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

FUNCTIONAL LOGIC GRAMMAR: A NEW SCHEME FOR LANGUAGE ANALYSIS

H. Lewis Chau
D. Stott Parker

December 1988
CSD-880097

Functional Logic Grammar: A New Scheme for Language Analysis

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 *Functional Logic Grammar*.

A functional logic grammar is a finite set of rewrite rules. It is efficiently executable, like most logic grammars. In fact, functional logic grammar rules can be compiled to Prolog and executed by existing Prolog interpreters as generators or acceptors. Unlike most logic grammars, functional logic grammar also permits higher-order specification and modular composition.

This paper defines functional logic 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 functional logic grammar. Functional logic grammar enjoys the advantages of DCG, as well as its first-order logic foundation. At the same time, functional logic grammar ranks higher in aspects such as expressiveness and modularity, and permits lazy evaluation.

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

This work done under 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`.

Functional Logic Grammar: A New Scheme for Language Analysis

H. Lewis Chau

D. Stott Parker

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

1. Introduction

Since the development of metamorphosis grammar [6], the first logic grammar formalism, several variants of logic grammars have been proposed [1, 7, 8, 11, 14, 15, 18]. 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 [17, 19].

All 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, DCG does not permit direct specification of grammar rules of the form

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

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.

Functional logic 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 functional logic grammar are defined by a term-rewriting system that is an extension of Narain's Log(F) system [12]. This approach gives both a compact formal definition of functional logic grammar, and an efficient logic programming implementation. In this paper, we point out a number of advantages of functional logic grammar for language analysis.

As a brief introductory example, let us show how easily regular expressions can be defined with functional logic 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 pattern operator defining the Kleene plus pattern, and the rules for ',' define pattern concatenation, very much like the usual Prolog rules for **append**.

Functional logic grammar rules are used much like rewrite rules with 'narrowing' [12], but with a special outermost term rewriting strategy described in the next section. With this strategy, one possible rewriting of `([a]+,[b])` is as follows:

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

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

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

Beyond standard grammatical applications, functional logic grammar has potential in new areas including the specification, analysis, and verification of properties of concurrent systems [3, 4, 13]. One major application of functional logic grammar is the *Stream Pattern Analyzer (SPA)* [3, 4] under development as part of the Tangram project at UCLA. SPA uses functional logic grammar to specify pattern analysis for streams of data. Parallel execution events in a distributed system may be captured in an event stream for analysis. Given a set of functional logic grammar rules, SPA can analyze arbitrarily complex behavior patterns in this stream. At the same time a SPA grammar can act as a declarative specification of valid event histories.

Section 2 defines functional logic grammar and section 3 describes some of its interesting features. Section 4 then goes on to compare functional logic grammar with the widely-used DCG formalism in logic programming, showing how it offers several important advantages. Section 5 gives one example illustrating the expressive power of functional logic grammar. Finally we conclude the paper with some potential applications, along with avenues for future work.

2. Functional Logic Grammar

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

2.1. Formalism of Functional Logic 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 *functional logic grammar* is a finite set of rules of the form:

```
LHS => RHS.
```

where `LHS` is any term except a variable, and `RHS` is a term.

Definition 2.3

Functors that do not appear as any rule's outermost `LHS` functor are called *constructor symbols*.

Terms whose function symbol is a constructor symbol are said to be *simplified*. By convention also, every variable is taken to be a simplified term. Note that no `LHS` of any rule can be a simplified term.

Definition 2.4

A *stream* is defined as a list of ground terms. By convention, we require the function symbols for lists (namely, the empty list `[]` and cons `[_|_]`, following Prolog syntax) to be constructor symbols. Thus all streams are simplified.

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), for which there is a most general unifier θ of LHS and s, then we say p *narrows* to $q = r[RHS]\theta$.

