this discussion we generaly ignore the latter two requirements: to some extent, they are covered by the companion section on 'chart parsing'.

There are several other dimensions on which is useful to characterise the behaviour of parsing algorithms. One can characterise their search strategy in terms of the characteristic alternatives of depth first or breadth first (q.v.). Orthogonally, one can characterise them in terms of the direction in which a structure is built: from the words upwards ('bottom up'), or from the root node downwards ('top down'). A third dimension is in terms of the sequential processing of input words: usually this is left-to-right, but right-to-left or 'middle-out' strategies are also feasible and may be preferable in some applications (e.g. parsing the output of a speech recogniser).

Top down parsing

We can illustrate a simple top down algorithm for a CFG as follows. Let the grammar contain the following rules (which for simplicity also introduce lexical entries instead of presupposing a separate component for this):

```
S     --> NP VP
NP    --> they | fish
VP    --> Aux VP
VP    --> Vi
VP    --> Vt NP
Aux   --> can
Vi    --> fish
Vt    --> can
```

This will assign to the sentence 'they can fish' two distinct analyses corresponding to two interpretations 'they are able/permitted to fish' and 'they put fish in cans'. (It will also generate several other good sentences and lots of odd ones).

The top down algorithm is very simple. We begin with the start symbol, in this case, S, and see what rules it figures in as the mother. Then we look at the daughters of these rules and see whether the first one matches the next word in the input. If it does, we do the same for the remaining daughters. If not, we go through the same loop again, this time matching the daughters of the rules already found with mothers of other rules.

The algorithm is easily illustrated on the input sentence 'they fish': the 'dot' precedes the daughter of the rule we are currently looking at, and the level of indentation indicates which rule has invoked the current action.

```
Rule                    Input
```

```
S  -> .NP VP          they fish
 NP -> .they          they fish
 NP -> they.          fish
S  -> NP .VP          fish
 VP -> .Vi            fish
  Vi -> .fish         fish
  Vi -> fish.
 VP -> Vi.
S  -> NP VP.
```

Of course, in parsing this sentence we have magically chosen the correct rule at each point, and ignored choices which would not lead to a successful parse. To fully specify the algorithm we would need to eliminate all this magic and provide an exhaustive mechanism for trying out all possibilities. We can do this by non-deterministically trying out all alternatives at each point as they arise, or, more flexibly, by casting our algorithm in the form of a set of operations on representations that encode the state of a parse. Each parsing step takes an 'item', as we shall call them, and produces a set of new items. These alternatives can be pursued in whatever order is convenient. Parsing proceeds from an initial seed item and ends when no more steps are possible. Each remaining item, if there are any, will then represent a successful parse of the input.

An item consists of a list of 'dotted trees' with the most recent to the right, and a list of unconsumed words. 'Dotted trees' are derived from dotted rules in an obvious way (a rule like S -¿ NP .VP is represented [S NP .VP]) and represent the partial analyses built so far. The list of words is the input remaining to be parsed. There are three basic operations that can be performed on an item:

(i) if there is a word after a dot in the most recent tree that matches the next word in the input, make a new item like the current one except that the dot follows the word, and the next word is removed from the input. For example, an item of the form

```
< ... [NP .they], [they can fish]>
```

will yield a new item:

```
< ... [NP they.] , [can fish] >
```

(ii) if the most recent tree has the dot at the end of its daughters, integrate it with the tree to its left, if there is one. So an item like:

```
<[S .NP VP], [NP they.], [can fish]>
```

yields a new one:

```
<[S  [NP they] .VP], [can fish]>
```

(iii) if there is a rule whose left hand side matches the category following the dot in the most recent tree, make a new item like the old one with the addition of a new tree derived from the rules, with the dot preceding its daughters.

An item like

```
<[S .NP VP], [they can fish]>
```

yields one like:

```
<[S .NP VP], [NP .they], [they can fish]>
```

The sequence of items corresponding to the sample parse of 'they fish' above begins:

```
0. <[.S], [they  fish]>

1. < [S .NP VP],  [they  fish]>
2. < [S .NP VP], [NP .they], [they  fish]>
3. < [S .NP VP], [NP .fish], [they  fish]>
4. < [S .NP VP], [NP they.], [ fish]>
5. < [S [NP they]  .VP],  [ fish]>
6. < [S [NP they]  .VP], [VP .Vi],  [ fish]>
7. < [S [NP they]  .VP], [VP .Vt NP],  [ fish]>

etc.
```

Notice that both 2 and 3 were produced from 1, but that 3 was discarded because none of the actions applied to it.

