Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

NARROWING GRAMMAR: A COMPARISON WITH OTHER LOGIC GRAMMARS

H. Lewis Chau

April 1989
CSD-890019

Narrowing Grammar : A Comparison with Other Logic Grammars

H. Lewis Chau

Computer Science Department
University of California
Los Angeles, CA 90024-1596
chau@cs.ucla.edu

ABSTRACT

In an earlier paper, we introduced *Narrowing Grammar (NG)*. NG is a new kind of grammar, which combines concepts from logic programming, rewriting, lazy evaluation and logic grammar formalisms such as Definite Clause Grammar. In this paper, we investigate the versatility of NG for language analysis by applying it in several examples.

We compare NG with several established logic grammars: Definite Clause Grammar, Metamorphosis Grammar, Extraposition Grammar and Gapping Grammar. We show how some of these logic grammars can be translated to NG. We also contrast NG with these kinds of grammar, and illustrate how NG permits higher-order specification, modular composition and lazy evaluation.

Keywords : Logic Grammars, Definite Clause Grammar.

This work was done within the Tangram project, supported by DARPA contract F29601-87-C-0072.

1. Introduction

A logic grammar has rules that can be represented as Horn clauses, and thus implemented by logic programming languages such as Prolog. These logic grammar rules are translated into Prolog clauses which can then be executed for either acceptance or generation of the language specified.

Since the development of metamorphosis grammar [5], the first logic grammar formalism, several variants of logic grammars have been proposed [1, 7, 10, 11, 15, 16, 18]. Among these we must mention *Definite Clause Grammar (DCG)*, the most popular approach to parsing in Prolog. DCG is a special case of Colmerauer's metamorphosis grammar. It extends context-free grammar in basically three ways:

(1) DCG provides context dependency.

(2) DCG allows arbitrary tree structures to be built in the process of parsing.

(3) DCG allows extra conditions to be added to the rules, permitting auxiliary computations.

Consider the following example taken from [15].

```
noun(N) --> [W], {rootform(W,N), is_noun(N)}.
```

`noun(N)` is a nonterminal and `[W]` is a terminal. It can be read as "a phrase identified as the noun `N` may consist of the single word `W`, where `N` is the root form of `W` and `N` is a noun".

DCG syntax is little more than "syntactically sugared" Prolog syntax. There is a simple procedure for compiling each DCG rule to a Prolog clause. The basic idea is to add a difference-list argument to each nonterminal symbol which gives the input and output streams [17, 19]. Below is an example of some DCG rules and their compilation to Prolog:

```
a --> b, c.          a(S0,S) :- b(S0,S1), c(S1,S).
a --> [W].           a(S0,S) :- connects(S0,W,S).
a --> b, {d}.        s(S0,S) :- b(S0,S), call(d).
```

The definition of `connects/2` is

```
connects([H|T],H,T).
```

Metamorphosis Grammar (MG) generalizes DCG by permitting rules of the form

LHS, T --> RHS

where *LHS* is a nonterminal and *T* is one or more terminals. For instance, the MG rule

```
be(present), [not] --> [isnt].
```

is translated to Prolog as follows:

```
be(present,S0,S) :- connects(S0,isnt,S1), connects(S,not,S1).
```

In spite of the power of DCG or MG, neither is convenient for the definition of certain constructions in natural language such as "filler-gap" constructions in which a constituent seems to have been moved from another position in the sentence. *Extraposition Grammar (XG)* [16], a

generalization of MG, allows everything found in DCG and allows, in addition, rules like the following:

```
LHS ... T --> RHS.
```

where *LHS* is a nonterminal and *T* is any finite sequence of terminals or nonterminals. The above MG rule can be read as " *LHS* can be expanded to *RHS* if *T* appears later in the input stream". This allows a very natural treatment of certain "filler-gap" constructions.

Gapping Grammar (GG) [7] can be viewed as a generalization of XG. It permits one to indicate where intermediate, unspecified substrings can be skipped, left unanalyzed during one part of the parse, and possibly reordered by the rule's application for later analysis by other rules. Consider the example which is taken from [7]:

```
a, gap(X), b, gap(Y), c --> gap(Y), c, b, gap(X).
```

For example, this rule can be applied successfully to either of the strings `a,e,f,b,d,c` and `a,b,d,e,f,c,` and application of the rule yields `d,c,b,e,f` and `d,e,f,c,b` respectively.

The logic grammar formalisms mentioned above are typically *first-order,* in the sense that a nonterminal symbol in these formalisms cannot be passed as an argument to some other nonterminal symbol. For example, usually Definite Clause Grammar (DCG) does not permit direct specification of grammar rules of the form

```
goal(X) --> X.
```

This problem was pointed out as early as [12]. We will discuss later in the paper (section 3.2 and 3.4) why this is more than just a minor problem, as it affects the convenience of use, extensibility, and modularity of grammars. We proposed a higher-order solution to this problem in [4].

