# Efficient Incremental Processing with Categorial Grammar

Mark Hepple

University of Cambridge Computer Laboratory,
New Museums Site, Pembroke St, Cambridge, UK.
e-mail: mrh@uk.ac.cam.cl

## Abstract

Some problems are discussed that arise for incremental processing using certain *flexible* categorial grammars, which involve either undesirable parsing properties or failure to allow combinations useful to incrementality. We suggest a new calculus which, though 'designed' in relation to categorial interpretations of some notions of dependency grammar, seems to provide a degree of flexibility that is highly appropriate for incremental interpretation. We demonstrate how this grammar may be used for efficient incremental parsing, by employing normalisation techniques.

## Introduction

A range of categorial grammars (CGs) have been proposed which allow considerable flexibility in the assignment of syntactic structure, a characteristic which provides for categorial treatments of extraction (Ades & Steedman, 1982) and non-constituent coordination (Steedman, 1985; Dowty, 1988), and that is claimed to allow for incremental processing of natural language (Steedman, 1989). It is this latter possibility that is the focus of this paper.

Such 'flexible' CGs (FCGs) typically allow that grammatical sentences may be given (amongst others) analyses which are either fully or primarily left-branching. These analyses have the property of designating many of the initial substrings of sentences as interpretable constituents, providing for a style of processing in which the interpretation of a sentence is generated 'on-line' as the sentence is presented. It has been argued that incremental interpretation may provide for efficient language processing — by both humans and machines — in allowing early filtering of thematically or referentially implausible readings. The view that human sentence processing is 'incremental' is supported by both introspective and experimental evidence.

In this paper, we discuss FCG approaches and some problems that arise for using them as a basis for incremental processing. Then, we propose a grammar that avoids these problems, and demonstrate how it may be used for efficient incremental processing.

## Flexible Categorial Grammars

CGs consist of two components: (i) a categorial lexicon, which assigns to each word at least one syntactic type (plus associated meaning), (ii) a calculus which determines the set of admitted type combinations and transitions. The set of types (T) is defined recursively in terms of a set of basic types ($T_0$) and a set of operators (\ and /, for standard *bidirectional* CG), as the smallest set such that (i) $T_0 \subseteq T$, (ii) *if* x,y $\in$ T, *then* x\y, x/y $\in$ T.[1] Intuitively, lexical types specify subcategorisation requirements of words, and requirements on constituent order. The most basic (non-flexible) CGs provide only rules of application for combining types, shown in (1). We adopt a scheme for specifying the semantics of combination rules where the rule name identifies a function that applies to the meanings of the input types in their left-to-right order to give the meaning of the result expression.

(1)  f:  X/Y + Y $\Rightarrow$ X    (where $f = \lambda a \lambda b.(ab)$)
     b:  Y + X\Y $\Rightarrow$ X    (where $b = \lambda a \lambda b.(ba)$)

### The Lambek calculus

We begin by briefly considering the (product-free) *Lambek calculus* (LC – Lambek, 1958). Various formulations of the LC are possible (although we shall not present one here due to space limitations).[2]

The LC is complete with respect to an intuitively sensible interpretation of the slash connectives whereby the type x/y (resp. x\y) may be assigned to any string $x$ which when left-concatenated (resp. right-concatenated) with any string $y$ of type y yields a string $x.y$ (resp. $y.x$) of type x. The LC can be seen to provide the limit for what are possible

---

[1] We use a categorial notation in which x/y and x\y are both functions from y into x, and adopt a convention of *left association*, so that, e.g. ((s\np)/pp)/np may be written s\np/pp/np.

[2] See Lambek (1958) and Moortgat (1989) for a sequent formulation of the LC. See Morrill, Leslie, Hepple & Barry (1990), and Barry, Hepple, Leslie & Morrill (1991) for a natural deduction formulation. Zielonka (1981) provides a LC formulation in terms of (recursively defined) reduction schema. Various *extensions* of the LC are currently under investigation, although we shall not have space to discuss them here. See Hepple (1990), Morrill (1990) and Moortgat (1990b).

type combinations — the other calculi which we consider admit only a subset of the Lambek type combinations.[3]

The flexibility of the LC is such that, for any combination $x_1,..,x_n \Rightarrow x_0$, a fully left-branching derivation is always possible (i.e. combining $x_1$ and $x_2$, then combining the result with $x_3$, and so on). However, the properties of the LC make it useless for practical incremental processing. Under the LC, there is always an *infinite* number of result types for any combination, and we can only in practice address the possibility of combining some types to give a *known* result type. Even if we were to allow only S as the *overall* result of a parse, this would not tell us the intermediate target types for binary combinations made in incrementally accepting a sentence, so that such an analysis cannot in practice be made.

## Combinatory Categorial Grammar

*Combinatory Categorial Grammars* (CCGs – Steedman, 1987; Szabolcsi, 1987) are formulated by adding a number of type combination and transition schemes to the basic rules of application. We can formulate a simple version of CCG with the rules of *type raising* and *composition* shown in (2). This CCG allows the combinations (3a,b), as shown by the proofs (4a,b).