The order in which new items are processed can be varied in order to give depth first or breadth first search behaviour.

Depending on the search strategy followed, the top down algorithm may go into a loop when certain types of rule are found in the grammar. For example, if we added rules for more complex types of NP, we would encounter 'left recursion' of the category NP:

```
NP -> NP  's  N   :possessive NPs like [[John] 's sister]
```

Now from an item like:

```
< S -> .NP VP ....>
```

we would produce one like

```
< NP -> .NP  's N ....>
```

and this would in turn produce another

```
< NP -> .NP  's N ....>
```

and so on. There are two solutions: either rewrite the grammar to eliminate left recursion, which is always possible in principle, or add an extra step to the algorithm to check for repeated items.

Some grammars will potentially send any sound and complete parsing algorithm into a loop: namely, those that contain cycles of the form `A -> ..... -> A`, where some symbol exhaustively dominates itself. This is not necessarily a bug in the parsing algorithms: such grammars will assign an infinite set of parse trees to any structure containing an A. Under these circumstances, the parser is merely trying to do what it is supposed to do and find all possible parses.

Bottom up parsing

The operation of a bottom up algorithm for CFG can be illustrated by the following sequence of operations for 'they fish':

```
Structure                          Input
so far                             remaining

                                   [they fish]
[NP  they]                         [fish]
[NP  they][Vi  fish]               []
[NP  they][Vp [Vi  fish]]          []
[S [NP  they][Vp [Vi  fish]]]      []
```

We proceed by matching words to the right hand sides of rules, and then matching the resulting symbols, or sequences of symbols, to the right hand sides of rules until we have an S covering the whole sentence. Again, we have magically made the right choices at each point.

We can provide a complete algorithm for one type of left-to-right, bottom up parsing procedure in terms of operations on items like those used in the top down parser. (Since we shall only be dealing with completely parsed constituents we can dispense with the dots.)

(i) if there is a rule whose right hand side matches the next word in the input, create a new item with a new tree made from that rule on the end of the tree list, and the remainder of the input.

For example:

```
<[], [they fish]>
```

becomes:

```
<[NP they], [fish] >
```

(ii) if there is a rule with n daughters matching the n most recent trees on the list, in order, create a new item which combines those daughters into an instance of the mother:

```
< [NP they], [VP [Vi fish]], []>
```

becomes:

```
< [S [NP they][VP [Vi fish]]], []>
```

A successful parse is represented by an item with no more input and a single S rooted tree on the list.

The sequence of items produced by this method in parsing 'they fish' is:

```
1. []                            [they fish]
2. [NP they],                    [fish]
3. [NP they], [NP fish]          []
4. [NP they], [Vi fish]          []
5. [NP they], [VP [Vi fish]]     []
6. [S [NP they] [VP [Vi fish]]] []
```

Nothing can happen to item 3 and it is discarded.

This particular type of bottom up algorithm is known as a 'shift-reduce' parser. Operation (i) shifts a word from the input on to the list of trees, which is operating like a 'stack' (a last-in-first-out store) insofar as it is only possible to get at the

most recent items. Operation (ii) 'reduces' the list of trees by combining several of them from the top of the stack into one constituent.

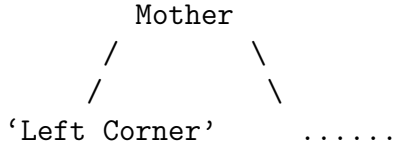
A generalisation of this type of algorithm is familiar from computer science: the LR(k) family can be seen as shift-reduce algorithms with a certain amount ('k' words) of lookahead to determine, for a set of possible states of the parser, which action to take. The sequence of actions from a given grammar can be precomputed to give a 'parsing table' saying whether a shift or reduce is to be performed, and which state to go to next. The use of this technique for natural language parsing has been promoted by Tomita (1987), among others.

