Table of Contents

LDL ++ Tutorial

(For Version 5.1)

December 1998 Revision

Carlo Zaniolo

Computer Science Department

UCLA

zaniolo@cs.ucla.edu

Acknowledgements

Several parts of this tutorial are based on a previous tutorial prepared by Shalom (Dick) Tsur. The sections on meta-predicates, external DBs, and foreign language functions are due to Natraj Arni, and Kayliang Ong. The sections on XY-stratification and user-defined aggregates are largely due to Haixun Wang.

HTML Tutorial: Table of Contents

PDF Version of Tutorial-- recommended for printing

LDL^{++} Tutorial: Table of Contents

Programs and associated facts in this tutorial are always downloadable by clicking the following icons at the topright corner of the section.

LDL++ programs Associated facts

The programs and facts are available under the shared directory of LDL++ Java Interface.

Carlo Zaniolo, 1997

Cities Database

Consider the following two examples:

CITY		
NAME	STATE	POPULATION
Houston	Texas	3,000,00
Dallas	Texas	2,000.000
Huntsville	Texas	150,000
Austin	Texas	750,000
Corsicana	Texas	60,000
Shreveport	Luisiana	90,000
Bastrop	Texas	6,000
San Antonio	Texas	1,500,000

DISTANCE		
CITY1	CITY2	MILES
Huston	Bastrop	130
Huston	Huntsville	60
Huntsville	Dallas	100
Austin	Waco	110
Waco	Dallas	100
Dallas	Shreveport	200
Austin	Bastrop	30
Austin	San Antonio	80
San Antonio	Huston	190

Database: Facts and Schema

Facts

- **Facts** Corresponds to tuples in the database: For example,

```
city('Houston', 'Texas', 3000000).
distance('Austin', 'Waco', 110).
assembly(wheel, spoke, 36).
```

- Variables start with capital letters.
- Constants that begin with a capital letter are enclosed in single quotes.
- city('Houston', 'Texas', 30000) corresponds to the row ('Houston', 'Texas', 30000) in the city table.

Schema

```
database( { city(Name:string, State:string, Population:integer),
            distance(City1:string, City2:string, Distance:integer)
          } ).
```

- Each predicate (relation) is named.

- Each column is (optionally) named.
- Each column is assigned a data type.
- Datatypes are: integer, real, string, any.
- Columns of type any can contain complex terms (functors).

Rules and Queries

Derivated predicates and rules

- New (or derived) predicates are defined by rules.
- *Example: derive the city in Texas with more than 400,000 people*

  ```
  lt_city(C, Pop) <-  city(C, `Texas`, Pop), Pop > 400000.
  ```
- Rules have a purely declarative role, similar to virtual views in relational databases
- Queries have an imperative role: when the query is executed then the results tha satisfy the query are returned
- *Example: List all cities in Texas with where the population exceeds 400,000.*

  ```
  query lt_city(C, Pop)
  ```
- This query will return:

  ```
     lt_city(`Huston`,   3000000)
     lt_city(`Dallas`,   2000000)
     lt_city(`Austin`,    750000)
     lt_city(`San Antonio`, 1500000)
  ```
- But queries without variables (aka closed queries) return a yes/no answer (no in this case since Austin does not have 2,000,000 inhabitants)

The command

```
 query lt_city(`Austin`, 2000000)
```

will actually be used in the regular interface, while in the Java interface the user will select an item from the menu.

Queries and Query Forms

Query Forms are a.k.a. Exports

- A query form, or export, is a generic query that specifies for the $\mathcal{LDL}^{++}$ compiler which of the arguments will be given and which are expected as output when a query of that type is issued.
- The given bindings in an export allow the compiler to optimize the access to the data.

- Exports can be specified for base predicates as well as derived predicates that are defined via the programs' rules.

For instance, if the program file contains:

export lt_city($X, Y)

Then, the user can compile this export, and then run the following query:

query lt_city (`Austin`, Pop)

The answers are then returned (but if no export match this bound/free pattern an error message is returned)

Example of Query Forms

- export lt_city($X, Y).
  X is given and Y is expected.
- export lt_city(X, $Y).
  X is expected and Y is given.
- export lt_city($X, $Y).
  Both X and Y are given and the response is True/False.

Constructs

Basic Language Constructs

- **Numbers**---integer or real.
- **Constants**.

  A constant is a string of symbols beginning with a lowercase letter. Constants are names of objects such as chain_stay, joe. If we want to use constants starting with an uppercase letter, then we must enclose them in quotes, e.g., 'Houston'.
- **Variables.**

  A variable is a string of symbols beginning with an uppercase letter. Examples are Subpart, Price, X123. Variables denote unspecified values that are assigned during the execution of an LDL++ program.
- Numbers, constants, and variables are included in the syntactic subclass called **terms**.

Carlo Zaniolo, 1997

More on Rules

Rules: Definitions

- A **rule** is an expression of the form:

  $\boldsymbol{A \longleftarrow B_1, B_2, \ldots, B_n.}$

  where $\boldsymbol{A}$ and $\boldsymbol{B_1}$ through $\boldsymbol{B_n}$ are predicates. $\boldsymbol{A}$ is called the **head** of the rule and the expression to the right of the "$\longleftarrow$" is called the **body**.

  ```
  part_cost1(X, Y) <- part_cost(top_tube, X, Y, Z).
  ```

- A rule that contains no variables is a **ground rule** or an **instantiated rule**.

  ```
  run_for_your_life <- physician(`Dr. Frankenstein`).
  ```

- A rule with no body is a **unit clause**.

  ```
  distance(City, City, 0).
  ```

- A ground unit clause is **fact**.

  ```
  distance(`Austin`, `Waco`, 110).
  ```

  Facts are normally stored in the database (but facts occasionally occur in the program.)

Head and Body:

$$\underbrace{\overbrace{eligible\_bachelor(X)}^{head} \leftarrow \overbrace{single(X), handsome(X), rich(X)}^{body}.}_{rule}$$

Semantics of Rules

Rules: Union The two rules,

```
A <- B.

A <- C.
```

are logically equivalent to

$(B \longrightarrow A) \,\&\, (C \longrightarrow A)$

which is

$(B \vee C) \longrightarrow A.$

We will refer to a predicate such as A, which is the result of the union of two (or more) rules, as a **union predicate**.

Rules: Interpretation

A rule,

$\boldsymbol{A \longleftarrow B_1, B_2, \ldots, B_n.}$

is logically equivalent to: the conjunction of $\boldsymbol{B_1}$, $\boldsymbol{B_2}$, **...** , $\boldsymbol{B_n}$ implies $\boldsymbol{A}$.

When the **truth value** of each predicate in the body is TRUE, the head predicate is also TRUE.

Rules: Bottom-up Evaluation

(a) h1(X,Y) <- p(X,Y).
(b) h2(X) <- h1(X,Y).
(c) p(X,Y) <- b(X,Y).

b(1,2).
b(2,4).

1. Rule (c) is the only one whose body contains only a base predicate. Matching the body with the data produces {p(1,2),p(2,4)} as new facts.
   The partial result is the set {b(1,2),b(2,4),p(1,2),p(2,4)}
2. Now rule (a) can be evaluated using the results of step (1).
   The result is {b(1,2),b(2,4),p(1,2),p(2,4),h1(1,2),h1(2,4)}.
3. Rule (b) is evaluated using the result of step (2).
   The result is {b(1,2),b(2,4),p(1,2),p(2,4),h1(1,2),h1(2,4),h2(1),h2(2)}.
4. No more results are produced.

Equality Predicate

Say that t(X, Y, Z) is a base predicate; X,Y,Z are integers. Then,

```
sums(X) <- t(X,Y,Z), X = Y + Z.
```

returns all a,b,c such that the tuple t(a,b,c) appears in t
and a = b + c. The equality serves as a test.

```
t(3,2,1).
t(5,2,3).
t(8,4,3).
t(7,4,3).
t(7,3,4).
```

export sums(X); query sums(X) returns {sums(3), sums(5), sums(7)}
export sums($X); query sums(5) returns TRUE, query sums(8) returns FALSE.

Equation Solving

t1(X, Z) is a base predicate; X, Z are integers.

```
sums1(Y) <- t1(X, Z), X = Y*Y + Z.
```

Fails at compile time! Although X, Z are known, the system cannot solve the equation for Y.

(The current LDL++ will however solve simple linear equations.)

Equality Predicate Operating as single Assignment

t1(X, Z) is a base predicate, X, Z are integers.

```
sums2(Y) <- t1(X, Z), Y = X + Z.
```

export sums2(X)

compiles correctly! In this form the equality is used as a single assignment.

```
t1(1,2).
t1(3,5).
```

query sums2(X) returns {sums2(3), sums2(8)}.
export sums2($X); query sums2(3) returns TRUE, query sums2(4) returns FALSE.

Safety

Safety is a program property that ensures that the values of program variables can be effectively computed.

Safety thus depend on the exectution used (e.g., bottom-up, top-down, or a combination of the two).

1. For the bottom-up execution, the safe predicates are those that are derivable from from database relations (possibly using arithmetic and equality assignements).
2. For top-down exectution, safety also depends from the binding pattern of the export.
3. Assume that goals in the rules are evaluated from left to right. Then the re-ordering of the goals could make a safe rule unsafe or vice-versa
4. The safe binding patterns for equality and disequality predicatess are summarized in the following table.

Safety: Example

```
good_salary(X) <- X > 80000.
```

export good_salary(X) does not compile!

- Query form is unsafe.

export good_salary($X) is safe.

- query good_salary(100,000) is TRUE.
- query good_salary(50,000) is FALSE.

In LDL++ the bound values in the exports are always propagated down to the defining predicates---if these are not recursive. For recursive predicates the situation is more complex and will be discussed later.

Safety: Modified Example

```
well_paid_employee(Name,Salary) <-
                employee(Name,Salary), Salary > 80000.
```

Now

export well_paid_employee(Name,Salary)

is safe.