The logic grammars mentioned above are also not *lazy*. Lazy evaluation can be very important in parsing. It allows coroutining recognition of multiple patterns against a stream. We are not aware of previous work connecting lazy evaluation and logic grammars, although the connection is a natural one. We will discuss this in more detail in section 3.3.

Narrowing grammar (NG) is a formalism for writing rules. It combines concepts from logic programming, rewriting, lazy evaluation, and logic grammar formalisms such as DCG. The semantics of NG are defined by a special outermost narrowing strategy [4], which is strikingly different from existing logic grammar formalisms. In this paper, we point out a number of advantages of NG for language analysis.

As a brief introductory example, let us show how easily regular expressions can be defined with NG. The regular expression pattern *a* b* that matches sequences of zero or more copies of *a* followed by a *b* can be specified with the grammar rule

```
pattern => ([a]*, [b]).
```

where we also define the following grammar rules:

```
(X*) => [].
(X*) => X, (X*).
([],L) => L.
([X|L1],L2) => [X|(L1,L2)].
```

Here `'*'` is the postfix pattern operator defining the Kleene star pattern, and the rules for `','` define pattern concatenation, very much like the usual Prolog rules for **append**.

NG rules are used much like the rewrite rules in [13], but with a special outermost narrowing strategy described in the next section. With this strategy, the narrowing of `([a]*,[b])` to `[a,b]` along with the rules used in each step of the narrowing is as follows:

	Rewritten term	*Rule used in rewriting*
`([a]*,[b])`		
→	`(([a],[a]*),[b])`	`(X*) => X, (X*).`
→	`([a|([],[a]*)],[b])`	`([X|L1],L2) => [X|(L1,L2)].`
→	`[a|(([],[a]*),[b])]`	`([X|L1],L2) => [X|(L1,L2)].`
→	`[a|([a]*,[b])]`	`([],L) => L.`
→	`[a|([],[b])]`	`(X*) => [].`
→	`[a,b]`	`([],L) => L.`

In a similar way, all other lists matching the pattern *a* b* could be produced.

There has always been a lot of interest in extending SLD-resolution theorem proving to theories with equality. As pointed out in [3], the problem with equality is the complexity of the refutation procedure. It seems that some constraints on equality are needed to achieve the computational properties of logic programs. An important approach to implement equality in logic programming has been proposed [20]. However, this approach is restricted to terminating theories. Log(F), an recent approach for combining logic programming, rewriting, and lazy evaluation has been suggested [13]. Log(F) can be used to do lazy functional programming in logic. However, unification is not fully exploited in this system because reduction (a special case of narrowing) has been used in this system.

A new procedural semantic of logic grammar rests on lazy narrowing has been proposed in [4]. This paper continues the work presented in [4]. Section 2 summarizes the formal definition of NG and section 3 describes some of its interesting features. Section 4 then goes on to compare NG with some existing logic grammars, showing how some of these logic grammars can be translated to NG. We also demonstrate the versatility of NG for language analysis by illustrating its use in several examples. Due to space limitations, we do not discuss implementation. For more information, see [4].

2. Narrowing Grammar

We quickly summarize the formal definition of narrowing grammar here.

Definition 2.1

A *term* is either a variable, or an expression of the form $f(t_1,..,t_n)$ where f is a n-ary function symbol, $n \geq 0$, and each t_i is a term. A *ground term* is a term with no variables.

Definition 2.2

A *narrowing grammar* is a finite set of rules of the form:

```
LHS => RHS
```

where:

(1) `LHS` is any term except a variable, and `RHS` is a term.

(2) If `LHS` = $f(t_1,...,t_n)$, then each t_i is a term in normal form (see definition 2.4 below).

Definition 2.3

Constructor symbols are functors (function symbols) that do not appear as any rule's outermost `LHS` functor.

A *simplified term* is a term whose outermost function symbol is a constructor symbol. By convention also, every variable is taken to be a simplified term. Note that no `LHS` of any rule can be a simplified term. In this paper we will assume the function symbols for lists (namely, the empty list `[]` and cons `[_|_]`, following Prolog syntax) are constructor symbols. Much in the way that constructors provide a notion of 'values' in a rewrite system, constructors here provide a notion of 'terminal symbols' of a grammar.

Definition 2.4

A term is said to be in *normal form* if all of its subterms are simplified. Since every variable is taken to be a simplified term, a term in normal form can be non-ground.