While in general, bottom up algorithms are more efficient than top down algorithms, one particular phenomenon that they deal with only clumsily are 'empty rules': rules in which the right hand side is the empty string. Such rules are theoretically dispensible in a context free framework (any grammar containing them can in principle be converted to one, usually much larger, recognising the same language which does not) but they are practically very useful in describing, for example, movement phenomena, in a reasonably concise way. Bottom up parsers find instances of such rules applying at every possible point in the input, which can lead to much wasted effort following parsing hypotheses that lead nowhere.

Bottom up with top-down filtering

A better parsing algorithm than either pure top down or pure bottom up can be got by combining features of both. Operating bottom up means that the process is guided by what is actually in the input, rather than what the grammar predicts might be there (which will give a very large number of possibilities for a realistic grammar and lexicon). But imposing some top down constraints means that we can use what we have already found, with our knowledge about possible grammatical structures, to guide the search for what follows.

One such combined algorithm is the following. It is a member of the family of 'left-corner' parsing algorithms, since it begins at the left corner of the constituent it is trying to build, when viewed pictorially:



It is also a 'predictive' parser, in that it uses grammatical knowledge to predict what should come next, given what it has found already.

To describe this left-corner predictive parsing algorithm we extend our notion of

an 'item' to be:

```
<CurrentConstituent, RemainingInput, MotherCatSought,
       DaughtersFound, DaughtersSought, StackOfIncompleteConstituents>
```

There are 4 operations creating new items from old:

'Shift' is as before, taking the next word from RemainingInput and making it (suitably labelled) the CurrentConstituent.

'Predict' finds a rule whose leftmost daughter matches the CurrentConstituent, and then fills the MotherCatSought, DaughtersFound, and DaughtersSought fields of the new item as appropriate, copying these parts of the old item onto the StackOfIncompleteConstituents.

'Match' applies when the CurrentConstituent is the next item in DaughtersSought: the new item has a null CurrentConstituent, and undergoes appropriate changes to the two Daughters fields.

'Reduce' applies when there are no more DaughtersSought and creates an item with a new CurrentConstituent built from the MotherCatSought and the DaughtersFound, popping the most recent StackOfIncompleteConstituents entry to form the new Mother and Daughters fields.

With the Predict operation defined as above the algorithm will be sound and complete. However, it will derive no benefit from the top down information it carries, for that is not yet being used to constrain parsing hypotheses. What we need to do is to check that the mother categories on the rules found by Predict can be a 'left corner' of the current MotherCatSought, i.e. can be the leftmost daughter of some tree dominated by the MotherCat. This information can be calculated from the grammar fairly easily: we will not show how to do this here but for the grammar given above the 'left-corner-of' relation is:

```
<NP,S>
<Aux,VP>
<Vi,VP>
<Vt,VP>
```

We illustrate with the more complex and ambiguous sentence 'they can fish', assuming that Predict uses a left-corner check to filter candidate rules. For purposes of illustration we will expand the grammar with an extra rule `Sigma -> S`, where `Sigma`, by convention, is the start symbol. Thus the full left-corner-of relation is:

```
<NP,Sigma>
```

<NP,S>
<Aux,VP>
<Vi,VP>
<Vt,VP>

The initial item is based on the rule expanding the start category Sigma:

	Operation	Current	Input	Mother	Found	Sought	Stack	
		-	they can fish	Sigma	[]	[S]	-	
1.	Shift:	[NP they]	can fish	Sigma	[]	[S]	-	
2.	Predict 1:	-	can fish	S	[NP they]	[VP]	<Sigma, [], [S]>	
3.	Shift 2:	[Vt can]	fish	S	[NP they]	[VP]	<Sigma, [], [S]>	
4.	Shift 2:	[Aux can]	fish	S	[NP they]	[VP]	<Sigma, [], [S]>	
5.	Predict 3:	-	fish	VP	[Vt can]	[NP]	<S,[NP they],[VP]> <Sigma, [], [S]>	
6.	Predict 4:	-	fish	VP	[Aux can]	[VP]	<S,[NP they],[VP]> <Sigma, [], [S]>	
7.	Shift 6:	[NP fish]	-	VP	[Aux can]	[VP]	<S,[NP they],[VP]> <Sigma, [], [S]>	*
8.	Shift 6:	[Vi fish]	-	VP	[Aux can]	[VP]	<S,[NP they],[VP]> <Sigma, [], [S]>	
9.	Shift 5:	[NP fish]	-	VP	[Vt can]	[NP]	<S,[NP they],[VP]> <Sigma, [], [S]>	
10.	Shift 5:	[Vi fish]	-	VP	[Vt can]	[NP]	<S,[NP they],[VP]> <Sigma, [], [S]>	*
11.	Predict 8:	-	-	VP	[Vi fish]	[]	<VP,[Aux can],[VP]> <S,[NP they],[VP]> <Sigma, [], [S]>	
12.	Reduce 11:	[VP [Vi fish]]	-	VP	[Aux can]	[VP]	<S,[NP they],[VP]> <Sigma, [], [S]>	
13.	Match 12:	-		VP	[Aux can], [VP [Vi fish]]	[]	<S,[NP they],[VP]> <Sigma, [], [S]>	
14.	Reduce 13:	[VP can [VP fish]]		S	[NP they]	[VP]	<Sigma, [], [S]>	
15.	Match 14:	-		S	[NP they], [VP can [VP fish]]	[]	<Sigma, [], [S]>	
16.	Reduce 15:	[S they can fish]		Sigma	[]	[S]	-	
17.	Match 16:	-		Sigma	[S they can fish]	[]	-	
18.	Reduce 17	[Sigma [S they can fish]]		-	-	-	-	
19.	Match 9:	-		VP	[Vt can], [NP fish]	[]	<S,[NP they],[VP]> <Sigma, [], [S]>	
20.	Reduce 19:	[VP can [NP fish]]		S	[NP they]	[VP]	<Sigma, [], [S]>	
21.	Match 20:	-		S	[NP they], [VP can [NP fish]	[]	<Sigma, [], [S]>	
22.	Reduce 21:	[S they can fish]		Sigma	[]	[S]	-	
23.	Match 22:	-		Sigma	[S they can fish]	[]	-	
24.	Reduce 23:	[Sigma [S they can fish]]		-	-	-	-	

Notice that the filtering done by Predict eliminates several possible items based on the `S -> NP VP` rule at the point where the NP 'fish' is the current constituent (items 7 and 9), because S is not a left corner of VP, which is the current Mother-CatSought. Items marked * lead to parsing paths that do not go anywhere: these

too could be ruled out by doing a left corner check when Shifting: if the category of the item shifted is not a left corner of the first item in DaughtersSought, then no parse will result.

Determinism

A parsing procedure which, in a particular state, is faced with a choice as to what to do next, is called 'non-deterministic'. It has been argued (Marcus 1980) that natural languages are almost deterministic, given the ability to look ahead a few constituents: i.e., that in general if there is not sufficient information to make a decision based on the input so far, there usually will be within the next few words. Such a property would have obvious functional advantages for the efficient processing of language using minimal short term memory resources. This claim, although hotly contested, has been used to try to explain certain types of preferred interpretation or otherwise mysterious difficulties in interpreting linguistic constructs. For example, it is argued, if the human parsing system is deterministic, and making decisions based on limited lookahead, we would have an explanation for the fact that sentences like:

The horse raced past the barn fell

are perceived as rather difficult to understand. Under normal circumstances, the tendency is to be 'led down the garden path', assembling 'the horse raced past the barn' into a sentence, then finding an apparently superfluous verb at the end. However, there are many other factors involved in the comprehension of sentences, and when all of these are taken into account, the determinism hypothesis is by no means completely satisfactory as an explanation. (For a survey and further discussion, see Pulman 1987; Briscoe, 1987)

Parsing strategies

Determinism has been offered as an account of one type of parsing preference. Other types of preferred interpretation have been claimed to be the result of specific strategies that are characteristic of the human parsing process (Kimball, 1973; Frazier and Fodor, 1978; etc.). For example, in the following sentences the tendency is to interpret the adverbial as modifying the most deeply embedded clause, even though linguistically, it is possible to interpret it as modifying the main clause:

He spoke to the man you saw on Friday. She said that he was here at some time yesterday

These particular judgements are held to be the result of a preference for an

analysis on which a modifier is attached as far 'down' the tree as is possible. Other preferences might be to incorporate constituents as arguments rather than modifiers where possible, or to build trees that are as 'flat' as possible consistent with the input.