Arithmetic

Examples of Arithmetic Terms:

Y + Z
X * (Y / Z)

- All participating variables must be bound to numeric values.
- The result of the evaluated term replaces the expression.
- External functions in C can be incoroporated in the term.

Arithmetic: Precedence

level	*operator*	*order*
1	* /	*right to left*
2	mod	*right to left*
3	+ -	*right to left*

Example: The term A / B * C + D will be evaluated in the following order:

1. $r_1 = B * C$.
2. $r_2 = A / r_1$
3. $r_3 = r_2 + D$

Arithmetic: Result Types

- Integer arithmetic returns integers.
- Mixed (real and integer) arithmetic returns real numbers.

Example:

result(X) <- b(A, B), X = A / B.

b(1, 2).

query result(X) returns result(0) when A,B are declared as integers in the schema, it returns result(0.500) when A,B are declared real numbers.

Arithmetic: Example

Express the distance from Austin to Bastrop in feet

Two equivalent formulations:

distance_feet1(X, Z, W) <- distance(X, Z, Y), W = 5280 * Y.

or,

```
distance_feet(X, Z, 5280 * Y) <- distance(X, Z, Y).

export distance_feet($X, $Z,  Y).
```

When the user types:

```
query distance_feet('Austin', 'Bastrop', Z)
```

the system returns:

```
distance_feet( 'Austin', 'Bastrop', 158400 ).
-- 1 solution
```

``List all cities which are within 100 miles of Austin''.

```
close(X, Y, Z) <- distance(X, Y, W), W < Z.

export close($X, Y, $Z).

query close( 'Austin', City, 100)

    close( 'Austin', 'San_antonio', 100 ).
    close( 'Austin', 'Bastrop', 100 ).
    -- 2 solutions
```

The query form: export close($X, Y, Z) will not compile. Why?

Arithmetic: Safety

Also, the program

```
close1(X, Y, Z) <- W < Z, distance(X, Y, W).

export close1($X, Y, $Z)
```

will not compile, since W is unbound when used in the comparison. Predicates must be permuted in the body to render it safe.

Carlo Zaniolo, 1997

Built-in Predicates for Terms

Predicate name and Arguments	***Safe Binding Patterns***	***Comments***
L = R	\$L, \$R	*L, R arbitrary terms*
	L, \$R	*L is a variable*
	\$L, R	*R is a variable*
L ~= R	\$L, \$R	$L \neq R$
L > R	\$L, \$R	
L < R	\$L, \$R	
L <= R	\$L, \$R	$L \leq R$
L >= R	\$L, \$R	$L \geq R$

Recursion

Recursion is a form of predicate definition, using the predicate itself in the rule body.

Example: *``Derive all even numbers up to 98''*

```
even(0).
even(Y) <- even(X), Y = X + 2,  Y < 98.

export even(X)

query even(X)

{even(0)}
{even(0), even(2)}
...
{even(0), ..., even(98)}
```

Recursion: Bottom-up Evaluation

```
even(0).
even(Y) <- even(X), Y = X + 2,  Y > 0.
```

export:even(X) would pass the safety test of the compiler, but it will start printing all integers in an infinite loop. (Use Control^C to stop it).

export even($X) will also run in the same loop, although it might not print-out anything.

Transitive Closure

We have a database relation called parents: e.g.,

```
parent(joe, jack).
parent(joe, jill).
parent(sam, jack).
parent(jack, mary).
parent(mary, lucy).
```

Y is an ancestor of X if Y is a parent of X or if there exists a Z such that Z is a parent of X and Y is an ancestor of Z.

```
ancestor(X, Y) <- parent(X, Y).
ancestor(X, Y) <- parent(X, Z), ancestor(Z, Y).

export:ancestor(X, Y)
query ancestor(X, Y)
```

Recursion: Transitive Closure Evaluation

Step1: Start with the rule whose body is the base relation.

S_1 = { ancestor(joe, jack), ancestor(joe, jill), ancestor(sam, jack), ancestor(jack, mary), ancestor(mary, lucy) }

$$Result = S_1.$$

Step2: ***Result*** is joined with parent. New tuples are pairs of nodes that can be reached along some path in 2 steps.

S_2 ={ancestor(joe, mary), ancestor(sam, mary), ancestor(jack, lucy)}

$$Result = S_1 \cup S_2.$$

Step3: ***Result*** is joined again with parent. New tuples are pairs that are reachable along a path in 3 steps.

S_3 ={ancestor(joe, jucy), ancestor(sam, lucy)}

$$Result = S_1 \cup S_2 \cup S_3.$$

No pairs can be reached along paths of more than 3 steps, and the computation terminates. We say that the computation has reached **fixpoint**.

Switching the rule order and the order of the goals within the rules will not change the result of this transitive closure.

Recursion: Bill of Materials

Bill of materials (BOM) problems are related to assemblies containing superparts composed of subparts that are eventually composed of elementary parts. The assembly(Part, Subpart, Qty) predicate in the parts database contains parts, their immediate subpart, and the quantity with which they are used in the part. part_cost(BasicPart, Supplier, Cost, Time) contains the basic parts.

The following rule serves as a constraint and should be part of the specifications.

```
violation <-
       part_cost(BasicPart, _, _, _), assembly(BasicPart, _ , _).
```

That is, a part cannot be both a basic part and a superpart within this database.

For each part find all the subparts---not just the immediate subparts

```
     all_subparts(Part, Part).
     all_subparts(Part, Sub2) <-
                             all_subparts(Part, Sub1),
                             assembly(Sub1, Sub2).
```

Safe query form for this rule:

export all_subparts($X, Y),
export all_subparts(X, $Y),
export all_subparts($X, $Y).

For each part, basic or otherwise, find all of the basic subparts, only

Let us think bottom-up for this.

```
% A basic part is a subpart of itself.

basic_sub_parts(BasicPart, BasicPart) <-
                   part_cost(BasicPart, _ , _).
basic_sub_parts(Part1, Sub) <-
                   basic_sub_parts(Part, Sub),
                   assembly(Part1, Part).
```

For each part, basic or otherwise, find all of the basic subparts, only

Let us think bottom-up for this.

```
% A basic part is a subpart of itself.

basic_sub_parts(BasicPart, BasicPart) <-
                   part_cost(BasicPart, _ , _).
basic_sub_parts(Part1, Sub) <-
                   basic_sub_parts(Part, Sub),


assembly(Part1, Part).
```

Carlo Zaniolo, 1997

Negation

Existential Variables

Existential variables are those which only appear once in the rule (alias singleton variables). They have a meaning `there exists' in positive goals and `there does not exists in negated goals.

```
    print_nice_people(X) <- person(X), ~nasty(X).
    print_nice_people(`everybody is nice') <- ~nasty(Y).
```

The variable Y in the last rule and W in the previous one could be replaced by an underscore; it is called an *existential variable*. Programs with existential variables in negated goals can be transformed into equivalent programs without. For instance the last rule can be re-written as:

```
   print_nice_people(`everybody is nice') <- ~someonenasty.
   someonenasty <- nasty(Y).
```

This re-writing defines the evaluation used by the system.

Negation: safety rules

The appearence of variables in negated goals does not make them safe. For instance, Y in the following rule is always unsafe (the safety of X depends on the export used)

```
     p(X)  <-  Y > X, ~r(X, Y).
```

An unbound variable appearing in a negated goals can never be used in later goals of the rule. For instance

```
    print_nasty_people(X) <- ~nice(X),  person(X).
```

This rule will not compile. Thus, the order of goals in the rules must be switched.

Stratification

We cannot use negation in a recursive definition. We say that such a program is **non-stratified**.

Example: A non-stratified program, see Predicate Connection Graph.

```
even(0).
even(Y) <- ~even(X), Y = X + 1.
```

The correct way to specify this program is:

```
int(1).
int(Y) <- int(X), Y=X+1.
```

```
odd(X) <- int(X), X mod 2 ~= 0.
even(X) <- int(X), ~odd(X).
```

Now the negation is out of the recursion.

Carlo Zaniolo, 1997

Complex Terms

Every argument in a predicate can itself be a complex term consisting of a functor and several arguments (which in turn can be complex terms). The following example illustrates the use of complex terms to store shapes of different structure and of using these in performing computation.

```
%A DB of flat parts described by their geometric shape and weight.
%Different geometric shapes require a different number of
%parameters. Also actualkg is the actual weight of the
%part, but  unitkg is the specific weight where the actual
%weight can be easily derived from the area of the part

query: part-weight(No, Kilos)

part-weight(No, Kilos ) <- part(No, _ , actualkg(Kilos)).
part-weight(No, Kilos ) <- part(No, Shape, unitkg(K)),
                           area(Shape, A), Kilos= K * A.

area(circle(Dmtr), A1) <- A= Dmtr * Dmtr * 3.14/4.
area(rectangle(Base, Height), A1) <- A1= Base*Height.

%   part#   shape   , weight
part(322, circle(11), actualkg(34)).
part(121, rectangle(10, 20), unitkg(2.1)).
```

In computing this query on the first part we find that the goal in the first rule yields No=322 and Kilos=34 (the operation by which this goal and fact are made equal is called *unification*). The goal of the first rule fails to unify with the first fact, and instead unifies with the second one, yielding No=121, Shape=rectangle(10, 20), and K=2.1. Unsing these values, the area rules compute A and then then Kilos.

Lists

Lists are terms with two arguments, called the *head* and the *tail* of the list. Because lists are so common, a special notation is supported for lists, as follows:

- [ ] denotes an empty list,
- [X | Y] is a list with head X and tail Y
- [a, b, c, ...] is a short hand for [a | [b | [c ...]]]

