A Lazy Way to Chart-Parse with Categorial Grammars

Remo Pareschi*† and Mark Steedman†

Dept. of AI and Centre for Cognitive Science, Univ. of Edinburgh,*†
and Dept. of Computer and Information Science, Univ. of Pennsylvania†

ABSTRACT

There has recently been a revival of interest in Categorial Grammars (CG) among computational linguists. The various versions noted below which extend pure CG by including operations such as functional composition have been claimed to offer simple and uniform accounts of a wide range of natural language (NL) constructions involving bounded and unbounded "movement" and coordination "reduction" in a number of languages. Such grammars have obvious advantages for computational applications, provided that they can be parsed efficiently. However, many of the proposed extensions engender proliferating semantically equivalent surface syntactic analyses. These "spurious analyses" have been claimed to compromise their efficient parseability.

The present paper describes a simple parsing algorithm for our own "combinatory" extension of CG. This algorithm offers a uniform treatment for "spurious" syntactic ambiguities and the "genuine" structural ambiguities which any processor must cope with, by exploiting the **associativity** of functional composition and the **procedural neutrality** of the combinatory rules of grammar in a bottom-up, left-to-right parser which delivers all semantically distinct analyses via a novel unification-based extension of chart-parsing.

1. Combinatory Categorial Grammars

"Pure" categorial grammar (CG) is a grammatical notation, equivalent in power to context-free grammars, which puts all syntactic information in the lexicon, via the specification of all grammatical entities as either functions or arguments. For example, such a grammar might capture the obvious intuitions concerning constituency in a sentence like *John must leave* by identifying the VP *leave* and the NP *John* as the arguments of the tensed verb *must*, and the verb itself as a function combining to its right with a VP, to yield a predicate -- that is, a leftward-combining function-from-NPs-into-sentences. One common "slash" notation for the types of such functions expresses them as triples of the form *<result, direction, argument>*, where *result* and *argument* are themselves syntactic types, and *direction* is indicated by "/" (for rightward-combining functions) or "\" (for leftward). *Must* then gets the following type-assignment:

(1) `must:= (S\NP)/VP`

In pure categorial grammar, the only other element is a single "combinatory" rule of Functional Application, which gives rise to the following two instances:[1]

(2)
```
a.  Rightward Application:
      X    ==>   X/Y   Y
b.  Leftward Application:
      X    ==>   Y  X\Y
```

These rules allow functions to combine with immediately adjacent arguments in the obvious way, to yield the obvious surface structures and interpretations, as in:

(3)
```
John    must     leave
----  ---------  -----
 NP  (S\NP)/VP   VP
     ------------->apply
        S\NP
  ------------<apply
        S
```

Combinatory Categorial Grammar (CCG) (Ades and Steedman 1982, Steedman 1985, Steedman 1986) adds a number of further elementary operations on functions and arguments to the combinatory component. These operations correspond to certain of the primitive combinators used by Curry and Feys (1958) to define the foundations of the λ-calculus, notably including functional composition and "type raising". For example:

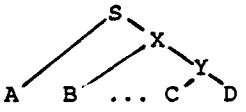
(4)
```
a.  Subject Type Raising:
      S/(S\NP)   ==>   NP
b.  Rightward Composition:
      X/Z    ==>   X/Y   Y/Z
```

These combinatory operations allow additional, non-standard "surface structures" like the following, which arises from the type-raising of the subject *John* into a function over predicates, which composes with the verb, which is of course a function *into* predicates:

(5)
```
   John        must     leave
--------   ---------   -----
   NP      (S\NP)/VP    VP
--------->raise
S/(S\NP)
------------------->compose
          S/VP
          ------------->apply
                S
```

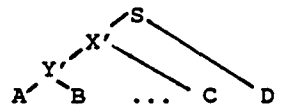
In general, wherever orthodox surface structure posits a right branching structure like (a) below, these new operations will allow not only the left branching structure (b), but every mixture of right- and left- branching in between:

(6) a.



[1] All combinatory rules are written as *productions* in the present paper, in contrast with the *reduction* rule notation used in the earlier papers. The change is intended to aid comparison with other unification-based grammars, and has no theoretical significance.

b.



The linguistic motivation for including such operations, (and the grounds for contesting the standard linguists' view of surface constituency), for details of which the reader is referred to the bibliography, stems from the possibility of extracting over, and also coordinating, a wide range of such non-standard composed structures. A crucial feature of this theory of grammar is that the novel operation of functional composition is *associative* so that all the novel analyses like (5) are semantically equivalent to the relevant canonical analysis, like (3). On the other hand, rules of type raising simply map arguments into functions over the functions of which they are argument, producing the same result, and thus are by themselves responsible for no change in generative capacity; indeed, they can simply be regarded as tools which enable functional composition to operate in circumstances where one or both the constituents which need to be combined initially are not associated with a functional type, as when combining a subject NP with the verb which follows it.