(2)    T: $x \Rightarrow y/(y\backslash x)$    (where $T = \lambda x \lambda f.(fx)$)
       B: $x/y + y/z \Rightarrow x/z$

$$(\text{where } B = \lambda f \lambda g \lambda x.f(gx))$$

(3) a.    $np{:}x, s\backslash np/np{:}f \Rightarrow s/np{:}\lambda y.fyx$

    b.    $vp/s{:}f, np{:}x \Rightarrow vp/(s\backslash np){:}\lambda g.f(gx)$

(4) (a)
```
    np        s\np/np    (b)  vp/s        np
  ---------T                           ---------T
  s/(s\np)                             s/(s\np)
  ------------------B                  ----------------B
        s/np                              vp/(s\np)
```

The derived rule (3a) allows a subject NP to combine with a transitive verb *before* the verb has combined with its object. In (3b), a sentence embedding verb is composed with a raised subject NP. Note that it is not clear for this latter case that the combination would *usefully* contribute to incremental processing, i.e. in the resulting semantic expression, the meanings of the types combined are not directly related to each other, but rather a hypothetical function mediates between the two. Hence, any

requirements that the verb may have on the semantic properties of its argument (i.e. the clause) could not be exploited at this stage to rule out the resulting expression as semantically implausible. We define as *contentful* only those combinations which directly relate the meanings of the expressions combined, without depending on the mediation of hypothetical functions.

Note that this calculus (like other versions of CCG) fails to admit some combinations, which are allowed by the LC, that are contentful in this sense — for example, (5). Note that although the semantics for the result expression in (5) is complex, the meanings of the two types combined are still directly related — the lambda abstractions effectively just fulfil the role of swapping the argument order of the subordinate functor.

(5)    $x/(y\backslash z){:}f, y/w\backslash z{:}g \Rightarrow x/w{:}\lambda v.f(\lambda w.gwv)$

Other problems arise for using CCG as a basis for incremental processing. Firstly, the free use of type-raising rules presents problems, i.e. since the rule can always apply to its own output. In practice, however, CCG grammars typically use type specific raising rules (e.g. $np \Rightarrow s/(s\backslash np)$), thereby avoiding this problem. Note that this restriction on type-raising also excludes various possibilities for flexible combination (e.g. so that not all combinations of the form $y, x\backslash y/z \Rightarrow x/z$ are allowed, as would be the case with unrestricted type-raising).

Some problems for efficient processing of CCGs arise from what has been termed 'spurious ambiguity' or 'derivational equivalence', i.e. the existence of multiple distinct proofs which assign the same reading for some combination of types. For example, the proofs (6a,b) assign the same reading for the combination. Since search for proofs must be exhaustive to ensure that all distinct readings for a combination are found, effort will be wasted constructing proofs which assign the same meaning, considerably reducing the efficiency of processing. Hepple & Morrill (1989) suggest a solution to this problem that involves specifying a notion of *normal form* (NF) for CCG proofs, and ensuring that the parser returns only NF proofs.[4] However, their method has a number of limitations. (i) They considered a 'toy grammar' involving only the CCG rules stated above. For a grammar involving further combination rules, normalisation would need to be completely reworked, and it remains to be shown that this task can be successfully done. (ii)

---

[3] In some frameworks, the use of non-Lambek-valid rules such as *disharmonic composition* (e.g. $x/y + y\backslash z \Rightarrow x\backslash z$) has been suggested. We shall not consider such rules in this paper.

[4] Normalisation has also been suggested to deal with the problem of spurious ambiguity as it arises for the LC. See König (1989), Hepple (1990) and Moortgat (1990).

The NF proofs of this system are *right*-branching — again, it remains to be shown that a NF can be defined which favours left-branching (or even primarily left-branching) proofs.

(6)  (a)  $x/y$   $y/z$   $z$   (b)  $x/y$   $y/z$   $z$

(6) (a) and (b) show proof diagrams with lines labelled f, B and derived types y, x/z, x.

## Meta-Categorial Grammar

In *Meta-Categorial Grammar* (MCG – Morrill, 1988) combination rules are recursively defined from the application rules (**f** and **b**) using the *metarules* (7) and (8). The metarules state that given a rule of the form shown to the left of $\Rightarrow$ with name $\phi$, a further rule is allowed of the form shown to the right, with name given by applying **R** or **L** to $\phi$ as indicated. For example, applying **R** to backward application gives the rule (9), which allows combination of subject and transitive verb, as **T** and **B** do for CCG. Note, however, that this calculus does not allow any 'non-contentful' combinations — all rules are recursively defined on the application rules which require a proper functional relation between the types combined. However, this calculus also fails to allow some contentful combinations, such as the case $x/(y\backslash z)$, $y/w\backslash z \Rightarrow x/w$ mentioned above in (5). Like CCG, MCG suffers from spurious ambiguity, although this problem can be dealt with via normalisation (Morrill, 1988; Hepple & Morrill, 1989).

(7)  $\phi\colon x + y \Rightarrow z \implies \mathbf{R}\phi\colon x + y/w \Rightarrow z/w$
(where $\mathbf{R} = \lambda g \lambda a \lambda b \lambda c.ga(bc)$)

(8)  $\phi\colon x + y \Rightarrow z \implies \mathbf{L}\phi\colon x\backslash w + y \Rightarrow z\backslash w$
(where $\mathbf{L} = \lambda g \lambda a \lambda b \lambda c.g(ac)b$)

(9)  $\mathbf{Rb}\colon y + x\backslash y/z \Rightarrow x/z$

## The Dependency Calculus

In this section, we will suggest a new calculus which, we will argue, is well suited to the task of incremental processing. We begin, however, with some discussion of the notions of head and dependent, and their relevance to CG.

