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

NARROWING GRAMMAR

H. Lewis Chau
D. Stott Parker

April 1989
CSD-890014

Narrowing Grammar

H. Lewis Chau

D. Stott Parker

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

ABSTRACT

We present a new kind of grammar. It combines concepts from logic programming, rewriting, lazy evaluation, and logic grammar formalisms such as Definite Clause Grammar (DCG). We call it *Narrowing Grammar.*

A narrowing grammar is a finite set of rewrite rules. It is directly executable, like most logic grammars. In fact, narrowing grammar rules can be compiled to Prolog and executed by existing Prolog interpreters as generators or acceptors. Unlike many logic grammars, narrowing grammar also permits higher-order specification and modular composition, and provides lazy evaluation. Lazy evaluation is important in certain language acceptance situations, such as in coroutined matching of multiple patterns against a stream.

This paper defines narrowing grammar and compares it with the successful and widely-used DCG formalism in logic programming. We show that pure DCG can be easily translated into narrowing grammar. Narrowing grammar enjoys the advantages of DCG, as well as its first-order logic foundation. At the same time, narrowing grammar can rank higher in aspects such as expressiveness and modularity.

Keywords: Logic Programming, Definite Clause Grammar, Logic Grammars, Parsing.

This work was done within the Tangram project, supported by DARPA contract F29601-87-C-0072.
Prolog code for the examples in this paper can be obtained by sending a message to the authors at `chau@cs.ucla.edu` or `stott@cs.ucla.edu`.
To appear in Proceedings of the Sixth International Conference on Logic Programming.

1. Introduction

Since the development of metamorphosis grammar [5], the first logic grammar formalism, several variants of logic grammars have been proposed [1, 6, 10, 11, 16, 17, 19]. Among these we must mention Definite Clause Grammar (DCG), a successful and widely-used formalism for language analysis in logic programming. We assume that the reader is familiar with DCG and Prolog. Good introductions to DCG and Prolog can be found, for example, in [18, 20].

Most logic grammar formalisms mentioned above are *first-order*. Specifically, a nonterminal symbol in these formalisms cannot be passed as an argument to some other nonterminal symbol. For example, usually 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 why this is more than just a minor problem, as it affects the convenience of use, extensibility, and modularity of grammars. Very recently Abramson [2] has commented on the problem, and has addressed it by using a new construct, *meta(X)*, to define metarules that go beyond the limit of first-order logic grammar formalisms. We propose a higher-order solution.

Narrowing grammar 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 narrowing grammar are defined by a term-rewriting system that is an extension of Narain's Log(F) system [14]. This approach gives both a compact formal definition of narrowing grammar, and an efficient logic programming implementation. In this paper, we point out a number of advantages of narrowing grammar for language analysis.