Say for instance that we have facts as follows:

```
part(socks, [red, black, blue]).
```

From these facts we might want to list all the items, colors pairs (i.e., deriving a flat relation from a nested one) as follows:

```
query: part-color(Item, Color)

part-color(I, C) <- allcolors(I, [C|_]).
subL(I, Rest) <- subL(I, [C|Rest]).
subL(I, CL) <- part(I, CL).
```

Sets and Set Terms

While in lists the order of the items is important and the same item can be repeated several times, order and repetions are immaterial in sets. Thus, in the folowing list of parents and their children there is no way to tell which children was born first, nor to have two different children sharing the same name:

```
father(joe, {peter, mary}).
father(james, {lucy, arnold, jim}).
father(joe, {ann}).
father(jack, {}).
mother(magret, {peter, mary}).
mother(linda, {john, junior}).
```

query father(X, S) returns all father facts.

query father(joe, S) returns
{father(joe, {peter, mary}), father(joe, {ann})}

query mother(X, {mary, peter}) returns mother(magret, {peter, mary}),
since {mary, peter}={peter, mary} because the ordering is not important. In general set equality and unification of set terms accounts for the commutativity and idempotence properties:

- **Commutativity**:---sets having the same elements but a different order are equal.
  Example: {a, b} = {b, a}.
  by contrast, [a, b] $\neq$ [b, a].
- **Idempotence**:---sets having repetitious elements are equal to the same sets without the repetitions.
  Example: {a, a} = {a}.
  by contrast, [a, a] $\neq$ [a]

The set properties are ``built-in"; when lists are used to represent sets, the maintenance of these properties is the programmer's responsibility.

Sets Terms: Joining and Equality

Predicates can be joined on set arguments.

Example: ``*create all pairs of parents having the same children"*

```
same_children(X, Y, S) <- father(X, S), mother(Y, S).
```

This rule is equivalent to

```
same_children(X, Y, S) <- father(X, S),
                             mother(Y, S1), S = S1.
```

The equality predicate can thus be applied to sets. The system supports various builtin functions on sets, including set union, difference, intersection, membership, membership, subset_of and the predicate aggregate that computes set aggregates.

Builtin Aggregates

Again say that we the followin facts:

```
father(joe, {peter, mary}).
father(jim, {lucy, arnold, jim}).
```

Then to list the number of children for each father we can use a LDL++ builtin called aggr as follows:

```
query: no_of_children(X, No)
no_of_children(X, No) <-  father(X, C_Set),
                          aggr(count, C_Set, No).
```

This will return:

```
no_of_children(joe, 2)
no_of_children(jim, 3)
```

Thus, an aggr goal applies the aggregate in of first argument (count in the example) to the set shown in the second argument (C_Set in the example), yielding the value of the aggregate as its thirds argument (No in the example).

LDL++ supports SQL's five basic aggregates:

sum, count, max, min, avg.

When applied to an empty set { }, sum and count, return 0, while the other aggregates fail.

As discuss later, LDL++ supports user-defined aggregates.

Grouped-by Head Sets

LDL++ uses pointed brackets < > to structure the atoms derived in the head of the rules as nested relations. For instance, with suppp(Sup, Part, Price) a supplier-part-price relation, we can group by the parts, where the price is greater than 5 by suppliers as follows:

```
sup_parts(Sup, <Part>) <-
               suppp(Sup, Part, Price), Price >5.
```

Thus, the pointed brakets denote set of values being grouped by the remaining arguments in the head. The sets so constructed can then be used to compute aggregates. Thus, to count the parts costing more than $5 sold by each supplier has, we can write:

```
count_parts(Sup, Cp) <- sup_parts(Sup, Part_set),
                        aggr(count, Part_set, Cp).
```

LDL++, however support the computation of aggregates while the head sets are being computed. Thus, the number of parts costing more than $5 supplied by each supplied, can be expressed as follows:

```
sup_setof_parts(Sup, count<Part> )  <-
                        suppp(Sup, Part, Price), Price >5.
```

Therefore, set aggregates can be used directly in the head of the rules or applied to a set using the `aggr` predicate.

Duplicates in LDL++

The LDL++ system uses the following policy for duplicate elimination:

- Duplication in base relations is controlled by the keys of the relations
- Duplicates are eliminated from the tuples returned as query answers
- Duplicates are eliminated during the computation of recursive predicates
- Duplicates are NOT eliminated during the computation of nonrecursive rules
- For a head sets of the form p(X, <Y>) <- ... each value of X is returned only once (with its associated set of Y-values).

  For a head aggregate such as p(X, count<Y>) <- ... the first argument X is a unique key for the values produced by this rule. Similar rules apply to all other builtins.

Duplicates in Builtin Aggregates

Following SQL conventions, LDL++ tow different versions of builtin aggregates:

- count_dist, sum_dist, avg_dist disgregard duplicates for the builtins aggregates
- count_all, sum_all, avg_all takes into account duplicates

Take for instance:

```
q(X, count_dist<Y>) <- p(X, Y, Z).
e(X, count_all<Y>) <-  p(X, Y, Z).
p(a,b,1).
p(a,b,2).
p(b,c,1).
```

Then query: q(X,C) returns q(a, 1), q(b, 1).

But query: e(X,C) returns e(a, 2), e(b, 1).

Moreover,

- these differences are only significant for head predicates; aggregates called using aggr operate on sets which contain no duplicates, thus both versions return the same results.
- There is only one version, for the extrema aggregates min, and max
- The names sum, avg, and count are also allowed for these aggregates: in the current LDL++ implementation they, respectively, behave as sum_all, avg_all, and count_all.

The SEQ aggregates

Startin with version 5, LDL++ supports the two builtins seq and seq_dist. The first adds an unique sequence number to the tuples returned in the head, while the second does so after eliminating duplicates.

```
g(X,Y,seq<>)<-p(X,Y).
f(X,Y,seq_dist<>)<-p(X,Y).

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

Here query: g(X,C,S) returns:

```
g(a,b, 1)
g(a,c, 2)
g(a,b, 3).
```

But query: f(X,C,S) returns:

```
f(a, b, 1)
f(a, c, 2).
```

Observe that:

- seq<> and seq_dist<> are only used with an empty list of arguments
- seq_dist provides an efficient means to eliminate aggregates in the tuples produced by a rule

The following table summarizes the built aggregates:

Built-in Aggregates in LDL++		
Name	**Semantics**	**SQL Equivalent**
sum (sum_all)	Compute the sum of items in column ***including*** duplicates	SUM
`sum_dist`	Compute the sum of items in column ***excluding*** duplicates	SUM DISTINCT
avg (avg_all)	Compute the average of items in column ***including*** duplicates	AVG
avg_dist	Compute the average of items in column ***excluding*** duplicates	AVG DISTINCT
count (count_all)	Count the items in column ***including*** duplicates	COUNT
count_dist	Count the items in column ***excluding*** duplicates	COUNT DISTINCT
max	Find the maximum value in column	MAX
min	Find the minimum value in column	MIN
seq	Compute the sum of items in column ***including*** duplicates	Not Applicable
seq_dist	Compute the sum of items in column ***excluding*** duplicates	Not Applicable

Carlo Zaniolo, 1997

Built-in Predicates for Sets

Predicate name and Aguments	***Safe Binding Patterns***	***Comments***
member(E, S)	$E, $S	$E \in S$
	E, $S	
subset(S1, S)	$S1, $S	$S1 \subseteq S$
	S1, $S	
union(S1, S2, S)	$S1, $S2, S	
	$S1, $S2, $S	
	$S1, S2, $S	
	S1, $S2, $S	
	S1, S2, $S	
difference(S1, S2, S)	$S1, $S2, S	*S = S1 - S2*
	$S1, $S2, $S	
intersection(S1, S2, S)	$S1, $S2, S	$S1 \cap S2$
	$S1, $S2, $S	
cardinality(S, N)	$S, N	*N = | S |*
	$S, $N	
aggregate(N, S, R)	$N, $S, R	*abbr: aggr*

Carlo Zaniolo
2/13/1998

Top-Down Execution of Programs

In a bottom-up execution the goals of a rule r_j are computed before the head of r_j, and the results just obtained in the head are then used to compute the goals of other rules. A top-down execution occurs when values of some of the variables in the head are computed first, and their values are used in the computation of the goals. For instance:

```
%A DB of flat parts described by their geometric shape and weight.
%Different geometric shapes require a different number of
%parameters. Also actualkg is the actual weight of the
%part, but  unitkg is the specific weight where the actual
%weight can be easily derived from the area of the part

query: part_weight(No, Kilos)

r1: part_weight(No, Kilos ) <- part(No, _ , actualkg(Kilos)).
r2: part_weight(No, Kilos ) <- part(No, Shape, unitkg(K)),
                               area(Shape, A), Kilos= K * A.

r3: area(circle(Dmtr), A1) <- A= Dmtr * Dmtr * 3.14/4.
r4: area(rectangle(Base, Height), A1) <- A1= Base*Height.

%%   part#   shape   , weight
f1: part(22, circle(11), actualkg(34)).
f2: part(121, rectangle(10, 20), unitkg(2.1)).
```

The computation of this query can begin in a bottom-up fashion since fact f1 saitsfies the goal in rule r1, and thus we obtain part_weight(22, 34). Now, f2 satisfies the first goal in r2, binding the variable Shape rectangle(10,20). Now, the second goal in r2, passes down the current value of Shape to the head of f2. The result is that Base and Height are now bound to 10 and 20, respectively and the goal in the body can now be safely computed yielding A1=200. This value is then returned to A in rule r2 and used in the computation of the last goal of the rule. Therefore, the area rules operate as a procedure that is called from the area goals in the rule. This top-down passing of parameters from goals to rule heads is often required for effective computation. The procedure calling analog is often applicable to top-down computation. For instance if have a set of facts similar to this describing the colors in which a given item comes:

```
part(socks, [red, black, blue]).
```

From these facts we might ask a query such as, find how many colors a given part comes in:

```
query: part-color($Myitem, Color)

part-color(Item, Color) < - part(Item, ColorList), member(C, ColorList).

member(C, [C| \_]).
member(C, [\_ | Crest]) <- member(C, Crest).
```

Here the value of $Myitem is passed down to the first argument in part-color and Item. This is very similar to procedure calls where the actual parameters are passed down to the formal parameters in the procedure head. Consider now the body of the part-color rules, where the goals are executed from left to right. The goal part(Item, ColorList) is first execute with the first arguent bound to $\tt Item = \$Myitem$, binding ColorList. For instance if Myitems = socks then ColorList= [red, black, blue]= [red | [black | [blue]]]. Then this list is passed to the second argument of member using $unification$, to yield C= red and Crest=[black|[blue]]. The recursive calls to member then proceed, always returning the head of the list until the tail is the empty list []. Observe that in the execution of the rule above, the actual work is performed during the actual recursive call when the second argument in the body is extracted from the second argument in the head, as per the equivalent formulation of the previous recursive rule as:

