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

# FUNCTIONAL LOGIC PROGRAMMING (FLOP)

Eugene Paik

May 1989
CSD-890021

# ABSTRACT

Functional LOgic Programming (FLOP) is a non-deterministic rewrite system which integrates functional programming and logic programming into a single unified paradigm. FLOP offers the expressiveness of functional notation as well as the declarative semantics of logic programs. This is accomplished by incorporating unification, non-determinism, and a flexible reduction strategy in one coherent environment.

FLOP programs are defined by a set of rewrite rules, where each rule is interpreted not only as a function specification, but also as a logical relationship. Whereas Prolog goals are simple predicates, FLOP goals are predicates whose truth values are represented by arbitrary terms. Hence, FLOP is an extention of SLD-resolution with functional qualities.

As a general programming environment, FLOP offers a number of advantages. Methodologies for both logic programming, such as in Prolog, and functional programming, such as in LISP, are directly applicable in FLOP. Special classes of rewrite rules, such as Sanjai Narain's LOG(F) and Definite Clause Grammars, are executable in FLOP with little or no syntactic enhancements. FLOP is an excellent environment for applications of partial evaluation. Also, lazy evaluation is inherently supported as the default reduction strategy is thoroughly lazy, i.e., terms are not reduced unless otherwise specified.

First, the syntax and the semantics of FLOP programs are defined. Next, comparisons between FLOP and other systems, namely, Prolog, LISP, and LOG(F), are discussed. An algorithm is then presented for compiling FLOP rewrite rules into Prolog clauses that correctly simulate the behavior of the rules with no computational overhead. Numerous examples of FLOP programs are presented throughout the text.

# TABLE OF CONTENTS

# CHAPTER 1 - INTRODUCTION

Until now, the expressiveness of functional notation has been lacking in most logic programming systems. At the same time, the computational flexibility of declarative semantics has been difficult to attain in most functional programming systems. This paper presents a general programming language called Functional **LO**gic **P**rogramming (FLOP) in which the advantages of both functional and logic programming are integrated in a single unified paradigm.

## 1.1 Functional Programming, Rewrite Systems

For the purposes of this thesis, we will identify functional programming with rewriting. In functional programming (rewriting), programs are defined by a set of functions (rewrite rules). Each rule specifies how an expression can be rewritten in a simpler form, while maintaining the meaning of the original expression. For example, with the plus (+) function, the expression `1+2` can be rewritten as `3`. Note that functions are directional. For instance, `3` is generally not rewritten as `1+2`, because `3` is a simpler form of `1+2`. The process of transforming one expression to another using a rewrite rule is called *reduction* (*evaluation*). Computation with a functional program is equivalent to evaluating a term using the rules specified in the program.

In this paper, rewrite rules are specified with the following notation:

LHS => RHS.

where LHS and RHS specify arbitrary patterns. A rewrite rule states that if a term matches LHS, then it can be rewritten as RHS. For example, the notion that the `color` of `apples` is `red`, and of `bananas` is `yellow`, can be expressed with the following set of rewrite rules:

```
color(apples) => red.

color(bananas) => yellow.
```

Here, `color` is a unary function, i.e., a function which takes one argument.

The semantics of functional programs depend not only on the set of functions defined in it, but also on the strategy used to apply them in reduction. There are two components to a reduction strategy: *term selection* and *rule selection.* Term selection can be viewed as a function which takes as its input a term **T** and returns either an indication that **T** is fully reduced or a subterm **S** of **T**, such that the first step in reducing **T** is to replace **S** with its reduced value **S'**. Thus, term selection must first determine when a term is fully reduced. There are a number of possible strategies for this. One possibility is to designate a special set of irreducible terms, called *constructors*[1]. With this approach all reductions terminate when they yield constructors. Another method is to always evaluate

[1]This strategy is used in Sanjai Narain's LOG(F).

a term unless protected by a special quote operator. We call this strategy *eager reduction*[1]. The dual of this last approach is to always not evaluate a term, unless indicated by a special eager operator. We call this strategy *lazy reduction*[2].

Once it is determined that a term is not fully reduced, term selection must then select the subterm which must be reduced first. In LISP, for example, the immediate arguments are evaluated first, unless the function is a special function. For example, in evaluating the term `(cdr (quote (a b)))`, the subterm `(quote (a b))` is evaluated first. However, in evaluating `(quote (a b))`, `(a b)` is not evaluated because `quote` is a special function. Thus, the final result is `(b)`. In lazy reduction, the term selected is the innermost subterm whose functor is the eager operator. Using the `?` symbol as the eager operator, in reducing the term `f(? g(? x))`, the subterm `x` is reduced first. With both eager and lazy strategies, term selection is also applied to the RHS of a rule.

Traditionally, functional programming languages have allowed only *determinate* functions. In this case, reduction of a term always yields a unique value. LISP, for instance, is in this category. However, a rewrite system can also be *non-deterministic*, i.e., allow *non-determinate* functions, such as:

```
hair_color(people) => black.
hair_color(people) => blond.
hair_color(people) => brown.
```

Here, the term `hair_color(people)` may be reduced to either `black` or `blond` or `brown`, depending on which rule is applied. When only determinate functions are permitted, rule selection is trivial, for no decision is required. However, when non-determinate functions are allowed, a reduction strategy must define a consistent method of selecting a rule so that the behavior of non-determinate functions is predictable. Backtracking is one such strategy. In backtracking, the rules of each non-determinate function are ordered, say, according to their textual positions in the program. When a selection must be made, the first rule is chosen. Further, if another solution is desired, it will backtrack and choose the next in order.

## 1.2 Logic Programming

In logic programming, programs are defined by a set of propositions, which state that certain relationships are true. Computation with a logic program is equivalent to deducing certain facts from the original set of facts specified in the program.

As Prolog is by far the most prevalent logic programming system, henceforth, we shall limit our discussion to this form of logic programming. Prolog[3] is an SLD-resolution (or LUSH-resolution) [HILL 74] system. Prolog programs are defined by a set of Horn clauses which have the following form:

$$\texttt{Head :- } G_1, \ldots, G_n.$$

[1]Eager reduction is the strategy used in LISP systems.
[2]Lazy reduction is the strategy used in FLOP.
[3]Prolog also contains, for pragmatic purposes, extra-logical (such as I/O ), meta-logical (such as checking for unbound variables) and backtrack modifying (the cut) predicates; however, these features are not needed for our discussion here.

where $n \geq 0$ and `Head` and $G_i$ are predications, each of the form $R(t_1, \ldots, t_m)$, $m \geq 0$, where `R` is a relation symbol and each $t_i$ a term. A term is either a variable, of the form $f(s_1, \ldots, s_q)$, $q \geq 0$, where `f` is a `q`-ary function symbol, and each $s_j$ a term. When `n` is 0, the clause is called a unit clause and has the following meaning: the relationship defined by the `Head` is unconditionally true. When `n` is greater than 0, the clause can be interpreted as follows: the relationship defined by the `Head` is true, if the relationship defined by each $G_i$ is true. The conjunction $G_1, \ldots, G_n$ is called the *body of the clause* and each $G_i$ is called a *subgoal* of `Head`, or simply a *goal.*

For instance, the `color` example in the previous section can be expressed with the following set of unit clauses:

```
color(apples, red).
color(bananas, yellow).
```

Here, `color` is a 2-ary predicate, i.e., a predicate which takes two arguments. These propositions state that the `color` of `apples` is `red`, and of `bananas` is `yellow`. One may now ask the Prolog system whether it is true that `color` of `apples` is `red` with the following query (the actual query is in **`bold font`** with system response in `regular font`):

```
| ?- color(apples, red).
yes
```

The system responds with `yes`, stating that from the given the set of facts in the program the query is found to be provable.

Declarative semantics of logic programs refer to the notion that the set of facts specified in the program are treated independently from the algorithm or the mechanism for deducing other facts from it. This is closely related to the "adirectionality" of predicates. This may be made more clear with a concrete example. When a logical variable is supplied as an argument of a query, the logic systems attempts to find an appropriate substitution for the variable for which the query may be made true. The following example demonstrates this notion:

```
| ?- color(apples, X).
 X=red
yes

| ?- color(X, red).
 X=apples
yes
```

The first query can be interpreted as the question "what `color` do `apples` have?" and the second as "what item has the `color red`?" The first query is responded to by binding the variable `X` with `apples`, the second by binding `X` with `red`. In general, functional programming can compute the solution of the first question, but not of the second, due to the inherent directionality of functions.

Deduction strategy in logic programming is analogous to the reduction strategy of rewrite systems. The two components of deduction strategy are *rule selection* and *goal selection.* Rule selection here is identical to rule selection in a rewrite reduction strategy.

Prolog is a non-deterministic system in which rule selection is accomplished through backtracking.

Goal selection refers to the order in which the subgoals of a predicate are satisfied. Goal selection does not change the semantics of a pure logic program; however, it does affect its efficiency. In Prolog, goals are ordered according to their textual position in the clause. For example, consider the following program:

```
f(X) :- g(X), h(X).

g(a).
h(a).
h(b).
```

The first clause states that `f(X)` is true if both `g(X)` and `h(X)` are true. The unit clauses simply state that `g(a)`, `h(a)`, and `h(b)` are true. Note that the semantics of the first clause are not changed if its subgoals are reversed, but its execution is less efficient. To test if `f(b)` is true, it is more efficient to test the subgoal `g(b)` first, since it will immediate fail, thereby avoiding the testing of `h(b)`.

## 1.3 Previous Work

A number of attempts to combine functional programming and logic programming have been documented in the literature. The most common approach has been to provide these two programming paradigms as distinct and separate subsystems with a mechanism for interfacing between them. Included in this category are [BARBUTI 84], LOGLISP [ROBINSON 82], QLOG [KOMOROWSKI 82], [CARLSSON 84], and Scheme/L [SRIVASTAVA 85]. For these systems, "the integration problem is essentially the problem of allowing mutual invocation" [BARBUTI 84, p162]. Fundamentally, these systems differ only in the way the two subsystems are interfaced. LOGLISP, QLOG, and Scheme/L are LISP systems enhanced with a logic component in which logical assertions are represented as LISP structures. LISP functions are provided to access the logic subsystem and visa versa. In [BARBUTI 84], a special module called Logic for Communicating Agents (LCA) provides an interface between the procedural and declarative components. Although these system offer both programming environments, they do not provide a unified environment. Namely, functional notation remains unavailable in the logic subsystem, and declarative semantics in the functional subsystem.

FUNLOG [SUBRAHMANYAM 84] differs fundamentally from the previous approach. The basic idea here is to extend logic programming, or more specifically, extend unification, with function evaluations. This extension of unification is called *semantic unification*:

> "In trying to unify two terms, say $f(t_1, ..., t_n)$ and $f(s_1, ..., s_n)$, suppose there are subterms $t_i$ and $s_i$ that are not unifiable in the conventional sense. In such a case, if $s_i$ (or $t_i$) is reducible and the reduced version of it is unifiable with $t_i$ (respectively $s_i$) then we say that $f(t_1, ..., t_n)$ and $f(s_1, ..., s_n)$, are semantically unifiable." [SUBRAHMANYAM 84, p145]

Semantic unification offers a number of desirable properties. First, terms are not reduced if two terms are unifiable. Further, it allows computation on conceptually infinite data structures. However, there is a fundamental weakness in this approach: the operational semantics are far too hidden from the programmer. In contrast, Prolog's success can be attributed to the fact that it provides a clear operational (procedural) semantics, thus enabling the programmer to gauge the efficiency of a program. With semantic unification, a combinatorial explosion is easily produced as all possible reductions must be attempted before determining that two terms are not semantically unifiable. Also, semantic unification is asymmetrical. In the description above, the number of reductions required in semantically unifying two terms is highly sensitive to whether $t_i$ or $s_i$ is reduced first. Consider the following function:

```
f(N) => f(N+1).
```

Here, `f(N+1)` and `f(N)` are clearly semantically unifiable after a single application of the rule to the second term. However, if the first term is reduced first, then an infinite loop is created.

LOG(F) [NARAIN 88] differs from the approaches above in that it rests upon subsuming rewriting and lazy evaluation within logic programming. In LOG(F), rewrite rules are compiled into equivalent Horn clauses. Hence, rewrite rules are interpreted as logical relationships between LHS and RHS, thereby providing a declarative interpretation to the rewrite rules. For example, the three LOG(F) rules

```
x => x.

y => y.

f(x) => y.
```

are represented as the following Horn clauses:

```
reduce(x, x).

reduce(y, y).

reduce(f(X), Y) :- reduce(X, x), reduce(y, Y).
```

The symbols `x` and `y` behave as constructors. Although the LOG(F) rewrite system offers declarative semantics as a general functional language, it has a restricted reduction strategy. In particular, the LOG(F) reduction strategy cannot immediately accommodate the deduction strategy of Prolog.

The approach used to combine functional programming and logic programming in FLOP has two major components. First, FLOP interprets each rewrite rule not only as a function specification but also as a kind of predicate. Namely, the {true, false} values of predicates are represented by the computational properties {reducible, not reducible}. This is similar to that of Prolog which uses {success, failure}. However, since reducible terms have a return value, the "truth value" of a predicate is generalized to permit arbitrary values.