Grammars of this kind, and the related variety proposed by Kartunnen (1986), achieve simplicity in the grammar of movement and coordination at the expense of multiplying the number of derivations according to which an unambiguous string such as the sentence above can be parsed. While we have suggested in earlier papers (Ades and Steedman 1982, Pareschi 1986) that this property can be exploited for incremental semantic interpretation and evaluation, a suggestion which has been explored further by Haddock (1987) and Hinrichs and Polanyi (1986), two potentially serious problems arise from these spurious ambiguities. The first is the possibility of producing a whole set of semantically equivalent analyses for each reading of a given string. The second more serious problem is that of efficiently coping with nondeterminism in the face of such proliferating ambiguity in surface analyses.

The problem of avoiding equivalent derivations is common to parsers of all grammars, even context-free phrase-structure grammars. Since all the spurious derivations are by definition semantically equivalent, the solution seems obvious: just find one of them, say via a "reduce first" strategy of the kind proposed by Ades and Steedman (1982). The problem with this proposal arises from the fact that, assuming left-to-right processing, *Rightward Composition may preempt the construction of constituents which are needed as arguments by leftward combining functional types.*[2] Such a depth-first processor cannot take advantage of standard techniques for eliminating backtracking, such as chart-parsing (Kay, 1980), because the subconstituents for the alternative analysis will not in general have been built. For example, if we have produced a left-branching analysis like (b) above, and then find that we need the constituent X in analysis (a) (say to attach a modifier), we will be forced to redo the entire analysis, since not one of the subconstituents of X (such as Y) was a constituent under the previous analysis. Nor of course can we afford a standard breadth-first strategy. Karttunen (1986a) has pointed out that a parser which associates a canonical interpretation structure with substrings in a chart can always distinguish a spurious new analysis of the same string from a genuinely different analysis: spurious analyses produce results that are the same as one already installed on the chart. However, the spurious ambiguity problem remains acute. In order to produce only the genuinely distinct readings, it seems that *all* of the spurious analyses must be explored, even if they can be discarded again. Even for short strings, this can lead to an unmanageable enlargement of the search space of the processor. Similarly, the problem of reanalysis under backtracking still threatens to overwhelm the parser. In the face of this problem Wittenburg (1986) has recently argued that massive heuristic guidance by strategies quite problematically related to the grammar itself may be required to parse at all with acceptable costs in the face of spurious ambiguities (see also Wittenburg, this conference.) The present paper concerns an alternative unification-based chart-parsing solution which is grammatically transparent, and which we claim to be generally applicable to parsing "genuine" attachment ambiguities, under extensions to CG which involve associative operations.

[2] If we had chosen to process right-to-left, then an identical problem would arise from the involvement of *Leftward* Composition.

2. Unification-based Combinatory Categorial Grammars

As Karttunen (1986), Uszkoreit (1986), Wittenburg (1986), and Zeevat et al. (1986) have noted, unification-based computational environments (Shieber 1986) offer a natural choice for implementing the categories and combination rules of CGs, because of their rigorously defined declarative semantics. We describe below a unification-based realisation of CCG which is both transparent to the linguistically motivated properties of the theory of grammar and can be directly coupled to the parsing methodology we offer further on.

2.1. A Restricted Version of Graph-unification

We assume, like all unification formalisms, that grammatical constituents can be represented as feature-structures, which we encode as *directed acyclic graphs* (dags). A dag can be either:

(i) a constant

(ii) a variable

(iii) a finite set of label-value pairs (features), where any value is itself a dag, and each label is associated with one and only one value

We use round brackets to define sets, and we notate features as *[label value]*. We refer to variables with symbols starting with capital letters, and to labels and constants with symbols starting with lower-case letters. The following is an example of a dag:

(7)
```
([a  e]
 [b  ([c  X]
      [d  f])])
```

Like other unification based grammars, we adopt dags as the data-structures encoding categorial feature information because of the conceptual perspicuity of their set-theoretic definition. However, the variety of unification between dags that we adopt is more restrictive than the one used in standard graph-unification formalisms like PATR-2 (Shieber 1986), and closely resembles term-unification as adopted in logic-programming languages.

We define unification by first defining a partial ordering of *subsumption* over dags in a similar (albeit more restricted) way to previous work discussed in Shieber (1986). A dag D_1 subsumes a dag D_2 if the information contained in D_1 is a (not necessarily proper) subset of the information contained in D_2. Thus, variables subsume all other dags, as they contain no information at all. Conversely, a constant subsumes, and is subsumed by, itself alone. Finally, subsumption between dags which are feature-sets is defined as follows. We refer to two feature-sets D_1 and D_2 as *variants* of each other if there is an isomorphism σ mapping each feature in D_1 onto a feature with the same label in D_2. Then a feature-set D_1 subsumes a feature-set D_2 if and only if:

(i) D_1 and D_2 are variants; and

(ii) if $\sigma(f, f')$, where f is a feature in D_1 and f' is a feature in D_2, then the value of f subsumes the value of f'.

The *unification* of two dags D_1 and D_2 is then defined as the most general dag D which is subsumed by both D_1 and D_2. Like most other unification-based approaches, we assume that from a procedural point of view, the process of obtaining the unification of two dags D_1 and D_2 requires that they be destructively modified to become the *same* dag D. (We also use the term unification to refer to this process.)