As a brief introductory example, let us show how easily regular expressions can be defined with narrowing grammar. The regular expression pattern $a^+ b$ that matches sequences of one 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, (X+).
([],L) => L.
([X|L1],L2) => [X|(L1,L2)].
```

Here `'+'` is the postfix operator defining the Kleene plus regular expression pattern, and the rules for `','` define pattern concatenation, very much like the usual Prolog rules for `append`.

---

† 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. See section 5.2 for further discussion.

Lists are used to represent sequences, as in most logic grammars.

Narrowing grammar rules are used much like rewrite rules in [14], but with a special leftmost outermost term rewriting strategy described in the next section. With this strategy, the narrowing of `([a]+,[b])` to `[a,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|([a],[b])]`	`(X+) => X.`
→	`[a,a|([],[b])]`	`([X|L1],L2) => [X|(L1,L2)].`
→	`[a,a,b]`	`([],L) => L.`

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

Narrowing grammar has a theoretical foundation in first-order logic. All pure narrowing grammar rules can be translated straightforwardly to pure Prolog clauses. This is advantageous since the full power of unification is exploited, and eventually it may be feasible for properties of a narrowing grammar to be verified using first-order logic theorem proving technology. At the same time, the higher-order specification and modularity of narrowing grammar enable some complex languages to be specified easily.

Section 2 defines narrowing grammar and section 3 shows how to implement it in Prolog. Section 4 then describes some of its interesting features, and section 5 goes on to compare narrowing grammar with the widely-used DCG formalism in logic programming, showing how it offers several important advantages.

2. Narrowing Grammar

Narrowing grammar is a clear and powerful formalism for describing languages. In this section we define narrowing grammar, and give examples showing how patterns can be specified with it.

2.1. Formalism of Narrowing Grammar

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

Let p, q be terms where p is not a variable, and let s be a nonvariable subterm of p (which we write $p = r[s]$). If there exists a rule (*LHS*=>*RHS*) (which we assume has no variables in common with p), for which there is a most general unifier θ of *LHS* and s and $q = (r[RHS])\theta$ (the result of replacing s by *RHS* and applying the substitution θ), then we say p *narrows to* q.

A *narrowing* is a sequence of terms $p_1, p_2, ..., p_n$ such that for each i, $1 \le i \le n-1$, p_i narrows to p_{i+1}. A narrowing is *successful* if p_n is simplified.

Generally speaking, a rewrite system will specify a mechanism for *selecting* a subterm s from a given term p, to determine what to narrow. This mechanism is then used successively with the actual rewriting mechanism to implement narrowing. Below we define *NU-narrowing*, a special leftmost outermost narrowing strategy for narrowing grammar.

Definition 2.6: 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).
```

We can view NU-step as a special leftmost outermost narrowing. The term p narrows to q in a NU-step if either (p => q) is an instance of some rule (first clause), or if left-to-right unification of subterms of p followed by the replacement of a subterm by the result of a NU-step yields q (second clause). Note that the NU-step definition does not permit a narrowing to begin with a variable.

We have used a logic program to define NU-step mainly out of interest in conciseness. Note that the definition is nondeterministic, since the `subterm_nu_step` predicate is nondeterministic. Nondeterminism permits NU-step to act both as an acceptor and as a generator.

Definition 2.7: NU-narrowing

A *NU-narrowing* is a narrowing $p_1, p_2, \ldots$ such that for each i, when p_i and p_{i+1} both exist, p_i narrows to p_{i+1} in a *NU-step*.

```
nu_narrowing(X,X).
nu_narrowing(X,Y) ← nu_step(X,Z), nu_narrowing(Z,Y).
```

Definition 2.8: simplification

A *simplification* is a *NU-narrowing* $p_1, p_2, \ldots, p_n$ if p_n is simplified and no other p_i is simplified.

```
simplification(X,X) ← simplified(X).
simplification(X,Y) ← ¬simplified(X),
                      nu_narrowing(X,Y), simplified(Y).
```

NU-narrowing is not just a leftmost outermost narrowing strategy. For instance, given the narrowing grammar rules:

```
a => c.
b => [].
c => c.
g(X,[]) => [].
```

then there is a NU-narrowing sequence

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

but the only leftmost outermost narrowing is

```
g(a,b), g(c,b), g(c,b), ...
```

Definition 2.9:

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

2.2. Specifying Patterns with Narrowing Grammar

We illustrate how useful patterns can be developed in narrowing grammar with a sequence of examples.

Example 2.0 : Regular Expressions

As we suggested earlier, regular expressions can be defined easily with grammar rules:

```
(X+) => X.
(X+) => X,(X+).

(X*) => [].
(X*) => X,(X*).

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

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

The operators `'+'` and `'*'` define the familiar Kleene plus and Kleene star regular expressions, respectively. `';'` is a disjunctive pattern operator, while `','` defines pattern concatenation.

Example 2.1 : Counting the Occurrences of a Pattern

Suppose we wish to count the number of times an uninterrupted sequence of one or more *a's* is followed by a *b* in a stream. This pattern can be represented by the regular expression `([a]+,[b])`, and we can count the number of its occurrences with the pattern

```
number( ([a]+,[b]), Total )
```

if we include the following grammar for `number`:

```
number(Pattern,Total) => number(Pattern,Total,0).
number(Pattern,Total,Total) => [end_of_file].
number(Pattern,Total,Count) =>
          Pattern, number(Pattern,Total,plus(Count,1)).
number(Pattern,Total,Count) =>
          [_], number(Pattern,Total,Count).
```

Here `Total` is unified with the number of occurrences of `Pattern` in a stream that is matched with the pattern `number(Pattern,Total)`, and `[end_of_file]` is a special terminal symbol that delimits the end of stream. We assume `plus(X,1)` that yields the value of `X+1` when

simplified.

From the example above it is clear that the grammar rules have a functional flavor. Stream operators are easily expressed using recursive functional programs. In addition, `number` is *higher-order* because it takes an arbitrary pattern as an argument. The definitions for `'+'`, `'*'`, `';'`, `','`, etc., above are also higher-order in that they have rules like

```
(X+) => X.
```

which rewrite terms to their arguments.

Example 2.2 : Coroutined Pattern Matching

Suppose we wish to count the occurrences of `[c]` as well as of `([a]+,[b])`. That is, we want to count the occurrences of multiple patterns in a stream. We use the pattern

```
number(([a]+,[b]),N1) // number([c],N2)
```

where we include the following grammar for `//`:

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

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 pattern matching primitive that requires both argument patterns to generate or accept streams of the same length. This example shows that multiple patterns in a stream can be simultaneously generated or accepted (i.e., coroutined) easily with `//`.

Example 2.3 : Non-Context Free Languages

Consider the following grammar†:

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

This grammar defines the non-context-free language $\{a^n b^n c^n \mid n \geq 0\}$ using only context-free-like constructions. The first rule for `s_abc` imposes simultaneous (parallel) constraints on streams generated by the grammar.

3. Compilation of Narrowing Grammar to Prolog

We describe an algorithm to compile narrowing grammars to Prolog programs. It turns out that SLD-resolution with left-to-right goal selection, the proof procedure commonly used in Prolog, will implement *NU-narrowing* on these programs. The compilation of a narrowing grammar rule into a Prolog clause combines information about the rule and the control of NU-narrowing when interpreting that rule. One interesting aspect of the compilation algorithm is its use of a

† Fernando Pereira suggested this example.

suitable 'equality' predicate `equal(X,Y)`, which succeeds when `X` can be narrowed to `Y`.

Algorithm 3.1 : Compilation of Narrowing Grammar to Prolog

(1) For each n-ary constructor symbol c, $n \geq 0$, and for distinct Prolog variables $X_1, \ldots X_n$, generate the clause:

`simplify(` $c(X_1,\ldots,X_n)$`,` $c(X_1,\ldots,X_n)$ `).`

(2) For each rule $f(L_1,\ldots,L_m)$=>RHS, let $A_1,\ldots,A_m$, `Out` be distinct Prolog variables not occurring in the rule, and generate the clause:

`simplify(`$f(A_1,\ldots,A_m)$`,Out) :-`
`equal(`A_1`,` L_1`),`
`...,`
`equal(`A_m`,` L_m`),`
`simplify(`RHS`,Out).`

Algorithm 3.1 does not deal with 'impure' features in narrowing grammar rules, such as cuts. However, it is not difficult to extend the compilation to include such features.

Definition 3.1 : `equal(X,Y)`

```
equal(X,Y) :- nu_narrow(X,Y).

nu_narrow(X,X) :- !.
nu_narrow(X,Y) :- simplify(X,Z),
                  nu_narrow_subterms(Z,Y).

nu_narrow_subterms(X,Y) :-
                     functor(X,F,N), functor(Y,F,N),
                     nu_narrow_subterms(X,Y,0,N).