A *narrowing* is a sequence of terms $p_1, p_2, ..., p_n$ such that for each i, when p_i and p_{i+1} both exist, 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 with successively the actual rewriting mechanism to implement narrowing. For functional logic grammar, the selection strategy is given by the (nondeterminate) predicate `select(p,s)` defined below:

Definition 2.6

`select(P,S)` is defined by the following pseudo-Horn clauses:

```
select(P,P) ←
        P is a nonvariable,
        there is a rule (LHS => RHS),
        and LHS and P unify.
select(P,S) ←
        P is of the form f(T1,...,Ti,...,Tn),
        there is a rule (f(L1,...,Li,...,Ln) => RHS),
        Ti and Li do not unify,
        and select(Ti,S).
```

Since we treat variables as simplified terms, `select(P,S)` is undefined when `P` is a variable.

Definition 2.7: N-step

If `select`(p,s) and there exists a rule $(LHS \Rightarrow RHS)$. Let $p = r[s]$, and $q = r[RHS]\theta$, where θ is a most general unifier of LHS with the subterm s of p. Then we say p *narrows to* q *in an N-step*, or $p \rightarrow q$.

Definition 2.8: N-narrowing

A N-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 N-step.

These definitions mirror the definitions of reductions in Narain's Log(F) [12]. Analogously to the completeness results of Narain, we have the following conjecture:

Conjecture: Narrowing-completeness of functional logic grammar for simplified forms.

Let p_0 be a term and $p_0, p_1, \ldots p_n$ be a successful narrowing. Then there is a successful N-narrowing $p_0, q_1, \ldots q_m$, such that $q_m \xrightarrow{*} p_n$, where $\xrightarrow{*}$ is the reflexive transitive closure of $\rightarrow$.

In Appendix 1 we describe how functional logic grammar rules are translated to logic programs. It turns out that SLD-resolution, the proof procedure commonly used in logic programming, can serve as an interpreter for functional logic grammar. The translation, or compilation, process has the property that when SLD-resolution interprets the logic program, it directly simulates lazy rewriting (narrowing) using the grammar. For more discussion, see [5].

2.2. Specifying Patterns with Functional Logic Grammar

We illustrate how patterns can be developed in functional logic grammar with a sequence of examples.

Example 2.0

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

Example 2.1

Suppose we wish to count the number of times one or more network failures is followed by a cpu failure in a stream that is a trace of events from a large operating system. That is, we want to count the occurrences of a specific pattern of failures in the input stream. This pattern can be represented by the regular expression `([net_failure]+,[cpu_failure])`, and we can count the number of its occurrences with the pattern

```
number( ([net_failure]+,[cpu_failure]), Total)
```

where 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,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.

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

Suppose we wish to count the number of occurrences of disk failures in addition to `([net_failure]+,[cpu_failure])`. That is, we want to count the occurrences of multiple patterns in an event stream. We use the pattern

```
number(([net_failure]+,[cpu_failure]),NF) // number([disk_failure],DF)
```

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 parallel pattern matching primitive that requires both argument patterns to match inputs of the same length. This example shows that multiple overlapping patterns in a stream can be simultaneously recognized easily with `//`.

Example 2.3

Consider the following grammar:

```
s => 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 $\{a^n b^n c^n\}$ using only context-free-like constructions. The first rule for `s_abc` imposes simultaneous (parallel) constraints on streams generated by the grammar.

3. Interesting Aspects of Functional Logic Grammar

In this section we summarize several important features of functional logic 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.

3.1. Executability

The translation mentioned in Appendix 1 from functional logic grammar rules to logic programs gives us a way to 'execute' these grammars. Previously, we have described how the 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 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 analysis is signaled explicitly with `match`. For example,

```
match( ([net_failure]+,[cpu_failure]), file_terms('experiment.output') ).
```

matches the pattern 'one or more copies of `net_failure` followed by a `cpu_failure`' against the stream of terms produced by the file `'experiment.output'`.

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 [14], and the ability to employ grammars both as acceptors and as generators has a number of uses [10].

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

3.2. The power of unification

Unification arises naturally in parsing. Functional logic grammar captures the power of unification, permitting arguments of terms in rewrite rules to be used not only as inputs, but also as grammar outputs [17]. Like most logic grammars, functional logic grammar allows arbitrary structures to be built in the process of parsing.

3.3. Higher-order Specification, Extensibility, and Modularity

Functional logic grammar is higher-order. Specifically, functional logic grammar is higher order in the sense that patterns can be passed as 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 `([net_failure]+,[cpu_failure])` can be used as an argument, as in:

```
number( ([net_failure]+,[cpu_failure]), Total).
```

It is well known that a higher-order capability increases expressiveness of a language, since it becomes easier to develop short generic functions that can be combined in a multitude of ways [9]. As a consequence, functional logic grammar rules are highly reusable and can be predefined

in a library. In short, functional logic grammar is modular.

Functional logic grammar is also extensible, since it permits definition of new grammatical constructs, as the `number` and `//` examples showed earlier.

3.4. Lazy Evaluation

Lazy evaluation is basically a computation scheme in which an expression is evaluated only when there is demand for its value. Functional logic grammar rules are compiled to Prolog `narrow` clauses in such a way that, when SLD-resolution interprets them, it directly simulates lazy rewriting [5].

One specific advantage of lazy evaluation in parsing is the ability to simultaneous recognition of multiple patterns in a stream. We illustrate this point with the now familiar operator, `//`.

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

`//` takes two patterns as arguments. In one step, the first argument is narrowed to `[X|Xs]`. Only the head `X` is computed. The tail `Xs` can then be further narrowed if this is necessary. Demand-driven computation like this is referred to as lazy evaluation or delayed evaluation, and is one of the important features of functional logic grammar.

An immediate advantage of lazy evaluation here is reduced computation. Essentially `//` narrows its arguments to `[X|Xs]` and `[X|Ys]` and suspends evaluation of `Xs` and `Ys`. If lazy evaluation is not performed, on the other hand, both arguments are completely evaluated before pattern matching is applied. If the heads of both completely evaluated terms then do not unify, many unnecessary narrowing steps have been performed to simplify the tails of two arguments. For other useful examples of lazy evaluation, see [12].

4. Comparison with Definite Clause Grammar

In this section we compare functional logic grammar with Definite Clause Grammar(DCG), a widely-used formalism of logic grammar. We show how pure DCG can be translated into functional logic grammar and how functional logic grammar and DCG differ. Due to length restrictions on the paper, we simply assume that the reader is familiar with DCG.

4.1. Equivalence between Functional Logic Grammar and Definite Clause Grammar

All pure functional logic grammar rules can be ultimately translated to pure Prolog clauses. As a consequence, the benefits DCG offers over things like Augmented Transition Networks as in [14] are also enjoyed by functional logic grammar.

We show how a DCG rule can be translated into a functional logic grammar rule describing the same language. We also show that, while the reverse translation is possible, it is not trivial.

How to Translate Definite Clause Grammar to Functional Logic Grammar

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

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

(2) {curly braces} and `';'` of DCG have the same semantics as of functional logic grammar but the latter are defined at the grammatical level.

```
(X;Y) => X.
(X;Y) => Y.
{G} => if(success(G),[]).
```

Here `success(G)` is a special construct whose compilation produces code that invokes a Prolog goal `G` and correspondingly narrows to either `true` or `false`. Note that this definition for {curly braces} does not work for cuts. However, it is not difficult to extend the compilation algorithm to handle impure DCG constructs such as {!}.

(3) *Metamorphosis grammar* [6] permits rules of the form

```
X,Y --> RHS
```

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

```
match((X,Y),S) => match(Y,match(X,S)).
match(replace(X,Y),S) => append(Y,match(X,S)).
```

and transform the metamorphosis grammar rule to