For example let D_1 and D_2 be the two following dags:

(8)
```
([a  ([b  c])]        ([a  Y]
 [d  g]                [d  Z]
 [e  X])               [e  Z])
```

Then the following dag is the unification of D_1 and D_2:

(9)
```
([a  ([b  c])]
 [d  g]
 [e  g])
```

However, under the present definition of unification, as opposed to the more general PATR-2 definition, the above is *not* the unification of the following pair of dags:

(10)
```
([a  ([b  c])]        ([d  Z]
 [d  g])               [e  Z])
```

These two dags are not unifiable in present terms, because under the above definition of subsumption, unification of two feature sets can only succeed if they are variants. It follows that a dag resulting from unification must have the same feature population as the two feature structures that it unifies.

The present definition of unification thus resembles term unification in invariably yielding a feature-set with exactly the same structure as both of the input feature-sets, via the instantiation of variables. The only difference from standard term unification is that it is defined over dags, rather than standard terms. By contrast, standard graph-unification can yield a feature-set containing features initially entirely missing from one or other of the unified feature-sets. The significance of this point will emerge later on, in the discussions of the *procedural neutrality* of combinatory rules in section 2.4, and of the related *transparency property* of functional categories in section 2.3. Since the properties in question inhere to the grammar itself, to which unification is merely transparent, there is nothing in our approach that is *incompatible* with the more general definition of graph unification offered by PATR-2. However, in order to establish the correctness of our proposal for efficient parsing of extended categorial grammars using the more general definition, we would have had to neutralise its greater power with more laborious constraints on the encoding of entries in the categorial lexicon as dags than those we actually require below. The more restricted version we propose preserves most of the advantages of graph over term data-structures pointed out in Shieber (1986).[3]

2.2. Categories as Features Structures

We encode constituents corresponding to non-functional categories, such as the noun-phrases below, as feature-sets defining the three major attributes *syntax*, *phonology* and *semantics*, abbreviated for reasons of space to *syn*, *pho*, and *sem* (the examples of feature-based categories given below are of course simplified for the purposes of concise exposition -- for instance, we omit any specification of agreement information in the value associated with the *syn(tax)* label):

(11)
```
John:=      ([syn np]
             [pho john]
             [sem john'])
```

(12)
```
Mary:=      ([syn np]
             [pho mary]
             [sem mary'])
```

Constituents corresponding to functional categories are feature-sets characterized by a triple of attributes, *result*, *direction*, and *argument*, abbreviated to *res*, *dir*, and *arg*. The value associated with *dir(ection)* can be instantiated to one of the constants / and \ and the values associated with *res(ult)* and *arg(ument)* can be associated with any functional or non-functional category. (Thus our functions are "curried", and may be higher order.)

We impose the simple but crucial requirement of *transparency* over the well-formedness of functional categories in feature-based CCG. Intuitively, this requirement corresponds to the idea that any change to the structure of the value of *arg(ument)* caused by unification must be reflected in the value of *res(ult)*. Given the definition of unification in the section above, this requirement can be simply stated as follows:

(13) Functional categories must be *transparent*, in the sense that every uninstantiated feature in the value of a function's *arg(ument)* feature -- that is, every feature whose value is a variable -- must share that variable value with some feature in the value of the function's *res(ult)* feature.

Thus, whenever a feature in a function's *arg(ument)* is instantiated by unification, some other feature in its *res(ult)* will be instantiated identically, as a side-effect of the destructive replacement of structures imposed by unification. Variables in the value of the *arg(ument)* of a functional category therefore have the sole effect of increasing the specificity of the information contained in the value of its *res(ult)*. As the combinatory rules of CCG build new constituents exclusively in terms of information already contained in the categories that they combine, a requirement that all the functional categories in the lexicon be transparent in turn guarantees the transparency of any functional category assigned to complex constituents generated by the grammar.

[3] Calder (1987) and Thompson (1987) have independently motivated similar approaches to constraining unification in encoding

The following feature-based functional category for a lexical transitive tensed verb obeys the transparency requirement (the operator * indicates string concatenation):

(14) loves:=

```
([res ([res ([syn s]
             [pho P1*loves*P2]
             [sem ([act     loving]
                   [agent   S1]
                   [patient S2]    )]))]
       [dir \]
       [arg ([syn np]
             [pho P1]
             [sem S1])]                    )]
 [dir /]
 [arg ([syn np]
       [pho P2]
       [sem S2])]                           )
```

When two adjacent feature-structures corresponding to a function category X_1 and an argument X_2 are combined by functional application, a new feature-structure X_0 is constructed by unifying the argument feature-structure X_2 with the value of the *arg(ument)* in the function feature structure X_1. The result X_0 is then unified with the *res(ult)* of the function. For example, Rightward Application can be expressed in a notation adapted from PATR-2 as follows. We use the notation $<l_1 \ldots l_n>$ for a path of feature labels of length n, and we identify as $X_n(<l_1 \ldots l_n>)$ the value associated with the feature identified by the path $<l_1 \ldots l_n>$ in the dag corresponding to a category X_n. We indicate unification with the equality sign, =. Rightward Application can then be written as:

(15) **Rightward Application:**

X_0 ==> X_1 X_2

X_1(<direction>) = /
X_1(<arg>) = X_2
X_1(<result>) = X_0

Application of this rule to the functional feature-set (14) for the transitive verb *loves* and the feature-set (12) for the noun-phrase *Mary* yields the following structure for the verb-phrase *loves Mary*:

(16) loves Mary:=

```
([res ([syn s]
       [pho P1*loves*mary]
       [sem ([act     loving]
             [agent   S1]
             [patient mary'] )])]
 [dir \]
 [arg ([syn np]
       [pho P1]
       [sem S1])]                    )
```

To rightward-compose two functional categories according to rule (4b), we similarly unify the appropriate *arg(ument)* and *res(ult)* features of the input functions according to the following rule:

(17) **Rightward Composition:**

X_0 ==> X_1 X_2

X_1(<direction>) = /
X_2(<direction>) = /
X_1(<arg>) = X_2(<result>)
X_2(<direction>) = X_0(<direction>)
X_1(<result>) = X_0(<result>)
X_2(<arg>) = X_0(<arg>)

For example, suppose that the non-functional feature-set (11) for the noun-phrase *John* is type-raised into the following functional feature-set, according to rule (4a), whose unification-based version we omit here:

(18) John:=

```
([res ([syn s]
       [pho P]
       [sem S])]
 [dir /]
 [arg ([res ([syn s]
             [pho P]
             [sem S])]
       [dir \]
       [arg ([syn np]
             [pho john]
             [sem john'])])])
```

Then (18) can be combined by Rightward Composition with (14) to obtain the following feature structure for the functional category corresponding to *John loves*:

(19) John loves:=

```
([res ([syn s]
       [pho john*loves*P2]
       [sem ([act     loving]
             [agent   john']
             [patient S2])])]
 [dir /]
 [arg ([syn np]
       [pho P2]
       [sem S2])])
```

Leftward-combining rules are defined analogously to the rightward-combining rules above.

2.3. Derivational Equivalence Modulo Composition

Let us denote the operations of applying and composing categories by writing *apply*(X, Y) and *comp*(X, Y) respectively. Then by the definition of the operations themselves, and in particular because of the associativity of functional composition, the following equivalences hold across type-derivations:

(20) apply(comp(X_1, X_2), X_3) = apply(X_1, apply(X_2, X_3))

(21) comp(comp(X_4, X_5), X_6) = comp(X_4, comp(X_5, X_6))

More formally, the left-hand side and right-hand side of both equations define equivalent terms in the combinatory logic of

---

linguistic theories.

Curry and Feys (1958).[4] It follows that *all* alternative derivations of an arbitrary sequence of functions and arguments that are allowed by different orders of application and composition in which a composition is merely traded for an application also define equivalent terms of Combinatory Logic.[5]

So, for instance, a type for the sentence *John loves Mary* can be assigned either by rightward-composing the type-raised function John, (18), with *loves*, (14), to obtain the feature-structure (19) for *John loves*, and then rightward applying (19) to *Mary*, (12), to obtain a feature-structure for the whole sentence; or, conversely, it can be assigned by rightward-applying *loves*, (14), to *Mary*, (12), to obtain the feature-structure (16) for *loves Mary*, and then rightward-applying *John*, (18), to (16) to obtain the final feature-structure. In both cases, as the reader may care to verify, the type-assignment we get is the following:

(22) John loves Mary:=
```
([syn s]
 [pho john*loves*mary]
 [sem ([act loving]
       [agent john']
       [patient mary'])])
```

An important property of CCG is that it unites syntactic and semantic combination in uniform operations of application and composition. Unification-based CCG makes this identification explicit by uniting the syntactic type of a constituent and its interpretation in a single feature-based type. It follows that all derivations for a given string induced by functional composition correspond to the same unique feature-based type, which cannot be assigned to any other constituent in the grammar.[6] This property, which we characterize formally elsewhere, is a direct consequence of the fact that unification is itself an associative operation.

It follows in turn that a feature-based category like (22) associated with a given constituent not only contains all the information necessary for its grammatical interpretation, but also determines an equivalence class of derivations for that constituent, a point which is related to Karttunen's (1986) proposal for the spurious ambiguity problem (cf. secn. 1 above), but which we exploit differently, as follows.