nu_narrow_subterms(_,_,N,N).
nu_narrow_subterms(X,Y,I,N) :-
             plus(I,1,I1), arg(I1,X,A), arg(I1,Y,B),
             nu_narrow(A,B),
             nu_narrow_subterms(X,Y,I1,N).
```

The table below lists some narrowing grammar rules together with the Prolog clauses resulting from their compilation†.

† Although the code produced by the compiler is not efficient, it can be optimized considerably. For example, partial evaluation alone will cause many `equal/2` subgoals to be replaced by unifications or `simplify/2` subgoals.

Narrowing Grammar Rules	Prolog Clauses

`match([],S) => S.`	`simplify(match(A,B),C) :-`
	`  equal(A,[]),`
	`  equal(B,S),`
	`  simplify(S,C).`

`match([X|L],[X|S]) =>`
`  match(L,S).`
	`simplify(match(A,B),C) :-`
	`  equal(A,[X|L]),`
	`  equal(B,[X|S]),`
	`  simplify(match(L,S),C).`

`(X+) => X.`	`simplify((A+),B) :-`
	`  equal(A,X),`
	`  simplify(X,B).`

`(X+) => X, (X+).`	`simplify((A+),B) :-`
	`  equal(A,X),`
	`  simplify((X,(X+)),B).`

`([],L) => L.`	`simplify((A,B),C) :-`
	`  equal(A,[]),`
	`  equal(B,L),`
	`  simplify(L,C).`

`([X|L1],L2) =>`
`  [X|(L1,L2)].`
	`simplify((A,B),C) :-`
	`  equal(A,[X|L1]),`
	`  equal(B,L2),`
	`  simplify([X|(L1,L2)],C).`

It should be clear that `simplify/2` guarantees its result (the second argument) will be simplified. That is, the function symbol of the result will be a constructor. Also, we can show that `simplify/2` behaves *like* NU-narrowing. Consider a derivation from the goal `simplify`($f(t_1,...,t_m)$,Z) which uses the Prolog clause

`simplify`($f(X_1,\ldots,X_m)$`,Out) :-`
`  equal(`X_1`, `Y_1`) , ..., equal(`X_m`, `Y_m`), simplify(`*RHS*`,Out).`

resulting from $f(Y_1,...,Y_m)$ => *RHS*. In left-to-right Prolog derivation with this clause, the `equal/2` subgoals are satisfied first, each effecting either a unification (first clause of `nu_narrow/2`) or a recursive simplification (second clause of `nu_narrow/2`). These are followed by a derivation from the subgoal `simplify`(*RHS*,`Out`), which also recursively effects a simplification of *RHS*. Concatenating these simplifications, we find that `simplify/2` effects a *simplification*. More formally:

Theorem 3.1: If X and Y are terms such that `simplify`(X,Y) succeeds, then `simplification`(X,Y) has a successful left-to-right SLD-derivation.

Proof :

We sketch a proof by induction on the length of a successful Prolog-derivation of `simplify`(X,Y). Because of the structure of the clauses for `simplify/2`, there must be a goal in this derivation that is an instance of `simplify`(Z,Y) for some term Z not equal to X. Since we know `simplify`(Z,Y) succeeds, then by the induction hypothesis, `simplification`(Z,Y) has a successful derivation. Now, we can show that each of the `equal/2` subgoals introduced in the derivation leads either to a unification or a derivation from a `simplify/2` subgoal, and these inductively give NU-narrowings. Since the composition of these NU-narrowings of arguments is a NU-narrowing, we can prove inductively that `nu_narrowing`(X,Z) has a successful derivation. Combining this with the result above about `simplification`(Z,Y), we find that `simplification`(X,Y) has a successful left-to-right SLD-derivation.

The converse of the theorem is true *if* we replace Prolog-derivation by left-to-right SLD-derivation, and we restrict the use of duplicate variables among arguments on the left hand sides of narrowing grammar rules in certain ways beyond the restrictions in Definition 2.2. One restriction is to require that only terms in normal form be passed to these arguments. This subject is studied in more detail in [4].

4. What is New about Narrowing Grammar

In this section we summarize several important features of narrowing grammar. Some of these features are novel in the context of grammar formalisms, while others are not. The combination of these features is certainly new and interesting, in any event.

4.1. New Model of Acceptance in Logic Grammar

Previously, we have described how grammar rules operate as pattern generators or specifiers. In this section, we show that they can also operate as 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 following derivation illustrating how `match` works:

```
match(([a]+,[b]), [a,a,b])
    → match(((([a],[a]+),[b]), [a,a,b])
    → match(((([a|([],[a]+)],[b]), [a,a,b])
    → match(([a|(([],[a]+),[b])], [a,a,b])
    → match((([],[a]+),[b]), [a,b])
    → match(([a]+,[b]), [a,b])
    → match(([a],[b]), [a,b])
    → match([a|([],[b])], [a,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, or parser. This acceptance/generation duality is familiar to users of Definite Clause Grammar [16], and the ability to employ grammars both as acceptors and as generators has a number of uses [8].