The dependency grammar (DG) tradition takes as fundamental the notions of *head, dependent* and the head-dependent relationship; where a head is, loosely, an element on which other elements depend. An analogy is often drawn between CG and DG based on equating categorial functors with heads, whereby a functor $x/y_1../y_n$ (ignoring directionality, for the moment) is taken to correspond to a head

requiring dependents $y_1..y_n$, although there are several obvious differences between the two approaches. Firstly, a categorial functor specifies an ordering over its 'dependents' (function-argument order, that is, rather than constituent order) where no such ordering is identified by a DG head. Secondly, the arguments of a categorial functor are necessarily phrasal, whereas by the standard view in DG, the dependents of a head are taken to be words (which may themselves be heads of other head/dependent complexes). Thirdly, categorial functors may specify arguments which have complex types, which, by the analogy, might be described as a head being able to make stipulations about the dependency requirements of its dependent and also to 'absorb' those dependency requirements.[5] For example, a type $x/(y\backslash z)$ seeks an argument which is a "y needing a dependent z" under the head/functor analogy. On combining with such a type, the requirement "need a dependent z" is gone. Contrast this with the use of, say, composition (i.e. $x/y$, $y/z \Rightarrow x/z$), where a type $x/y$ simply needs a dependent y, and where composition allows the functor to combine with its dependent y while the latter still requires a dependent z, and where that requirement is inherited onto the result of the combination and can be satisfied later on.

Barry & Pickering (B&P, 1990) explore the view of dependency that arises in CG when the functor-argument relationship is taken as analogous to the traditional head-dependent relationship. A problem arises in employing this analogy with FCGs, since FCGs permit certain type transformations that undermine the head-dependent relations that are implicit in lexical type assignments. An obvious example is the type-raising transformation $x \Rightarrow y/(y\backslash x)$, which directly reverses the direction of the head-dependent relationship between a functor and its argument. B&P identify a subset of LC combinations as *dependency preserving* (DP), i.e. those combinations which preserve the head-dependent relations implicit in the types combined, and call constituents which have DP analyses *dependency constituents*. B&P argue for the significance of this notion of constituency in relation to the treatment of coordination and the comparative difficulty observed for (human) processing of nested and non-

---

[5] Clearly, a CG where argument types were required to be basic would be a closer analogue of DG in not allowing a 'head' to make such stipulations about its dependents. Such a system could be enforced by adopting a more restricted definition of the set of types (T) as the smallest set such that (i) $T_0 \subseteq T$, (ii) *if* $x \in T$ and $y \in T_0$, *then* $x\backslash y$, $x/y \in T$ (c.f. the definition given earlier).

nested constructions.[6] B&P suggest a means for identifying the DP subset of LC transformations and combinations in terms of the lambda expressions that assign their semantics. Specifically, a combination is DP *iff* the lambda expression specifying its semantics does not involve abstraction over a variable that fulfils the role of functor within the expression (c.f. the semantics of type raising in (2)).[7]

We will adopt a different approach to B&P for addressing dependency constituency, which involves specifying a calculus that allows all and only the DP combinations (as opposed to a criterion identifying a subset of LC combinations as DP). Consider again the combination $x/(y\backslash z)$, $y/w\backslash z \Rightarrow x/w$, not admitted by either the CCG or MCG stated above. This combination *would* be admitted by the MCG (and also the CCG) if we added the following (Lambek-valid) associativity axioms, as illustrated in (11).

(10)   a: $x\backslash y/z \Rightarrow x/z\backslash y$
   a: $x/y\backslash z \Rightarrow x\backslash z/y$

   (where $a = \lambda f \lambda a \lambda b.fba$)

(11)   $x/(y\backslash z)$   $\dfrac{y/w\backslash z}{y\backslash z/w}$a
   $\dfrac{\phantom{x/(y\backslash z)\quad y\backslash z/w}}{x/w}$Rf

We take it as self-evident that the unary transformations specified by these two axioms are DP, since function-argument order is a notion extraneous to dependency; the functors $x\backslash y/z$ and $x/z\backslash y$ have the same dependency requirements, i.e. dependents y and z.[8] For the same reason, such reordering of arguments should also be possible for functions that occur as subtypes within larger types, as in (12a,b). The operation of the associativity rules can be 'generalised' in this fashion by including the unary metarules (13),[9] which recursively define

---

[6]See Barry (forthcoming) for extensive discussion of dependency and CG, and Pickering (1991) for the relevance of dependency to human sentence processing.

[7]B&P suggest a second criterion in terms of the form of proofs which, for the natural deduction formulation of the LC that B&P use, is equivalent to the criterion in terms of lambda expressions (given that a variant of the Curry-Howard correspondence between implicational deductions and lambda expressions obtains).