Second, FLOP provides a highly flexible reduction strategy, namely, lazy reduction. As a functional language, the lazy reduction strategy is flexible enough to easily incorporate others, such as the eager reduction found in LISP. However, eager reduction cannot easily simulate lazy reduction. The deduction strategy of logic programming is naturally accommodated in the reduction strategy of FLOP. In particular, lazy reduction is flexible enough to accommodate the goal selection strategy of Prolog. Hence, logic programming is a special instance of functional logic programming.

# CHAPTER 2 - FLOP PROGRAMS

This chapter presents the syntax and the semantics of FLOP programs. A FLOP program is a set of rewrite rules. The reduction strategy is defined locally within each rewrite rule. The syntax of FLOP rewrite rules is defined below.

FLOP rewrite rules are by default lazy[1]. In other words, terms are not evaluated unless otherwise specified. This lazy default is overridden by two special operators, `?` and `&`. Their precise semantics are defined in **Section 2.2 (Semantics of FLOP Programs).**

## 2.1 Syntax of FLOP Programs

Many of the syntactic constructs of FLOP are identical to those of Prolog. Where this is the case, they are noted as such.

**Variables.** There is a countably infinite number of variables. The syntax of variables is identical to that of Prolog, including the anonymous variable, '_'.

**Function symbols.** For each i, $i \geq 0$, there is a countably infinite list of i-ary function symbols. The syntax of function symbols is identical to that of Prolog. In addition, two special operators, {`?`, `&`}, are defined. They are called the **eager operator** and **argument operator**, respectively.

The **connectives** are the set {=>, `(`, `)`, ','}.

The **terminator** is the period symbol, '.'.

A **term** is either a variable, or an expression of the form $f(t_1, \ldots, t_n)$, where $f$ is an n-ary function symbol, $n \geq 0$, and each $t_i$ is a term.

A FLOP program is a set of **rewrite rules**, where each rule is of the form:

LHS => RHS.

LHS and RHS are terms with the following restrictions:

1. LHS is not a variable.
2. The eager operator is not the functor of LHS . Other than that, it can be embedded anywhere in LHS or RHS.
3. If LHS is $f(t_1, \ldots, t_n)$, then the argument operator, if it exists, can only be the functor of terms $t_1, \ldots, t_n$. The argument operator is not allowed anywhere else.

[1]Some have pointed to a strong correlation between the extreme laziness of evaluation in FLOP and the personality of its author.

The following are examples of valid and invalid rules:

| | |
|---|---|
| `a => b.` | valid. |
| `f(X, Y, Z) => g(a(Z), b(Y), c(X)).` | valid. |
| `f(& x, Y) => g(? b, Y).` | valid. |
| `f(& ? x, Y) => ? g(b, ?Y).` | valid. |
| `f(X, ?Y, & ? Z) => t.` | valid. |
| `a => ? ? b.` | valid. |
| `X => y.` | invalid - violates rule #1. |
| `? l => r.` | invalid - violates rule #2. |
| `f(? & X) => t.` | invalid - violates rule #3. |

We now define some terminology which will be used in the following discussions. A term which contains no eager operator is called a **lazy term**; otherwise it is called a **busy term**, i.e., it contains at least one eager operator. A term whose functor is the eager operator is called an **eager term**. Given an eager term **T**, its immediate subterm is called the **eager argument of T**. For example, the eager parameter of the eager term `(? f(x,y))` is `f(x,y)`. Given a busy term **T**, the left-most and inner-most subterm that is an eager term is called the **eagerest term of T**. For example, suppose T is the busy term `(? f(? x, g(y, h(? z))))`, then its eagerest term is `(? x)`. If the eagerest term of a busy term **T** is **T** itself, then **T** is called a **simple eager term**. For example, `(? f(x))` is a simple eager term, while `(? f(? x))` is not.

## 2.2 Semantics of FLOP Programs

In describing the semantics of FLOP programs, three key terms, **lazy reduction**, **reduction**, and **LHS-matching**, must be defined. These terms as they are mutually recursive.

First, a general overview is in order. A FLOP program consists of a set of rewrite rules. Informally, a rewrite rule states that if a term *matches* with the LHS of a rewrite rule, then that term can be rewritten, or replaced, with another term, where the replacement term is defined by the RHS of the rule. This process of rewriting a term using a rewrite rule is called **reduction**. In FLOP, all reductions are lazy by default. That is, a term is not rewritten unless specifically required by either the eager operator or the argument operator. The eager operator specifies that the eager term should be replaced with the reduced value of its eager argument. When there are multiple eager terms, the eagerest term is reduced first. This process of rewriting a term by successively replacing each eagerest term is called **lazy reduction**.

### 2.2.1 Lazy Reduction and the Eager Operator: ?

Let **P** be a FLOP program and **T** a term which may be either busy or lazy. **T** is **Lazy reducible** in **P** if one of two conditions are true:

1. **T** is a lazy term. Furthermore, **T** is said to be a **lazy reduced value** of **T** in **P**.

2. Let **T'** be an identical copy of **T**, except the eagerest term of **T** is replaced with the reduced value of its eager argument. If this eager argument is not reducible, then **T** is **not lazy reducible** in **P**. Otherwise, **T** is **lazy reducible** if and only if **T'** is **lazy reducible**. Furthermore, the **lazy reduced value** of **T** is recursively defined to be the **lazy reduced value** of **T'**.

Note that this definition implies that, if **T** is a lazy term, then the reduction of **T** is equivalent to the lazy reduction of (`?` **T**).

In essence, the eager operator is used to control the inside-out reduction strategy of a term. The following example demonstrates step by step the reduction order of lazy reduction of a term with multiple eager operators:

| | |
|---|---|
| `? a(? b(? c, ? d), f, ?g)` | initial term |
| `? a(? b(C, ? d), f, ?g)` | `C` = reduced value of `c` |
| `? a(? b(C, D), f, ?g)` | `D` = reduced value of `d` |
| `? a(B, f, ? g)` | `B` = reduced value of `b(C, D)` |
| `? a(B, f, G)` | `G` = reduced value of `g` |
| `A` | `A` = reduced value of `a(B, f, G)` |

An eager argument may also be an eager term, i.e., multiple eager operators may be nested as in the case `f(? ? g)`. The semantics of this form is consistent with what has been presented so far. Namely, in the above case, the term `g` is reduced to produce `(? g)`; then, the resulting value is reduced once again for `(? ? g)`.

## 2.2.2 LHS-Matching and the Argument Operator: &

**LHS-matching** is a generalization of head unification in Prolog. When a lazy term **T** is being reduced, only those rules whose LHS *matches* with **T** are used to rewrite **T**. Let T be $f(t_1, \ldots, t_n)$ and LHS be $f(s_1, \ldots, s_n)$, where `f` is a functor of arity n≥0, and each $t_i$ is a term, for 0≤i≤n. Of course, LHS must meet the syntactic requirements described in the previous section. Henceforth, $t_i$'s will be called **actual arguments** and $s_i$'s **formal parameters.** T and LHS match if the following conditions are met:

> **LHS-reduction.** The LHS-reduction of **T** = $f(t_1, \ldots, t_n)$ and LHS = $f(s_1, \ldots, s_n)$ are **T'** = $f(t'_1, \ldots, t'_n)$ and LHS' = $f(s'_1, \ldots, s'_n)$, where for each $t_i$ and $s_i$, the following conditions are met
>
> > **AA-reduction.** If the functor of $s_i$ is the argument operator, `&`, then let $t'_i$ be the reduced value of $t_i$ and $s'_i$ be the immediate subterm of $s_i$ . Otherwise, let $t'_i$ be $t_i$

and `s'`$_i$ be `s`$_i$ . If the functor of `s`$_i$ is the argument operator, but `t`$_i$ is not reducible, then LHS-matching fails.

**FP-reduction.** Let `s"`$_i$ be the lazy reduced value of `s'`$_i$ . If `s'`$_i$ is not lazy reducible, then LHS-matching fails.

**LHS-matching. T** and LHS **match** if their LHS-reductions **T'** and LHS' are unifiable..

Note that if there are neither argument operators nor eager operators in LHS, then LHS-matching is equivalent to head unification in Prolog.

The eager operator, `&`, is used to control the outside-in reduction strategy. This operator indicates whether the actual arguments should first be reduced prior to LHS-unification.

### 2.2.3 Reduction

Let **P** be a FLOP program and **T** a lazy term. **T** is said to be **reducible in P**, if the following conditions are met:

1. **T** is not an unbound variable.
2. There is a rule **R** in **P** whose LHS matches **T** and whose RHS is lazy reducible. In this case, the lazy reduced value of RHS is the **reduced value** of **T** by rule **R**.

Henceforth, the terms **reduction** and **evaluation** are used interchangeably. The terms **reduced value, evaluated value** and **return value** are also used interchangeably. Furthermore, where it is clear in the context, the phrase "**by rule R**" will be dropped.

**Chapter 6 (Implementing FLOP In Prolog)** contains a short FLOP interpreter written in Prolog. It provides a concrete implementation of the definitions in this section.

### 2.2.4 Examples of Reduction

This section provides some examples to demonstrate the principles defined in the previous sections. Consider the following FLOP program:

| # | rule |
|---|---|
| 1. | `thomas => tom.` |
| 2. | `likes(mary) => john.` |
| 3. | `likes(& tom) => mary.` |
| 4. | `loves(X) => X.` |
| 5. | `hates(? thomas) => john.` |

This program produces the following return value for each term:

| term | reduced value |
|---|---|
| `thomas` | `tom` |
| `likes(mary)` | `john` |
| `tom` | **fail** |

The reduction of the term `thomas` succeeds with the rule 1 and `likes(mary)` with rule 2. However, the reduction of `tom` fails because there is no rule for which LHS-matching is successful.

LHS-unification is attempted after AA-reduction:

| term | reduced value |
|---|---|
| `likes(thomas)` | `mary` |
| `likes(tom)` | **fail** |

The first case `likes(thomas)` succeeds because the argument operator in rule 3 signifies that `thomas` should first be reduced before LHS-unification is attempted, after which LHS-matching is successful. However, the second case, `likes(tom)`, fails because the argument `tom` cannot be successfully reduced. Therefore, the reduction for the whole term `likes(tom)` also fails.

The variable bindings made in LHS-matching are visible in the RHS, as demonstrated by the following:

| term | reduced value |
|---|---|
| `loves(anything)` | `anything` |

This term succeeds with rule 4, in which the variable `x` is bound to `anything` as a side effect of LHS-matching. Therefore, the return value is the bound value of `x`, `anything`.

LHS-unification is attempted after FP-reduction. Consider the following results:

| term | reduced value |
|---|---|
| `hates(tom)` | `john` |
| `hates(thomas)` | **fail** |

The first term succeeds with rule 5, as FP-reduction modifies `(? thomas)` to `tom` before LHS-unification. Hence, LHS-matching is successful for `hates(tom)`, but not for `hates(thomas)`.

FP-reduction allows abbreviation of multiple rules in a single rewrite rule. Consider the following FLOP program:

| # | rule |
|---|---|
| 1. | `group => tom.` |
| 2. | `group => jerry.` |
| 3. | `likes(?group) => bugs.` |

In this example, rule 3 can be viewed as an abbreviation of the two rules

```
likes(tom) => bugs.
likes(jerry) => bugs.
```

Since the term `group` is reduced during runtime, introducing additional members to the `group` does not invalidate the rule. On the other hand, if an unabbreviated form is used, a corresponding rule must be added for each additional member of the `group`.

If the eager operator is present in RHS, the return value is the the lazy reduced value of RHS. If any of the subterms marked by the `?` operator cannot be reduced, then reduction with this rule fails. Here is a FLOP program to demonstrate this point:

| # | rule |
|---|---|
| 1. | `thomas => tom.` |
| 2. | `jane => bimbo.` |
| 3. | `likes(?thomas) => mary.` |
| 4. | `likes(john) => ? thomas.` |
| 5. | `likes(?jane) => ? likes(?thomas)` |
| 6. | `likes(mary) => ? tom.` |

This program produces the following results:

| term | reduced value |
|---|---|
| `likes(john)` | `tom` |
| `likes(mary)` | **fail** |
| `likes(bimbo)` | `mary` |

The term `likes(john)` succeeds with rule 4. Note that the return value of the term is the lazy reduced value of RHS, (`? thomas`), which is `tom`. However, reduction of `likes(mary)` fails. Although LHS-matching is successful with rule 6, lazy reduction of RHS fails.

The trace of the final example, `likes(bimbo)`, is as follows:

```
•  term to reduce: likes(bimbo)
   rule chosen: likes(?jane) => ?likes(?thomas)
      •  reduce LHS subterm marked by the eager operator
            •  term to reduce is jane
               rule chosen: jane => bimbo
            √  return bimbo
      •  modified LHS becomes likes(bimbo)
      •  LHS-matching successful.
      •  reduce RHS subterms marked by the eager operator
            •  term to reduce: thomas
               rule chosen: thomas => tom
            √  return tom
            •  term to reduce: likes(tom)
               rule chosen: likes(?thomas) => mary
                  •  reduce LHS subterm marked by the eager
                     operator
                        •  term to reduce: thomas
                           rule chosen: thomas => tom
                        √  return tom
                  •  modified LHS becomes likes(tom)
                  •  LHS-matching successful.
                  √  return mary
            √  return mary
      •  modified RHS becomes mary
√  return mary
```

### 2.2.5 Non-determinism in FLOP

The first step in LHS-matching is the selection of a rewrite rule. The non-determinism of this selection process, as in Prolog, is implemented with backtracking. Therefore, the order of rewrite rules defined in a FLOP program is significant in determining the order of the results. For example, the FLOP program

```
likes(tom) => mary.
likes(mary) => thomas.
```

produces the following results for each term:

| term | first result | second result | third result |
|---|---|---|---|
| `likes(X)` | `mary` | `tom` | **fail** |
| `likes(tom)` | `mary` | **fail** | |

## 2.3 Interactive Environment

This section describes the top level interactive environment. As in Prolog, the interaction is query based. Henceforth, when traces of queries are presented, the user inputs are displayed in **bold** font, and system responses in `regular` font. The default prompt is the '`| ?=`' symbol. Each query has the form

```
| ?= term.
```

where **term** may be either busy or lazy. However, the argument operator (`&`) is not allowed.