```
X => replace(RHS,Y).
```

For example, the metamorphosis grammar rule

```
is(N), [not] --> [aint].
```

is transformed to

```
is(N) => replace([aint],[not]).
```

The translation from functional logic grammar to DCG is not trivial. For example,

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

are valid functional logic grammar rules but

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

are not valid DCG rules because the right hand side of a DCG rule cannot be a variable.

4.2. DCG is First-order and is not fully modular

A DCG rule is no more than 'syntactic sugar' for a certain kind of (first-order) Prolog clause. Each nonterminal is a Prolog predicate with Prolog terms as its arguments. Therefore, it is hard to write DCG rules that behave like `number` given earlier. DCG rules do not permit arbitrary patterns to be passed as arguments.

We conclude this section by emphasizing the following two points:

(1) Since DCG is not higher-order, it is not reasonable to pass DCG nonterminals as arguments. This restricts the modularity of this kind of grammar.

(2) Some things seem very hard to do with DCG, but are easy to do with functional logic grammar. One such thing that is quite useful is 'and parallel' pattern matching, `//`.

5. Case Study

We give one example, filler-gap dependencies, illustrating the expressive power of functional logic grammar. The example shows that some problems are quite cumbersome to encode in DCG but easy to encode in functional logic grammar.

5.1. Filler-Gap Dependencies

Filler-gap dependencies constitute a set of linguistic phenomena with a quite cumbersome encoding in DCG. We take the definition of Filler-gap dependencies from [17] as follows:

> *A filler-gap dependency occurs in a natural-language sentence when a subpart of some phrase (the gap) is missing from its normal location and another phrase (the filler), outside of the incomplete one, stands for the missing phrase. We have a dependency between the gap and the filler because the gap can only occur when the appropriate filler occurs.*

For example, in the sentence

```
Terry read every book that Bertrand wrote.
```

there is a filler-gap dependency between the relative pronoun 'that' (the filler) and the missing direct object of 'wrote' (the gap). In other words, this sentence intuitively contains the subsentence 'Bertrand wrote [the book]', but English syntax permits '[the book]' to be removed (gapped) because the filler 'that' is used.

The obvious way of representing filler-gap dependency in DCG is to use a nonterminal argument to indicate the presence or absence of a gap within the phrase covered by the nonterminal. If there is a gap, we must also indicate the syntactic category (i.e., nonterminal) of the gap. However, the technique used for passing gap information among the nonterminals in grammar rules has some problems. In [17], the formalism of Filler-Gap DCG is introduced. This kind of DCG allows a simpler statement of filler-gap constraints, and an interpreter is presented for it. However, the interpreter must be modified for any new grammatical constructs.

Functional logic grammar allows a more direct approach to filler-gap dependencies. Instead of writing an interpreter as in DCG, filler-gap dependencies problem can be expressed directly in

functional logic grammar. As a simple example, we define the following functional logic grammar rules, adapted from [17]:

```
s => np_island, vp.

np => np_island.

np_island => det, n, optrel.
np_island => pn.

vp => dv, np, np.
vp => tv, np.
vp => iv.

optrel => [].
optrel => [that], s without np.
optrel => [that], vp.

det => [every].

n => [book].

pn => [bertrand].
pn => [terry].

tv => [read].
tv => [wrote].
```

This grammar is very easy to read, and we leave the reader to compare it with the grammar in [17]. Both `np` and `np_island` are noun phrases but no gap can occur in the latter. The phrase `'s without np'` means that there exists a gap `np` in `s`. With the above functional logic grammar rules, we need to add the definition of `without`.

```
(A without A) => [].
((A,B) without A) => B.
([A|B] without C) => [A|B without C].
```

Now we illustrate how the sentence

```
terry read every book that bertrand wrote
```

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

```
→ np_island, vp
→ pn, vp
→ [terry], vp
→ [terry], tv, np
→ [terry], [read], np
→ [terry], [read], np_island
→ [terry], [read], det, n, optrel
→ [terry], [read], [every], [book], [that], s without np
→ [terry], [read], [every], [book], [that], (np_island, vp) without np
→ [terry], [read], [every], [book], [that], (pn, vp) without np
→ [terry], [read], [every], [book], [that], [ bertrand | vp ] without np
→ [terry], [read], [every], [book], [that], [ bertrand | vp without np ]
→ [terry], [read], [every], [book], [that], [bertrand], vp without np
→ [terry], [read], [every], [book], [that], [bertrand], (tv,np) without np
→ [terry], [read], [every], [book], [that], [bertrand], [wrote], np without np
→ [terry], [read], [every], [book], [that], [bertrand], [wrote], []
```

The point to be made here is that quite sophisticated grammars can be developed easily and concisely with functional logic grammar. Syntactic features that are treated at the 'meta level' with DCG rules can be handled by the higher-order capabilities of functional logic grammar.

In Extraposition Grammars [16], Pereira uses 'bracketing constraints' to permit sentences such as