[8]Clearly, the reversal of two co-directional arguments (i.e. $x/y/z \Rightarrow x/z/y$) would also be DP for this reason, but is *not* LC-valid (since it would not preserve linear order requirements). For a unidirectional CG system (i.e. a system with a single connective /, that did not specify linear order requirements), free reversal of arguments would be appropriate. We suggest that a unidirectional variant of the calculus to be proposed might be the best system for pure reasoning about 'categorial dependency', aside from linearity considerations.

[9]These unary metarules have been used elsewhere as part of the LC formulation of Zielonka (1981).

new unary rules from the associativity axioms.

(12)   a.   $a\backslash b/c/d \Rightarrow a/c\backslash b/d$

   b.   $x/(a\backslash b/c) \Rightarrow x/(a/c\backslash b)$

(13)   a.   $\phi: x \Rightarrow y \quad \Longrightarrow \quad \mathbf{V}\phi: x/z \Rightarrow y/z$
   $\phi: x \Rightarrow y \quad \Longrightarrow \quad \mathbf{V}\phi: x\backslash z \Rightarrow y\backslash z$

   (where $\mathbf{V} = \lambda f \lambda a \lambda b.f(ab)$)

   b.   $\phi: x \Rightarrow y \quad \Longrightarrow \quad \mathbf{Z}\phi: z/y \Rightarrow z/x$
   $\phi: x \Rightarrow y \quad \Longrightarrow \quad \mathbf{Z}\phi: z\backslash y \Rightarrow z\backslash x$

   (where $\mathbf{Z} = \lambda f \lambda a \lambda b.a(fb)$)

(14)   $x/(a\backslash b/c):f \Rightarrow x/(a/c\backslash b):\lambda v.f(\lambda a \lambda b.vba)$

Clearly, the rules $\{\mathbf{V},\mathbf{Z},a\}$ allow only DP unary transformations. However, we make the stronger claim that these rules specify the *limit* of DP unary transformations. The rules allow that the given functional structure of a type be 'shuffled' upto the limit of preserving linear order requirements. But the only alternative to such 'shuffling' would seem to be that some of the given type structure be removed or further type structure be added, which, by the assumption that functional structure expresses dependency relations, cannot be DP.

We propose the system $\{\mathbf{L},\mathbf{R},\mathbf{V},\mathbf{Z},a,f,b\}$ as a calculus allowing all and only the DP combinations and transformations of types, with a 'division of labour' as follows: (i) the rules f and b, allowing the establishment of direct head-dependent relations, (ii) the subsystem $\{\mathbf{V},\mathbf{Z},a\}$, allowing DP transformation of types upto the limit of preserving linear order, and (iii) the rules $\mathbf{R}$ and $\mathbf{L}$, which provide for the inheritance of 'dependency requirements' onto the result of a combination. We call this calculus the *dependency calculus* (DC) (of which we identify two subsystems: (i) the binary calculus $B = \{\mathbf{L},\mathbf{R},f,b\}$, (ii) the unary calculus $U = \{\mathbf{V},\mathbf{Z},a\}$). Note that B&P's criterion and the DC do not agree on what are DP combinations in all cases. For example, the semantics for the type transformation in (14) involves abstraction over a variable that occurs as a functor. Hence this transformation is not DP under B&P's criterion, although it is admitted by the DC. We believe that the DC is correct in admitting this and the other additional combinations that it allows.

There is clearly a close relation between DP type combination and the notion of contentful combination discussed earlier. The 'dependency requirements' stated by any lexical type will constitute the sum of the 'thematically contentful' relationships into which it may enter. In allowing all DP combinations (subject to the limit of preserving linear order requirements), the DC ensures that lexically

82

originating dependency structure is both preserved
and also exploited in full. Consequently, the DC is
well suited to incremental processing. Note, how-
ever, that there is some extent of divergence be-
tween the DC and the (admittedly vague) criterion
of 'contentful' combination defined earlier. Con-
sider the LC-valid combination in (15), which is
not admitted by the DC. This combination would
appear to be 'contentful' since no hypothetical se-
mantic functor intervenes between $f$ and $g$ (although
$g$ has undergone a change in its relationship to its
own argument which depends on such a hypothet-
ical functor). However, we do not expect that the
exclusion of such combinations will substract signif-
icantly from genuinely useful incrementality in pars-
ing actual grammars.

(15)  $x/(y/z){:}f,\ y/(w\backslash(w/z)){:}g \Rightarrow x{:}f(\lambda v.g(\lambda h.hv))$

## Parsing and the Dependency Calculus

Binary combinations allowed by the DC are all of
the form (16) (where the vertical dots abbrevi-
ate unary transformations, and $\phi$ is some binary
rule). The obvious naive approach to finding possi-
ble combinations of two types x and y under the DC
involves searching through the possible unary trans-
forms of x and y, then trying each possible pairing
of them with the binary rules of B, and then deriv-
ing the set of unary transforms for the result of any
successful combination.

At first sight, the efficiency of processing using
this calculus seems to be in doubt. Firstly, the
search space to be addressed in checking for possible
combinations of two types is considerably greater
than for CCG or MCG. Also, the DC will suffer spu-
rious ambiguity in a fashion directly comparable to
CCG and MCG (obviously, for the latter case, since
the above MCG is a subsystem of the DC). For ex-
ample, the combination x/y, y/z, z $\Rightarrow$ x has both
left and right branching derivations.