When a term is entered, the system responds as follows. First, an extra eager operator is embedded in front of the **term** to produce (`?` **term**) which is then lazy reduced. If the lazy reduction fails, the system prints `reduction unsuccessful` and produces a new prompt. If the lazy reduction is successful, then the return value is printed along with any bindings made during the process.

Once the return value is displayed, the system waits for a response. As in Prolog, a carriage return results in the termination of the query and a new prompt is produced. Typing a semicolon, '`;`', results in backtracking in search of another solution. Consider the program:

```
likes(tom) => mary.
likes(mary) => thomas.

thomas => tom.
```

The following is a trace of an interactive session with the above program:

```
| ?= likes(tom).
mary ;
reduction unsuccessful

| ?= likes(WHO).
mary
  WHO=tom ;
thomas
  WHO=mary ;
reduction unsuccessful

| ?= ? likes(? thomas).
mary ;
reduction unsuccessful
```

## 2.4 Some Useful FLOP Functions[1]

This section presents some useful FLOP functions. All of these functions are implemented as rewrite rules, as there are no built-in functions defined in FLOP.

One fundamental function in logic programming is unification. In FLOP, as in Prolog, the = operator unifies two terms:

```
(Rslt = Rslt) => Rslt.
```

This function simply unifies its left and right arguments and returns the resulting unified term. The use of this rule is straight forward:

```
| ?= f(X, y) = f(a, B).
f(a, y)
  X=a
  =y ;
reduction unsuccessful
```

So far, only reductions of ground terms have been presented. In other words, input parameters of functions have been fully instantiated. However, this is not a requirement in FLOP, as function parameters maybe unbound variables. When this is done, the system attempts to satisfy the query by binding the variables with appropriate values. For queries of this type, the following function is useful:

```
(& Rslt => Rslt) => Rslt.
```

This function unifies its right argument with the reduced value of its left argument. The return value of this function is the unified term. For example, the following query

```
| ?= likes(WHO) => mary.
```

succeeds if there exists a term unifiable with `likes(WHO)` which reduces to `mary`. If such a term does exist, in this particular case, the return value will be the atom `mary`. Furthermore, as a side effect, the variable `WHO` will be bound to the appropriate term.

Consider the following rewrite rule

```
certain_type(OBJECT) => ? (length(OBJECT) => small).
```

Note that the first => symbol is a function specifier which separates LHS and RHS of the rule. The second => is an operator within RHS of the rule. This rule states that an `OBJECT` is of `certain_type` if its `length` is `small`.

[1]In context of FLOP, the word "function" is used loosely. Normally, functions refer to relationships that map each element in the domain to a unique element in the range. In general, FLOP allows mapping to more than one element in the range. True functional relationships can be implemented by including only deterministic rules in a program.

Also, there may be eager operators embedded within its arguments. The semantics of such expression is consistent with what has been presented so far. For example, the query

```
| ?= color(? strange_material) => ? reddish_colors.
```

looks for a `strange_material` whose `color` is one of the `reddish_colors`. The order of evaluation is left-to-right and inside-out. Therefore, first the term (`? strange_objects`) is replaced by its return value, say a `moon_rock`. Second, the term (`? reddish_color`) is replaced by its return value, say `pink`. Then, the resulting term (`color(moon_rock) => pink`) is evaluated.

As another example, consider the following implementation of `append` in FLOP:

```
append([], L) => L
append([X|Xs], Ys) => [X|? append(Xs, Ys)]
```

The following trace demonstrates that the behavior of this function is fundamentally identical to that of Prolog:

```
| ?= append([a], [b]).
[a,b] ;
reduction unsuccessful

| ?= append(A, B) => [a,b].
[a,b]
  A=[]
  B=[a,b] ;
[a,b]
  A=[a]
  B=[b] ;
[a,b]
  A=[a,b]
  B=[] ;
reduction unsuccessful
```

Once again, the first query is one of reducing a ground term. The query is successful and the return value, `[a,b]`, is printed. However, the second query is not ground. As in the first query, the return value is `[a,b]`, but there are additional side effects of variable bindings. The various bindings of the input variables `A` and `B` represent the solution space for the query.

Although the two functions, = and =>, are similar, in the current implementation of FLOP, there is a subtle difference in their behavior. Consider the trace of the following two queries:

```
| ?= append(A, B) => [].
[]
  A=[]
  B=[] ;
reduction unsuccessful

| ?= ? append(A, B) = [].
[]
  A=[]
  B=[] ;
|Out of global stack during execution.
% Execution aborted.
```

The first query behaves as expected. However, the second one falls into an infinite loop.

This is due to a subtle difference of when the unification with the `[]` term is applied. There are two main components to these two queries, reduction of the `append` term and unification of the return value with `[]`. In the = version, these two components are completely separate. That is, first the `append` term is reduced and the return value is bound to a temporary variable. Then, that term is unified with `[]`. On the other hand, in the => version, reduction of the `append` term is done within the context of reducing the => term. As a result, unification with `[]` is done at an earlier point, during the process of reducing the `append` term. Hence, the search tree is pruned at a much earlier stage, thereby preventing the infinite loop. The relationship between the two queries above is analogous to the two Prolog queries:

```
| ?- append(A, B, []).
  A=[]
  B=[] ;
no

| ?- append(A, B, C), C=[].
  A=[]
  B=[] ;
|Out of global stack during execution.
% Execution aborted.
```

Another useful set of FLOP rewrite rules are conditional functions. The simplest version is the following:

```
if_then(& _, Rslt) => Rslt.
```

In this rule, if the first argument is reducible, then the return value is the second argument, unevaluated. Another possibility is to evaluate the second argument, as in the next example:

```
if_ethen(& _, Rslt) => ? Rslt.
```

In this rule, if the first argument is reducible, then the return value is the reduced value of the second argument.

The next set of rules represents *functional* conjunction (',') and disjunction (';'). The term functional here refers to the fact that the return values of these logical operators may be arbitrary terms.

```
(&_ , &Rslt) => Rslt.
```

Here, if both arguments are reducible, then the return value is the reduced value of the right argument. If either one of the arguments is not reducible, then this reduction fails.

```
(&Rslt ; _) => Rslt.

(_ ; &Rslt) => Rslt.
```

This is a functional disjunction, where only one of the arguments need be reducible. The return value is reduced value of the reduced argument.

The following rewrite rules simulate some of the well known LISP functions. **Chapter 5 (LISP and FLOP)** provides a more in depth discussion of these two systems.

```
car(& [X | _]) => X.

cdr(& [_ | Xs]) => Xs.

cons(& X, & Xs) => [X | Xs].

eval(& X) => ? X.

quote(X) => X.
```

The `&` operator precedes each formal parameter, as LISP normally evaluates actual arguments. The `quote` operator is a special exception to this rule.

# CHAPTER 3 - FUNCTIONAL LOGIC PROGRAMMING

FLOP integrates functional programming and logic programming coherently into a single environment. We call this **functional logic programming.** This integration is accomplished by incorporating into a single system the essential features of logic programming, namely, unification and backtracking, with a flexible reduction strategy.

When a rule for conjunction is introduced and only predicate functions are allowed, FLOP amounts to SLD-resolution. In other words, SLD-resolution is a special case within the FLOP rewrite system. In general, FLOP allows the expressiveness of functional notation within the framework of logic programming.

## 3.1 SLD-Resolution in FLOP

Implementing SLD-resolution in FLOP is straightforward. First, note that the domain of SLD-resolution is predicates defined by Horn clauses. Therefore, all rewrite rules are restricted to a form equivalent to Horn clauses as described below. An arbitrary atom, say `true`, is designated as the return value of a successful goal.

For each unit Horn clause of the form

$f(t_1, \ldots, t_m).$

we introduce a FLOP rule of the form:

$f(t_1, \ldots, t_m) \Rightarrow true.$

Also, for each non-unit Horn clause of the form

```
h :- g1, ..., gn.
```

we introduce a FLOP rule of the form:

```
h => ? (g1, ..., gn).
```

Finally, a mechanism for simulating conjunction must be devised. This is achieved by the following rule:

```
(& true , & true) => true.
```

Here, both arguments must be reducible and their return values must be `true`. If this condition is met, then the return value of the conjunction is also `true`. Otherwise, the goal fails.

Consider the following `append` and `reverse` relationships expressed as Horn clauses in FLOP notation:

```
append([], L, L) => true.
append([X|L1], L2, [X|L3]) => ? append(L1, L2, L3).

reverse([], []) => true.
reverse([X|L], R) => ? (reverse(L, LR), append(LR, [X], R)).
```

Their behavior is identical to those implemented in Prolog:

```
| ?= reverse([a,b,c], R).
true
 R =[c,b,a]

| ?= reverse(S, [c,b,a]).
true
 S=[a,b,c]
```

It is worth noting that conjunction is not a primitive function in FLOP. Conjunction can be correctly implemented with the matching and reduction primitives; however, this is not true in Prolog.

## 3.2 Combining Functional and Logic Programming

One of the restrictions in the previous section is that all rewrite rules must be defined as predicates. Upon all successful reductions, the return value must be the atom `true`. What are the ramifications if this restriction is relaxed, so that a successful goal can return arbitrary values? One interpretation of this relaxation is that the truth value of a predicate can now be represented by arbitrary terms. Hence, a FLOP rewrite rule serves a dual purpose, one as a logical relationship and the other as a functional specification. This is the foundation of functional logic programming.

Many relationships are more naturally expressed in functional notation than in predicate form. Implementing them in predicate form generally requires introducing additional temporary variables, which often result in expressions that are less intuitive. For example, in the second clause of the `reverse` predicate above, the variables `LR` is a temporary variable. Notational limitations of predicate logic require it. However, these temporary variables can be eliminated when expressed in functional notation. The two functions above can be expressed in FLOP as follows:

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

reverse([]) => [].
reverse([X|L]) => ? append(? reverse(L), [X]).
```

Note that no temporary variables are needed. As a result, functional notation provide a more comprehensible expression than that expressed in predicate form. Furthermore, the behavior of the function version is fundamentally identical to that in predicate form:

```
| ?= reverse([a,b,c]).
[c,b,a]

| ?= reverse(S) => [c,b,a].
[c,b,a]
 S=[a,b,c]
```

With the relaxation of the return values of predicates, the conjunction rewrite rule presented above must also be generalized to accommodate arbitrary truth values. One possibility is

```
(& _ , & _) => true.
```

Here, the comma function returns `true` if both arguments are reducible. The restriction that the return value of its arguments must be the `true` atom has been eliminated; now, their actual return values are unrestricted.

This rule can be further generalized, since the conjunction function, too, can return arbitrary values. One possibility is to choose the reduced value of the second argument:

```
(& _ , & Rslt) => Rslt.
```

Here is a demonstration of its use:

```
| ?= (reverse([x,y]) => R), append(R, R).
[y,x,y,x]
 R=[y,x]
```

Note that the return value of the conjunction operator is the reduced value of the right argument, which in this case is `append(R, R)`. However, prior to evaluating this term, the left argument is evaluated which has the effect of binding `R` to `[y,x]`.

# CHAPTER 4 - LISP AND FLOP

This chapter compares two functional programming languages, LISP and FLOP. Although both employ functional notations for program specification, there are a number of differences worth noting. Also, a small LISP-like system in FLOP is presented.

The fundamental difference between FLOP and LISP stems from the features of logic programming incorporated in FLOP. FLOP offers both declarative and procedural semantics, while LISP only procedural. As a result, LISP programs are strictly uni-directional. For each function, there are predefined input and output parameters and their roles are not interchangeable. FLOP's declarative semantics allow greater flexibility. Also, LISP functions are purely deterministic. On the other hand, FLOP is a non-deterministic system in general, although deterministic behavior can be obtained where needed. (See **Chapter 6 (Implementing FLOP in Prolog)** for a complete description.) Many runtime errors which produce interrupts in a LISP system, such as attempting to evaluate an undefined atom, are handled uniformly by the backtracking mechanism in FLOP.

In terms of their reduction strategies, FLOP is lazy by default, whereas LISP is eager. That is, in FLOP terms are by default not evaluated, but in LISP they are. Actually, this previous statement is somewhat misleading, as FLOP employs logical variables and LISP symbolic variables. When logical variables are bound to a term, they become indistinguishable from the term. In the framework of LISP programming, logical variables can be viewed as being automatically evaluated whenever they are bound. Symbolic variables, on the other hand, are indistinguishable from atoms. Therefore, the user must explicitly protect each value from being treated like a variable by quoting it. Since logical variables cannot be confused with terms (values), the `quote` operator need not be a primitive function in FLOP.

For logical variables, all bindings are accomplished through the pattern matching of unification. As a result, destructive assignments, such as provided by LISP's `setq`, is not easily simulated in FLOP. However, the absence of this feature is not necessarily a negative one.

## 4.1 A LISP-like System In FLOP

As a demonstration of FLOP's reduction strategy, we present a small LISP-like system, where functions are defined using FLOP rewrite rules. Certain syntactic restrictions are placed on these rules so that LISP's evaluation strategy is reproduced. First, a set of FLOP rewrite rules which represents the primitive functions of our LISP-like system are presented. For notational clarity, generalized structures, not simple lists, are used to distinguish the functor and its arguments.

The first set of functions, `quote` and `eval`, control the default evaluation strategy. The `!` symbol is used to represent the `quote` operator, although the "'" symbol is traditionally used in LISP. The `!` operator returns the unevaluated argument, and the `eval` operator evaluates its argument twice.