Definition 2.5: NU-step

$p \rightarrow q$, or p narrows to q in a *NU-step* †, is defined concisely by the following clauses:

---

†The significance of the prefix '*NU-*' in '*NU-step*' comes from the fact that we use a special strategy to select a subterm for narrowing, and this strategy selects terms in a leftmost outermost, or *N*ormal order, fashion. *U*nification is implicitly used left-to-right by this strategy.

```
nu_step(P,Q) ← nonvar(P), (P => Q).
nu_step(P,Q) ← nonvar(P), ¬(P => Q), functor(P,F,N),
               functor(Q,F,N), subterm_nu_step(P,Q,1,N).

subterm_nu_step(P,Q,I,N) ← arg(I,P,A), arg(I,Q,A),
                           plus(I,1,I1), subterm_nu_step(P,Q,I1,N).
subterm_nu_step(P,Q,I,N) ← arg(I,P,A), arg(I,Q,B),
                           nu_step(A,B), unify_remaining(P,Q,I,N).

unify_remaining(_,_,N,N).
unify_remaining(P,Q,I,N) ← plus(I,1,I1), arg(I1,P,A),
                           arg(I1,Q,A), unify_remaining(P,Q,I1,N).
```

Definition 2.6: NU-narrowing

A *NU-narrowing* ($\xrightarrow{*}$) is a reflexive transitive closure of NU-step ($\rightarrow$).

Definition 2.7

A *stream* is a list of ground terms. A *stream pattern* is a term that has a NU-narrowing to a stream.

For the rest of the paper, unless explicitly stated otherwise, by a narrowing is meant a NU-narrowing.

3. A Guided Tour of Narrowing Grammar

In this section, we illustrate the use of NG as pattern generators as well as acceptors, and summarize several important features of NG. We also point out some limitations of first-order logic grammars.

3.1. Narrowing Grammar as Pattern Generators

We first define several NG rules and illustrate how stream patterns can be generated:

```
precedes(X,Y) => eventually(X), eventually(Y).

eventually(X) => X.
eventually(X) => [_], eventually(X).

([X|Xs] // [X|Ys]) => [X|Xs//Ys].
([] // []) => [].
```

Consider the derivation showing how the stream `[a,b,c]` can be generated from interleaving of the narrowing of the two patterns `precedes([a],[c])` and `precedes([b],[c])`:

```
precedes([a],[c]) // precedes([b],[c])
  → (eventually([a]), eventually([c])) // precedes([b],[c])
  → ([a], eventually([c])) // precedes([b],[c])
  → [a|eventually([c])] // precedes([b],[c])
  → [a|eventually([c])] // (eventually([b]), eventually([c]))
  → [a|eventually([c])] // (([_], eventually([b])), eventually([c]))
  → [a|eventually([c])] // ([_|eventually([b])], eventually([c]))
  → [a|eventually([c])] // [_|(eventually([b]), eventually([c]))]
  → [a|(eventually([c]) // (eventually([b]), eventually([c])))]
  → [a|(([_], eventually([c])) // (eventually([b]), eventually([c])))]
  → [a|([_|eventually([c])] // (eventually([b]), eventually([c])))]
  → [a|([_|eventually([c])] // ([b], eventually([c])))]
  → [a|([_|eventually([c])] // [b|eventually([c])])]
  → [a,b|(eventually([c]) // eventually([c]))]
  → [a,b|([c] // [c])]
  → [a,b,c|([] // [])]
  → [a,b,c]
```

Similarly the stream `[b,a,c]` can also be generated.

The operator `//` takes two patterns as arguments, narrows them to `[X|Xs]` and `[X|Ys]` respectively and then yields `[X|Xs//Ys]`. Thus `//` is a coroutined pattern matching primitive that requires both argument patterns to match inputs of the same length. As a result, multiple overlapping patterns in a stream can be simultaneously recognized easily with `//`.

3.2. Narrowing Grammar as Pattern Acceptors

Our approach for pattern acceptance is to introduce a new pair of narrowing grammar rules specifying pattern matching. The entire definition is the following pair of rules for `match`:

```
match([],S) => S.
match([X|L],[X|S]) => match(L,S).
```

`match` can take a pattern as its first argument, and an input stream as its second argument. If the pattern narrows to the empty list `[]`, `match` simply succeeds. On the other hand, if the pattern narrows to `[X|L]`, then the second argument to `match` must also narrow to `[X|S]`. Intuitively, `match` can be thought of as *applying* a pattern (the first argument) to an input stream (the second argument), in an attempt to find a prefix of the stream that the grammar defining the pattern can generate.

Pattern acceptance is requested explicitly with `match`. As a simple example, consider the derivation illustrating how `match` works:

```
match(([a]*,[b]), [a,b])
    → match((([a],[a]*),[b]), [a,b])
    → match(([a|([],[a]*)],[b]), [a,b])
    → match([a|(([],[a]*),[b])], [a,b])
    → match((([],[a]*),[b]), [b])
    → match(([a]*,[b]), [b])
    → match(([],[b]), [b])
    → match([b], [b])
    → match([], [])
    → []
```

There is a certain elegance to this; the rules of the grammar by themselves act as pattern generators, but when applied with `match` they act like an acceptor.

3.3. Higher-order Specification, Extensibility, and Modularity

Narrowing grammar is higher-order. Specifically, narrowing grammar is higher-order in the sense that patterns can be passed as input arguments to patterns, and patterns can yield patterns as outputs.