2.4. Procedural Neutrality of Combinatory Rules

The rules of combinatory categorial grammar are purely declarative, and unification preserves this property, so that, as with other unification-based grammatical formalisms (cf. Shieber 1986), there is no procedural constraint on their use. So far, we have only considered examples in which such rules are applied "bottom-up", as in example (16), in which the rule of application (15) is used to define the feature structure X_0 on the left-hand side of the rule in terms of the feature structures X_1 and X_2 on the right, respectively instantiated as the function *loves* (14) and its argument *Mary* (12). However, other procedural realizations are equally viable.[7] In particular, it is a property of rules (15) and (17), (and of all the combinatory rules permitted in the theory -- cf. Steedman 1986) that if *any* two out of the three elements that they relate are specified, then the third is entirely and uniquely determined. This property, which we call *procedural neutrality* follows from the form of the rules themselves and from the transparency property (13) of functional categories, under the definition of unification given in section 2.1 above.[8]

This property of the grammar offers a way to short-circuit the entire problem of non-determinism in a chart-based parser for grammars characterised by spurious analyses engendered by associative rules such as composition. The procedural neutrality of the combinatory rules allows a processor to recover constituents which are "implicit" in analysed constituents in the sense that they would have been built if some *other* equivalent analysis had happened to have been the one followed by the processor. For example, consider the situation where, faced with the string *John loves Mary* dealt with in the last section, the processor has avoided multiple analyses by composing *John*, (18), with *loves*, (14), to obtain *John loves*, (19), and has then applied that to *Mary*, (12), to obtain *John loves Mary* (22), ignoring the other analysis. If the parser turns out to need the constituent *loves Mary*, (16), (as it will if it is to find a sensible analysis when the sentence turns out to be *John loves Mary madly*), then it can recover that constituent by defining it via the rule of Rightward Application in terms of the feature structures for *John loves Mary*, (22), and *John*, (18). These two feature structures can be used to respectively instantiate X_0 and X_1 in the rule as stated at (15). The reader may verify that instantiating the rule in this way determines the required constituent to be exactly the same category as (16).

This particular procedural alternative to the bottom-up invocation of combinatory rules will be central to the parsing algorithm which we present in the following section, so it will be convenient to give it a name. Since it is the "parent" category X_0 and the "left-constituent" category X_1 that are instantiated, it seems natural to call this alternative **left-branch instantiation** of a combinatory rule, a term which we contrast with the **bottom-up instantiation** invoked in earlier examples.

The significance of this point is as follows. Let us suppose that we can guarantee that a parser will always make available, say in a chart, the constituent that *could* have combined under

---

[4] The terms are equivalent in the technical sense that they reduce to an identical normal form.

[5] The inclusion of certain higher-order function categories in the lexicon (of which "modifiers of modifiers" like *formerly* would be an example in English) means that composition may affect the argument structure itself, thereby changing meaning and giving rise to *non*-equivalent terms. This possibility does not affect the present proposal, and can be ignored.

[6] If there is genuine ambiguity, a constituent will of course be assigned more than one type.

[7] There is an obvious analogy here with the fact that unification-based programming languages like Prolog do not have any predefined distinction between the input and the output parameters of a given procedure.

[8] From a formal point of view, procedural neutrality is a consequence of the fact that unification-based combinatory rules, as characterised above, are *extensional*. Thus, we follow Pereira and Shieber (1984) in claiming that the "bottom-up" realization of a unification-based rule r corresponds to the unification of a structure E_r encoding the equational constraints of r, and a structure D_r corresponding to the merging of the structures instantiating the elements of the right-hand side of r. A structure N_r is consequently assigned as the instantiation of the left-hand side of r by individuating a relevant substructure of the unification of the pair $\langle D_r, E_r \rangle$. If r is a rule of unification-based CCG, then the fact that N_r is the instantiation of the left-hand side of r both in terms of $\langle D_r, E_r \rangle$ and $\langle D_r', E_r \rangle$ guarantees that D_r and D_r' are identical (in the sense that they subsume each other).

bottom-up instantiation as a left-constituent with an implicit right-constituent to yield the same result as the analysis that was actually followed. In that case, the processor will be able to recover the implicit right-constituent by left-branch instantiation of a *single* combinatory rule, without restarting syntactic analysis and without backtracking or search of any kind. The following algorithm does just that.

3. A Lazy Chart Parsing Methodology

Derivational equivalence modulo composition, together with the procedural neutrality of unification-based combinatory rules, allows us to define a novel generalisation of the classic chart parsing technique for extended CGs, which is "lazy" in the sense that:

a) only edges corresponding to *one* of the set of semantically equivalent analyses are installed on the chart;

b) surface constituents of already parsed parts of the input which are not on the chart are directly generated from the structures which are, rather than being built from scratch via syntactic reanalysis.

3.1. A Bottom-up Left-to-Right Algorithm

The algorithm we decribe here implements a bottom-up, left-to-right parser which delivers all semantically distinct analyses. Other algorithms based on alternative control strategies are equally feasible. In this specific algorithm, the distinction between *active* and *inactive* edges is drawn in a rather different way from the standard one. For an edge *E* to be active does not meanthat it is associated with an incomplete constituent (indeed, the distinction between complete and incomplete constituents is eliminated in CCG); it simply means that *E* can trigger new actions of the parser to install other edges, after which *E* itself becomes inactive. By contrast, inactive edges cannot initiate modifications to the state of the parser.