```
(!X) => X.

eval(& X) => ? X.
```

Note that, even when they are nested, these two functions behave exactly as they do in LISP, as demonstrated by the following example:

```
| ?= eval(! ! term).
term
```

There is a fundamental difference between the eager operator and the `eval` function. The precedence of the eager operator is inside-out. Hence, all terms marked by the eager operator are always reduced. On the other hand, the precedence of the `eval` function is outside-in. Hence, terms marked by the `eval` operator is evaluated only on demand. The following trace demonstrates this point.

```
| ?= ! eval(! term).
eval(! term)

| ?= ! ? ! term.
term
```

Next is the primitive list functions, `car`, `cdr` and `cons`. Prolog list constructors are used to represent lists. All of these functions evaluate their arguments, as signified by the presence of the argument operators.

```
car(&[X|Xs]) => X.

cdr(&[X|Xs]) => Xs.

cons(&X, &Xs) => [X|Xs].
```

Here is a demonstration of their usage:

```
| ?= car([a,b,c]).
reduction unsuccessful

| ?= car(![a,b,c]).
a

| ?= cdr(![a,b,c]).
[b,c]

| ?= car(cdr(![a,b,c])).
b

| ?= cons(car(![a,b,c]), cdr(![x,y,z])).
[a,y,z]
```

The following operators, `equal` and `true`, represent predicates in this system. Others are easily added. LISP predicates return the atom `T` for true and `nil` for false. In this system, predicates are implemented as FLOP predicates. That is, if a predicate is true, then it succeeds with the return value of the atom `true`; otherwise, it fails.

```
equal(&X, &X) => true.

true => true.
```

The `equal` operator succeeds if the evaluated forms of the two arguments are unifiable. Like the LISP atom `T`, the `true` predicate always succeeds. It can be used as the default case in the `cond` function described below.

Next is the `cond` function.

```
cond([(Cond, Rslt) | _]) => ? (Cond, Rslt).
cond([_ | More]) => ? cond(More).
```

The `cond` function takes a list as its argument, where each element of the list is of the form `(Cond, Rslt)`, where `Cond` is a predicate, and `Rslt` the corresponding return value. The `cond` function returns the evaluated form of the first `Rslt` whose corresponding `Cond` predicate succeeds. The `cond` function fails if there are no `(Cond, Rslt)` pairs that are both reducible.

The following demonstrates the use of all of the above functions:

```
| ?= cond([]).
reduction unsuccessful

| ?= cond([(equal(cdr(![a,b,c]), ![]), !incorrect)
            (equal(cdr(![a,b,c]), ![b,c]), !correct)
            (equal(car(![a,b,c]), !a), !too_late)]).
correct

| ?= cond([(equal(cdr(![a,b,c]), ![]), !incorrect)
            (true, !the_default)]).
the_default
```

We now describe the `apply` function which evaluates a lambda expression with a set of arguments. This function takes two arguments, a `Lambda` expression and an argument list, `AAList`.

```
apply(& lambda(FPList, Body), & AAList) => ? (
            apply_list(AAList, FPList),
            Body
      ).
apply(& Functor, & AAList) => ? (
            (=..(Goal) => [Functor|AAList]),
            Goal
      ).
```

```
apply_list([], []) => true.
apply_list([AA|AAList], [FP|FPList]) => ? (
            (AA=>FP),
            apply_list(AAList, FPList)
      ).
```

The `Lambda` expression cannot be an unbound variable (first clause). Other than that, it can be of two forms. First, (second clause) it can be of the form `lambda(FPList, Body)`, where `FPList` is the formal parameter list and `Body` is the body of the lambda expression. To adhere more closely to a true LISP system which allows only simple variables, `FPList` should be a list of unbound variables. However, this is not enforced and arbitrary terms may be used for unification. With this form, each element of the `AAList` is evaluated and unified with the `FPList`. This is done by the `apply_list` function. Note that if any of the arguments is not reducible, or if `FPList` and `AAList` have different lengths, evaluation of the `apply` term fails. Another form of `Lambda` expression is a FLOP atom (third clause), in which case it is assumed to be the `Functor` of a rule. Here, a `Goal` header is constructed using the `Functor` atom and the `AAList` and then evaluated. Here is an application of the `apply` function:

```
| ?= apply(!cdr, ![![a,b,c]]).
[b,c]

| ?= apply(!lambda([X,Y], cons(car(!X), cdr(!Y))),
            ![![a,b,c], ![x,y,z]]).
[a,y,z]
```

Note that in the body of the lambda expression, each variable is preceded by the `!` mark. This is necessary as these logical variables are automatically evaluated. Therefore, the `!` operator protects them from double evaluation.

We now present a method of defining additional LISP-like functions. These functions are implemented as rewrite rules, much in the way `car`, `cdr` and `cons` are defined above. In the function header (LHS of the rule), each formal argument is preceded by the argument operator to signify that the corresponding actual argument is to be evaluated. The function body (RHS of the rule) should be a simple eager term whose eager argument is composed of the primitive functions presented above or other functions defined in the present manner. Each variable referenced is preceded it by the `!` operator to prevent double evaluation. The following is a recursive `concat` function implemented in the manner just described:

```
concat(&X, &Y) => ?
      cond([(equal(!X, ![]), !Y),
            (true, cons(car(!X), concat(cdr(!X), !Y)))]).
```

Here is its execution:

```
| ?= concat(![a,b,c], ![x,y,z]).
[a,b,c,x,y,z]

| ?= apply(!lambda([X], concat(!X, !X)),
            ![![a,b,c]]).
[a,b,c,a,b,c]
```

An immediate benefit of implementing LISP functions as our LISP-like functions is the inherent declarative semantics of FLOP rewrite rules. For example, the `concat` function can now be executed in the framework of functional logic programming:

```
| ?= concat(!X, !Y) => [a,b].
[a,b]
 X=[]
 Y=[a,b] ;
[a,b]
 X=[a]
 Y=[b] ;
[a,b]
 X=[a,b]
 Y=[] ;
reduction unsuccessful
```

There are a number of differences between this LISP-like system and a true LISP system. First, LISP stores functions as lambda expressions, and function names evaluate to their corresponding lambda expressions. In our LISP-like system, they are represented as FLOP rewrite rules which have only similar behavior to the evaluation of lambda expressions. However, this, too, simulated by requiring each LISP-like functions be defined such that the function header is a simple atom and the body a lambda expression. Then, an additional layer of interpreter must `apply` the corresponding lambda expression to the arguments.

The point of this exercise, however, is not to demonstrate that each and every detail of LISP can be duplicated in FLOP, but rather that functional logic programming offers the essential features of functional programming, such as LISP, plus the declarative semantics of logic programming. As a practical point, it would be unwise to attempt to duplicate all the details of LISP within FLOP, for each paradigm demands different programming techniques and styles. For example, it is rather cumbersome and inefficient to precede each variable reference with the `quote` operator.

One of the fundamental components of LISP missing from the LISP-like version is the destructive assignment function, `setq`. However, reliance on destructive assignment is generally discouraged This is yet another situation for which different programming styles may be warranted for a procedural language, like LISP with `setq`, and a declarative language, like FLOP.

# CHAPTER 5 - LOG(F) AND FLOP

This chapter compares the two non-deterministic rewrite systems, FLOP and Sanjai Narain's LOG(F) [NARAIN 88]. **Appendix I** contains a synopsis of LOG(F). LOG(F) rests upon subsuming within logic programming, both rewriting and lazy evaluation. FLOP, on the other hand, is based on combining logic programming, rewriting, and lazy evaluation into a single environment. FLOP can be viewed as a proper extension of LOG(F), for LOG(F) programs can be simulated in FLOP, but it is not obvious how to perform the converse simulation. Furthermore, transforming LOG(F) rules into equivalent FLOP rules is straight forward.

## 5.1 LOG(F) in FLOP

In translating LOG(F) programs into equivalent FLOP programs, two distinct classes of LOG(F) function symbols must be considered. First is the set of *constructors*, which have the following properties:

1. For an n-ary constructor symbol, $n \geq 0$, no restrictions are permitted for any of its formal parameters.

2. Constructors are deterministic, as constructors cannot be the functor of LHS of a rule.

3. The actual arguments of constructors are not reduced.

Constructors are easily implemented in FLOP. For each constructor symbol c of arity n, where $n \geq 0$, introduce a FLOP rewrite rule of the following form:

```
c(C1, ..., Cn) => c(C1, ..., Cn).
```

The first property is satisfied as the formal parameters are simply unbound variables $C_1$, ..., $C_n$. The deterministic property is achieved as it is illegal to construct a rewrite rule whose LHS is a constructor symbol. Finally, the actual arguments of this reduction rule are not reduced, as no & operators are used.

The second kind of function symbols introduce the LHS of some LOG(F) rule. LOG(F) reductions are like FLOP reductions, but with two important differences:

1. Actual arguments are always reduced first when their corresponding formal parameters are non-variables (in which case the formal parameters must be constructors, with zero or more variable arguments).

2. Reductions terminate when a RHS of a rule obtained in the reduction (including the bindings made during LHS-matching) is in *simplified*, i.e., its function symbol is a constructor.

The procedure for converting a LOG(F) reduction rule into an equivalent FLOP rule is as follows. For each LOG(F) reduction rule of the form:

```
f(s1, ..., sm) => rhs.
```

for $m \geq 0$, and where $s_i$ and `rhs` are arbitrary terms, introduce a FLOP rewrite rule of the form:

```
f(t1, ..., tm) => ? rhs.
```

where if $s_i$ is a variable, then $t_i$ is same as $s_i$; otherwise, $t_i$ is `(& si)`.

The first distinction between LOG(F) and FLOP reductions is implemented by the presence of the `&` operator for each argument. The second distinction is implemented as follows. If `rhs` is not simplified, then all of RHS is reduced by the `?` operator. On the other hand, if `rhs` is simplified, the implementation of constructor symbols described previously terminates further reduction of this term.

The two resulting Prolog programs, one compiled with the basic algorithm presented in [NARAIN 88] from LOG(F) to Prolog, and the other from LOG(F) to FLOP and then from FLOP to Prolog, produce identical code.

## 5.2 The Differences

There is a fundamental difference between the intended goals of FLOP and LOG(F). FLOP is intended to be a general rewriting environment in which various reduction strategies can easily be implemented. On the other hand, LOG(F), or more specifically F*, is a special rewrite system which provides a number of desirable properties, such as confluence and completeness within its framework. In order to attain these properties, however, a number of restrictions are placed on its rewrite rules and reduction strategy.

A major component of LOG(F) eliminated in FLOP is the notion of constructor symbols. Constructors serve several crucial functions in LOG(F):

1. All values returned by LOG(F) reductions are simplified, i.e., their function symbols are constructors .

2. LOG(F) requires that the formal parameters be constructors or variables. LHS-unification succeeds by reducing actual arguments to simplified terms where necessary.

3. Constructors serve as the mechanism for supporting LOG(F)'s lazy evaluation. Constructors are the analogues of LISP's `quote` operator in protecting their subterms from evaluation.

LOG(F)'s strong dependence on constructors imposes several limitations:

1. Values returned by reduction must use the globally pre-defined set of constructor symbols.

2. Once defined, a constructor cannot be further reduced.

3. Control of lazy evaluation is limited. Only subterms of constructor symbols are protected from evaluation. All other terms are evaluated.

In FLOP, the functionality of constructor symbols has been replaced with a reduction strategy that is by default lazy, and the two operators, ? and &, which override this default. With the removal of constructors, the return value of a reduction need no longer be restricted, and reduction termination is locally controlled within each rewrite rule. Therefore, the evaluation of a term is context dependent, not globally constrained.

FLOP's local control of reduction termination and the eagerness of evaluation affords greater potential for lazy evaluation. In FLOP, all terms are treated lazily by default evaluation mode and are evaluated only on demand. In LOG(F), all terms that are not subterms of constructors must be evaluated.

The implementation of LOG(F) presented in [NARAIN 88] provides one mechanism for modifying the default reduction strategy, namely *eager evaluating*. As with constructor symbols, a special set of terms are globally designated as being *eager*. For each eager term, the programmer is required to provided a Prolog clause which computes the return value of that term. When these eager terms are detected at compile time, a Prolog goal for computing them is inserted in front of the goal for the reduction of RHS. Therefore, the eager mechanism in LOG(F) serves a dual purpose: to override the default reduction strategy; and to interface with the host Prolog environment.

There are three disadvantages to this scheme. First, there is no local control of eager reduction. That is, once a term is designated as eager, it is always eagerly evaluated. In FLOP, the eagerness of a term is defined locally. Hence, a term may be treated as eager or lazy depending on the context. Second, since LOG(F)'s eager terms are dependent on their Prolog substitution at compile time, modification of their definition requires recompilation of all rules containing those terms. The scheme used in FLOP eliminates this need (see **Chapter 6 (Implementing FLOP in Prolog)**). Finally, within each LOG(F) rewrite rule, all Prolog goals are eagerly evaluated. However, in FLOP, Prolog goals, too, are also subject to lazy evaluation.

# CHAPTER 6 - IMPLEMENTING FLOP IN PROLOG

This chapter describes a strategy for implementing FLOP within Prolog. First, a FLOP interpreter written in Prolog is presented. Second, an algorithm for compiling FLOP programs into equivalent Prolog programs is described. Third, a meta-syntactic construct for FLOP rewrite rules is presented. These are used to inject optional Prolog goals directly into the compiled code. This mechanism is used to provide access of the host Prolog environment within each rewrite rule.

## 6.1 FLOP Interpreter in Prolog

The following is a Prolog interpreter of FLOP programs. This interpreter properly simulates the behavior of FLOP programs, as defined in **Chapter 2 (FLOP Programs)**. It assumes that FLOP rewrite rules are stored in the Prolog data base with the =>/2 predicates.