For example, the enumeration pattern `(_ // _)` defined in section 3.1 is higher-order, as its arguments are patterns. The patterns `precedes([a],[c])` and `precedes([b],[c])` can be used as arguments to `//`, as in

```
precedes([a],[c]) // precedes([b],[c]).
```

It is well known that a higher-order capability increases expressiveness of a language, since it makes it possible to develop generic functions that can be combined in a multitude of ways [8]. As a consequence, narrowing grammar rules are highly reusable and can be usefully collected in a library.

Narrowing grammar supports modular composition of patterns. For example, the enumeration pattern `(_ // _)` is composed of patterns `precedes(_,_)`. Narrowing grammar is also extensible, since it permits definition of new grammatical constructs, as the `//` example showed earlier.

3.4. Lazy Evaluation, Stream Processing, and Coroutining

Lazy evaluation is intimately related with a programming paradigm referred to as *stream processing* [14]. Note that in this paper, a stream pattern is a term that will yield a list of ground terms under lazy evaluation. We are not aware of previous work connecting stream processing and grammars, although the connection is a natural one. Lazy evaluation and stream processing also have intimate connections with *coroutining* [9]. Coroutining is the interleaving of evaluation (here, narrowing) of two expressions. It is applied frequently in stream processing. For example, narrowing of the stream pattern

```
match( ([a]*,[b]), [a,b] )
```

interleaves the narrowing of `([a]*,[b])` with the narrowing of `match(_,[a,b])`. The sample narrowing of this pattern in section 3.2 shows the actual interleaving – first `([a]*,[b])` is narrowed for three NU-steps, then `match(_,[a,a,b])` for one NU-step, then `([a]*,[b])` for three NU-steps, then `match(_,[b])` for one NU-step, and finally `match(_,[])` for one NU-step. The effect of leftmost outermost narrowing of the combined stream pattern is precisely to interleave these two narrowings. Similarly narrowing of the pattern

```
([a]*,[b])
```

interleaves the narrowing of `[a]*` with the narrowing of `(_,[b])`.

A specific advantage of lazy evaluation in parsing, then, is that coroutined recognition of multiple patterns in a stream becomes accessible to the grammar writer. The coroutining rules

```
([X|Xs] // [X|Ys]) => [X|Xs//Ys].
([] // []) => [].
```

make explicitly coroutined pattern matching possible. Essentially `//` narrows each of its arguments, obtaining respectively `[X|Xs]` and `[X|Ys]`. Having obtained simplified terms, it suspends narrowing of `Xs` and `Ys` until further evaluation is necessary. An immediate advantage of lazy evaluation here is reduced computation. Without lazy evaluation, both arguments would have to be completely simplified before pattern matching took place; failure to unify the heads of these completely evaluated arguments would then mean that many unnecessary narrowing steps on the tails of the arguments had been performed.

3.5. Limitations of First-order Logic Grammars

The usual method for compiling DCG to Prolog does not permit direct specification of grammar rules of the form:

```
goal(X) --> X.
```

where `X` is a variable. Therefore, it is hard to pass nonterminals (patterns) to DCG rules that behave like `//` and `number` given earlier. This is a basic limitation of first-order logic grammars.

This problem was pointed out in [12], where Moss proposed a special translation technique by using a single predicate name for non-terminals. For instance, Moss translates the above DCG rule to the Prolog clause

```
nonterminal(goal(X),S0,S) :- nonterminal(X,S0,S).
```

Some Prolog systems, including Quintus Prolog and Sicstus Prolog, have been extended to permit such rules. In these systems the rule shown above is translated to

```
goal(X,S0,S) :- phrase(X,S0,S).
```

where `phrase/3` is a metapredicate that performs DCG compilation of its first argument (at run

time) and then executes the result. Similar suggestion has also been made by Abramson [2].

There is one final limitation. Even with the techniques suggested above, the lazy evaluation or coroutining aspect of NG is not easily attained with first-order logic grammars. To attain them, a new evaluation strategy such as the one in section 2 is needed.

4. Comparison with Previous Work

In this section we compare NG with other logic grammar formalisms in the literature. Most existing logic grammars are logic-based in the sense that they can be represented as Horn clauses. For example, each DCG rule is straightforwardly translated to a Prolog clause by adding a different-list to each nonterminal symbol giving the input and output streams. Each nonterminal is a Prolog predicate with Prolog terms as its arguments. Therefore, it is hard to write DCG rules that permit arbitrary patterns to be passed as arguments†. This restricts the expressive power and modularity of the grammar. In the following subsections, we show how some logic grammars can be easily translated to NG. We also demonstrate the versatility of NG for language analysis by illustrating its use in several examples.

4.1. Narrowing Grammar and Definite Clause Grammar

We show how a pure DCG rule can be translated into a NG rule in which both describing the same language.