Active edges can be added to the chart according to the three following actions:

- **Scanning**: if *a* is a word in the input string then, for each lexical entry X associated with *a*, add an active edge labeled X spanning the vertices corresponding to the position of *a* on the chart.
- **Lifting**: if E_1 is an active edge labeled X_1, then for every unary rule of type raising which can be instantiated as X_0 ==> X_1 add an active edge E_0 labeled X_0 and spanning the same vertices of E_1.
- **Reducing**: if an edge E_2 labeled X_2 has a left-adjacent edge E_1 labeled X_1 and there is a combinatory rule which can be instantiated as X_0 ==> X_1 X_2 then add an active edge E_0 labeled X_0 spanning the starting vertex of E_1 and the ending vertex E_2.

The operational meaning of Scanning and Lifting should be clear enough. The Reducing action is the workhorse of the parser, building new constituents by invoking combinatory rules via bottom-up instantiation. Whenever Reducing is effected over two edges E_1 and E_2 to obtain a new edge E_0 we ensure that:

> E_1 is marked as a **left-generator** of E_0. If the rule in the grammar which was used is Rightward Composition, then E_2 is marked as a **right-generator** of E_0.

The intuition behind this move is that *right-generators* are rightward functional categories which have been composed into, and will therefore give rise to spurious analyses if they take part in further rightward combinations, as a consequence of the property of derivational equivalence modulo composition, discussed in section 2.3. *Left-generators* correspond instead to choice points from where it would have been possible to obtain a derivationally different but semantically equivalent constituent analysis of some part of the input string. They thus constitute suitable constituents for use in recovering *implicit* right-constituents of other constituents in the chart via the invocation of combinatory rules under the procedure of left-branch instantiation discussed in the last section.

In order to state exactly how this is done, we need to introduce the **left-starter** relation, corresponding to the transitive closure of the left-generator relation:

(i) A left-generator L of an edge E is a left-starter of E.

(ii) If L is a left-starter of E, then any left-starter of L is a left-starter of E.

The parser can now add *inactive* edges corresponding to *implicit* right-constituents according to the following action:

- **Revealing**: if an edge E is labeled by a leftward-looking functional type X and there is a combinatory rule which can be instantiated as X' ==> X_2 X then if

  (i) there is an edge E_0 labeled X_0 left-adjacent to E
  (ii) E_0 has a left-starter E_1 labeled X_1
  (iii) there is a combinatory rule which can be instantiated as X_0 ==> X_1 X_2

  then add to the chart an inactive edge E_2 labeled X_2 spanning the ending vertex of E_1 and the starting vertex of E, unless there is already an edge labelled in the same way and spanning the same vertices. Mark E_2 as a right-generator of E_0 if the rule used in (iii) was Rightward Composition.

To summarise the section so far: if the parser is devised so as to avoid putting on the chart subconstituents which would lead to redundant equivalent derivations, non-determinism in the grammar will always give rise to cases which require some of the excluded constituents. In a left-to-right processor this typically happens when the argument required by a leftward-looking functional type has been mistakenly combined in the analysis of a substring left-adjacent to that leftward-looking type. However, such an implicit or hidden constituent could have only been obtained through an equivalent derivation path for the left-adjacent substring. It follows that we can "reveal" it on the chart by invoking a combinatory rule in terms of left-branch instantiation.

We can now informally characterize the algorithm itself as follows:

- the parser does Scanning for each word in the input string going left-to-right
- moreover, whenever an active edge A is added to the chart, then the following actions are taken in order:

  (i) the parser does Lifting over A
  (ii) if A is labeled by a leftward-looking type, then for every edge E left-adjacent to A the parser does Revealing over E with respect to A

(iii) for every edge E left-adjacent to A the parser does Reducing over E and A, with the constraint that if A is not labeled by a leftward-looking type then E must not be a right-generator of any edge E'

- the parser returns the set of categories associated with edges spanning the whole input, if such a set is not empty; it fails otherwise.

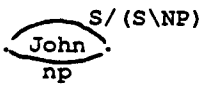
3.2. An Example

In the interests of brevity and simplicity, we eschew all details to do with unification itself in the following examples of the workings of the parser, reverting to the original categorial notation for CCG of section 1, bearing in mind that the categories are now to be read strictly as a shorthand for the fuller notation of unification-based CCG. For similar reasons of simplicity in exposition, we assume for the present purpose that the only type-raising rule in the grammar is the subject rule (4a).

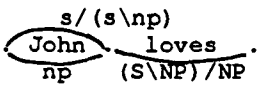
The algorithm analyses the sentence *John loves Mary madly* as follows. First, the parser Scans the first word *John*, adding to the chart an active NP edge corresponding to its sole lexical entry, and spanning the word in question, thus:

(23) 

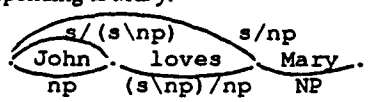
(We adopt the convention that active edges are indicated by upper-case categories, while inactive edges will be indicated with lower-case categories.) Since the edge in question is active, it falls under the second clause of the algorithm. The Lifting condition (i) of this clause applies, since there is a rule which type raises over NP, so a new active edge of type S/(S\NP) is added, spanning the same word, *John* (no other conditions apply to the NP active edge, and it becomes inactive):

(24) 