```
member(C, L) <- L= [\_ | Crest],  member(C, Crest).
```

In Datalog, this is called right-linear recursion, which is largely equivalent to tail-recursion in Lisp. Right-linear recursive rules (and symmetrically, left-recursive rules where whole the computation is performed after the recursive call) rules are supported efficiently in LDL++, using a single fixpoint computation. We might have linear rules where computation must be performed before and after the recursive call. Consider for instance the following rules that count length of the list:

```
length([H|T], C1) <- length( T, C), C1=C+1
length([], 0).
```

For these recursive rules, some computation takes place before the recursive call and some computation after. Thus they will be supported using using the supplementary magic set method that requires two fixpoint computations. However, the length of a list can also be computed as follows:

```
length(L, Ll) <- clen(L, 0, Ll).
clen([H|T], C, Ll) <- C1=C+1, clen(T, C1, Ll).
clen([], C, C).
```

This program uses a right-linear rule, where the computation is performed before the recursive call. In general, while the left/right-linear formulation of programs should be attempted whenever possible, many computations (e.g., list-append) cannot be expressed in this form and will be implemented using the supplementary magic method.

Carlo Zaniolo, 1998

The IF-THEN-ELSE Construct

If---then---else only in the body of a rule

The meaning of

h <- if(p then q else w).

is strictly logical. The semantics of this rule is the same as:

```
h <-  p, q.
h <- ~p, w.
```

Thus, e.g., the compiler requires that the original program, so expanded must be stratified.

However, if-then-else is implemented directly, i.e., without rewriting the program. Thus it adds efficiency, since p is executed only once.

When there are side effectsi, e.g., in updates, the meaning is no longer the same.

Example: Compare two length measures, expressed in yards and inches.

```
gt_or_eq(length(Y1,I1), length(Y2, I2)) <-
        if(Y1 = Y2 then I1 >= I2 else Y1 > Y2).
```

Note that the syntax of the if-predicate) is the same as that of a predicate.

Omitting the else clause: The abbreviation,

h <- if(p then q).

is also allowed. Its meaning is

h <- if(p then q else true),

If-then-else and Existential Variables

A common programming mistake is the implicit use of the *else true* clause, which may result in an unsafe rule, as per the following example.

``If exceptions occur, then take corrective action and report the problem, otherwise take normal action.''

```
report_problems(X) <-
       if(problem(X) then corrective_action else normal_action).
```

export:report_problem($X) is safe but export:report_problem(X) is not; what do we report when there is no problem? A correct formulation would be:

```
report_problem(X) <-
   if(problem(Y) then X = Y, corrective_action else X = "noproblem", normal_action).
```

LDL++ forces you to write complete specifications.

A common mistake is to write the previous rules as:

```
report_problem(X) <-
   if(problem(X) then corrective_action else X = noproblem, normal_action).
```

This can be re-expressed using negation as follows:

```
report_problem(X) <-  problem(X), corrective_action.
report_problem(X) <- ~problem(X), X = noproblem, normal_action.
```

The error is that a non-existential variable is used before it is bound.

Nondeterministic Reasoning: Choice

The set-oriented semantics of logic programs lead to the generation of all answers satisfying a given set of rules. I many pratical situations, there is the need for picking an element out of of a set of candidate in nondeterministic fashion. Interesting enough, choosing an element, can be viewed as enforcing a functional dependency constraint.

Example: *Say that, our university database contains facts describing professors and fact describing students. In fact, say that our toy database contains only the following facts:*

```
student('Jim Black', ee, senior).

professor('Ohm', ee).
professor('Bell', ee).
```

Now, the rule is that the major of a student must match his/her advisor's major area of specialization. Then eligible advisors can be computed as follows:

```
eligadv(S, P) <- student(S, Major,Year),
                 professor(P,Major).
```

This yields:

```
elig_adv('Jim Black', 'Ohm')
elig_adv('Jim Black', 'Bell')
```

But, since a student can only have one advisor, choice goal will be used to force the selection of a unique advisor, out of the eligible advisors, for a student. Thus we obtain the following choice rule:

```
 actualadv(S, P) <-  student(S, Major, Levl),
                     professor(P, Major), choice((S),(P)).
```

The goal can also be viewed as enforcing a functional dependency (FD) ; thus, in , the second column (professor name) is functionally dependent on the first one (student name).

The result of executing this rule is nondeterministic. It can either give a singleton relation containing eithe of the following two tuples<

```
elig_adv('Jim Black','Ohm'), elig_adv('Jim Black','Bell')
```

A rule (program) where the rules contain choice goals is called a choice rule (program).

The semantics of a choice program *P* can be defined by transforming into a program with negation, *SV(P)*, called the stable version of *P*, which exhibits a multiplicity of total stable models, each obeying the FDs defined by the choice goals.

The use of choice is critical in many applications. For instance, the following nonrecursive rules can be used to determine whether there are more boys than girls in a database containing the unary relations boy and girl:

Example: *Are there more boys than girls in our database?*

```
match(Bname, Gname) <-   boy(Bname ),  girl(Gname).
                          choice ((Bname),(Gname)),
                          choice ((Gname), (Bname)) .
matchedboy(Bname) <-  match(Bname, Gname).
moreboys  <-      boy(Bname),  ~matchedboy(Bname).
```

The most significant applications of choice involve the use of choice in recursive predicates. For instance, the following program computes the spanning tree, starting from the source node , for a graph where an arc from node a to node b is represented by the database fact g(a,b):

Example: *Computing a spanning tree:*

```
st(root,a).
st(X,Y)<- st(,X), g(X,Y), Y ~= a, choice((Y),(X)).
```

In this example, the goal ensures that, in , the end-node for the arc produced by the exit rule has an in-degree of one; likewise, the goal ensures that the end-nodes for the arcs generated by the recursive rule have an in-degree of one.

Example: *Ordering a domain*

```
ordered-d(root,root).
ordered-d(X,Y) <-  orderedd(_, X), d(Y),
                    choice((X),(Y)), choice((Y),(X)).
```

Example:*The sum of the elements in d(X)*

```
sumd(root, root, 0) <-  d(_).
sumd(X, Y, SY) <-  sumd(, X, SX), d(Y),
                   choice((X),(Y)), choice((Y),(X)),
                   SY=SX+Y.
totald(Sum)  <- sumd(_, X, Sum), ~sumd(X, _ , _).
```

If we eliminate the choice goal from this program obtain a program that is stratified with respect to negation. Stratified choice programs always have stable models Moreover, these stable models can be computed by strata, as any other stratified program.

The choice construct is significant for both nondeterministic and deterministic queries. A nondeterministic query is one where any answer out of a set is acceptable. This is, for instance, in the previous example where an advisor ws assigned to a student. The use of choice to compute a deterministic query is illustrated by the sum example: the sum of the elements of a set is independent from the order in which these are visited.

Carlo Zaniolo, 1998

User-Defined Aggregates

The user can define a new aggregate, called newaggr, by writing the rules that support:
single(newaggr, Y, NV) and multi(newaggr, Y, OV, NV).

where

- single is the computation to be performed on a singleton set having Y as its only element.
- multi denotes how to compute the aggregate value NV for a set S' obtained by adding a new element Y to the the set S, where the value of the aggregate for S is OV.

For instance, if we had to define the max aggregate we would write:

```
single(max, Y, Y).
multi(max, Y, MO, MN) <-  Y > MO, MN=Y.
multi(max, Y, MO, MN) <-  Y <=  MO, MN=MO.
```

For efficiency considerations, we might use the if-then-else construct. to combine the last two rules Likewise, in LDL++, count and sum could have been defined as shown by the two pairs of rules below:

```
single(count,  Y, 1).
multi(count, Y, Old, New) <- New= Old+1.

single(sum, Y, Y).
multi(sum,  Y, Old, New) <- New= Old+Y.
```

The count and sum so defined behave as count_all and sum_all since these rules accumulate the Old value with the new Y, without checking whether the same Y value had already occurred.

User-defined aggregates can also be called by means of aggr goals. In this case, when applied to the empty set, the compiler will search for empty rule defining the behavior of that particular aggregate on an empty set. For instance, the our built-in aggregates behave as if they were defined by the following rules:

```
empty(sum, 0).
empty(count, 0).
empty(max, 0) <- false.
```

On empty set, aggr will return 0 for sum and count, and will fail on max (also fails on min and avg).

Several new aggregates can be defined using the single and multi rules. For instance in SQL, after the maximum is found, a second sub-query is needed to return all the values associated with the maximum. In LDL++, if sppp denotes a supplier-part-price relation, to find, for each supplier their most expensive items and their common price of these items, we can write:

```
findmax(S, mymax<(Itm,Pric)>) <- sppp(S, Itm, Pric).

single(mymax, (Item, Pr), (Item, Pr)).
```

```
multi(mymax, (Sit,Sp),(Oit,Op), (Sit, Sp)) <- Sp >= Op.
multi(mymax, (Sit,Sp),(Oit,Op), (Oit, Op)) <- Sp < Op.
```

This example illustrates that:

1. Aggregates can return full tuples, such as the pair (Item, Pr) produced by mymax
2. More than one value/tuple can be returned from the computation of an aggregate.

Return Rules

The aggregates defined via single and multi rules suffer from the following limitations:

- Values are not returned until the end of the computation. To express on-line aggregates, temporal aggregates on time series, and a number of aggregates required for data mining, values must be returned while the computation is still progressing.
- The aggregate returns values in one column. Often we want to return values in several columns.

To overcome these limitations LDL++, Version 5.0 and above, supports the the predicate return whereby the productions of answers can be explicitly controlled through rules. In its simplest form, the predicate return has four argument (same as multi):