However, a further equivalence problem arises due
to the interderivability of types under the unary
subsystem U. For any unary transformation x $\Rightarrow$ y,
the converse y $\Rightarrow$ x is always possible, and the se-
mantics of these transformations are always inverses.
(This obviously holds for a, and can be shown to
hold for more complex transformations by a simple
induction.) Consequently, if parsing assigns distinct
types x and y to some substring that are merely
variants under the unary calculus, this will engen-
der redundancy, since anything that can be proven
with x can equivalently be proven with y.

(16)  
$$\begin{array}{cc} x & y \\ \vdots & \vdots \\ x' & y' \\ \hline \multicolumn{2}{c}{\phi} \\ z \\ \vdots \\ z' \end{array}$$

## Normalisation and the Dependency Calculus

These efficiency problems for parsing with the DC
can be seen to result from equivalence amongst terms
occurring at a number of levels within the system.
Our solution to this problem involves specifying nor-
mal forms (NFs) for terms — to act as privileged
members of their equivalence class — at three differ-
ent levels of the system: (i) types, (ii) binary com-
binations, (iii) proofs. The resulting system allows
for efficient categorial parsing which is incremental
up to the limit allowed by the DC.

A standard way of specifying NFs is based on
the method of *reduction*, and involves defining a
*contraction* relation ($\rhd_1$) between terms, which is
stated as a number of contraction rules of the form
X $\rhd_1$ Y (where X is termed a *redex* and Y its *con-
tractum*). Each contraction rule allows that a term
containing a redex may be transformed into a term
where that occurrence is replaced by its contractum.
A term is said to be in NF if and only if it contains
no redexes. The contraction relation generates a *re-
duction* relation ($\rhd$) such that X *reduces to* Y (X $\rhd$
Y) iff Y is obtained from X by a finite series (pos-
sibly zero) of contractions. A term Y is a NF of X
iff Y is a NF and X $\rhd$ Y. The contraction relation
also generates an equivalence relation which is such
that X = Y iff Y can be obtained from X by a se-
quence of zero or more steps, each of which is either
a contraction or reverse contraction.

Interderivability of types under U can be seen as
giving a notion of *equivalence* for types. The con-
traction rule (17) defines a NF for types. Since
contraction rules apply to any redex subformula oc-
curring within some overall term, this rule's do-
main of application is as broad as that of the as-
sociativity axioms in the unary calculus given the
generalising effects of the unary metarules. Hence,
the notion of equivalence generated by rule (16) is
the same as that defined by interderivability un-
der U. It is straightforward to show that the reduc-
tion relation defined by (16) exhibits two impor-
tant properties: (i) *strong normalisation*[10], with the

---

[10] To prove strong normalisation it is sufficient to give a
metric which assigns each term a finite non-negative integer
score, and under which every contraction reduces the score
for a term by a positive integer amount. The following metric
suffices: (a) X$'$ = 1 if X is atomic, (b) (X/Y)$'$ = X$'$ + Y$'$,
(c) (X\Y)$'$ = 2(X$'$ + Y$'$).

consequence that every type has a NF, and (ii) the *Church-Rosser property*, from which it follows that NFs are unique. In (18), a *constructive* notion of NF is specified. It is easily shown that this constructive definition identifies the same types to be NFs as the reductive definition.[11]

(17)  $x/y\backslash z \ \triangleright_1 \ x\backslash z/y$

(18)  $x\backslash y_1..y_i/y_{i+1}..y_n$

where $n \geq 0$, x is a basic type and each $y_j$ $(1 \leq j \leq n)$ is in turn of this general form.

(19)  $\phi: x/u_1..u_n + y \Rightarrow z \implies$
$\qquad L\langle n\rangle\phi: x\backslash w/u_1..u_n + y \Rightarrow z\backslash w$

(where $L\langle n\rangle = \lambda g\lambda a\lambda b\lambda c.g(\lambda v_1..v_n.av_1..v_nc)b$)

We next consider normalisation for binary combinations. For this purpose, we require a modified version of the binary calculus, called B′, having the rules $\{L\langle n\rangle,R,f,b\}$), where $L\langle n\rangle$ is a 'generalised' variant of the metarule L, shown in (19) (where the notation $x/u_1..u_n$ is schematic for a function seeking n forward directional arguments, e.g. so that for $n = 3$ we have $x/u_1..u_n = x/u_1/u_2/u_3$). Note that the case $L\langle 0\rangle$ is equivalent to L.

We will show that for every binary combination $X + Y \Rightarrow Z$ under the DC, there is a corresponding combination $X' + Y' \Rightarrow Z'$ under B′, where X′, Y′ and Z′ are the NFs of X, Y and Z. To demonstrate this, it is sufficient to show that for every combination under B, there is a corresponding B′ combination of the NFs of the types (i.e. since for binary combinations under the DC, of the form in (16), the types occurring at the top and bottom of any sequence of unary transformations will have the *same* NF).