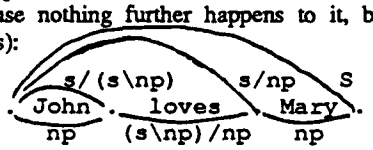
Neither Lifting, Revealing, nor Reducing yield any new edges, so the new active edge merely becomes inactive. The next word is Scanned to add a new lexical active edge of type (S\NP)/NP spanning *loves*:

(25) 

The new lexical edge Reduces with the type-raised subject to yield a new active edge of type S/NP. The subject category is marked as the new edge's left-generator, and (because the combinatory rule was Rightward Composition) the verb category is marked as its right-generator. Nothing more results from *loves*, and neither Lifting, Revealing nor Reducing yield anything from the new edge, so it too becomes inactive, and the next word is Scanned to add a new lexical active NP edge corresponding to *Mary*:

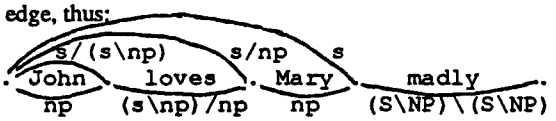
(26) 

This edge yields *two* new active edges before becoming inactive, one of type S/(S\NP) via Lifting and the subject rule, and one of type S, via Reducing with the s/np edge to its left by the Forward application rule (we omit the former from the illustration, because nothing further happens to it, but it is there nonetheless):

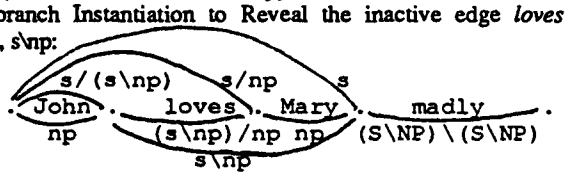
(27) 

The s/np edge is in addition marked as the left generator of the S. Note that Reducing would potentially have allowed a third new active edge corresponding to *loves Mary* to be added by Reducing the new active NP edge corresponding to *Mary* with the left-adjacent (s\np)/np edge, *loves*. However, this edge has been marked as a right generator, and is therefore not allowed to Reduce by the algorithm.

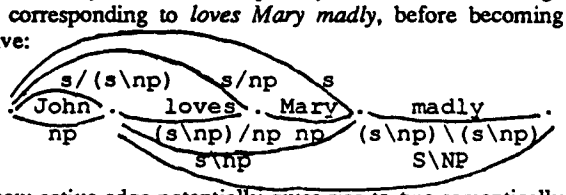
Nothing new results from the new active S edge, so it becomes inactive and the next word *madly* is scanned to add a new active edge, thus:

(28) 

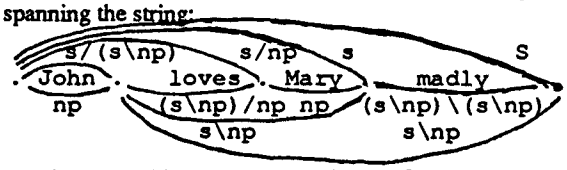
This active edge, being a leftward-looking functional type, precipitates Revealing. Since there is a rule (Backward Application, 2a) which would allow *madly*, (S\NP)\(S\NP) to combine with a left-adjacent s\np, *and* there is a rule (Forwards Application, 2a) which would allow a left-starter *John* s/(s\np) to combine with such an s\np to yield the s which is left-adjacent to *madly*, (and since there is no left-adjacent s\np there already), the rule of Forward Application can be invoked via Left-branch Instantiation to Reveal the inactive edge *loves Mary*, s\np:

(29) 

The (still) active backward modier *madly* can now Reduce with the newly introduced s\np, to yield a new active edge S\NP corresponding to *loves Mary madly*, before becoming inactive:

(30) 

The new active edge potentially gives rise to *two* semantically equivalent Reductions with the subject *John* to yield S -- one with its ground np type, and one with its raised type, s/(s\np). Only one of these is effected, because of a detail dealt with in the next section, and the algorithm terminates with a single S edge spanning the string:

(31) 

In an attachment-ambiguous sentence like the following, which we leave as an exercise, *two* predicates, *believes John loves Mary* and *loves Mary*, are revealed in the penultimate stage of the analysis, and *two* semantically distinct analyses result:

(32) Fred believes John loves Mary passionately

Space permits us no more than to note that this procedure will

also cope with another class of constructions which constitute a major source of non-determinism in natural language parsing, namely the diverse coordinate constructions whose categorial analysis is discussed by Dowty (1985) and Steedman (1985, 1987).