(1) Essentially, DCG rules can be translated to NG rules by changing all occurrences of `-->` to `=>` and by including NG definition for `','`.

```
([],L) => L.
([X|L1],L2) => [X|(L1,L2)].
```

(2) `';'` of DCG have the same semantics as of NG but the latter is defined at the grammatical level.

```
(X;Y) => X.
(X;Y) => Y.
```

4.2. Narrowing Grammar and Metamorphosis Grammar

MG [5] permits rules of the form

LHS, T --> RHS

where *LHS* is a nonterminal and *T* is one or more terminals. We can capture the semantics of this rule in narrowing grammar by defining a constructor `replace(X,Y)` and one more rule for `match` as follows:

† Some proposals [2, 12] define meta rules that improve the expressiveness of DCG to permit such arguments, however.

```
match([replace(X,Y)|L],S) => match(L,(Y,match(X,S))).
```

and transform the metamorphosis grammar rule to

```
LHS => [replace(RHS,T)].
```

Note, however, that with NG `T` can be *any* pattern, not just a stream of terminals.

Example 4.1

Consider the following MG and NG rules which accept all strings of `[a]'s` and `[b]'s` which have an equal number of `[a]'s` and `[b]'s`.

```
ns --> [].                    ns => [].
ns --> na, ns, nb.            ns => na, ns, nb.
na --> [a].                   na => [a].
na,[term(nb)] --> nb, na.     na => [replace((nb,na),nb)].
nb --> [b].                   nb => [b].
nb --> [term(nb)].
```

The `[term(nb)]` 'terminal' permits the MG to treat the nonterminal `nb` temporarily as a terminal. However, the grammar will recognize/generate a superset of the original language. Note that this artifice is not needed with NG.

A derivation showing how `match` works with the constructor symbol `[replace(_,_)]` is shown:

```
match(ns, [b,a])
        → match((na,ns,nb), [b,a])
        → match(([replace((nb,na),nb)],ns,nb), [b,a])
        → match((ns,nb), (nb,match((nb,na), [b,a])))
        → match(([],nb), (nb,match((nb,na), [b,a])))
        → match(nb, (nb,match((nb,na), [b,a])))
        → match([b], (nb,match((nb,na), [b,a])))
        → match([b], ([b],match((nb,na), [b,a])))
        → match([b], [b|match((nb,na), [b,a])])
        → match([], match((nb,na), [b,a]))
        → match((nb,na), [b,a])
        → match(([b],na), [b,a])
        → match([b|([],na)], [b,a])
        → match(([],na), [a])
        → match(na, [a])
        → match([a], [a])
        → match([], [])
        → []
```

4.3. Narrowing Grammar, Extraposition Grammar and Left Extraposition

We can extend the method in section 4.2 for XG rules of the form

```
LHS ... T --> RHS.
```

Here *LHS* is a nonterminal symbol and *T* is any finite sequence of terminals or terminals.

We can capture the semantics of this XG rule by defining a constructor symbol `replace(X,Y)` and one more rule for `match` as follows:

```
match([replace(RHS,T)|L],S) => match(L,insert(T,match(RHS,S))).
insert(X,Y) => (X, Y).
insert(X,[Y|Z]) => [Y|insert(X,Z)].
```

and transform the XG rule to

```
LHS => [replace(RHS,T)].
```

Note the use of `insert(_,_)` here as compared to the use of concatenation in MG. Due to ... in the *LHS* of an XG rule, *T* is nondeterministically inserted into the remaining input stream (not necessary into the head) after *RHS* has been recognized.

Example 4.2 : Left Extraposition

Pereira pointed out that relative clauses can be handled with rules like the following:

```
s --> np, vp.
np --> det, n, optional_relative.
np --> [trace].
vp --> v.
vp --> v, np.
optional_relative --> [].
optional_relative --> relative.
relative --> rel_marker, s.
rel_marker ... [trace] --> rel_pro.
rel_pro --> [that].
```

The left-extraposition XG rule

```
rel_marker ... [trace] --> rel_pro.
```

is transformed to

```
rel_marker => [replace(rel_pro,[trace])].
```

A derivation showing how `match` works with the constructor symbol `[replace(_,_)]` is given below:

```
match(relative, [that,john,wrote])
        → match((rel_marker,s), [that,john,wrote])
        → match(([replace(rel_pro,[trace])],s), [that,john,wrote])
        → match([replace(rel_pro,[trace])|s], [that,john,wrote])
        → match(s, insert([trace],match(rel_pro,[that,john,wrote])))
        → match((np,vp), insert([trace],match(rel_pro,[that,john,wrote])))
        → match(([john],vp), insert([trace],match(rel_pro,[that,john,wrote])))
        → match([john|vp], insert([trace],match(rel_pro,[that,john,wrote])))
        → match([john|vp], insert([trace],match([that],[that,john,wrote])))
        → match([john|vp], insert([trace],match([],[john,wrote])))
        → match([john|vp], insert([trace],[john,wrote]))
        → match([john|vp], [john|insert([trace],[wrote])])
        → match(vp, insert([trace],[wrote]))
        → match((v,np), insert([trace],[wrote]))
        → match(([wrote],np), insert([trace],[wrote]))
        → match([wrote|np], insert([trace],[wrote]))
        → match([wrote|np], [wrote|insert([trace],[])])
        → match(np, insert([trace],[]))
        → match([trace], insert([trace],[]))
        → match([trace], ([trace],[]))
        → match([trace], [trace|([],[])])
        → match([], ([], []))
        → ([], [])
        → []
```

The point to be made here is that commonly used XG rules can be transformed to NG rules easily. In example 4.6, we illustrate how NG formalism by itself describes the same language in a more natural manner.