4.2. 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 `number(_,_)` defined in example 2.1 is higher-order, as its first argument is a pattern. The whole pattern `([a]+,[b])` can be used as an argument, as in:

```
number( ([a]+,[b]), Total ).
```

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 [7]. As a consequence, narrowing grammar rules are highly reusable and can be usefully collected in a library. In short, narrowing grammar is modular. Narrowing grammar is also extensible, since it permits definition of new grammatical constructs, as the `number` and `//` examples showed earlier.

4.3. Lazy Evaluation, Stream Processing, and Coroutining

Leftmost outermost reduction is also called normal-order reduction, and sometimes *lazy evaluation*. This name comes from the basic observation that outside-in evaluation of an expression tends to evaluate arguments of function symbols only on demand – i.e., only when the argument values are needed. That is, outside-in evaluation is 'lazy'.

Narain showed that compilation like that in section 3 is a technique for implementing lazy evaluation [13, 14]. Thus narrowing grammar rules are compiled to Prolog clauses in such a way that, when SLD-resolution with left-to-right goal selection interprets them, it directly simulates

leftmost outermost narrowing, or lazy evaluation.

Lazy evaluation is intimately related with a programming paradigm referred to as *stream processing* [15]. 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 applicable frequently in stream processing. For example, narrowing of the stream pattern

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

interleaves the narrowing of `[a]+` with the narrowing of `(_, [b])`. The sample narrowing of this pattern in section 1 shows the actual interleaving – first `[a]+` is narrowed for two steps, then `(_, [b])` for one step, then `[a]+` for two steps, and finally `(_, [b])` for two steps. The effect of leftmost outermost narrowing of the combined stream pattern is precisely to interleave these two narrowings.

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 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.

5. Comparison with Definite Clause Grammar

In this section we compare narrowing grammar with Definite Clause Grammar (DCG), a widely-used formalism of logic grammar. We show how pure DCG can be translated into narrowing grammar and how narrowing grammar and DCG differ. We also point out some limitations of first-order logic grammars.

5.1. Narrowing Grammar and Definite Clause Grammar

All pure narrowing grammar rules can be ultimately translated to pure Prolog clauses. As a consequence, the benefits DCG offers over things like Augmented Transition Networks listed in [16] are also enjoyed by narrowing grammar.

We show how a DCG rule can be translated into a narrowing grammar rule describing the same language.

How to Translate Definite Clause Grammar to Narrowing Grammar

(1) Essentially, DCG rules can be translated to narrowing grammar rules by changing all occurrences of `-->` to `=>` and by including the narrowing grammar definition for `',' ` as in section 2.2.

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

(2) Disjunction (`';'`) in DCG has the same semantics as in narrowing grammar but the latter can be defined at the grammatical level:

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

(3) *Metamorphosis grammar* [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:

```
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 narrowing grammar *T* can be *any* pattern, not just a stream of terminals.

Example 5.1

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. 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([], [])
        → []
```

5.2. 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 write DCG rules that behave like `number` given earlier. This is a basic limitation of first-order logic grammars.

Abramson [2] has addressed this limitation by introducing a meta-nonterminal symbol, written *meta(X),* where *X* may be instantiated to any terminal or nonterminal symbol. During parsing, an *X* is to be recognized at the point where *meta(X)* is used in a grammar rule. Abramson suggested two ways to implement *meta(X).* The first method makes an interpretive metacall wherever *meta(X)* is used. (This is a special case of the approach used by the `phrase/3` metapredicate in Quintus and Sicstus Prolog.) The other approach is to preprocess the rules containing *meta(X)* so as to generate a new set of rules with no calls to *meta(X).* However, this preprocessing can generate extra nonterminals and rules.