The following contraction rules define a NF for combinations under B′ (which includes the combinations of B as a subset — provided that each use of L is relabelled as $L\langle 0\rangle$):

(20)  *IF* $w \triangleright_1 w'$ *THEN*

a. f: $w/y + y \Rightarrow w \ \triangleright_1$ f: $w'/y + y \Rightarrow w'$
b. f: $y/w + w \Rightarrow y \ \triangleright_1$ f: $y/w' + w' \Rightarrow y$
c. b: $y + w\backslash y \Rightarrow w \ \triangleright_1$ b: $y + w'\backslash y \Rightarrow w'$
d. b: $w + y\backslash w \Rightarrow y \ \triangleright_1$ b: $w' + y\backslash w' \Rightarrow y$
e. $L\langle i\rangle\phi: x\backslash w/u_1..u_i + y \Rightarrow z\backslash w \ \triangleright_1$
$\qquad L\langle i\rangle\phi: x\backslash w'/u_1..u_i + y \Rightarrow z\backslash w'$
f. $R\phi: x + y/w \Rightarrow z/w \ \triangleright_1$
$\qquad R\phi: x + y/w' \Rightarrow z/w'$

---

[11] This NF is based on an *arbitrary* bias in the restructuring of types, i.e. ordering backward directional arguments after forward directional arguments. The opposite bias (i.e. forward arguments after backward arguments) could as well have been chosen.

(21)  $L\langle i\rangle R\phi: x\backslash w/u_1..u_i + y/v \Rightarrow z/v\backslash w \ \triangleright_1$
$\qquad RL\langle i\rangle\phi: x\backslash w/u_1..u_i + y/v \Rightarrow z\backslash w/v$

(22)  $L\langle o\rangle f: x/w\backslash v + w \Rightarrow x\backslash v \ \triangleright_1$
$\qquad f: x\backslash v/w + w \Rightarrow x\backslash v$

(23)  $L\langle i\rangle f: x\backslash w/u_1..u_i + u_i \Rightarrow x/u_1..u_{i-1}\backslash w \ \triangleright_1$
$\qquad f: x\backslash w/u_1..u_i + u_i \Rightarrow x\backslash w/u_1..u_{i-1}$
for $i > 0$.

(24)  b: $z + x/y\backslash z \Rightarrow x/y \ \triangleright_1$
$\qquad Rb: z + x\backslash z/y \Rightarrow x/y$

(25)  $L\langle i\rangle\phi: x/v\backslash w/u_1..u_i + y \Rightarrow z\backslash w \ \triangleright_1$
$\qquad L\langle i+1\rangle\phi: x\backslash w/v/u_1..u_i + y \Rightarrow z\backslash w$

(26)  *IF* $\phi: x + y \Rightarrow z \ \triangleright_1 \ \phi': x' + y' \Rightarrow z'$
*THEN*  $R\phi: x + y/w \Rightarrow z/w \ \triangleright_1$
$\qquad R\phi': x' + y'/w \Rightarrow z'/w$

(27)  *IF* $\phi: x/u_1..u_i + y \Rightarrow z \ \triangleright_1$
$\qquad \phi': x'/u_1'..u_i' + y' \Rightarrow z'$
*THEN* $L\langle i\rangle\phi: x\backslash w/u_1..u_i + y \Rightarrow z \ \triangleright_1$
$\qquad L\langle i\rangle\phi': x'\backslash w/u_1'..u_i' + y' \Rightarrow z'$

These rules also transform the *types* involved into their NFs. In the cases in (20), a contraction is made without affecting the identity of the particular rule used to combine the types. In (21–25), the transformations made on types requires that some change be made to the rule used to combine them. The rules (26) and (27) recursively define new contractions in terms of the basic ones.

This reduction system can be shown to exhibit strong normalisation, and it is straightforward to argue that each combination must have a unique NF. This definition of NF accords with the constructive definition (28). (Note that the notation $R^n$ represents a sequence of n Rs, which are to be bracketed right-associatively with the following rule, e.g. so that $R^2f = (R(Rf))$, and that i takes the same value for each $L\langle i\rangle$ in the sequence $L\langle i\rangle^m$.)

(28)  $\phi: x + y \Rightarrow z$

where x, y, z are NF types, and $\phi$ is $(R^nf)$ or $(R^nL\langle i\rangle^mb)$, for $n, m \geq 0$.

Each proof of some combination $x_1,..,x_n \Rightarrow x_0$ under the DC can be seen to consist of a number of binary 'subtrees', each of the form (16). If we substitute each binary subtree with its NF combination in B′, this gives a proof of $x_1',..,x_n' \Rightarrow x_0'$ (where each $x_i'$ is the NF of $x_i$). Hence, for every DC proof, there is a corresponding proof of the combination of the NFs of the same types under B′.

Even if we consider only proofs involving NF combinations in B′, we observe spurious ambiguity of the kind familiar from CCG and MCG. Again, we can deal with this problem by defining NFs for such

proofs. Since we are interested in incremental processing, our method for identifying NF proofs is based on favouring left-branching structures.

Let us consider the patterns of functional dependency that are possible amongst sequences of three types. These are shown in (29).[12] Of these cases, some (i.e. (a) and (f)) can only be derived with a left-branching proof under B' (or the DC), and others (i.e. (b) and (e)) can only be derived with a right-branching proof. Combinations of the patterns (c),(d) and (g) commonly allow both right and left-branching derivations (though not in all cases).