```
[the], [mouse], [that], [the], [cat], [that], [likes], [fish], [chased], [squeaks]
```

while ruling out sentences such as

```
[the], [mouse], [that], [the], [cat], [that], [chased], [likes], [fish], [squeaks].
```

These constraints are also handled by the higher-order capabilities of functional logic grammar (using the definition of `without` above).

6. Discussion and Conclusion

In this paper, we have shown how functional logic grammar, together with `match` and `select`, comprises a new formalism for language analysis. Functional logic grammar combines concepts from logic programming, rewriting, lazy evaluation, and logic grammar formalisms such as DCG. All functional logic grammar rules can be compiled to Prolog clauses in such a way that, when SLD-resolution interprets them, it directly simulates lazy rewriting. As a consequence, full advantage is taken of the very efficient implementations of Prolog that are now available.

Both functional logic grammar and DCG have a theoretical foundation in first-order logic. Therefore, the benefits DCG offers over things like Augmented Transition Network as in [14] are enjoyed by functional logic grammar. We have shown how a pure DCG rule can be translated to a functional logic grammar rule; however, 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 functional logic grammar. Filler-gap dependencies and 'and parallel' matching of patterns are among these.

Functional logic grammar is modular, extensible and highly reusable, and can be predefined 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. As a consequence, some complex patterns can be specified more easily.

Beyond standard applications of grammar, functional logic grammar has potential in new areas including specification, analysis, and verification of concurrent systems. Furthermore, 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 functional logic 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 functional logic grammar is unique.

Once a specification mechanism becomes sufficiently powerful, its 'expressive power' is no longer problematic, and other issues become important: elegance, usefulness in specifying real systems, tractability of inference problems used in verification, and so on. Functional logic grammar is very interesting in its properties here. Since it is grammatically based, it can be used for both acceptance and generation of histories. Acceptance is equivalent to *analysis* of histories. Generation is useful in producing hypothetical histories for *testing* [10]. Furthermore, since functional logic grammar can be ultimately translated to logic program it is possible to perform *verification* of properties of this grammar using existing first-order logic theorem proving technology. Altogether functional logic grammar appears to hold many advantageous properties for specification.

Acknowledgement

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

Appendix 1: Compilation of Functional Logic Grammar to Prolog `narrow` rules

Below we give the compilation procedure for functional logic grammar, and show by examples how logic programs (here Prolog clauses for the `narrow` predicate) are produced from functional logic grammar rules. When SLD-resolution (the proof procedure commonly used in logic programming) interprets these programs, it simulates lazy rewriting using `select`.

The compilation algorithm consists of the following steps:

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

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

(2) Let $f(L_1,\ldots,L_m) \Rightarrow RHS$ be a rule. Let $A_1,\ldots,A_m$, `Out` be distinct Prolog variables not occurring in the rule. If L_i is a variable, let Q_i be $A_i = L_i$. If L_i is $g(Y_1,\ldots,Y_l)$, and g is a non-constructor symbol, let Q_i be `equal(`A_i`,` $g(Y_1,\ldots,Y_l)$`)`. Otherwise, let Q_i be `narrow(`A_i`,` $g(Y_1,\ldots,Y_l)$`)`.

(3) Let $f(L_1,\ldots,L_m) \Rightarrow RHS$ be a rule. For all variables `X` that occur multiple times in `LHS` and also occur in L_i, let P_i include `equal(`X_i`,X)`. If there are no such variables, let P_i be true.

(4) Generate the clause:

`narrow(` $f(A_1,\ldots,A_m)$`, Out) :-` $Q_1,P_1,\ldots,Q_m,P_m$`, narrow(RHS,Out).`