4.4. Narrowing Grammar, Gapping Grammar and Right Extraposition

In last section, we describe how NG simulates left-extraposition of XG, here we show how NG simulates right-extraposition of GG. Consider a special class of GG rules [7] of the form

LHS, gap(X), T --> gap(X), RHS.

where *LHS* is a nonterminal symbol and *T* is a single (pseudo-) terminal†.

We can capture the semantics of this GG rule by defining a constructor symbol `replace(X,Y)` and one more rule for `match` as follows:

```
match([replace(RHS,T)|L],S) => match(gap(L,T,RHS),S).
gap([X|Xs],[X],RHS) => RHS, Xs.
gap([X|Xs],T,RHS) => [X|gap(Xs,T,RHS)].
```

† Current implementation restricts *T* to be a single terminal. In general, *T* can be any finite sequence of terminals and non-terminals.

and transform the GG rule to

```
LHS => [replace(RHS,T)].
```

Example 4.3 : Right Extraposition

Dahl [7] gave the following GG for the language $\{a^n b^n c^n \mid n \geq 0\}$:

```
s --> as, bs, cs.
as --> [].
as --> xa, [a], as.
bs --> [].
xa, gap(X), bs --> gap(X), [b], bs, xb.
cs --> [].
xb, gap(X), cs --> gap(X), [c], cs.
```

This GG uses right-extraposition in linguistic theory. Here we transform the GG to NG as described above†:

```
s => as, bs, cs.
as => [].
as => xa, [a], as.
bs => [].
bs => [term(bs)].
xa => [replace(([b],bs,xb),[term(bs)])].
cs => [].
cs => [term(cs)].
xb => [replace(([c],cs),[term(cs)])].
```

A derivation showing how `match` works with the constructor symbol `[replace(_,_)]` is given below:

† Note the use of `[term(bs)]` and `[term(cs)]` as pseudo-terminals in NG

```
match(s, [a,b,c])
  → match((as,bs,cs), [a,b,c])
  → match((xa,[a],as,bs,cs), [a,b,c])
  *→ match([replace(([b],bs,xb),[term(bs)])|([a],as,bs,cs)], [a,b,c])
  → match(gap(([a],as,bs,cs),[term(bs)],([b],bs,xb)), [a,b,c])
  *→ match([a|gap((as,bs,cs),[term(bs)],([b],bs,xb))], [a,b,c])
  → match(gap((as,bs,cs),[term(bs)],([b],bs,xb)), [b,c])
  *→ match(gap((bs,cs),[term(bs)],([b],bs,xb)), [b,c])
  *→ match(gap([term(bs)|cs],[term(bs)],([b],bs,xb)), [b,c])
  → match(([b],bs,xb,cs), [b,c])
  *→ match((bs,xb,cs), [c])
  *→ match((xb,cs), [c])
  *→ match([replace(([c],cs),[term(cs)])|cs], [c])
  → match(gap(cs,[term(cs)],([c],cs)), [c])
  → match(gap([term(cs)],[term(cs)],([c],cs)), [c])
  *→ match(([c],cs), [c])
  → match([c|cs], [c])
  → match(cs, [])
  → match([], [])
  → []
```

4.5. NG is modular

In previous sections, we describe how NG simulates other logic grammars. In this section, we argue that many examples given before can be expressed with NG in a more natural way.

Example 4.4 : Non-Context Free Languages

Consider the following NG:

```
s_abc => ab_c // a_bc.
ab_c => pair([a],[b]), [c]*.
a_bc => [a]*, pair([b],[c]).
pair(X,Y) => [].
pair(X,Y) => X, pair(X,Y), Y.
```

This NG defines the non-context-free language $\{a^n b^n c^n \mid n \geq 0\}$ using only context-free-like constructions. // imposes simultaneous constraints ($a^n b^n c^*$ and $a^* b^n c^n$) on streams generated by the grammar.