In general, discussion of parsing preferences takes place in the context of hypotheses about human performance rather than explicit computationally implementable algorithms. However, many of the parsing preferences that have been discussed in the literature can be naturally modelled within a shift-reduce based parser, by recasting them as strategies to be followed when there is a choice as to which action to perform next (Pereira, 1985). For example, if whenever there is a choice between shifting and reducing then shifting is preferred, constituent final modifiers will always be attached as 'low' as possible. This is because reductions only take place on the topmost elements of the stack, and thus a sequence like

```
NP V NP V NP PP
```

will be assembled into constituents from the right hand end, leading to 'low' attachment. Since the algorithm is actually non-deterministic, 'preference' is most naturally interpreted as meaning that analyses are found in a preferred order rather than that one is found to the exclusion of others.

While the status of parsing preferences as a source of evidence for the workings of the human sentence processing mechanism is still unresolved, to the extent that they are real, it is of great practical importance to be able to include them in computational parsing systems, if for no better reason than that when faced with the multiplicity of parse trees that a complete algorithm yields for most sentences when operating on realistically sized grammars, anything is better than making a random decision as to which one represents the intended interpretation of the input sentence.

Chart Parsing and WFSSTs.

For efficiency, most practical parsing algorithms are implemented using a 'well formed substring table' (wfsst) to record parses of complete subconstituents. This means that duplicate paths through the search space defined by the grammar are avoided. Since there may be many such paths, use of such a table can sometimes mean the difference between processing times of seconds, as opposed to minutes or even hours.

To illustrate, consider the process of parsing a sequence analysable as

```
... (1) V (2) NP (3) PP (4)
```

(e.g. '... saw the man in the park') given rules in the grammar like

```
VP -> V NP
VP -> VP PP
NP -> NP PP
```

In a top down parsing regime, there will be two points at which the first of these rules will be used to predict an NP beginning at position 2. Let us assume that the algorithm first finds the nonrecursive NP from 2 to 3. Now, via the third rule, it will again look for an NP, this time followed by a PP. If we are not careful, the naive algorithm will repeat the same sequence of actions it has just performed in recognising the original NP from 2 to 3.

However, if we assume that whenever a complete constituent is recognised, it is recorded in a table, then we can change our algorithm to consult this table whenever it is looking for a constituent of a particular type at a given point. In this instance we will immediately find that we already have an NP from 2 to 3, and can therefore proceed directly to look for a PP beginning at position 3.

Having found this PP, we can now assemble two VPs: one from 1 to 3, of the form `[VP [V NP]]`, and one from 1 to 4 of the form `[VP [V [NP NP PP]]]`.

Now we will be in the state where the second rule predicts a PP beginning at position 3. On the naive algorithm, again we will just go ahead and recompute the sequence of actions leading to the recognition of the PP. But using our wfsst, we have already recorded the fact that there is a PP from 3 to 4 and instead we simply use that information to build a VP from 1 to 4 of the form `[VP [VP PP]]`. Thus we have saved ourselves at least two recomputations. If this does not sound very impressive, the reader is invited to work through the steps involved in parsing a sequence like:

```
V NP P NP P NP P NP
```

(e.g. 'saw the man in the park with a telescope on Friday') given the above rules, along with `P -> P NP`, and to check how many times the same constituents are reparsed when a wfsst is not being used. For constructions like this, the numbers can grow very rapidly indeed. Use of a wfsst here is essential if parsing algorithms are to be implented in a practically usable form.

A further technique is to keep a record of subconstituents that have NOT been found beginning at a particular point in the input. For example, in the earlier 'saw the man in the park' example, when the VP of the form `[VP V [NP [NP PP]]]` is found, it will, via the second rule, cause a prediction of a PP beginning at position 4. This prediction fails, because we are at the end of a sentence. When the VP of the form `[VP [VP [V NP]] PP]` is found, it too will cause the same

prediction, via the same rule. While in this case discovering that we have no more PPs is fairly trivial, this will not always be so, and a lot of recomputation can be saved by also checking that, when looking for a constituent C at position P, we have not already tried and failed to find C at P.

Basic chart parsing

A 'chart' is a generalisation of a wfsst in which incomplete constituents are also represented. Charts are the basis on which many parsing algorithms are implemented, and they provide a flexible framework within which different types of processing regime and different types of grammatical formalism can be handled.