(29)  (a)  (b)  (c)  (d)  (e)  (f)  (g)
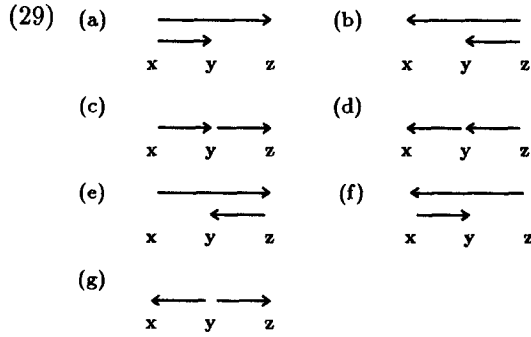
(30)  $(R^n f)$: $x/y + y/u_1..u_n \Rightarrow x/u_1..u_n$

(31)  $(R^n L\langle i \rangle^m b)$:
$x\backslash w_1..w_m/u_1..u_i + y\backslash (x/u_1..u_i)/v_1..v_n$
$\Rightarrow y\backslash w_1..w_m/v_1..v_n$

NF binary combinations of the pattern in (28) take the two more specific forms in (30) and (31). Knowing this, we can easily sketch out the schematic form of the three element combinations corresponding to (29c,d,g) which have equivalent left and right branching proofs, as shown in Figure 1.

We can define a NF for proofs under B' (that use only NF combinations) by stating three contraction rules, one for each of the three cases in Figure 1, where each rule rewrites the right branching three-leaf subproof as the equivalent left branching subproof. This will identify the optimally left branching member of each equivalence class of proofs as its NF exemplar. Again, it is easily shown that reduction under these rules exhibits strong normalisation and the Church-Rosser property, so that every proof must have a unique normal form. However, it is not so easy to prove the stronger claim that there is only a single NF proof that assigns *each distinct reading* for any combination.[13] We shall not attempt

to demonstrate this property, although we believe that it holds. We can identify the redexes of these three contraction rules purely in terms of the rules used to combine types, i.e. without needing to examine the schematic form of the types, since the rules themselves identify the relevant structure of the types. In fact, the right-branching subproofs for cases (29c,g) collapse to the single schematic redex (32), and that for (29d) simplifies to the schematic redex (33). (Note that the notation $\phi_\pi$ is used to represent any (NF) rule which is recursively defined on a second rule $\pi$, e.g. so that $\pi_b$ is any NF rule defined on b.)

(32)  $x$   $y$   $z$
$\dfrac{\overline{\hphantom{xx}}R^m f}{\dfrac{w}{\dfrac{\hphantom{xx}}{v}R^n \phi}}$   where $n \geq m$

(33)  $x$   $y$   $z$
$\dfrac{\overline{\hphantom{xx}}\phi(L\langle i \rangle b)}{\dfrac{w}{\dfrac{\hphantom{xx}}{v}\pi_b}}$   where $n \geq 1$

Let us consider the use of this system for parsing. In seeking combinations of some sequence of types, we first begin by transforming the types into their NFs.[14] Then, we can search for proofs using only the NF binary combinations. Any proof that is found to contain a proof redexes is discontinued, so that only NF proofs are returned, avoiding the problems of spurious ambiguity. Any result types assigned by such proofs stand as NF exemplars for the set of non-NF types that could be derived from the original input types under the DC. We may want to know if some input types can combine to give a *specific* result type x. This will be the case if the parser returns the NF of x.

Regarding incremental processing, we have seen that the DC is well-suited to this task in terms of allowing combinations that may usefully contribute to a knowledge of the semantic relations amongst the phrases combined, and that the NF proofs we have defined (and which the parser will construct) are optimally left-branching to the limit set by the calculus. Hence, in left-to-right analysis of sentences, the parser will be able to combine the presented material to the maximal extent that doing so usefully contributes to incremental interpretation and the filtering of semantically implausible analyses.

---

[12]Note that various other conceivable patterns of dependency do not need to be considered here since they do not correspond to any Lambek-valid combination.

[13]This holds if the contraction relation generates an equiv-

alence relation that equates any two proofs iff these assign extensionally equivalent readings.

[14]The complexity of this transformation is constant in the complexity of the type.

## Case (28c):

(a)
$$\frac{\dfrac{x/y \quad y/w_1..w_n}{x/w_1..w_n}R^n f \qquad w_n/v_1..v_m}{x/w_1..w_{n-1}/v_1..v_m}R^m f$$

(b)
$$\frac{x/y \qquad \dfrac{y/w_1..w_n \qquad w_n/v_1..v_m}{y/w_1..w_{n-1}/v_1..v_m}R^m f}{x/w_1..w_{n-1}/v_1..v_m}R^{m+n-1} f$$

## Case (28d):

(a)
$$\frac{w_n\backslash q_1..q_k/u_1..u_j \qquad \dfrac{y\backslash w_1..w_{n-1}\backslash(w_n/u_1..u_j)/v_1..v_i \qquad x\backslash(y/v_1..v_i)/t_1..t_m}{x\backslash w_1..w_{n-1}\backslash(w_n/u_1..u_j)/t_1..t_m}R^m L\langle i\rangle^n b}{x\backslash w_1..w_{n-1}\backslash q_1..q_k/t_1..t_m}R^m L\langle j\rangle^k b$$