4. Type Raising and Spurious Ambiguity

As noted at example (30) above, type raising rules introduce a second kind of spurious ambiguity connected to the interactions of such rules with functional application rather than functional composition. If the processor can Reduce via a rule of application on a type-raised category, then it can also always invoke the opposite rule of application to the *un*raised version of the same category to yield the same result. Spurious ambiguity of this kind is trivially easy to avoided, as (unlike the kind associated with composition), it can always be detected *locally* by the following redundancy check on attachment of new edges to the chart in Reducing: *when Reducing creates an edge via functional application, then it is only added to the chart if there is no edge associated with the same feature structure and spanning the same vertices already on the chart.*

5. Alternative Control Strategies and Grammatical Formalisms

The algorithm described above is a pure bottom-up parsing procedure which has a close relative in the Cocke-Kasami-Younger algorithm for context-free phrase-structure grammars. However, our chart-parsing methodology is completely open to alternative control options. In particular, Pareschi (forthcoming) describes an adaptation of the Earley algorithm, which, in virtue of its top-down prediction stage, allows for efficient application of more general type-raising rules than are considered here. Formal proofs of the correctness of both these algorithms will be presented in the same reference.

The possibility of exploiting this methodology for improving processing of other unification-based extensions of CG involving spurious ambiguity, like the one reported in Karttunen (1986a), is also under exploration.

6. Conclusion

The above approach to chart-parsing with extensions to CGs characterised by spurious ambiguities allows us to define algorithms which do not build significantly more edges than chart parsers for more standard theories of grammar. Our technique is fully transparent with respect to our grammatical formalism, since it is based on properties of associativity and procedural neutrality inherent in the grammar itself.[9]

ACKNOWLEDGEMENTS

We thank Inge Bethke, Kit Fine, Ellen Hays, Aravind Joshi, Dale Miller, Henry Thompson, Bonnie Lynn Webber, and Kent Wittenburg for help and advice. Parts of the research were supported by: an Edinburgh University Research Studentship; an ESPRIT grant (project 393) to CCS, Univ. Edinburgh; a Sloan Foundation grant to the Cognitive Science Program, Univ. Pennsylvania; and NSF grant IRI-10413 A02, ARO grant DAA6-29- 84K-0061 and DARPA grant N0014-85-K0018 to CIS, Univ. Pennsylvania.

[9] Chart parsers based on the methodology described here and written in Quintus Prolog have been developed on a Sun workstation.

REFERENCES

Ades, A. and Steedman, M. J. (1982) On the Order of Words. Linguistics and Philosophy, 44, 517-518.

Calder, J. (1987) Typed Unification for Natural Language Processing. Ms, Univ. of Edinburgh

Curry, H. B. and Feys, R. (1958) Combinatory Logic, Volume I. Amsterdam: North Holland.

Dowty, D. (1985). Type raising, functional composition and non-constituent coordination. In R. Oehrle et al, (eds.), Categorial Grammars and Natural Language Structures, Dordrecht, Reidel. (In press).

Haddock, N. J. (1987) Incremental Interpretation and Combinatory Categorial Grammar. In Proceedings of the Tenth International Joint Conference on Artificial Intelligence, Milan, Italy, August, 1987.

Hinrichs, E. and Polanyi, L. (1986) Pointing the Way. Papers from the Parasession on Pragmatics and Grammatical Theory at the Twenty-Second Regional Meeting of the Chicago Linguistic Society, pp.298-314.

Karttunen, L. (1986) Radical Lexicalism. Paper presented at the Conference on Alternative Conceptions of Phrase Structure, July 1986, New York.

Kay, M. (1980) Algorithm Schemata and Data Structures in Syntactic Processing. Technical Report No. CSL-80- 12, XEROX Palo Alto Research Centre.

Pareschi, Remo. 1986. Combinatory Categorial Grammar, Logic Programming, and the Parsing of Natural Language. DAI Working Paper, University of Edinburgh.

Pareschi, R. (forthcoming) PhD Thesis, Univ. Edinburgh.

Pereira, F. C. N. and Shieber, S. M. (1984) The Semantics of Grammar Formalisms Seen as Computer Languages. In Proceedings of the 22nd Annual Meeting of the ACL, Stanford, July 1984, pp.123-129.

Shieber, S. M. (1986) An Introduction to Unification-based Approaches to Grammar, Chicago: Univ. Chicago Press.

Steedman, M. (1985) Dependency and Coordination in the Grammar of Dutch and English. Language, 61, 523-568.

Steedman,M. (1986) Combinatory Grammars and Parasitic Gaps. Natural Language and Linguistic Theory, to appear.

Steedman, M. (1987) Coordination and Constituency in a Combinatory Grammar. In Mark Baltin and Tony Kroch, (eds.), Alternative Conceptions of Phrase Structure, University of Chicago Press: Chicago. (To appear.)

Thompson, H. (1987) FBF - An Alternative to PATR as a Grammatical Assembly Language. Research Paper, Department of A.I, Univ. Edinburgh.

Uszkoreit, H. (1986) Categorial Unification Grammars. In Proceedings of the 11th International Conference on Computational Linguistics, Bonn, August, 1986, pp187-194.

Wittenburg, K. W. (1986) Natural Language Parsing with Combinatory Categorial Grammar in a Graph-Unification-Based Formalism. PhD Thesis, Department of Linguistics, University of Texas.

Zeevat, H., Klein, E. and Calder, J. (1987) An Introduction to Unification Categorial Grammar. In N. Haddock et al. (eds.), Edinburgh Working Papers in Cognitive Science, 1: Categorial Grammar, Unification Grammar, and Parsing.