The same 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 DCG rule

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

to the Prolog clause

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

There is one final limitation. Even with the techniques suggested by Abramson and Moss, the

lazy evaluation or coroutining aspects of narrowing grammar are not easily attained with first-order logic grammars. To attain them, a new evaluation strategy is needed for these grammars, implemented via either a meta-interpreter or a compiler like that in section 3.

6. Discussion and Conclusion

In this paper, we have shown how narrowing grammar, together with `match`, comprises a new formalism for language analysis. Narrowing grammar combines concepts from logic programming, rewriting, lazy evaluation, and logic grammar formalisms such as DCG. All narrowing grammar rules can be compiled to Prolog clauses in such a way that, when SLD-resolution with left-to-right goal selection interprets them, it directly simulates a kind of leftmost outermost narrowing called NU-narrowing.

Both narrowing grammar and DCG have a theoretical foundation in first-order logic. Since DCG can be translated directly to narrowing grammar, the benefits DCG offers over things like Augmented Transition Networks listed in [16] are enjoyed by narrowing grammar. Although we have shown how a pure DCG rule can be translated to a narrowing grammar rule, translation the other way is not trivial. We have also illustrated by examples that some problems are difficult to express with DCG but are very easy to express with narrowing grammar. 'Coroutined' matching of patterns is among these.

Narrowing grammar is modular, extensible and highly reusable, so saving rules in a library makes sense. These grammars extend the expressive power of first-order logic grammars, by permitting patterns to be passed as arguments to the grammar rules. As a consequence, some complex patterns can be specified more easily.

The main issue of implementation here that we have not addressed is *efficiency*. An interesting open problem is to devise improvements for the (relatively inefficient) implementation given in this paper. 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]) => t2(S).
t2([a|S]) => t2(S).
t2(S) => t3(S).
t3([b|S]) => S.
```

More work on the issue of efficiency appears in [4].

Beyond standard applications of grammar, narrowing grammar has potential in new areas including specification, analysis, and verification of concurrent systems. It can be used in 'history-oriented' or 'object-oriented' specification of concurrent systems. In [3] we demonstrate this in more detail. Given a set of actors (automata, concurrent objects, etc.) $A_1,...,A_n$ we can produce narrowing grammars with starting patterns $S_1,...,S_n$ for these, and then use $S_1//.../\!/S_n$ as a specification of valid histories for the entire system. To our knowledge, this

aspect of narrowing grammar is unique.

Acknowledgement

Many people have contributed to the ideas in this paper, but we particularly would like to acknowledge the remarks of Paul Eggert, Sanjai Narain, Fernando Pereira, and the referees.

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. Chau, H.L. and D.S. Parker, "Executable Temporal Specifications with Functional Grammars," Technical Report CSD-880046, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, June 1988.
4. Chau, H.L., "Narrowing Grammar: A New Scheme for Language Analysis," Ph.D. Dissertation, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1989, forthcoming.
5. Colmerauer, A., "Metamorphosis Grammars," in *Natural Language Communication with Computers, LNCS 63*, Springer, 1978.
6. Dahl, V. and H. Abramson, "On Gapping Grammars," *Proc. Second Intl. Conf. on Logic Programming*, pp. 77-88, Uppsala, Sweden, 1984.
7. Darlington, J., A.J. Field, and H. Pull, "The Unification of Functional and Logic Languages," *Logic Programming: Functions, Relations, and Equations*, Prentice-Hall, 1986.
8. Gorlick, M.D., C. Kesselman, D. Marotta, and D.S. Parker, "Mockingbird: A Logical Methodology for Testing," Technical Report, The Aerospace Corporation, P.O. Box 92957, Los Angeles, CA 90009-2957, May 1987. To appear, *Journal of Logic Programming*, 1989.
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., "A Technique for Doing Lazy Evaluation in Logic," *J. Logic Programming*, vol. 3, no. 3, pp. 259-276, October 1986.
14. 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.

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

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

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

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

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

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