```
reduce(Term, Rslt) :-
      var(Term),
      !, fail.
reduce(Term, Rslt) :-
      red_rule(Term, Rslt).

red_rule(Term, Rslt) :-
      (LHS => RHS),
      LHS  =.. [F | FPList],
      makelist(FPList, AAList),
      Term =.. [F | AAList],
      lhs_matching(AAList, FPList),
      lazy_reduce(RHS, Rslt).

lazy_reduce(Term, Rslt) :-
      var(Term),
      Term = Rslt, !.
lazy_reduce(? Term, Rslt) :- !,
      lazy_reduce(Term, RTerm),
      reduce(RTerm, Rslt).
lazy_reduce(Term, Rslt) :-
      Term =.. [F | AList],
      makelist(AList, RAList),
      Rslt =.. [F | RAList], !,
      lazy_reduce_list(AList, RAList).

lazy_reduce_list([], []) :- !.
lazy_reduce_list([A | AList], [RA | RAList]) :-
      lazy_reduce(A, RA),
      lazy_reduce_list(AList, RAList).
```

```
28:   lhs_matching([], []) :- !.
29:   lhs_matching([AA | AAList], [FP | FPList]) :-
30:         match_one_arg(AA, FP),
31:         lhs_matching(AAList, FPList).

32:   match_one_arg(AA, FP) :-
33:         var(FP),
34:         AA = FP, !.
35:   match_one_arg(AA, & FP) :- !,
36:         reduce(AA, RAA),
37:         lazy_reduce(FP, RAA).
38:   match_one_arg(AA, FP) :-
39:         lazy_reduce(FP, AA).

40:   makelist([], []) :- !.
41:   makelist([_ | Xs], [_ | Ys]) :-
42:         makelist(Xs, Ys), !.
```

At this point, some comments are in order. The cuts (`!`'s) on lines 3, 15, 16, 34, and 35 are essential to the program. For example, the cut in line 3 prevents backtracking after a successful completion of the first `reduce` clause. Otherwise, since the success of the first clause implies that `Term` is an unbound variable, `Term` would unify with any term in the first argument of the second `reduce` clause (line 4). The remaining cuts are "green cuts", i.e., the semantics of the program are not changed by their presence.

Lines 1 to 5 define the `reduce` predicate. This is the entry point of the reduction process. The first parameter corresponds to the term being reduced. The return value is unified with the second parameter.

Lines 6 to 12 defined the `red_rule` predicate which handles the actual reduction of each term using FLOP rewrite rules. This predicate assumes that `Term` is not a variable, as this condition is eliminated by the reduce predicate.

Lines 13 to 27 define the `lazy_reduce` and `lazy_reduce_list` predicates, the heart of the FLOP reduction engine. Here the input term in the first parameter is recursively traversed. If the term is an unbound variable (first clause, line 13), no reduction is applied. If the eager operator is detected (second clause, line 16), `lazy_reduce` is first applied to its subterm, then the resulting term is reduced. If the eager operator is not detected (third clause, line 19), `lazy_reduce` is applied to subterms; however, the resulting term is not further reduced. The order of traversal determines the order of precedence in reducing a term. Here, *postorder* traversal correctly implements the left-to-right and inside-out evaluation order of FLOP.

The goal `makelist` in line 21 is required, because when both `Rslt` and `RAList` (lazy reduced argument list) is unbound, the `=..` predicate fails. Therefore, `makelist` predicate, defined in lines 40 to 42, simply creates a list of unbound variables whose length is equal to `AList` (list of arguments in their original form).

In lines 28 through 39, `lhs_match` and `match_one_arg` predicates are defined. These routines handle LHS-matching. In `match_one_arg`, if the argument operator is detected, `reduce` is applied to the actual argument. In either case, `lazy_reduce` is applied to each formal parameter.

## 6.2 Compiling FLOP to Prolog

This section describes a method of transforming FLOP programs into equivalent Prolog programs.This method is based on the process of *partial evaluation*. The basic idea behind partial evaluation is the following. Given a program and an interpreter for that program, many of the decisions required by the interpreter can be handled prior to the actual execution of the program. These are the goals, or procedures, which depend solely on the structure of the program, and not the runtime environment. When these goals are detected, they may be textually replaced by their subgoals. When all of the computation of a goal can be done *a priori*, that goal can be completely eliminated. Through this process the original program can be expanded to include appropriate pieces of the interpreter code, thereby compiling the program from its original language to the language in which the interpreter is written.

We can apply partial evaluation of the interpreter of the previous section. The first step is to locate the point where the FLOP rewrite rules becomes defined first. Line 7 is used to select a rule within a program. Since this selection process is non-deterministic, a scheme must be devised to instantiate individual rules. This non-deterministic selection process within the red_rule clause can be eliminated by introducing a separate red_rule clause for each rewrite rule. Then, there will be only one rewrite rule associated with each red_rule clause.

Actually, for the sake of efficiency, each red_rule clause is mapped to a clause whose functor is the same as RHS, but whose arity is one larger than RHS. The additional argument is used for the return value. The advantage of this method is that in many Prolog systems, head unifications are particularly efficient. Thus, to accommodate this modification, the reduce predicate must be modified as follows:

```
reduce(Term, Rslt) :-
        var(Term),
        !, fail.
reduce(Term, Rslt) :-
        new_head(Goal, Term, Rslt),
        call(Goal).

new_head(reduce(Term, Rslt), Term, Rslt) :-
        var(Term), !.
new_head(Goal, Term, Rslt) :-
        Term =.. [F | AList],
        det_append(AList, [Rslt], NewAList),
        Goal =.. [F | NewAList], !.
```

In the new_head predicate, if Term is an unbound variable Goal is unified with the reduce(Term, Rslt). Otherwise, the predicate maps Term into Goal, where they are identical except an additional argument, Rslt, is added to the Goal.

Now that the non-deterministic selection process has been eliminated, the variables LHS and RHS are fully instantiated within each clause. Note, however, that the variable Rslt is unbound as it is runtime dependent. However, since LHS is instantiated, line 8 is fully deterministic. In other words, the goal can be eliminated by executing it at compile time. As a side effect of executing line 8, variables F and FPList are now bound.

For the `makelist` goal in line 9, there are two possible replacements as there are two clauses representing the predicate. In actuality, since `FPList` is already instantiated, only one of those two clauses can satisfy the goal. More specifically, if `FPList` is bound to `[]`, then the first clause (line 40) is applied, which in this case is a unit clause. Therefore, that goal is executed and then eliminated. The side effect of this execution is binding of the the variable `AAList` to `[]`. One the other hand, if `FPList` is bound to a non-empty list, then the goal is unified with the head of the second clause (line 41) and it is replaced by the subgoal of that clause (line 42). As a result, the variable `Xs` and `Ys` are bound to the tails of `AAList` and `FPList`, respectively. A similar process is applied to the replaced `makelist` goal, until that goal, too, is eliminated.

This process of partial evaluation can be continued with each clause as long as the replacement goals can be determined for each goal. There are seven predicates defined in the interpreter. Of these, except for `red_rule` and `reduce`, the remaining five are deterministic. Also, as partial evaluation is applied, variables will become instantiated to the point where the clause which must be used to satisfy the goal can be determined at compile time, as was the case in the `makelist` example above.

The `reduce(Term, Rslt)` goal can be evaluated at compile time if `Term` is bound. Namely, the goal can be replaced by the `call(Goal)`, where the `Goal` is defined by `new_head(Goal, Term, Rslt)`. However, if `Term` is an unbound variable, it cannot be determined at compile time whether it will remain unbound at runtime. Therefore, the `reduce(Term, Rslt)` goal must be evaluated at runtime. Note that this differentiation is made in the `new_head` predicate.

The following is a set of FLOP reduction rules and their equivalent Prolog predicates:

```
FLOP rewrite rules            Prolog clauses

likes(tom) => mary.           likes(tom, mary).

likes(? tom) => mary.         likes(Tom, mary) :-
                                    tom(Tom).

likes(tom) => ? mary.         likes(tom, Rslt) :-
                                    mary(Rslt).

likes(? tom) => ? mary.       likes(Tom, Rslt) :-
                                    tom(Tom),
                                    mary(Rslt).

likes(& ? tom) => ? mary.     likes(AA_Tom, Rslt) :-
                                    reduce(AA_Tom, Tom),
                                    tom(Tom),
                                    mary(Rslt).
```

**Appendix II (An Implementation of FLOP)** contains a FLOP-to-Prolog compiler written in Prolog. This compiler handles other features, such as a Prolog interface, which will be described in the following section. However, the essence of the compiler is the idea of partial evaluation presented above.

The process of partial evaluation is basically that of rewriting. This suggests that perhaps a FLOP-to-Prolog compiler can be implemented in FLOP. This is, in fact, true and **Appendix III (FLOP-To-Prolog Compiler In FLOP)** contains an implementation of a compiler written in FLOP. Note that the interpreter presented in the previous section, the compiler implemented in Prolog in Appendix II, and the compiler in FLOP presented in Appendix III share identical structures.

## 6.3 PROLOG Interface

Consider the following pairs of FLOP rules and their compiled Prolog clauses.

| FLOP rewrite rules | Prolog clauses |
|---|---|
| `append([], L) => L.` | `append([], L, L).` |
| `append([X\|Xs], L) => [X\|?append(Xs, L)].` | `append([X\|Xs], L, [X\|Rslt]) :- append(Xs, L, Rslt).` |

Note that the compiled clauses of the `append` rewrite rule is the usual `append` predicate written in Prolog. There are two points worth noting. First, the mapping scheme from FLOP rewrite rules to Prolog clauses produces efficient code. Namely, once the rules are compiled, there is little computational overhead introduced in execution. Second, because FLOP rules are represented in a form identical to usual Prolog predicates, existing Prolog predicates ( i.e., the predicates written directly in Prolog) with arity greater than zero are directly executable within FLOP. For example, the `append` predicate could be implemented either in FLOP or in Prolog; the two implementations are equivalent.

There are a number of built-in Prolog predicates with non-zero arities which have interesting interpretations in context of FLOP. For example, consider the following queries:

```
| ?= var => T.
_50
 T=_50

| ?= =..(f(x,y)).
[f, x, y]
```

In FLOP, `var` can be viewed as a function which generates a complete set of unbound variables. In the trace above, the query can succeed only when `var` generates the unbound variable `T` itself. Similarly, the `=..` functor returns a list whose first element is the functor of its argument term and the following elements correspond the arguments of this term.

In addition to accessing existing Prolog predicates in a manner described above, it is sometimes convenient to include Prolog predicates directly into the compiled clauses. This is necessary, for example, when Prolog predicates of arity zero must be called. In particular, cuts can be inserted in the compiled clause to alter the backtracking behavior of the rule. The following is a description of a meta-FLOP syntax for interfacing between a FLOP program to its host language Prolog. The basic form is similar to the insertion of Prolog goals in Definite Clause Grammars [CLOCKSIN 81]. This facility has several uses, including debugging, Input/Output, and deterministic programming in FLOP.

A FLOP rewrite rule which is compiled into Prolog predicates has two distinct components, LHS-matching and RHS lazy reduction. Consider the following rewrite rule and its compiled form:

```
lhs(& a1, a2) => ? rhs.

lhs(A1, a2, Rslt):-              % header
      reduce(A1, a1),            % LHS-matching
                                 % post-LHS
      rhs(Rslt).                 % RHS lazy reduction
                                 % post-RHS
```

Following the two components, LHS matching and RHS reduction, two regions of the code have been labelled. These regions are strategic locations for embedding Prolog predicates as they represent the following points in reduction:

**post-LHS** - at this point, LHS-matching has succeeded, and the argument variables are bound to values resulting from the matching.

**post-RHS** - at this point, reduction for this rule has succeeded and all variable bindings resulting from the reduction have been made.

The two points specified above are natural locations for optional Prolog goals. Optional Prolog goals can be inserted in these locations by the following meta-syntax:

LHS => *{post_LHS}*, RHS, *{post_RHS}*.

The Prolog goals must be surrounded by the {} brackets.Variable names are maintained between the reduction rule and the optional goals by the compiler. That is, all references to variables with the same name refer, in fact, to the same variable.

Each {} bracketed code is optional, and any combination of them are allowed.

The function name, `result`, of arity 1 is reserved for the special purpose of accessing the return value. If this function symbol is detected either in the post-LHS or post-RHS, its argument will be bound to the reduced value. This is accomplished by replacing each Prolog goal of the form `result(`*`term`*`)`, where *`term`* is an abitrary term, with the goal `Rslt = `*`term`*, where `Rslt` is a variable which will be bound to the reduction value. In post-LHS, this variable may or may not be bound to the return value. However, this variable is guaranteed to be bound to the return value in the post-RHS.

Here is a sample compilation of a FLOP rule:

```
FLOP rewrite rules          Prolog clauses

f(X) =>                     f(X, Rslt) :-
     {write(X), nl},             write(X), nl,
     ? g(X).                     g(X, Rslt),
     {result(R),                 R = Rslt,
      write(R), nl}.             write(R), nl.
```

## 6.2.1 Deterministic FLOP

Inserting cuts in FLOP rewrite rules in either the post-LHS or post-RHS will modify the backtracking behavior of the rules and produce more deterministic behavior. The following is an example of a deterministic reduction rule:

| FLOP rewrite rules | Prolog clauses |
|---|---|
| `f(x) => a, {!}.` | `f(x, a) :- !.` |
| `f(y) => b.` | `f(y, b).` |

The cut in post-RHS prevent backtracking to other clauses once LHS-unification is successful:

```
| ?= f(x).
a

| ?= f(y).
b

| ?= f(V).
a
 V=x ;
reduction unsuccessful
```

## 6.2.2 Lazy Evaluation of Prolog Predicates

Prolog goals which are introduced in the Post-LHS or Post-RHS are always treated eagerly. However, it is often desirable to treat these in a lazy fashion. This can be accomplished simply in the following manner:

```
(Term1 == Term2) => {Term1 == Term2}, true.

var(X) => {var(X)}, true.

atom(X) => {atom(X)}, true.

read(X) => {read(X)}, true.

write(X) => {write(X)}, true.

writeln(X) => {write(X), nl}, true.
```

For each function, if the Prolog predicate succeeds, the function, too, succeeds with the return value `true`. If the Prolog predicate fails, the function, too, fails. Note these Prolog predicates can be treated as normal FLOP functions with lazy evaluation.

The following rewrite rule generalizes the lazy evaluation of of Prolog goals[1]:

```
{Goal} => {call(Goal)}, true.
```

If the `Goal` succeeds (in the Prolog environment), then it returns the atom `true`.

[1]There is a slight limitation in the way our current syntax is defined. For example, the rule

```
lhs => {goal}, rhs.
```

has two interpretations. The question is whether `{goal}` should be interpreted as post-LHS, which is treated eagerly or as a lazy evaluable goal using the generalized Prolog predicate rule. In general, whenever there is a conflict, `{}` terms are interpreted as post-LHS or post-RHS. At this point, the reader may wonder why the post-LHS and post-RHS syntax are not simply eliminated by making all `{}` terms calls to Prolog predicates in a lazy fashion. The only reason for this is that introduction of cuts in post-LHS and post-RHS retain the scope of the cut within each rule. However, if the cut symbols are lazy evaluated, its scope is limited to within the `{}` term.

# CHAPTER 7 - EXAMPLES

This chapter presents four sample FLOP programs. The first is list flattening function which transforms arbitrarily nested list-of-list structures into a single list. The second implements the quicksort algorithm in two different styles, one in a procedural style, the other as a functional expression. The third example is a compiler of Prolog clauses into WAM instruction sequences [WARREN 83]. It is based on the example of partial evaluation presented in [KURSAWE 86]. Finally, a method of directly evaluating Definite Clause Grammars are presented.

## 7.1 Flattening Lists

The following is an implementation of a simple double recursive algorithm for flattening list of lists. The lists may be nested to arbitrary depth.

```
flatten(T) => ? if_then(var(T), [T]), {!}.
flatten(T) => ? if_then({atomic(T)}, [T]), {!}.
flatten([L | R]) => ? append(?flatten(L), ?flatten(R)), {!}.
```

And here is a trace of a executing this function:

```
| ?= flatten([[a,[b,c]],[[e,f],g]]).
[a,b,c,d,e,f,g] ;
reduction unsuccessful
```

Note that the two base conditions, `var(T)` and `{atomic(T)}`, are implemented in two different ways. Both use the built-in function provided by the host Prolog system. However, the mechanism for obtaining its service is different. The reduction of the term `var(T)` is implemented via

```
var(T) => {var(T)}, true.
```

The reduction of the term `{atomic(T)}` is accomplished through the general Prolog predicate evaluator:

```
{Goal} => {call(Goal)}, true.
```

Though these two styles differ, their effects are identical. They have been implemented in this manner for pedagogical reasons.

## 7.2 Quicksorts

This section presents two versions of the quicksort algorithm. The first is implemented in a procedural style.

```
quicksort1([]) => [].
quicksort1([X | Xs]) => ? (
            (partition1(X, Xs) => SmallXs/BigXs),
            (quicksort1(SmallXs) => SmallLs),
            (quicksort1(BigXs) => BigLs),
            append(SmallLs, [X | BigLs])
      ).

partition1(P, []) => []/[].
partition1(P, [X |Xs]) => ? (
            {X =< P},
            (partition1(P, Xs) => SmallXs/BigXs),
            !([X | SmallXs]/BigXs)
      ).
partition1(P, [X |Xs]) => ? (
            {X > P},
            (partition1(P, Xs) => SmallXs/BigXs),
            !(SmallXs/[X | BigXs])
      ).
```

The `partition1` function accepts two arguments. The first is the partition number, and second the list to partition. The return value of this function is two lists separated by the '/' operator. Its left argument is a list of elements less than or equal to the partition number, and the right argument the list of elements greater than the partition. The return value must be quoted, as the conjunction operator always evaluates its arguments.

The next version of quicksort is implemented as a functional expression[1], rather than procedural, by eliminating the use of the conjunction operator:

```
quicksort2([]) => [].
quicksort2([X | Xs]) =>
      ? append(
            ? quicksort2(? partition2(X, Xs, BigXs)),
            [X | ? quicksort2(BigXs)]
      ).

partition2(P, [], []) => [].
partition2(P, [X |Xs], BigXs) => ? (
            {X =< P},
            ![X | ? partition2(P, Xs, BigXs)]
      ).
partition2(P, [X |Xs], [X | BigXs]) => ? (
            {X > P},
            partition2(P, Xs, BigXs)
```

[1]The difference between functional and procedural style is actually rather artificial in FLOP, for conjunction, too, is defined in a functional manner. Nevertheless, the behavior of the conjunction function can be naturally viewed sequentially, much in the way Prolog clauses can be interpreted procedurally.

```
          ).
```

In this version, the `partition2` function returns the list of elements less than or equal to the partition number. The list of elements greater than the partition is returned by side-effect. This list is bound to the third argument.

Though these two styles differ, they produce identical results, as exemplified by the following execution:

```
| ?= quicksort1([1,2,3,2,1]).
[1,1,2,2,3] ;
reduction unsuccessful

| ?= quicksort2([1,2,3,2,1]).
[1,1,2,2,3] ;
reduction unsuccessful
```

## 7.3 WAM

This chapter presents a FLOP program which translates Prolog clauses into an abstract Prolog instruction set [WARREN 83]. It is based on the example of partial evaluation presented in [KURSAWE 86]. The basic idea behind partial evaluation is described in **Section 6.2 (Compiling FLOP to Prolog)**.

The following is an implementation of *apm-1* and *apm-2* provided in [KURSAWE 86]. The removal of the eager operators in the lines denoted by the comment /* apm-2 */ unravels the source program only down to the apm-1 level. The insertion of eager operators in these locations cause further evaluation to the apm-2 level. The syntactic structure of the source code is examined in post-LHS. The cuts in post-RHS are added mainly for efficiency.

```
:- op(700,xfx,':=').

unwind(P) =>
      {var(P), write('cannot unwind a variable.'), nl, !,
      fail},
      _.
unwind((H:-G)) =>
      {functor(H,F,N), functor(NewH,F,N),
            H =..[_|HL], NewH =..[_|NewHL]},
      (NewH :- ?comma_cat(?unipp_args(HL,NewHL), G)),
      {!}.
unwind(H) =>
      ?unwind((H:-true)),
      {!}.
```

```
unipp(S,T) =>
      {var(S)},
      S := T,
      {!}.
unipp(S,T) =>
      {atomic(S)},
      ? unipp_constant(S,T),          /* apm-2 */
      {!}.
unipp(S,T) =>
      {struct(S)},
      ? unipp_struct(S,T),            /* apm-2 */
      {!}.

unipp_constant(S,T) =>
      (var(T), T:=C; T==C),
      {!}.

unipp_struct(S,T) =>
      {functor(S,F,N), S=..[_|SL]},
      ((var(T), functor(T,F,N) ;
            struct(T), functor(T,G,M), F==G, N==M),
            T=..[_|TL],
            ? unipp_args(SL,TL)),     /* apm-2 */
      {!}.

unipp_args([], []) =>
      true,
      {!}.
unipp_args([S|SL], [T|TL]) =>
      (?comma_cat(?unipp(S,T), ?unipp_args(SL,TL))),
      {!}.

comma_cat(T1,T2) =>
      {comma_cat(T1, T2, R)},
      R.
comma_cat(T1, T2, (T1, T2)) :- var(T1), !.
comma_cat((T, T1), T2, (T, T3)) :- comma_cat(T1, T2, T3), !.
comma_cat(T1, T2, (T1, T2)).

struct(X) :-
      novar(X),
      functor(X, F, N),
      N > 0, !.
```

The `unwind` rewrite rule converts the initial Prolog clause into a normal form, one in which there is at least one subgoal of the head goal. Compilation of the two `append` clauses is shown below:

```
| ?= unwind(append([], L, L)).

append(A, B, C) :-
        (       var(A),
                A := D
                |  A == D
        ),
        E := B,
        E := C,
        true.

| ?= unwind((append([X|L1], L2, [X|L3]) :-
                                        append(L1, L2, L3))).
append(A, B, C) :-
        (       var(A),
                functor(A,  '.', 2)
                | struct(A),
                functor(A, D, E),
                '.' == D,
                2 == E
        ),
        A =.. [F,G,H],
        I := G,
        J := H,
        true,
        K := B,
        (       var(C),
                functor(C,  '.', 2)
                |   struct(C),
                functor(C, L, M),
                '.' == L,
                2 == M
        ),
        C =.. [N,O,P],
        I := O,
        Q := P,
        true,
        append(J, K, Q).
```

## 7.4 DCG In FLOP

The relationship between definite clause grammars (DCGs) [CLOCKSIN 81] and logic programming has been well established. DCGs become executable context free grammars under SLD-resolution when compiled into Horn clauses by a notational transformation in which additional parameters are added for input and output. Since it has already been shown in **Chapter 3 (Functional Logic Programming)** that SLD-resolution is a special case within functional logic programming, DCGs are clearly executable within FLOP if a similar transformation is made. Currently, there is related work being done at UCLA on *functional grammars* [CHAU 88].

This section demonstrates that there is a more direct relationship between DCGs and FLOP than between DCGs and Horn clauses. More specifically, as DCG are a particular form of rewrite rules, they can be expressed in FLOP such that they are directly executable with no modifications.

First, a connective for DCG symbols must be selected, say #. This operator is used to connect the items within each grammar rule. The following grammar represents a simple sentence structure.

```
sentence => ? (noun # verb # object).

object => ? (article # noun).

noun => [tom].
noun => [mary].
noun => [lunch].
noun => [movie].

verb => [saw].
verb => [ate].

article => [a].
article => [the].
```

Following the usual convention, terminal symbols are represented as members of a list. Now the rewrite rule for the connective # must be defined. Since each argument generates a list of atoms, the return value is simply the `append` of these two lists.