```
return(newaggr, NewY, OldV, VR) <- ...
```

While in multi the last argument is stored in the accumulator, in return it is returned as a (partial) result. This value can be computed by user-defined rule from the new value in the input and the old value in the accumulator---same as for the multi rule. When no return rule is given for an aggregate being defined, then the last argument of multi is returned at the end of the computation--for compatibility with previous versions of LDL++.

Early Returns

Example. *Find suppliers who supply more than 7 items*

```
select(Sup) <- allcounts(Sup, CC), CC>7 .
allcounts(Sup, cntol<Itm>) <- sppp(Sup, Itm, Price).
```

where cntol can be defined as follows:

```
single(cntol, _, 1).
multi(cntol, S, Old, New) <- New= Old+1.
return(cntol, S, Old, Value) <- Old ~= nil, Value=Old+1.
```

The return rule is applied after each new value generated by either the single or the multi rule. But the single rule leaves the value of of Old equal to nil. To avoid a type error that will follow from the computation of Old+1, therefore, we have the condition Old ~= nil. The following example, illustrates the use of nil, to emulate the choice construct. Under the following definition of mychoice:

```
single(mychoice, Y, Y).
multi(mychoice,Y, nil, nil) <- fail.
return(mychoice, Y, nil, Y).
```

The following two rules are equivalent:

```
p(X, Y) <- q(X, Y), choice((X), Y).
p(X, mychoice<Y>) <- q(X, Y).
```

At the end of the dataset the value of the next input is set to nil.
This can be used to control the values returned at the end of the computation.
Example. *The aggregate avg could have been defined as follows:*

```
single(avg, X, (X,1)).
multi(avg,  X, (OS,OC), (NS,NC)) <- NS=OS+X, NC=OC+1.
return(avg, nil,(OS,OC), Avg) <- Avg= OS/OC.
```

The computation of average can be performed by computing the sum and the count and then returning their ratio every seven records. The value nil in the first argument of the return rule denotes that we have reached the end of the computation; nil is in fact *produced by the system when the end of the dataset is reached* .

Using return rules, an assortment of very useful aggregates, e.g., those used for data mining applications, can be defined. Moving window aggregates, for instance, are of common usage in time-series analysis.

Example. *Moving time window aggregation : Average the prices of IBM stocks over the last five days.*

```
p(mw5avg<A>) <- stock-closing('IBM',A).

single(mw5avg, X, [X]).
multi(mw5avg, X, OL, NL) <- if(OL= [X1, X2, X3, X4, X5]
                            then  L = [X1, X2, X3, X4]
                            else  L= OL), NL = [X | L].
return(mw5avg, _,[X5,X4,X3,X2,X1],Avg) <-
                               Avg= (X1+X2+X3+X4+X5)/5.
```

Arbitrary Number of Columns

The previous example that find the max priced items and their prices can be rewritten as follows:

```
findmax(S, maxtwo<(Itm,Pric)>) <- sppp(S, Itm, Pric).

single(maxtwo, (Item, Pr), (Item, Pr)).

multi(maxtwo, (Sit,Sp),(Oit,Op), (Sit, Sp)) <- Sp >= Op.
multi(maxtwo, (Sit,Sp),(Oit,Op), (Oit, Op)) <- Sp <= Op.

return(maxtwo, nil, (Sit, Sp), Sit, Sp).
```

The first three arguments in the head of a return rule denoted the name of the aggregate, the new value in the stream, and the old value in the accumulator. **Any additional argument is returned in a separate column**. Thus, in our case maxtwo returns Sit (max priced item) and Sp (its price) in two separate column. Thus, findmax is now treated as a ternary predicate. The occurrence of nil in the last rule denotes that we are defining a return that takes place when all the input values have been visited. **When the heads of return rules only have three arguments, this is boolean aggregate,** which produces no argument in the output. For instance the following aggregate determines whether the count exceeds the value of 7.

```
   single(count7, _, 1).
   multi(count7, _, Old, New) <- Old<7,  New=Old+1.
   return(count7, _, Old)  <- Old=7.
```

Thus, to find suppliers who supply more than 7 items we can write the following rule:

```
   select(Sup, count7<Itm> <- sppp(Sup, Itm, Price).
```

Multiple Aggregates in the Head

Multiple aggregates are allowed in the same head,

```
p(K1,K2,...,Km, aggr1<A1>, aggr2<A2>, ..., aggrN<An>) <- Rule Body.
```

under the following conventions:

1. The arguments in the head, that is, K1, ...,Km and A1,...,An must all appear in the body of the rule.
2. aggr1,...,aggrN can either be builtin aggregates or user-defined aggregates.
3. Each aggr1<A1>, aggr2<A2>, ..., aggrN<An> is grouped by K1,K2,...,Km where m denote a non-negative integer (thus an empty group-by list is also allowed).
4. The cartesian product of the results of aggr1, ...,aggrN will be returned for each new value X. Thus, if any of the N aggregates fail for a given X no value is returned at that point.

Monotone Aggregation

Traditional aggregates defined without an explicit return rule are nonmonotonic. A final return rules is one where the second argument in the head of the rule is the distinguished symbol nil. Aggregates defined using final return rules are nonmonotonic.

However, aggregates that have **early return rules** and no final return rules are monotonic. These aggregates can be used in recursive programs without restrictions. This leads to the simpler expression of complex algorithms.

Suppose we define a count-like predicate mcount as follows:

```
   single(mcount, Y,1).
   multi(mcount, Y, Old, New) <-  New=Old+1.
   return(mcount, Y, Old, New) <- if(Old=nil then New=1
                                      else New=Old+1).
```

Since the return rule operates on the new value of input and the old value of the accumulator, the situation Old=nil defines the value
to be returned after the application of the single rule. Therefore if p is a database predicate with *n* facts, then the rule

```
   q(mcount<X>) <- p(X).
```

returns I_n= { q(1), q(2), ..., q(n) }. If the original set of facts is increased to a new set of cardinality m > n, then, our rule returns: I_m= {q(1), q(2), ..., q(m)}, where I_m is a superset of I_n. Therefore:

Program rules with mcount *define monotone deterministic mappings.*

All aggregates inductively defined using single, multi and early-return rules define monotone mappings; although in general these mapping are nondeterministic (unlike mcount). The examples which follow show that, deterministic or otherwise, monotone aggregates provide a powerful and flexible tool for advanced applications.

Join the Party: Some people will come to the party no matter what, and their names are stored in a sure(Person) relation. But many other persons will join only after they know that at least K of their friends will be there. Here, friend(A, B) denotes that A views B as a friend.

```
   willcome(P)<-  sure(P).}
   willcome(P)<-  c_friends(P, K), K >= 3.
   c_friends(P, mcount<F>) <-  willcome(F), friend(P, F).
```

Here, we have set K=3 as the number of friends required for a person to come to the party.

By specializing the count aggregate, we can further improve the efficiency of the computation. Let us define an aggregate kcount as follows:

```
   single(kcount,(K,Y),1).
   multi(kcount,(K,Y),Old,New) <- Old<K, New=Old+1.
   return(kcount,(K,Y),K1,yes) <- K1+1=K.
```

Thus, the early return rule succeeds (producing a yes) only when the count reaches the value of K. Since we assume that k>1 we do not need to return the values produced by single. Also, the computation of multi fails after we return the value. Thus, the computation of party goers becomes:

```
   wllcm(F,yes) <- sure(F).
   wllcm(X,kcount<(3,F)>) <- wllcm(F,_), friend(X,F).
```

Unlike in the previous formulation, where a new tuple c_friends is produced every time a new friend is found, a new wllcm tuple is here produced only when the threshold of 3 is crossed. Rather than returning yes we should have programmed our aggregate to return no argument, i.e., to act as a boolean predicate. Then our program simplifies as follows:

```
  single(zcount, (K,X), 1).
  multi(zcount,  (K,X), Old, New) <- Old < K, New=Old+1.
  return(zcount, (K,X), K1) <- K1~=nil, K=K1+1.

  wllcom(F) <- sure(F).
  wllcom(X, zcount<(3,F)>) <- wllcom(F), friend(X, F).
```

Next, we define msum and mmin that provide monotone extensions for sum and min.

For msum we have:

```
single(msum, Y, Y).
multi(msum, Y, Old, New) <-  New = Old + Y.
return(msum, Y, Old, New)  <- if(Old = nil then New=Y
                                 else New=Old+1).
```

For mmin, we will return the last value if this is a new min.

```
single(mmin, Y,Y).
multi(mmin, Y, Old,New)  <-  if(Y < Old  then  New=Y
                                else  New=Old).
return(mmin, Y, Old, Y)  <-  if(Old ~= nil then Y < Old).
```

Least-Distance Connections: Given a graph g(X,Y, C) where C is the cost of an edge from node X to node Y, the least-cost distance between any two nodes can be computed as follows:

```
ld(X, Y, mmin<C>) <-  g(X,Y, C).
ld(X, Y, mmin<C>) <-  ld(X,Z, C1),
                      ld(Z, Y, C2), C= C1+C2.
least_dist(X, Y, min<C>) <- ld(X,Z, C1).
```

This transitive-closure like computation adds a new arc ld(X, Y, C) provided that this then becomes the new least-cost arc between the nodes X and Y. The arcs so produced are then used in the next step of the seminaive computation. At the end of this fixpoint computation, the least_dist rule is used to select the least-distance arc between these two nodes, out of the succession of arcs of decreasing C values produced in the computation. For a given graph, the values obtained during the computation of ld can vary depending on the order in which the arcs are considered. The final values in least_dist, however, are always the same (a nondeterministic computation producing a deterministic answer).

Company Control: Another interesting example is transitive ownership and control of corporations. Say that owns(C1, C2, Per) denotes that corporation C1 owns a percentace Per of the shares of corporation C2. Then, C1 controls C2 if it owns more than, say, 49% of its shares. In general, to decide whether C1 controls C3 we must also add the shares owned by corporations such as C2 that are controlled by C1. This yields the transitive control predicate defined as follows:

```
control(C, C) <- owns(C, _, _).
control(C1, C2) <- twons(C1, C2, Per), Per>49.
towns(C1, C3, msum<Per>) <- contrl(C1, C2),
                               owns(C2, C3, Per).
```

Thus, every company controls itself, and a company C1 that has transitive ownership of more than 49% of C2's shares controls C2 . In the last rule, twons computes transitive ownership with the help of msum that adds up the shares of controlling companies. Observe that any pair (C2,C3) is added at most once to control, thus the contribution of C2 to C1's transitive ownership of C3 is only accounted once. To further simplify the program and expedite the computation we can introduce a boolean aggregate as follows:

```
single(sum49, Y, Y).
multi(sum49, Y, Old, Z) <-  Old<49, Z= Old+Y.
return(sum49, Y, Old) <- if(Old=nil then Y>49
                            else Old+Y>49).
```

Then the recursive rules become:

```
   cntrl(C1, C2) <- owns(C1, C2, Per), Per >49.
   cntrl(C1, C3,sum49<Per>) <- cntrl(C1,C2),
                                 owns(C2,C3,Per).
```

Thus, sum49 succeeds only when the 49% threshold is crossed during the summation. Here, the value of 49 was cast into the very definition of our aggregate. Alternatively, this value could be given as a parameter, as in the case of kcount.

Bill-of-Materials (BoM) Applications: BoM applications represent an important application area that requires aggregates in recursive rules. Say, for instance that psb(P1, P2, QT) denotes that P1 contains part P2 in quantity QT. We also have elementary parts that are purchasable for a price and will be delivered in a certain number of days: these are described by the relation basic(P, Price, Days). Then, the following program computes the cost of a part as the sum of the cost of the basic parts it contains.

```
  part_cost(Part, O, Cst) <-  basic(Part, Cst).
  part_cost(Part, mcount<Sb>, msum<MCst>) <-
                part_cost(Sb,ChC,Cst), prolfc(Sb,ChC),
                psb(part,Sb,Mult), MCst=Cst*Mult.
```

Thus, the key condition in the body of the second rule is that a subpart Sb is counted in part_cost only when all Sb's children have been counted. This occurs when the number of Sb's children counted so far by mcount is equal to its total number of children in the psb graph. This last number is kept in the prolificity table, prolfc, which can be computed as follows:

```
   prolfc(P1, 0) <-        basic(P1, _).
   prolfc(P1, count<P2>)<- psb(P, P2, _).
```

Also, this BOM computation can be simplified and made more efficient using the zcount aggregate, yielding:

```
   pcost(Part, Cost)  <-  basic(Part, Cost).
   pcost(Part, zcount<(K,Sb)>, msum<Cst>) <-
                               pcost(Sb, yes, Cst),
                               psb(Part, Sb, Mult),
                               prolfc(Part, K),
                               MCst=Cst*Mult.
```

Observe that the prolfc relation is now used to qualify Part in the rule head, rather than its subparts in the body. The technique of counting the children could also be used with least_dist problem, above, if the underlying graph is acyclic. For cyclic graphs we must use the current formulation that exploits the property that extrema are unaffected by duplicates (idempotence).

Carlo Zaniolo, 1998

XY-Stratified Programs

In LDL++, negation and aggregates can be used in recursive programs provided that these are XY-stratified. The LDL++ compiler recognizes these (locally-stratified) programs and generates an efficient execution plan to construct their stable models. We will now concentrate on the practical aspects of XY-stratified programs, whereas the theory of these programs is discussed in Advanced Database Systems. Recursive predicates defined an XY-stratified program, use their first argument as a discrete-time *temporal argument*, or a state counter. Syntactically, a temporal argument is either

1. a non-negative integer constant---typically0 as in the first rule of the example below, or
2. a temporal variable, such as J, or
3. the expression J+1 denoting the successor of state J.

There is at most one temporal variable per rule. For instance, the ancestors of marc using the differential fixpoint (a.k.a. seminaive fixpoint) method can be computed as follows:

Example: *Seminaive computation of the ancestors of marc*

```
delta_anc(0, X, Y) <- parent(X,Y).
delta_anc(J+1, X, Z) <-  delta_anc(J, X, Y), parent(Y, Z),
                       ~all_anc(J, X, Z).
all_anc(J+1, X, Y)      <-  all_anc(J, X, Y).
all_anc(J, X, Y)        <-  delta_anc(J, X, Y).
```

The second rule and third rule in the example are Y-rules: a Y-rule is characterized by te fact that the temporal argument in the head is J+1, while J appears as the temporal argument in the goal of the rule. This leads to the natural interpretation that J+1 denotes the "new" values, whereas J denotes the "old" values from the last state.

A rule such as the last rule in the example above is an X-rule: an X-rule is one where all the temporal arguments in the rule are identical, e.g., in last rule above they all coincide with J. Thus, a a possible interpretation, of this rule, is that the "old" values of all_anc are derived from the "old" values of delta_anc. But, if we replace the temporal argument J by I+1, everywhere in the rule, this states that the "new" values of all_anc are derived from the "new" values of delta_anc. This second view of X-rules should be used since it leads to the simple view that the "new" values in the head are being computed using a mixture of old and new values from the body. Therefore, rules, 1-4, in the previous example can be interpreted as follows:

1. At state zero, parents become delta_anc delta_anc only contains marc
2. the new delta_anc are derived by finding the parents of the old delta_anc---minus those that already appear in old_anc
3. the new all_anc contain the old all_anc,
4. the new all_anc also include the new delta_anc

The new/old labelling given to the predicates also used by the compiler to check whether the rules defining recursive predicate form an XY-stratified program *P* where, a program is XY-stratified if it satisfies the following two conditions:

1. every rule in *P* is either an X-rule or a Y-rule
2. the program obtained by adding the "new" and "old" tags to the recursive predicates in *P* is non-recursive

After checking these conditions, the compiler turns the program into an efficient execution plan, where only the old copy and the new copy of each predicate table are stored; when the computation for the new copy is completed, this becomes the the old copy and the process repeats.

XY-stratification is very powerful and conducive to the elegant formulation of classical graph-oriented algorithms, that use aggregates in their computation. For instance, the computation of Floyd's Algorithm to compute the shortest distances between nodes in a connected graph can be expressed by the following program.

Example: *Shorters Distance in a graph g(X,Y,C) denotes an arc from X to Y of cost C.*

```
delta(0, X, X, 0) <-    g(X,_,_).
delta(0, Y, Y, 0) <-    g(_,Y,_).
delta(J+1, X, Z, min<C>) <- delta(J,X,Y,C1),
                            g(Y,Z,C2), C=C1+C2,
                            if (all(J,X,Z,C3) then C3>C).

all(J+1, X, Z, C) <-    all(J,X,Z,C), ~delta(J+1, X, Z, _).
all(J, X, Z, C)  <-     delta(J, X, Z, C).
```

Floyd's algorithm provides a more efficient computation for the same problem.

Example: *Floyd's algorithm: g(X,Y,C) denotes an arc from X to Y of cost C.*

```
delta(0, X, Y, C) <-    g(X,Y,C).

new(J+1, X, Z, C) <- delta(J,X,Y,C1),
                            all(Y,Z,C2), C=C1+C2,
                            if (all(J,X,Z,C3) then C3>C).

new(J+1, X, Z, C) <- delta(J,X,Y,C1),
                            all(Y,Z,C2), C=C1+C2,
                            if (all(J,X,Z,C3) then C3>C).

delta(J, X, Z, min<c>) <- new(J, X, Y, C).

all(J+1, X, Z, C) <-    all(J,X,Z,C), ~delta(J+1, X, Z, _).
all(J, X, Z, C)  <-     delta(J, X, Z, C).
```

Because of the lack of monotonicity, the usual differential fixpoint optimization cannot be used as such in the bottom-up execution of XY-stratified programs (also, many of these programs express sophisticated algorithms requiring ad-hoc optimization under the direct control of the programmer). However, top-down optimization techniques, such as the magic set optimization and the left/right linear rule optimization, are applied to XY-stratified programs. For instance, if we ask the question delta_anc(_,marc,X) LDL++ will use the specialization approach, whereby the second argument in the recursive predicate is instantiated to marc.

The copy and delete rule optimization, discussed next is unique to XY-stratified programs.

Copy Rules and Delete Rules

A copy rule is one where the head is identical to some argument in the body, except for the temporal argument. The last rule in our ancestor example is a copy rule. The LDL++ system maintains a new and an old copy of each predicate, and stores it without the temporal argument, that is stored globally for all recursive aggregates. In this scenario, the copy rule, basically tantamounts to the old predicate (all_anc, in our example), be copied into its new value. Therefore, the LDL++ system, stores only one copy of predicates such as all_anc, and the "new" and "old" versions are simply pointers to the same table. Then, a copy rule executes as a zero-cost no-op. The copy rule optimization also applies to the situation where the copy rule body contain goals that only share temporal variables with the copy goal. Thus, for the previous ancestor example, the last rule modified as follows,

```
all_anc(J+1, X) <-  all_anc(J, X), delta_anc(J+1, _).
```

is still a copy rule. In fact, the compiler simply evaluates the delta goal, and if true then the pointer for the "new" version all_anc is set to point to the "old" copy, otherwise it is set to point to the empty set. Consider now the following recursive program that computes the temporal projection by coalescing overlapping periods into maximal periods.

Example. *Coaleshing after temporal projection for* emp_dep_sal(Eno, D, S, Frm, To)

```
e_hist(0, Eno, Frm, To) <- emp-dep-sal(Eno, D, S, Frm, To).

overlap(J+1, Eno, Frm1, To1, Frm2, To2) <-
                        e_hist(J, Eno, Frm1, To1),
                        e_hist(J, Eno, Frm2, To2),
                        Frm1<=Frm2, Frm2<=To1,
                        distinct(Frm1, To1, Frm2, To2).
e_hist(J, Eno, Frm1, To) <-
                       overlap(J, Eno, Frm1, To1, Frm2, To2),
                       select_larger(To1, To2, To).
e_hist(J+1, Eno, Frm, To) <-
                 e_hist(J, Eno, Frm, To),
                 ~overlap(J+1, Eno, Frm, To, _, _),
                 ~overlap(J+1, Eno, _, _, Frm, To).