(b)
$$\frac{\dfrac{w_n\backslash q_1..q_k/u_1..u_j \qquad y\backslash w_1..w_{n-1}\backslash(w_n/u_1..u_j)/v_1..v_i}{y\backslash w_1..w_{n-1}\backslash q_1..q_k/v_1..v_i}R^i L\langle j\rangle^k b \qquad x\backslash(y/v_1..v_i)/t_1..t_m}{x\backslash w_1..w_{n-1}\backslash q_1..q_k/t_1..t_m}R^m L\langle i\rangle^{k+n-1} b$$

## Case (28g):

(a)
$$\frac{\dfrac{y\backslash w_1..w_j/u_1..u_i \qquad x\backslash(y/u_1..u_i)/v_1..v_m}{x\backslash w_1..w_j/v_1..v_m}R^m L\langle i\rangle^j b \qquad v_m/q_1..q_n}{x\backslash w_1..w_j/v_1..v_{m-1}/q_1..q_n}R^n f$$

(b)
$$\frac{\dfrac{y\backslash w_1..w_j/u_1..u_i \qquad x\backslash(y/u_1..u_i)/v_1..v_m \qquad v_m/q_1..q_n}{x\backslash(y/u_1..u_i)/v_1..v_{m-1}/q_1..q_n}R^n f}{x\backslash w_1..w_j/v_1..v_{m-1}/q_1..q_n}R^{m+n-1} L\langle i\rangle^j b$$

Figure 1: Equivalent left and right-branching three-leaf subproofs

# References

Ades, A.E. and Steedman, M.J. 1982. 'On the order of words.' *Linguistics and Philosophy*, 4.

Barry, G. *forthcoming:1991*. Ph.D. dissertation, Centre for Cognitive Science, University of Edinburgh.

Barry, G., Hepple, M., Leslie, N. and Morrill, G. 1991. 'Proof figures and structural operators for categorial grammar'. In *EACL-5*, Berlin.

Barry, G. and Morrill, G. 1990. (Eds). *Studies in Categorial Grammar*. Edinburgh Working Papers in Cognitive Science, Volume 5. Centre for Cognitive Science, University of Edinburgh.

Barry, G. and Pickering, M. 1990. 'Dependency and Constituency in Categorial Grammar.' In Barry, G. and Morrill, G. 1990.

Dowty, D. 1988. 'Type raising, function composition, and non-constituent conjunction.' In Oehrle, R., Bach, E. and Wheeler, D. (Eds), *Categorial Grammars and Natural Language Structures*, D. Reidel, Dordrecht.

Hepple, M. 1990. 'Normal form theorem proving for the Lambek calculus.' In Karlgren, H. (Ed), *Proc. of COLING 1990*.

Hepple, M. 1990. *The Grammar and Processing of Order and Dependency: A Categorial Approach*. Ph.D. dissertation, Centre for Cognitive Science, University of Edinburgh.

Hepple, M. and Morrill, G. 1989. 'Parsing and derivational equivalence.' In *EACL-4*, UMIST, Manchester.

König, E. 1989, 'Parsing as natural deduction.' In *Proc. of ACL-25*, Vancouver.

Lambek, J. 1958. 'The mathematics of sentence structure.' *American Mathematical Monthly* 65.

Moortgat, M. 1989. *Categorial Investigations: Logical and Linguistic Aspects of the Lambek Calculus*, Foris, Dordrecht.

Moortgat, M. 1990. 'Unambiguous proof representations for the Lambek calculus.' In *Proc. of 7th Amsterdam Colloquium*, University of Amsterdam.

Moortgat, M. 1990. 'The logic of discontinuous type constructors.' In *Proc. of the Symposium on Discontinuous Constituency*, Institute for Language Technology and Information, University of Tilburg.

Morrill, G. 1988, *Extraction and Coordination in Phrase Structure Grammar and Categorial Grammar*. Ph.D. dissertation, Centre for Cognitive Science, University of Edinburgh.

Morrill, G. 1990. 'Grammar and Logical Types.' In *Proc. 7th Amsterdam Colloquium*, University of Amsterdam. An extended version appears in Barry, G. and Morrill, G. 1990.

Morrill, G., Leslie, N., Hepple, M. and Barry, G. 1990. 'Categorial deductions and structural operations.' In Barry, G. and Morrill, G. 1990.

Pickering, M. 1991. *Processing Dependencies*. Ph.D. dissertation, Centre for Cognitive Science, University of Edinburgh.

Steedman, Mark. 1985. 'Dependency and Coordination in the Grammar of Dutch and English.' *Language*, 61:3.

Steedman, Mark. 1987. 'Combinatory Grammars and Parasitic Gaps.' *NLLT*, 5:3.

Steedman, M.J. 1989. 'Grammar, interpretation and processing from the lexicon.' In Marslen-Wilson, W. (Ed), *Lexical Representation and Process*, MIT Press, Cambridge, MA.

Szabolcsi, A. 1987 'On Combinatory Categorial grammar.' In *Proc. of the Symposium on Logic and Language, Debrecen*, Akadémiai Kiadó, Budapest.

Zielonka, W. 1981. 'Axiomatizability of Ajdukiewicz-Lambek Calculus by Means of Cancellation Schemes.' *Zeitschr. f. math. Logik und Grundlagen d. Math.* 27.