```
((& T1) # (& T2)) => ? append(T1, T2).
```

DCGs can be both generators and parsers. The following execution illustrates the use of a DCG as a generator using the above grammars:

```
| ?= object.
[a,tom] ;
[a,mary] ;
[a,lunch] ;
[a,movie] ;
[the,tom] ;
[the,mary] ;
[the,lunch] ;
[the,movie] ;
reduction unsuccessful
```

Note that all possible forms of the `object` are generated. The next trace demonstrates DCG as a parser:

```
| ?= sentence => [tom, ate, the, lunch].
[tom,ate,the,lunch] ;
reduction unsuccessful
```

With the definition of # above, each term is fully evaluated before it is unified with the list to be parsed. This can be improved by employing difference lists. The use of difference lists allows unification with the list to take place as nonterminal in the grammar is reduced. Therefore, failure can be detected sooner. To accommodate the use of difference lists, the # operator is redefined with the following rule:

```
(& (S0/T) # & (T/S)) => S0/S.
```

Also, a simple function is added to convert each terminal into a difference list. This is accomplished by the following `terminal` function:

```
terminal(T) => [T|S]/S.
```

The following is an implementation of the identical grammar above using a difference list format. Also, in this next version, a parse tree is generated as well:

```
sentence(sentence(Noun, Verb, Object)) => ? (
      noun(Noun) # verb(Verb) # object(Object)
    ).

object(object(Part, Noun)) => ? (
      article(Part) # noun(Noun)
    ).

noun(noun(tom)) => ? terminal(tom).
noun(noun(mary)) => ? terminal(mary).
noun(noun(lunch)) => ? terminal(lunch).
noun(noun(movie)) => ? terminal(movie).

verb(verb(saw)) => ? terminal(saw).
verb(verb(ate)) => ? terminal(ate).

article(part(a)) => ? terminal(a).
article(part(the)) => ? terminal(the).
```

Now, a simple trace of its execution:

```
| ?= sentence(X) => [tom,ate,the,lunch]/[].
[tom,ate,the,lunch]/[]
 X=sentence(noun(tom), verb(ate),
           object(part(the), noun(lunch))) ;
reduction unsuccessful
```

Note that X is bound to the parse tree. Also, the second argument of the => functor is specified as a difference list.

As in Prolog, arbitrary FLOP goals may also be inserted within each grammar rule. Since grammar rules are directly executed without a syntactic preprocessor, however, FLOP goals which are not legitimate items of the grammar rule must also return some value. More specifically, their return value must be a list to accommodate the # operator. Furthermore, they must not modify the actual list of terminals generated by a rule. This can be accomplished if FLOP goals which are not legitimate items of the grammar return an empty list of terminals. We can designate a special function called, say, `non_dcg`[1] which evaluates arbitrary flop goals. The following rule implements this:

```
non_dcg(& Goal) => [].

non_dcg(& Goal) => [S]/S.
```

The first rule can be used for the version of our DCG example, where the list of terminals are represented as simple lists. The second is for the second version which uses difference lists. In both versions, the function returns the appropriate representation of a null list if its argument is reducible, where no restrictions are placed on the return value of the argument.

[1]In Prolog, curly braces { } are traditionally used to designate additional goals that supplement the grammar. This term is not used in FLOP because it is already used for evaluating Prolog predicates within FLOP rewrite rules.

# CHAPTER 8 - FUTURE WORK

FLOP shows a great potential as a general programming language, as many problems may be easily and cleanly formulated in a system which offers simultaneously offers the advantages of functional programming and logic programming. However, FLOP must now be vigorously tested by using it to write more sophisticated and demanding programs.

So far, FLOP has been implemented only within Prolog. A direct implementation of FLOP in a lower level language, such as C, may yield significant improvement in efficiency. Certainly, it is more reasonable to run Prolog programs in FLOP, as opposed to the other way around, for it would be trivial to enhance FLOP so that Prolog programs can directly executable. A prudent approach might to design a low level abstract machine, one analogous to the WAM [WARREN 83] for Prolog. Further, as there are a number of common underlying mechanisms between FLOP and Prolog, much of the knowledge gained in Prolog technology should be directly applicable in implementing FLOP systems.

There are several issues which must be resolved, however, in order to make a direct implementation of FLOP feasible. First, a number of features which are currently handled by the Prolog environment, such as meta-logical and extra-logical predicates, must be added. Second, a useful debugging environment must be provided. Finally, a mechanism for modifying the backtracking behavior, one analogous to the cut in Prolog, must be designed. It is the author's view that elegant solutions for the latter two issues are far from trivial and will require a spark of ingenuity.

The current FLOP system reflects a design process in which simplicity, uniformity and generality have been the primary goals. More pragmatic issues concerning programmability and efficiency have yet to be thoroughly studied. Nevertheless, a number of areas of enhancement and research have already surfaced.

There are a number of functions, of which conjunction is a prime example, for which modification of lazy reduction would enhance the programmability and the usefulness of lazy evaluation. With the conjunction operator, the right argument should be lazy reduced, if and only if the lazy reduction of the left argument succeeds. Currently, all of the eager terms within both arguments are reduced first, before the conjunction operator itself is reduced. A simple solution would be to designate conjunction as a special case under lazy reduction. However, a more general mechanism may be devised so that other functors with similar properties may be added.

With respect to the partial evaluation of FLOP rewrite rules into Prolog predicates, in the current implementation, the process unconditionally terminates when `reduce` goals are detected. Perhaps a special class of rewrite rules can be designated for which further evaluation can be performed. This method may yield significant improvement in the execution speed of FLOP programs.

Currently, programming languages suited for developing prototypes are in general inadequate in producing efficient programs. Therefore, programs which require "industrial strength" efficiency must be reimplemented in a lower level language. It may be possible to bridge this gap by two methods: 1) by extending partial evaluation of functions and 2) by incorporating lower level constructs in FLOP. Then, a program may be incrementally modified in its prototype environment until the desired efficiency is attained.

Another issue deals with notational convenience. It is often cumbersome to add an eager operator in front of RHS. One possible solution is to introduce another function specifier symbol, say ->, which would automatically insert an additional eager operator.

The Prolog interface mechanism may be generalized, such that the insertion of optional Prolog goals may be less restrictive. Currently, optional goals are limited to two locations, post-LHS and post-RHS. One possibility is to generalize the current meta-syntax so that optional goals may be added with each eager and argument operator. This scheme seems reasonable as additional goals (`reduce`/2) are introduced solely by the presence of the two operators. This additional flexibility would provide finer control over the execution of FLOP programs, especially with respect to modifying its backtracking behavior with cuts. Note also that, in the current version, there is no mechanism for inserting cuts prior to the goals generated by the `&` operator.

# REFERENCES

[ANDREWS 86] *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, Peter B. Andrews, Academic Press, Inc., San Diego, CA, 1986.

[BARBUTI 84] *On The Integration Of Logic Programming And Functional Programming*, R. Barbuti, M. Bellia, G. Levi, 1984 International Symposium On Logic Programming, Atlantic City, NJ, 1984.

[CARLSSON 84] *On Implementing Prolog In Functional Programming*, Mats Carlsson, 1984 International Symposium On Logic Programming, Atlantic City, NJ, 1984.

[CLOCKSIN 81] *Programming in Prolog*, W. F. Clocksin and C. S. Mellish, Springer-Verlag, New York, NY, 1981.

[HILL 74] *LUSH-Resolution And Its Completeness*, R. Hill, DCL Memo 78, Dept of AI, University of Edinburgh, 1974.

[KOMOROWSKI 82] *QLOG - The Programming Environment for Prolog in LISP*, H. J. Komorowski, Logic Programming, eds. K. Clark, S. A. Tarnlund, Academic Press, New York, NY, 1982.

[KURSAWE 86] *How to Invent a Prolog Machine*, Peter Kursawe, Proceedings of Third International Conference on Logic Programming, Springer-Verlag, New York, NY, 1986.

[[LEWIS 88] *Executable Temporal Specifications With Functional Grammars*, H. Lewis Chau, D. Stott Parker, UCLA Tech Report CSD-880046, Computer Science Dept., University of California at Los Angeles, June 1988.

NARAIN 85] *A Technique For Doing Lazy Evaluation In Logic*, Sanjai Narain, 1985 International Symposium On Logic Programming, Atlantic City, NJ, 1985.

[NARAIN 88] *LOG(F): An Optimal Combination of Logic Programming, Rewriting and Lazy Evaluation*, Sanjai Narain, UCLA Tech Report CSD-880040, Computer Science Dept., University of California at Los Angeles, June 1988.

[PEREIRA] *C-Prolog User's Manual*, Fernando Pereira, SRI International, Menlo Park, CA.

[ROBINSON 82] *LOGLISP: Motivation, Design And Implementation*, J. A. Robinson, E. E. Sibert, Logic Programming, eds. K. Clark, S. A. Tarnlund, Academic Press, New York, NY, 1982.

[SRIVASTAVA 85] *An(other) Integration Of Logic And Functional Programming*, Amitabh Srivastava, Don Oxley, Aditya Srivastava, 1985 International Symposium On Logic Programming, Atlantic City, NJ, 1985.

[STERLING 86] *The Art of Prolog*, Leon Sterling and Ehud Shapiro, MIT Press, Cambridge, MA, 1986.

[SUBRAHMANYAM 84] *Conceptual Basis And Evaluation Strategies For Integrating Functional And Logic Programming*, P. A. Subrahmanyam, J-H. You, 1984 International Symposium On Logic Programming, Atlantic City, NJ, 1984.

[SUSSMAN 85] *Structure and Interpretation of Computer Programs*, Harold Abelson, Gerald Jay Sussman and Julie Sussman, MIT Press, Cambridge, MA, 1985.

[WARREN 83] *An Abstract Prolog Instruction Set*, David H. D. Warren, Technical Note 309, SRI International, Menlo Park, CA, 1983.

[WINSTON 84] *LISP, 2nd ed*, Patrick Henry Winston and Berthold Klaus Paul Horn, Addison-Wesley Publishing Co., Menlo Park, CA, 1986.

# APPENDIX I - LOG(F)

The following is a synopsis of the LOG(F) rewrite system. For a more complete presentation of LOG(F) refer to [NARAIN 88].

F* is a non-Noetherian rewrite rule system (allows infinite reductions). An F* program is a set of reduction rules defined as follows:

**Variables.** There is a countably infinite list of variables.

**Function Symbols.** For each i, $i \geq 0$, there is a countably infinite list of i-ary function symbols.

**Connectives.** The connectives are =>, (, ), ','.

**Constructor Symbols.** There is a subset of the function symbols called Constructors.Each element of the constructors is called a constructor symbol. For each n, $n \geq 0$, Constructors contain an infinite number of n-ary function symbols. In particular, `0`, `true`, `false`, `[]` are 0-ary constructor symbols and `|` a 2-ary constructor symbols.

**Term.** A term is either a variable, or an expression of the form $f(t_1, \ldots, t_n)$ where f is an n-ary function symbol, $n \geq 0$, and each $t_i$ is a term. A term is called ground if it contains no variables. *Unless otherwise stated, by a term we mean a ground term.*

**Reduction Rules.** A reduction rule is of the form `LHS=>RHS`, where `LHS` and `RHS` are terms possibly containing variables. The following restrictions are placed on `LHS` and `RHS`:

a. `LHS` is not a variable.

b. `LHS` is not of the form $c(t_1, \ldots, t_n)$ where c is an n-ary constructor symbol, $n \geq 0$.

c. If `LHS` is $f(t_1, \ldots, t_n)$, $n \geq 0$, each $t_i$ is a variable or a term of the form $c(x_1, \ldots, x_m)$, where c is an m-ary constructor symbol, $m \geq 0$, and each $x_i$ is a variable. Note that => rules with left-hand-sides of arbitrary depth can easily be expressed in terms of rules with left-hand-sides of depth at most two.

d. There is at most one occurrence of any variable in `LHS`.

e. All variables of `RHS` appear in `LHS`.

**$E \Rightarrow_P E1$.** Let P be an F* program and E and E1 be terms. We say $E \Rightarrow_P E1$ if there is a rule `LHS=>RHS` in P such that `LHS` and E unify with m.g.u. $\Omega$ and E1 is $RHS\Omega$. We also say that E reduces to E1 by the rule `LHS=>RHS`. If P is clear from the context we write E=>E1 in place of $E \Rightarrow_P E1$.

**E->$_P$E1, E-*>$_P$E1.** Let P be an F* program and E be a term. Let G be a subterm of E such that G=>$_P$H. Let E1 be the result of substituting H for G in E. Then E->$_P$E1. -*> is the reflexive-transitive closure of ->. Again, if P is clear from the context we write E->E1 and E-*>E1 in place of E->$_P$E1 and E-*>$_P$E1.

**Reduction.** Let P be an F* program. A reduction in P is a sequence of E1, E2, ... such that for each i, when Ei and Ei+1 both exist, Ei->$_P$Ei+1.

**Simplified Form.** A term is said to be in simplified form if it is of the form c($t_1$,...,$t_n$) where c is an n-ary constructor symbol, n≥0, and each $t_i$ is a term. F is called a simplified form of E if E-*>F and F is in simplified form.

**Successful Reduction.** Let P be an F* program. A successful reduction in P is a reduction E0,...,En, n≥0, in P, such that En is simplified.

**Reduction Strategy of F*.** Let P be an F* program. We now define a reduction strategy, select$_P$ for P. Informally, given a term E it will select that subterm of E whose reduction is necessary in order that some => rule in P apply to the whole of E. Where f($t_1$,...,$t_n$) is a term, n≥0, the relation select$_P$ is defined using the following pseudo-Horn clauses:

```
selectP(f(t1,...,tn), f(t1,...,tn)) if
        f(t1,...,tn) =>P X.

selectP(f(t1,...,tn), X) if
        there is a rule f(L1,...,Li,...,Ln) => RHS in P, and
        ti does not unify with Li, and
        selectP(ti, X).
```

**N-step.** Let P be an F* program and E, G, H be terms. Suppose that select$_P$(E,G) and G=>$_P$H. Let E1 be the result of replacing G by H in E. Then we say that E reduces to E1 in an N-step in P . The qualification "in P" is omitted when P is clear from context. The prefix N in N-step is intended to connote normal order.

**N-reduction.** Let P be an F* program. An N-reduction in P is a reduction E1, E2, ... in P such that for each i, when Ei and Ei+1 both exist, Ei reduces to Ei+1 in an N-step in P. In particular, the sequence E where E is a term, is an N-reduction in P. The qualification "in P" is omitted when P is clear from the context.

**Theorem - Completeness of F*.** Let P be an F* program and E0 a term. Let E1, E2, ..., En=c($t_1$,...,$t_x$), x≥0, n≥0, be a successful reduction in P, for some constructor symbol c. Then, there is a successful N-reduction D0..., in P.

**Compiling F* Into Horn Clauses.** A simple algorithm to compile F* programs into Horn clauses is described. LOG(F) is defined to be a logic programming system augmented with an F* compiler. The translation of P into Horn clauses proceeds in two stages.

Stage 1. For each n-ary constructor symbol c in P, and where $X_1$,...,$X_n$ are distinct variables, generate the clause:

```
reduce(c(X1,...,Xn),c(X1,...,Xn)).
```

Stage 2. Let `f(L`$_1$`,...,L`$_m$`)=>RHS` be a rule in P where `f` is an m-ary, m≥0, non-constructor function symbol and each of `RHS` and `L`$_1$`,...,L`$_m$ is a term, possibly containing variables. For each such rule perform the following steps:

a. Let `A`$_1$`,...,A`$_m$ be distinct logical variables none of which occur in the rule. For any i, if `L`$_i$ is a variable, let `Q`$_i$ be `A`$_i$`=L`$_i$. If `L`$_i$ is `c(X`$_1$`,...,X`$_n$`)` where `c` is a constructor symbol and each `X`$_i$ a variable, let `Q`$_i$ be `reduce(A`$_i$`,c(X`$_1$`,...,X`$_n$`))`.

b. Let `Out` be a logical variable not occurring in the rule and different from `A`$_1$`,...,A`$_m$ Generate the predicate `reduce(RHS, Out)`.

c. Generate the clause:

```
reduce(f(A1,...,Am), Out) :-
      Q1,...,Qm, reduce(RHS, Out).
```

# APPENDIX II - An Implementation of FLOP

The following Prolog program consists of a FLOP-to-Prolog compiler, FLOP interactive shell, and other utility routines. The compilation algorithm employed is described in the **Section 7.2 (Compiling FLOP to Prolog)**. Each FLOP rewrite rule is transformed into a Prolog clause whose functor same as the RHS, but whose arity is one greater than RHS.

The program presented runs in C-Prolog [PEREIRA]. Most of the code, however, should be readily portable to other systems.