%%%%auxiliary predicates%%%%%
distinct(Frm1, To1, Frm2, To2)<- To1 ~= To2.
distinct(Frm1, To1, Frm2, To2)<- Frm1 ~= Frm2.
```

```
select_larger(X, Y, X) <- X>=Y.
select_larger(X, Y, Y) <- Y>X.
```

The fourth rule above is a *delete rule*; such rule is defined as follows:

- the head of the rule is identical to one of its goal (the copy goal), except for the temporal argument
- in addition to the copy goal, the rule has one or more *negated goal* having non-temporal variables in common with the copy goal,

The procedure to support copy rules is as follows: construct the new value for the copy predicate by deleting the value satisfying the negated goals from the old copy predicate. Thus, for the example at hand, the newly-generated values of overlap are visited, and they are thus deleted from the old e_hist. In general, the execution strategy used for delete rules is efficient under the assumption that the negated predicates are significantly smaller than the copy predicate. Another limitation of a delete rule is that the old version of its copy predicate cannot be used by later rules. When this condition is not satisfied, then the compiler uses the standard two-version approach to support this predicate. Thus, e.g., if the negated predicates are significantly smaller than the copy predicate, the delete rule rewriting can be avoided by simply switching the order of these two rules.

Returning Results and Termination

In a computation such as the ancestor example, the results can be collected incrementally, by simply specifying the goal

```
delta_anc(I, X)
```

The LDL++ system uses pipelining in the computation of XY-stratified programs, whereby, a new tuple is only generated when it is immediately needed by calling goal. Thus when no more tuples are requested by the our goal above, there is no longer any computation of the XY-rules. For instance, goals such as delta_anc(20, _) , or all_anc(_ , anode) are basically existential goals, and they never call the XY-rules computation again after they succeed.

For the more general situation, the goal delta_anc(I, X) fails when this delta relation becomes empty for some value of I and the XY-rules are never executed again after that. Therefore, while a straight bottom-up execution of the ancestor program suggest an infinite computation because of copy rules, the computation is in fact safe when the goal delta_anc(I, X) is used (but the computation would not be safe if the goal all_anc(I, X) is used).

In many programs, answers cannot be returned incrementally since only the values at the end of the computation are of interest. In this case, the programmer, should write rules to (i) identify the step at which the computation complete, and (ii) use the values computed at that step. For instance, for the temporal projection example, the followin rules can be used:

```
lastperiod(I) <- overlap(I,_,_,_,_,_),
                 ~overlap(I+1,_,_,_,_,_).
final_e_hist(Eno,Frm,To) <- lastperiod(I),
                                e_hist(I,Eno,Frm,To).
```

Thus, the final state is that for which there is no successor. Thus, a program calling predicates defined in XY-rules can only refer to two successive states (since only two states are stored, the previous states are lost). However, for copy predicates, only one state can be used by external computations (insofar as only one version of the predicate is kept by the system. For Floyd's algorithm we can write:

```
lastfloyd(I) <- delta(I,_, _, _),~delta(I+1,_,_,_).

floyd_results(X,Y,C) <- lastfloyd(I), all(I, X, Y, C).
```

Choice and User-Defined Aggregates

The choice operator can be used in the recursive rules of XY-stratified programs, provided that the rule's temporal variable appears in the first argument of the choice goal.

For instance, the following program extracts the least spanning tree, rooted in a node a, from an undirected graph represented by pairs of nodes g(X,Y,C), g(Y,X,C).

Example. *Prim's algorithm.*

```
prim(0, nil, a).          % nodes solved so far
newedgs(I+1, X, Y, C) <- solved(I, X), g(X,Y,C), ~prim(I,_, Y).
leastedg(I, min) <- newedgs(I, X, Y, C).

prim(I+1, X, Y) <- prim(I, X, Y).
prim(I,X, Y) <- leastedg(I, C), newedgs(I,X,Y,C),
                 choice((I), (Y)).
```

Here the choice construct is used to ensure that only one edge is added at each step (otherwise the resulting graph might not be a tree). Observe that the last rule is a choice rule, satisfying the condition that the temporal variable is in the first argument of choice. The significance of this requirement is that choice can only be used to enforce FD constraints for the values generated in a certain state (i.e., for a particular value of the temporal argument). Choice should not be used used to enforce constraints over the global temporal history.

As in SQL, the builtin aggregate min value, but not the edge at which this occur. But LDL++ supports the definition of new aggregates, which can be freely used in XY-stratified programs. Thus Prim's algorithm can also be expressed as follows:

Example. *Prim's algorithm using a new aggregate*

```
solved(0, a).          % nodes solved so far

prim(I+1, aleast<(X, Y, C)>) <- solved(I, X), g(X, Y, C),
                                ~solved(I,Y).
```

```
solved(I+1, X) <- solved(I, X).
solved(I, Y) <- prim(I, (X, Y, C)).

single(aleast,(X,Y,C), (X,Y,C)).
multi(aleast,(X1,Y1,C1),(X2,Y2,C2),(X2,Y2,C2)) <- C2 < C1.
multi(aleast,(X1,Y1,C1),(X2,Y2,C2),(X1,Y1,C1)) <- C2 >= C1.
```

Carlo Zaniolo, 1998

Meta Predicates

In LDL++, we provide the capability to reason with predicate names as if they were data. In other words, the argument to a predicate could be another predicate. The arguments to the inner predicates could be either constants or variables. If they are variables, we use a different representation for them. Unlike the regular variables, which are either capitalized or begin with an underscore (e.g.: A, X, Y, _salary, etc.) these variables are represented as "ldl_var(1), "ldl_var(2)" etc. Let us call these types of variables "Meta-variables." In the following sections we describe the syntax and semantics of these "Meta" constructs. We make use of a special built-in predicate called "meta_pred_list." This predicate is defined, informally, as follows:

```
meta_pred_list($Predicates List, $VariableList, BindingList).
```

Predicates List: At run time, this list should be bound to either a list of predicates or a list of lists of predicates. The arguments to these predicates can either be constants or Meta-variables.

Variable List: This is a list of Meta-variables that the user is interest in . Note that every member of this list should also be a member of the list of all variables occurring as arguments in the previous argument, Predicates List.

Binding List: This regular variable (not a Meta variable) unifies with the previous argument after those variables are bound.

Example: Using meta_pred_list

Let us show you an instance of the meta-predicate "meta_pred_list."

```
meta_pred_list( [ father(ldl_var(1), ldl_var(2)) ],
                [ ldl_var(1), ldl_var(2)) ]
                [ FatherName, ChildName ] )
```

The effect of using (or querying) the above meta predicate is the same as making a query on a unique predicate, say new_pred_1(FatherName, ChildName), where there is a rule of the form:

```
new_pred_1(A,B) <- father(A, B).
```

Let us now show you an example, where the "Predicates List" argument is a "list of lists of predicates":

```
meta_pred_list( [father(ldl_var(1), ldl_var(2))],
                [mother(ldl_var(1), ldl_var(2))] ],
                [ ldl_var(1), ldl_var(2) ],
                [ ParentName, ChildName ] )
```

The meaning of the above query is the same as querying a unique predicate, say new_pred_2(ParentName, ChildName) where the following two rules are defined:

```
new_pred_2( ParentName,ChildName ) <-
         father( ParentName, ChildName ).
new_pred_2( ParentName,ChildName ) <-
         mother( ParentName, ChildName ).
```

As can be seen from above, you may pass the specifications for any arbitrary rule inside the *LDL++* meta predicate "meta_pred_list." In the next example, we show a complete *LDL++* program demonstrating the use of the meta predicate.

The formal definitions of the various terms involved in a Meta-Predicate are given below.

- Meta-Variable:

A Meta-Variable is represented as an LDL++ complex object, ldl_var(Vname), where Vname is the name of the Meta-Variable represented by a constant. We denote a LDL++ Meta-Variable as V.

- Meta-Predicate:

A Meta-Predicate is represented as an LDL++ functor except that it may contain LDL++ Meta-Variables, We denote a LDL++ Meta-Predicate as P.

Given meta_pred_list(PredicateList, VariableList, BindingList):

1. VariableList should be of the form [V_1, ..., V_n] for n > = 0,
   where V_1, ...,V_n are Meta-Variables.
2. BindingList should be of the form [T_1, ..., T_n]for n > = 0,
   where T_1, ...,T_n are LDL++ terms.
3. PredicateList could be either of the following two forms:
   3.1 [P_1, ..., P_i] for I>= 1 or
   3.2 [[P_{1l}, ..., P_{1k}],..., [P_{j1}, ..., P_{jk}]] where j>= 1 and k>= 1

Carlo Zaniolo, 1997

Database Updates

$\mathcal{LDL}^{++}$ has two operators ``+" and ``-" that can be applied to base relations to add and delete facts from a relation. Changes to a base fact are accomplished by a delete followed by an addition.

Example:

Delete from the city relation all of the cities whose population is below some given threshold Size.

```
delete_small_cities(Size) <-
           city(Name, State, Population),
           Population < Size,
           -city(Name, State, Population).

export delete_small_cities($Size)
```

query delete_small_cities(100000)

When applied to the cities database, the remaining facts are:

```
city(Houston, Texas,     3000000).
city(Dallas , Texas,     2000000).
city(Huntsville, Texas,   150000).
city(Austin, Texas,       750000).
city(San Antonio, Texas, 1500000).
```

Semantics of Update Constructs

Set Oriented Semantics, as in Relational Databases.

- Updates (-b, +b) can only be applied to base relations b.
- A rule containing an update operation is called an *update rule*.
- The update rule has two parts: a *query part* and an *update part*.
- The query part is used to ``mark" the tuples that will be affected during the update and can use the full power of $\mathcal{LDL}^{++}$.
- Once marked, the tuples are added/removed in the update part of the rule: *snapshot semantics*.

$$\underbrace{delete\_small\_cities(Size) \leftarrow \overbrace{city(\_,\_,Pop), Pop < Size,}^{query\ part} \overbrace{-city(\_,\_,Pop)}^{update\ part}.}_{update\ rule}$$

Order of Execution

the *order* of update specification may affect the result. The query part of the program is still declarative as before.

Suppose that the employee relation contains two tuples with the same name but different salaries.

employee(pat, 'CHPC', 1000).
employee(pat, 'MCC', 1100).

We define an update rule to give all employees some specified raise.