In practice, if L_i is a variable, Q_i can be dropped, provided A_i is replaced by L_i in $f(A_1,\ldots,A_m)$.

This algorithm does not deal with "impure" features in functional grammar rules, such as cuts or `success(G)`, mentioned in the text. Obviously however the compilation can be extended to include such features.

One of the interesting aspects of the compilation algorithm is that it permits the user to declare a suitable 'equality' predicate. The predicate `equal(X,Y)` is to succeed whenever both `X` and `Y` can be narrowed (not necessarily N-narrowed) to a common term. In many situations the grammar will obey the restrictions of Log(F) [12], and this predicate will not be used. However in general, such as for `without` in the filler-gap example above, an extended notion of equality is necessary. The advantage of using `equal` is that, although full narrowing of both arguments is necessary for the sake of completeness, in many situations (such as DCG-like parsing) a simple approximation (such as unification) will suffice. Users can select a definition for `equal` that meets their needs.

The table below lists some functional logic grammar rules together with the Prolog `narrow` rules resulting from their compilation. It should be clear that `narrow` is similar to the transitive closure of `=>`, but it guarantees its result (the second argument) will be in simplified form. That is, the function symbol of the result will be a constructor.

Functional Logic Grammar Rules	Prolog narrow Rules
`match([],S) => S.`	`narrow(match(A,B),B) :-` `  narrow(A,[]).`

`match([X|L],[X|S]) =>`
`  match(L,S).`
	`narrow(match(A,B),C) :-`
	`  narrow(A,[D1|E]),`
	`  equal(D1,D),`
	`  narrow(B,[D2|F]),`
	`  equal(D2,D),`
	`  narrow(match(E,F),C).`

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

`([],L) => L.`
`([X|L1],L2) => [X|(L1,L2)].`
	`narrow((A,B),B) :- narrow(A,[]).`
	`narrow((A,B),[C|(D,B)]) :-`
	`  narrow(A,[C|D]).`

`number(Pattern,Total) =>`
`  number(Pattern,Total,0).`
`number(Pattern,Total,Total)`
`  => [end_of_file].`
`number(Pattern,Total,Count)`
`  => Pattern,`
`  number(Pattern,Total,Count+1).`
`number(Pattern,Total,Count)`
`  => [_], number(Pattern,Total,Count).`
	`narrow(number(A,B),C) :-`
	`  narrow(number(A,B,0),C).`
	`narrow(number(A,B,B),[end_of_file]).`
	
	`narrow(number(A,B,C),D) :-`
	`  E is C + 1,`
	`  narrow((A,number(A,B,E)),D).`
	`narrow(number(A,B,C),D) :-`
	`  narrow(([_],number(A,B,C)),D).`

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. and D.S. Parker, "Functional Grammars: A New Formalism for Stream Pattern Analysis," Draft, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, May 1988.
5. Chau, H.L. and D.S. Parker, "Functional Logic Grammar," Technical Report, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, December 1988.

6. Colmerauer, A., "Metamorphosis Grammars," in *Natural Language Communication with Computers, LNCS 63*, Springer, 1978.

7. Dahl, V., "More on Gapping Grammars," *Proc. Intl. Conf. on Fifth Generation Computer Systems*, Tokyo, 1984.

8. Dahl, V. and H. Abramson, "On Gapping Grammars," *Proc. Second Intl. Logic Programming Conf.*, pp. 77-88, Uppsala, Sweden, 1984.

9. Darlington, J., A.J. Field, and H. Pull, "The Unification of Functional and Logic Languages," *Logic Programming: Functions, Relations, and Equations*, Prentice-Hall, 1986.

10. 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*, 1988.

11. Hirschman, L. and K. Puder, "Restriction Grammars in Prolog," *Proc. First Intnl. Conf. on Logic Programming*, pp. 85-90, Marseille, 1982.

12. 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.

13. Parker, D.S., R.R. Muntz, and H.L. Chau, "The Tangram Stream Query Processing System," Technical Report CSD-880025, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, January 1988.

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

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

16. Pereira, F.C.N., "Logic for Natural Language Analysis," Technical Note 275, SRI International, Menlo Park, California, 1983.

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

18. Stabler, E.P., "Restricting Logic Grammars," *Proc. Fifth National Conference on Artificial Intelligence*, pp. 1048-1052, Philadelphia, PA, 1986.

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