```
:- op(200, fy, '&').
:- op(200, fy, '?').
:- op(1200, xfy, '=>').

flop_to_prolog((LHS => RHS), PROLOG) :-
      var(RHS), !,
      normal_flop_to_prolog((LHS => RHS),
                                          true ,true, PROLOG).
flop_to_prolog((LHS => RHS, {Post_RHS}), PROLOG) :-
      var(RHS), !,
      normal_flop_to_prolog((LHS => RHS),
                                          true, Post_RHS, PROLOG).
flop_to_prolog((LHS => {Post_LHS}, RHS), PROLOG) :-
      var(RHS), !,
      normal_flop_to_prolog((LHS => RHS),
                                          Post_LHS, true, PROLOG).
flop_to_prolog((LHS => {Post_LHS}, RHS, { Post_RHS}),
                                          PROLOG) :- !,
      normal_flop_to_prolog((LHS => RHS), Post_LHS,
                                          Post_RHS, PROLOG).
flop_to_prolog((LHS => {Post_LHS}, RHS), PROLOG) :- !,
      normal_flop_to_prolog((LHS => RHS),
                                          Post_LHS, true, PROLOG).
flop_to_prolog((LHS => RHS, {Post_RHS}), PROLOG) :- !,
      normal_flop_to_prolog((LHS => RHS),
                                          true, Post_RHS, PROLOG).
flop_to_prolog((LHS => RHS), PROLOG) :- !,
      normal_flop_to_prolog((LHS => RHS),
                                          true ,true, PROLOG).
```

```
normal_flop_to_prolog((LHS=>RHS), Post_LHS, Post_RHS,
                                        (Head:-Body)) :-
        LHS  =.. [F | FPList],
        makelist(FPList, AAList),
        Tmp_Head =.. [F | AAList],
        new_head(Head, Tmp_Head, Rslt),
        lhs_matching_code(AAList, FPList, Argcode),
        lazy_reduce_code(RHS, Rslt, RHScode),
        check_result(Post_LHS, PLHScode, Rslt),
        check_result(Post_RHS, PRHScode, Rslt),
        remove_true((Argcode, PLHScode, RHScode, PRHScode),
                                        Body), !.

lazy_reduce_code(Term, Term, true) :-
        var(Term), !.
lazy_reduce_code(? Term, Rslt, (Code, Goal)) :-
        lazy_reduce_code(Term, RTerm, Code),
        new_head(Goal, RTerm, Rslt), !.
lazy_reduce_code(Term, Rslt, Code) :-
        Term =.. [F | AList],
        makelist(AList, RAList),
        Rslt =.. [F | RAList],
        lazy_reduce_list_code(AList, RAList, Code), !.

lazy_reduce_list_code([], [], true) :- !.
lazy_reduce_list_code([A | AList], [RA | RAList],
                                        (Code, Morecode)) :-
        lazy_reduce_code(A, RA, Code),
        lazy_reduce_list_code(AList, RAList, Morecode), !.

lhs_matching_code([], [], true) :- !.
lhs_matching_code([AA | AAList], [FP | FPList],
                                        (Argcode, Morecode)) :-
        match_one_arg_code(AA, FP, Argcode),
        lhs_matching_code(AAList, FPList, Morecode), !.

match_one_arg_code(AA, FP, true) :-
        var(FP),
        AA = FP, !.
match_one_arg_code(AA, & FP, (Goal, FPcode)) :-
        new_head(Goal, AA, RFP),
        lazy_reduce_code(FP, RFP, FPcode), !.
match_one_arg_code(AA, FP, FPcode) :-
        lazy_reduce_code(FP, AA, FPcode), !.

new_head(reduce(Term, Rslt), Term, Rslt) :-
        var(Term), !.
new_head(Goal, Term, Rslt) :-
        Term =.. [F | AList],
        det_append(AList, [Rslt], NewAList),
        Goal =.. [F | NewAList], !.

det_append([], L, L) :- !.
det_append([X|Xs], Ys, [X|Zs]) :- det_append(Xs, Ys, Zs), !.

makelist([], []) :- !.
makelist([_|Xs], [_|Ls]) :- makelist(Xs, Ls), !.
```

```
remove_true(Term, Rslt) :-
      flatten_commas(Term, FlatTerm),
      rm_true(FlatTerm, Rslt).

rm_true((true, Cs), Rs) :-
      rm_true(Cs, Rs), !.
rm_true((X, true), X) :- !.
rm_true((X, true, Xs), Rs) :-
      rm_true((X,Xs), Rs), !.
rm_true((X, Cs), (X, Rs)) :-
      rm_true(Cs, Rs), !.
rm_true(X, X) :- !.

flatten_commas((L,R), Rslt) :-
      flatten_commas(L, FL),
      flatten_commas(R, FR),
      link_commas(FL, FR, Rslt), !.
flatten_commas(T, T).

link_commas((L, Ls), Rs, (L, LRs)) :-
      link_commas(Ls, Rs, LRs), !.
link_commas(L, Rs, (L, Rs)).

check_result(T, T, Rslt) :- var(T), !.
check_result((X, Xs), (X, Rs), Rslt) :-
      var(X),
      check_result(Xs, Rs, Rslt), !.
check_result((result(T), Xs), (T=Rslt, Rs), Rslt) :-
      check_result(Xs, Rs, Rslt), !.
check_result((X, Xs), (X, Rs), Rslt) :-
      check_result(Xs, Rs, Rslt), !.
check_result(T, T, Rslt).


/******************************************************/
/* entry point to the reduction process.              */
/* these predicates must exist during a flop session. */
/******************************************************/
reduce(T, T)  :- var(T), !, fail.
reduce(T, R)  :- new_head(Goal, T, R), call(Goal).


/******************************************************/
/* routine for compiling a rewrite rule and           */
/* pretty printing it.                                */
/******************************************************/
print_flop(X) :-
      nl, nl,
      write('FLOP Rule:'), nl,
      \+ \+ portray_clause(X), nl,
      flop_to_prolog(X, Y),
      write('Prolog Clause:'), nl,
      \+ \+ portray_clause(Y), nl, nl, !.
```

```
/*****************************************************/
/* for compiling FLOP rewrite rules in the File.     */
/*****************************************************/
compile_flop(File) :-
      seeing(OldFile, File),
      repeat,
            read(Rule),
            (Rule == end_of_file ->
                  see(OldFile), !
            ;     flop_to_prolog(Rule, Clause),
                  assert(Clause),
                  fail
            ).


/*****************************************************/
/* FLOP shell - an interactive interpreter.          */
/*****************************************************/
flop :-
      prompt(Old, ''),
      nl, nl,
      write('Starting FLOP Interactive Environment.'),
      nl, nl,
      repeat,
            nl, write('| ?= '),
            read(Term, Vars),
            (Term==end_of_file ->
                 ! ; reduce_and_print(Term, Vars),   fail).

reduce_and_print(Term, Vars) :-
      lazy_reduce_code(?Term, Rslt, Code),
      (     call(Code)
      ;     nl, write('reduction unsuccessful'), nl,
            !, fail
      ),
      nl, write(Rslt), write(' '),
      print_vars(Vars),
      get_semi, !, fail.

print_vars([]) :- !.
print_vars([V | Vars]) :-
      nl, write(V), write(' '),
      print_vars(Vars), !.

get_semi :- get0(N), get_semi(N).
get_semi(10) :- !.
get_semi(59) :- skip(10), !, fail.
get_semi(26) :- abort.
get_semi(_) :- get_semi.
```

# APPENDIX III - FLOP-to-Prolog Compiler In FLOP

The following FLOP program compiles FLOP programs into an equivalent Prolog program. The basic algorithm employed is one of partial evaluation described in **Section (7.2 Compiling FLOP to Prolog)**. Each FLOP rewrite rule is transformed into a clause whose functor is same as the functor of RHS, but whose arity is one greater than RHS. The Prolog interface mechanism, namely, the insertion of Prolog goals in post-LHS and post-RHS positions, is not handled by this program.

In this compiler, the `?` operator has been replaced by the `eop` symbol and `&` by `aop` . This is necessary, since these symbols are always interpreted. Namely, there is no way of quoting them.

```
:- op(200, fy, 'eop').
:- op(200, fy, 'aop').

unwind_flop((LHS => RHS)) =>
      (Head :-
            ? remove_true((
                  ? (LHS  =.. [F | FPList]),
                  ? makelist(FPList, AAList),
                  ? (THead =.. [F | AAList]),
                  ? new_head(Head, THead, Rslt),
                  ? lhs_matching(AAList, FPList),
                  ? lazy_reduce(RHS, Rslt)
            ))
      ), {!}.

lazy_reduce(Term, Rslt) =>
      (     ? var(Term),
            ? (Term = Rslt)
      ), {!}.
lazy_reduce((eop Term), Rslt) =>
      (     ?lazy_reduce(Term, RTerm),
            ? new_head(Goal, RTerm, Rslt),
            Goal
      ), {!}.
lazy_reduce(Term, Rslt) =>
      (     ? (Term =.. [F | AList]),
            ? makelist(AList, RAList),
            ? (Rslt =.. [F | RAList]),
            ? lazy_reduce_list(AList, RAList)
      ), {!}.
```

```
lazy_reduce_list([], []) =>
      true,
      {!}.
lazy_reduce_list([A | AList], [RA | RAList]) =>
      (     ? lazy_reduce(A, RA),
            ? lazy_reduce_list(AList, RAList)
      ), {!}.

lhs_matching([], []) =>
      true,
      {!}.
lhs_matching([AA | AAList], [FP | FPList]) =>
      (     ? match_one_arg(AA, FP),
            ? lhs_matching(AAList, FPList)
      ), {!}.

match_one_arg(AA, FP) =>
      (     ? var(FP),
            ? (AA = FP)
      ), {!}.
match_one_arg(AA, aop FP) =>
      (     ? new_head(Goal, AA, RAA),
            Goal,
            ? lazy_reduce(FP, RAA)
      ), {!}.
match_one_arg(AA, FP) =>
      ? lazy_reduce(FP, AA),
      {!}.

new_head(reduce(Term, Rslt), Term, Rslt) =>
      {var(Term)},  true, {!}.
new_head(Goal, Term, Rslt) =>
      (     ? (Term =.. [F | AList]),
            ? det_append(AList, [Rslt], NewAList),
            ? (Goal =.. [F | NewAList])
      ), {!}.

det_append([], L, L) => true, {!}.
det_append([X|Xs], Ys, [X|Zs]) => ? det_append(Xs, Ys, Zs),
{!}.

makelist([], []) => true, {!}.
makelist([_ | AList], [_ | RAList]) =>
      ? makelist(AList, RAList), {!}.

remove_true(T) => ?rm_true(?flatten_commas(T)), {!}.

rm_true((true, Rest)) =>
      ? rm_true(Rest), {!}.
rm_true((Untrue, true)) => Untrue, {!}.
rm_true((Untrue, true, Rest)) =>
      ? rm_true((Untrue, Rest)), {!}.
rm_true((Untrue, Rest)) =>
      (Untrue, ? rm_true(Rest)), {!}.
rm_true(Notcomma) => Notcomma, {!}.
```

```
flatten_commas((L,R)) =>
      ? link_commas(?flatten_commas(L), ?flatten_commas(R)),
      {!}.
flatten_commas(T) => T, {!}.

link_commas((L, Ls), R) => (L, ?link_commas(Ls, R)), {!}.
link_commas(L, R) => (L, R), {!}.
```