A chart consists of a set of 'edges' and 'vertices'. Vertices represent the positions between words in an input sentence, and edges represent partial or complete analyses. An edge, at least when context free rules or related formalisms are involved, can be thought of as something derived from a rule and having the following structure:

```
edge(Id,LeftVertex,RightVertex,MotherCat,DaughtersFound,DaughtersSought)
```

(In describing charts it is convenient to use a Prolog like notation: words in lower case are constants, words beginning with upper case are variables, an underscore is a variable whose value we are not interested in, and a list of items is enclosed in square brackets, with the convention that an expression like `[Head|Tail]` represents a list whose first member is Head and whose remaining members (a list) are Tail). An edge has an identifier and connects vertices. DaughtersFound will usually be represented in terms of a list of other edges representing the analysis of those daughters. DaughtersSought is a list of categories. An edge is complete if DaughtersSought is empty.

To specify a particular instantiation of a chart framework, we need to state:

(i) a regime for creating new edges
(ii) a way of combining two old edges to form new edge
(iii) access to an 'agenda' of edges created by (i) or (ii) that are waiting for further processing when they are entered into the chart.

A bottom up chart parser

One particular chart based algorithm can be specified as follows: it proceeds bottom up, one word at a time.

```
New edges:
  whenever there is a complete edge put in the chart of form
   edge(Id,From,To,Category,_,_)
   then for each rule in the grammar of form Lhs -> Rhs
   where Category is the first member of Rhs,
  put a new edge on the agenda of form
   edge(NewId,From,From,Lhs,[],Rhs)
```

(Not all rules in the grammar meeting this criterion will lead to a complete parse. This step of the procedure can be made sensitive to information precomputed from the grammar so as to only select rules which are, say, compatible with the next word in the input, or alternatively, compatible with the next category sought of at least one incomplete edge ending at the point where the current word starts. See Aho and Ullman 1977 for various types of relevant grammar relations).

```
Combine Edges:
 Whenever a new edge is put into the chart of the form:
 edge(Id1,B,C,Cat1,Found1,[])
then for each edge in the chart of form:
 edge(Id2,A,B,Cat2,Found2,[Cat1|OtherDaughtersSought])
create a new edge
 edge(Id3,A,C,Cat2,Found2+Id1,OtherDaughtersSought)

 Whenever a new edge is put into the chart of the form:
 edge(Id1,A,B,Cat1,Found1,[Cat2|RestSought])
then for each edge in the chart of form:
 edge(Id2,B,C,Cat2,Found2,[])
create a new edge
 edge(Id3,A,C,Cat1,Found1+Id2,RestSought)
```

The first part of Combine Edges is triggered by the addition of a complete edge to the chart, and produces a new edge for each incomplete edge ending where the complete edge begins which can combine with it. These incomplete edges are already in the chart. The new edges are put on the agenda.

The second part is triggered by the addition of an incomplete edge to the chart, and produces a new edge for each complete edge beginning where the incomplete edge ends. These new edges are put on the agenda.

Looking at things from the point of view of a complete edge, the first part of Combine Edges ensures that it is combined with whatever is already in the chart that it can be combined with, whereas the second part ensures that it will be combined with any future incomplete edges entering the chart. Thus no opportunity for combination will be missed, at whatever stage of parsing it arises.

All we have to do now to specify a complete chart parsing procedure is to define access to the agenda: we can choose to treat the agenda as a stack (last in, first out) in which case the general search strategy will be depth first; or as a queue (first in, first out) in which case we will search hypotheses breadth first. We could also have more complex heuristics ordering edges on the agenda according to some weighting function: this would mean that the highest scoring hypotheses were explored first, independently of the order in which they were generated.

We also have to embed the procedure in some kind of top level driver, so as to start off the process, and check for complete analyses when the process is ended. The final program, then, might have the following structure:

```
Until no more words:
 create new edges for next word
 do  New Edges for these edges.
  Until agenda is empty:
   pop next edge off agenda and put in chart
   do  New Edges
   do  Combine Edges
Check for complete edges of desired category spanning start to finish
```

Given the following grammar the algorithm would proceed as follows on the input sentence 'they can fish'

```
S   --> NP VP
NP  --> they | fish
VP  --> Aux VP
VP  --> Vi
VP  --> Vt NP
Aux --> can
Vi  --> fish
Vt  --> can
```