Example 4.5 : Non-Context Free Languages

In [6, 7] respectively, Dahl gave the following two GGs for the language $\{a^n b^m c^n d^m \mid m,n \geq 0\}$ and pointed out that XG cannot be used because of the XG constraint on the nesting of gaps.

```
s --> as, bs, cs, ds.
as, gap(G), cs --> [a], as, gap(G), [c], cs.
as, gap(G), cs --> gap(G).
bs, gap(G), ds --> [b], bs, gap(G), [d], ds.
bs, gap(G), ds --> gap(G).

s --> as, bs, cs, ds.
as --> [].
as, gap(X), xc --> [a], as, gap(X).
bs --> [].
bs, gap(X), xd --> [b], bs, gap(X).
cs --> [].
cs --> xc, [c], cs.
ds --> [].
ds --> xd, [d], ds.
```

Here we present a NG describing the same language.

```
abcd => a_c // b_d.
a_c => triple([a],[b],[c]), [d]*.
b_d => [a]*, triple([b],[c],[d]).
triple(X,Y,Z) => Y*.
triple(X,Y,Z) => X, triple(X,Y,Z), Z.
```

It is obvious that NG rules are modular. Similar to example 4.4, the first rule imposes simultaneous constraints $a^n b^* c^n d^*$ and $a^* b^m c^* d^m$. On the other hand, the interaction between different gapping rules in GG formalism makes it difficult for the grammar writers.

Example 4.6 : Filler-Gap Problem

In example 4.2, we show how XG is translated to NG and interpret "relative clauses" in natural language. However, it may not necessarily be efficient. In this example, we propose a higher-order solution. We define NG as follows:

```
s => np, vp.
np => [term(np)].
np => det, n, optional_relative.
vp => v.
vp => v, np.
optional_relative => [].
optional_relative => rel_pro, s without [term(np)].
rel_pro => [that].

A without [] => A.
[A|As] without [A|Bs] => As without Bs.
[A|B] without C => [A|B without C].
```

Note the definition of `without/2` handles the Filler-Gap problem. Now we illustrate how the sentence

```
the cat that likes fish runs
```

can be generated from the NG by giving a sequence of terms that can be produced by narrowing of the NG start symbol `s`:

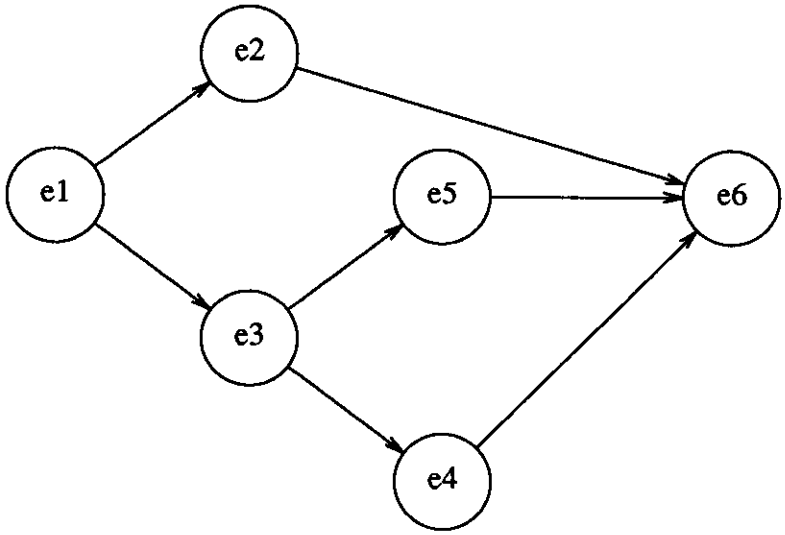
```
s
  → np, vp
  → det, n, optional_relative, vp
  → [the], n, optional_relative, vp
  → [the], [cat], optional_relative, vp
  → [the], [cat], rel_pro, s without [term(np)], vp
  → [the], [cat], [that], s without [term(np)], vp
  → [the], [cat], [that], (np, vp) without [term(np)], vp
  → [the], [cat], [that], ([term(np)], vp) without [term(np)], vp
  → [the], [cat], [that], [term(np)|vp] without [term(np)], vp
  → [the], [cat], [that], vp without [], vp
  → [the], [cat], [that], vp, vp
  → [the], [cat], [that], v, np, vp
  → [the], [cat], [that], [likes], np, vp
  → [the], [cat], [that], [likes], [fish], vp
  → [the], [cat], [that], [likes], [fish], v
  → [the], [cat], [that], [likes], [fish], [runs]
```

Example 4.7 : Lazy Narrowing and Coroutined Pattern Matching

In this final example, we consider a basic problem for testing a certain set of events that satisfies certain precedence (partial ordering) constraints. A directed acyclic graph (DAG) is well suited for representing these constraints. In a DAG whose nodes represent events, an edge X → Y represents the ordering constraint that event X occurs before event Y, which can also be specified by `precedes(X,Y)` defined in section 3.1. Now, we can directly represent an ordering of any two events by a pattern as follows:

```
... // precedes(X,Y) // ...
```

Each `precedes(X,Y)` corresponds to an edge X → Y in the DAG. For instance, suppose we are given a partial ordering of events which is represented by the following DAG:



The pattern that describes the partial ordering is

```
precedes(e1,e2) // precedes(e1,e3) // precedes(e3,e5) //
precedes(e3,e4) // precedes(e2,e6) // precedes(e5,e6) // precedes(e4,e6).
```

Here `e1,...,e6` are some event patterns.

5. Conclusion

In this paper, we have shown how NG, together with a simple definition of `match`, comprises a new formalism for language analysis. Unlike many existing logic grammars which is logic-based, NG is a functional logic grammar which combines concepts from logic programming, rewriting, lazy evaluation, and logic grammar formalisms such as DCG. The semantics of NG are defined by a special outermost lazy narrowing strategy.

NG compares favorably in expressive power and generality with other logic grammar formalisms. We have demonstrated how to translate different logic grammars into NG. We also point out some limitations of existing logic grammars. NG is modular, extensible and highly reusable, and can be usually collected in a library. We have extended the expressive power of first-order logic grammars to be higher-order, by permitting patterns to be passed as arguments to the grammar rules. At the same time, the laziness aspect of NG permits coroutine matching of multiple patterns against a stream easily.

Like many logic grammars, NG can be straightforwardly translated to Prolog and executed by existing Prolog interpreters as generators or acceptors. Although narrowing is in general difficult to implement efficiently, we proposed in [4] one relatively efficient way to implement NG. However, the performance is still unsatisfactory as compared to DCG-like Prolog code. A challenging open problem is to devise further improvements for the implementation given there. Efficient implementations are possible in many cases. For example, when we know we will use `match`, the definition

```
pattern => ([a]*, [b]).
```

can be replaced by

```
match(pattern,S) => t1(S).
t1([a|S]) => t1(S).
t1(S) => t2(S).
t2([b|S]) => S.
```

and possibly compiled even further to DCG-like Prolog code in which nonterminals are augmented with output variables, such as

```
match(pattern,S,T) :- t1(S,T).
t1([a|S],T) :- t1(S,T).
t1(S,T) :- t2(S,T).
t2([b|S],S).
```

Acknowledgement

I would like to thank Stott Parker for very helpful discussions on this paper.

References

1. Abramson, H., "Definite Clause Translation Grammars," *Proc. First Logic Programming Symposium*, pp. 233-240, IEEE Computer Society, Atlantic City, 1984.
2. Abramson, H., "Metarules and an Approach to Conjunction in Definite Clause Translation Grammars: Some Aspects of Grammatical Metaprogramming," *Proc. Fifth International Conference and Symposium on Logic Programming*, pp. 233-248, MIT Press, 1988.
3. Bellia, M. and G. Levi, "The Relation between Logic and Functional Languages: A Survey," *J. Logic Programming*, pp. 217-236, 1986.
4. Chau, H.L. and D.S. Parker, "Narrowing Grammar," Technical Report CSD-890014, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, April 1989. To appear, *Proc. Sixth International Conference on Logic Programming, Lisbon*, June 1989.
5. Colmerauer, A., "Metamorphosis Grammars," in *Natural Language Communication with Computers, LNCS 63*, Springer, 1978.
6. Dahl, V., "More on Gapping Grammars," *Proc. Intl. Conf. on Fifth Generation Computer Systems*, Tokyo, 1984.
7. Dahl, V. and H. Abramson, "On Gapping Grammars," *Proc. Second Intl. Conf. on Logic Programming*, pp. 77-88, Uppsala, Sweden, 1984.
8. Darlington, J., A.J. Field, and H. Pull, "The Unification of Functional and Logic Languages," *Logic Programming: Functions, Relations, and Equations*, Prentice-Hall, 1986.

9. Henderson, P., *Functional Programming: Application and Implementation,* Prentice/Hall International, 1980.

10. Hirschman, L. and K. Puder, "Restriction Grammar: A Prolog Implementation ," *Logic Programming and its Applications*, 1985.

11. McCord, M.C., "Modular Logic Grammars," *Proc. 23rd Annual Meeting of the Association for Computational Linguistics*, pp. 104-117, Chicago, 1985.

12. Moss, C.D.S., "The Formal Description of Programming Languages using Predicate Logic," Ph.D. Dissertation, Imperial College, London, 1981.

13. Narain, S., "LOG(F): An Optimal Combination of Logic Programming, Rewrite Rules and Lazy Evaluation," Ph.D. Dissertation, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1988.

14. Parker, D.S., "Stream Data Analysis in Prolog," Technical Report CSD-890004, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, January 1989.

15. Pereira, F.C.N. and D.H.D. Warren, "Definite Clause Grammars for Language Analysis," *Artificial Intelligence*, vol. 13, pp. 231-278, 1980.

16. Pereira, F.C.N., "Extraposition Grammars," *American Journal for Computational Linguistics*, vol. 7, 1981.

17. Pereira, F.C.N. and S.M. Shieber, *Prolog and Natural-Language Analysis,* CSLI Stanford, 1987.

18. Stabler, E.P., "Restricting Logic Grammars with Government-Binding Theory," *Computational Linguistics*, vol. 13, no. 1-2, 1987.

19. Sterling, L. and E. Shapiro, *The Art of Prolog,* MIT Press, Cambridge, MA, 1986.

20. vanEmden, M.H. and K. Yukawa, "Logic Programming with Equations," *Journal of Logic Programming*, vol. 4, no. 4, 1987.