Operation	Chart	Agenda
new word edge		e1(1,2,NP,[they],[])
pop	e1	
new edge		e2(1,1,S,[],[NP,VP])
pop	+e2	
combine e1 e2		e3(1,2,S,[e1],[VP])
pop	+e3	
new word edges		e4(2,3,Aux,[can],[]),
		e5(2,3,Vt,[can],[])
pop	+e4	
new edge		e6(2,2,VP,[],[Aux,VP]),
		e5

```
pop             +e6
combine e4 e6                   e7(2,3,VP,[e4],[VP]),
                                e5
pop             +e7             e5
pop             +e5
new edge                        e8(2,2,VP,[],[Vt,NP])
pop             +e8
combine e5 e8                   e9(2,3,VP,[e5],[NP])
pop             +e9
new word edges                  e10(3,4,Vi,[fish],[])
                                e11(3,4,NP,[fish],[])
pop             +e10            e11
new edge                        e12(3,3,VP,[],[Vi]),
                                e11
pop             +e12            e11
combine e12 e10                 e13(3,4,VP,[e10],[]),
                                e11
pop             +e13            e11
combine e7 e13                  e14(2,4,VP,[e4,e13],[]),
                                e11
pop             +e14            e11
combine e3 e14                  e15(1,4,S,[e1,e14],[]),
                                e11
pop             +e15            e11
pop             +e11
new edge                        e16(3,3,S,[],[NP,VP])
combine e9 e11                  e17(2,4,VP,[e5,e11],[])
                                e16
pop             +e17            e16
combine e3 e17                  e18(1,4,S,[e1,e17],[])
                                e16
pop             +e18            e16
pop             +e16
combine e16 e11                 e19(3,4,S,[e11],[VP])
pop             +e19
```

At this point no more processing can take place. Inspecting the chart we find that we have two complete edges spanning the input, of the desired category, S. By recursively tracing through their contained edges we can recover the syntactic structure of the analyses implicit in the chart. Notice that as well as sharing some complete subconstituents (edge 1), the final analyses were built up using some of the same partial constituents (edge 3).

Other chart based parsing algorithms

It is easy to formulate different processing regimes within the chart framework. By redefining the New Edges routine to operate on incomplete edges one can implement a top down, 'Earley' style of algorithm (Thompson and Ritchie, 1984). Alternatively, it is possible to define New Edges so that any arbitrary constituent, rather than the first, is used to build new edges, and with corresponding changes to Combine Edges constituents can be built right to left as well as left to right (Steel and de Roeck, 1987) This can be useful for practical efficiency when some construction is 'keyed' by an item which is not the leftmost one, for example, conjunctions in English, or verbal complements in subordinate clauses in languages like German. A left to right strategy would have to allow for the possibility of many types of constituent before there was any evidence for them, possibly leading to a lot of wasted computation most of the time. This strategy can also be generalised to 'head driven' parsing, using information from the head of a phrase to guide the search for its sister constituents to left or right (Kay 1990).

For all of these different types of parsing algorithm, appropriate search strategies can be imposed by different methods of manipulating the agenda, as mentioned earlier.

Chart parsing also offers a degree of robustness in the case of failure to find a complete parse, either because the input is ungrammatical or (what amounts to the same thing from the point of view of the parser) is not within the coverage of the grammar. Even if the overall parse fails, an algorithm like the one above will result in all well formed subconstituents that are present in the input being found. From these, different types of heuristic strategy can be used to do something useful with them. Mellish (1989) describes one such technique, which attempts to massage an ill-formed input into a well-formed one by inserting or deleting constituents after the ordinary parsing process has failed.

It is also worth pointing out that many of the virtues of charts as a basis for parsing are also desiderata for the reverse process of generation. In generation as well as parsing it is important to avoid unnecessary recomputation. Shieber (1988) describes a chart-based framework which is intended to accommodate a variety of generation as well as parsing algorithms.

Packing

While charts, as so far described, offer considerable economy of representation and processing, they still take exponential time and space when there is an exponential number of parse trees to be assigned to a sentence. We can in fact improve on this. Notice that in the example above, we built two VP constituents spanning 2

to 4, and that these in turn gave rise to two S constituents, each containing the same NP. This has the advantage that all analyses of the sentence are explicitly represented: there is an S edge for each one. But it has the disadvantage that we are doing the combination redundantly on the second and any further occasions. (This is where the exponential behaviour arises). If an NP can combine with one VP from 2 to 4 to form a sentence, then it is obvious that it can also combine with any other VP from 2 to 4. At the point at which this move is made the fact that the two VPs have different internal structure is irrelevant.

If we are prepared to do extra work in spelling out explicit parse trees when other processing has been completed, we can eliminate this redundancy. In the chart framework as we have developed it here, this is most easily achieved by generalising the representation of complete edges so that the 'constituents' or 'daughters found' field is a disjunction of lists of edge identifiers, rather a single list. Then we add the extra step, whenever we are building a complete edge, of checking to see whether there is already in the chart a complete edge of the same category and with the same From and To labels. If there is, we simply make a disjunction of the constituents field of the edge we are building with that of the existing edge and we need not create the new edge at all, or do anything further. Any analyses that the existing edge already takes part in will also involve the new edge, and the new edge will be included in any further analyses involving the existing edge. Thus in the case of our second VP edge, at the point where we can combine edge 9 with edge 11 to form edge 17, we would instead have extended edge 14 to look like:

```
e14(2,4,VP,[[e4,e13],[e5,e11]],[])
```

Then the combination of edge 3 with edge 14 to make a sentence would represent both analyses, without needing to repeat the combination explicitly. In this case, only one set of operations is saved, but it is easy to see that in cases like the multiple PP sentence encountered earlier, the savings can be considerable.

This extension is essentially the notion of 'packing', as described in, for example, Tomita 1987. If explicit parse trees are required, then there is an extra cost in a post processing phase, for each disjunction will need to be unpacked into a separate analysis. This step may be exponential even where parsing time was polynomial: the complexity is moved, not eliminated. However, the number of constituents created and the number of computational operations performed while actually parsing will usually be far smaller if packing is used than if the simple chart scheme is used. Moreover, in many practical NLP systems, parse trees do not need to be enumerated explicitly, except perhaps for debugging purposes, as they are merely a preliminary to semantic interpretation. In some of these systems (e.g. that described in Alshawi et al. 1988), many aspects of semantic interpretation can be done on packed structures directly, saving yet

more computational work.

References

M. Tomita 1987
An efficient augmented context-free parsing algorithm Computational Linguistics, 13, 31-46.

M. Marcus 1980
A Theory of Syntactic Recognition for Natural Language, MIT Press.

S. G. Pulman 1987
Computational Models of Parsing, in A. Ellis (ed), Progress in the Psychology of Language, Vol III, London and New Jersey: Lawrence Erlbaum Associates, 159-231.

E. J. Briscoe 1987
Modelling Human Speech Comprehension: a Computational Approach, Ellis Horwood, Chichester, and Wiley and Sons, N.Y.

J. Kimball 1973
Seven principles of surface structure parsing in natural language Cognition 2, 15-48.

L. Frazier and J. D. Fodor 1978
The Sausage Machine: a new two-stage parsing model. Cognition 6, 291-325.

F. C.N. Pereira 1985
A new characterization of attachment preferences, in David R. Dowty, Lauri Karttunen and Arnold M. Zwicky (eds) Natural Language Parsing, Cambridge: Cambridge University Press, 307-319.

A. V. Aho and J. D. Ullman 1977
Principles of Compiler Design Reading, Mass: Addison Wesley.

H. S. Thompson and G. D. Ritchie 1984
Implementing Natural Language Parsers, in T. O'Shea and M. Eisenstadt, eds., Artificial Intelligence: Tools, Techniques, and Applications, N.Y., Harper and Row, 245-300.

S. Steel and A. de Roeck 1987
Bidirectional Chart Parsing,

in Christopher S. Mellish and John Hallam (eds) Advances in Artificial Intelligence (Proceedings of AISB-87) Ellis Horwood, Chichester; N.Y. Wiley and Sons, 223-235.

M. Kay 1990

Head Driven Parsing in Parsing Technologies: Proceedings of an International Workshop, Carnegie-Mellon University.

C. Mellish 1989
Some Chart-based techniques for parsing ill-formed input, Proceedings of 27th ACL: Vancouver, 102-109.

S. Shieber 1988
A Uniform Architecture for Parsing and Generation, COLING 88, Budapest: 614-619.

H. Alshawi, D. M. Carter, J. van Eijck, R. C. Moore, D. B. Moran, and S. G. Pulman 1988
Overview of the Core Language Engine, Proc. Intl. Conf. on 5th Generation Computer Systems, Tokyo, 1108-1115.