```
raise(P) <-
   employee(Name, D, Sal),
   NewSal = Sal * P,
   -employee(Name, D, Sal),
   +employee(Name, D, NewSal).

export: raise($P)
```

The goals following an update goal see the results of the update. Thus, even if Name is a key, no key constraint violation will follow from this rule.

Updates: Limitations

Limitations: recursion

Updates are not permitted in a recursive rule. The following is incorrect:

```
p(X) <- p(Y), +b(Y).
```

This update is incorrect because *all* of the tuples must be marked in the query part *prior* to the updating; this is, as in negation, a requirement for a stratified program. On the other hand,

```
p(X) <- q(Y), +b(Y).
```

where q(Y) itself is recursively defined is ok.

Limitations: no union rules

No union of update rules. Suppose that we want to give somebody a raise of 10% if his/her salary is below 1000 and otherwise we want to give him/her a 5% raise.

```
raise(Name, Sal, NewSal) <-
           employee(Name, Sal),
           Sal <= 1000,
           NewSal = Sal * 1.1,
           +employee(Name, NewSal).
raise(Name, Sal, NewSal) <-
           employee(Name, Sal),
           Sal > 1000,
           NewSal = Sal * 1.05,
           +employee(Name, NewSal).
```

```
export:raise(Name, Sal, NewSal)
```

This construct is illegal since we do not know the order of execution of these rules. We can use the if-then-else construct to overcome this problem.

Using Conditionals instead of union rules

The correct version of the previous program uses the if-then-else predicate.

```
raise(Name, Sal, NewSal) <-
            employee(Name, Sal),
            if(Sal <= 1000 then NewSal = Sal * 1.1
                           else NewSal = Sal * 1.05),
            -employee(Name, Sal),
            +employee(Name, NewSal).
```

Failing Goals after Updates

No Failures After Updates

In LDL++ failures cannot be tolerated after the update part of a rule for two reasons: 1) A failure entails the ``undoing'' of the updates to the base relations---an operation that should be avoided; 2) For compatibility with relational databases, e.g., violations of integrity contraints, can be detected immediately.

Example: If w(X, Y) can fail in the rule,

```
p(X) <- q(Y), -b(Y), w(X, Y).
```

then the rule might not have a clear meaning. The compiler will issue a warning.

The normal solution consists in moving the potentially failing predicate before the update:

```
    p(X) <- q(Y), w(X, Y), -b(Y).
```

In some cases this solution is impractical, because, e.g., we want to see the result of the update. In such a case we must ensure that the goal after the update is *infallible*.

Updates and Assignment as Infallible Predicates

Updates and assignment statements are unfailing (see Infallible Predicates) and can be used after updates. Thus, the following is acceptable.

```
raise1(Name, Sal, NewSal) <-
            employee(Name, Sal), Sal <= 1000,
            -employee(Name, Sal),
            NewSal = Sal * 1.1,
            +employee(Name, NewSal).
```

The derived raise1 predicate is itself labelled as an update predicate, so it cannot be followed by a failing goal when used in rules. The compiler checks the condition that no goal is unfailing after updates and complains otherwise.

IF (Condition ELse True) is Infallible The result of the execution of the if predicate can succeed even though the condition within the predicate fails.

Example: This will work only for export:p($X).

```
   p(X) <- q(Y), -b(Y), if(w(X,Y) then true).
```

Here the ``if'' succeeds even if w(X, Y) fails. The rule can be modified to work also for export:p(X).

```
   p(X) <- q(Y), -b(Y), if(w(X,Y) then true else X="empty W").
```

The Forever Construct

The ***forever*** predicate in a rule:

$$h \longleftarrow g, forever(p), q.$$

Is interpreted as:

$$h \longleftarrow g, p_1, p_2, \dots, p_n, q.$$

where p_k, $k = 1, ..., n-1$ are successive versions of the goal ***p*** all of which succeed and with p_n the last succeeding goal. Note that each p_i operates on the state left by the execution of the previous goal p_{i-1}. If for some i, p_i causes no updates, then there are no further state changes, $p_j, i < j \leq n$ are ignored and we resume with ***q***.

Observet that the forever-goals are also infallible.

Forever: Example

Continue giving 10% salary raises to employees of the toy department until John's salary exceeds some number N.'

```
iterRaise(N) <- forever(eds(john,D,S), S<=N, raise(toy)).

raise(D) <-
      eds(E,D,Sal), Sal1= 1.1*Sal,
      -eds(E,D,Sal), +eds(E,D,Sal1).
```

Note than N is imported into the forever predicate. We cannot however export values as a result of its execution; therefore, its effects are manifested throught the update(s) of base relations.

Imperative Programs

Forever: ``Do procedurally what cannot be done declaratively''

Using the Parts database, Find the cost of each part P. Then find all the subparts of P, each with their quantity and cost, and compute the total cost. We will augment the Parts database by an additional table cost(Part:string, Cost:integer) which will be updated to contain the cost for each part, basic or assembly.

```
% Update cost with all basic parts and their costs.
basic_costs <- part_cost(Basic_part, -, Cost, _), +cost(Basic_part, Cost).

% Update cost with all assembled (non-basic) parts and their costs.
assembly_costs <- forever( assembly(Part, _, _), ~unresolved_part(Part),
                  tally_costs(Part, Total), +cost(Part, Total) ).

% unresolved_part contains those parts for which we do not have as yet a cost
% in the cost relation.
unresolved_part(P) <- assembly(P, Sub, _), ~cost(Sub, _).

% tally_costs sums up the costs of all sub parts making up a part.
tally_costs(P, Total) <- get_all(P, Set_of_subs), aggregate(sum, Set_of_subs, Total).

% get_all computes the sub cost of a sub part by multiplying its quantity times
% the unit sub part cost.
get_all(P, <Prod>) <- assembly(P, Sp, Qty), cost(Sp, SpCost), Prod = SpCost * Qty.
```

Carlo Zaniolo, 1997

External Databases (client-server)

LDL++ permits the use of external database relations in addition to locally defined relations. The program or the rule base makes no semantic distinction between these different kinds of relations. Of course the syntax for declaring these external relations as schema in a LDL++ program is possibly different for different kinds of data sources, eg: Sybase, Ingres etc.

The current release of the LDL++ system has support Sybase, Oracle and DB2.

The following is the schema declaration needed for Sybase relations:

```
database( {
sybase::<Relation>:(<field 1>:<Type 1>,
<field 1>:<Type 2>,
<field n>:<Type n>)
local_name <Local relation name>
from <Sybase server name>
use <Database name>
user_name <User name>
application_name <Application name>
interface_filename <Sybase interface file name>
password <Password>
})
```

The schema definition makes use of the following key words:

local_name: This is the local name to be given to the external relation.
This avoids conflicts if we have two relations from different sources with the same name.Type:string.

from: This specifies the database server we want to use. Type:string.

use: This specifies the database we want to use from a given server.
Type:string.

application_name: The name of the application. Type:string.

interface_filename: This specifies the location of the interface file
for sybase applications. Typically this file contains the communication port addresses for the servers. Type:string.

password: The password of the user. Type:string.

Now the sybase relation can be used as a local relation by using its local name as specified above.

Connecting to External Database with JDBC

The **Java LDL++ driver** (JavaLdlServer.class) connects to external databases using JDBC. LDL++ communicates with this driver which in turn communicates with external databases.

Configuring a Java LDL++ driver

The **Java LDL++ driver** is configured by editing file "jdbc.server" which must be in the same directory as the driver (JavaLdlS erver.class). The following is a sample configuration for an MySQL database and an Oracle server:

```
mysql.driver = twz1.jdbc.mysql.jdbcMysqlDriver

mysql.url = jdbc:z1MySQL://vesuvio.cs.ucla.edu:3306/test

oracle.driver = oracle.jdbc.OracleDriver
oracle.url = jdbc:oracle:
```

Basically, the configuration specifies two things. One is the jdbc driver, the other is the URL of the database server. This Java LDL++ driver is configured to talk with MySQL and Oracle database server. The line starting with "mysql.driver" specifies the JDBC specifies the JDBC/MySQL driver. The line starting with "mysql.url" specifies the URL and the protocol for the MySQL system.

Using External Database via JDBC

The following syntax is used in the LDL++ schema to denote an exteranal database accessible via JDBC:

```
database( {
        jdbc::employee(Name:string, Dept:string, Sal:integer)
                local_name emp
                from 'vesuvio.cs.ucla.edu:mysql'
                user_name hxwang
                password lapid
        } ).
```

Here, "employee" is a table managed by MySQL and is refered by this LDL++ program with the local name of "emp". Also, "vesuvio.cs.ucla.edu" is {\em the machine where the **Java LDL++ driver** is running on}.

The next example showa a connection to an Oracle server. The **Java LDL++ driver** in this example is running on "cheetah.cs.ucla.edu".

```
database( {
        jdbc::temp2(Name: integer)
                local_name temp
                from 'cheetah.cs.ucla.edu:oracle'
                user_name hxwang
                password dbpasswd,
        jdbc::temp5(ID: integer, NAME:varchar, SALARY:float)
                local_name emp
                from 'cheetah.cs.ucla.edu:oracle'
                user_name hxwang
                password dbpasswd
        } ).
```

Carlo Zaniolo. Jul,1998

Foreign Language Functions

In LDL++, the user may call functions written in C/C++. These functions (henceforth called functions) are used as predicates in the LDL++ system. Since, unlike functions, predicates may possibly return more than one value, the construction of these functions becomes slightly complicated. We provide a set of accessory functions to be used in writing these functions. In this section we shall explaining the mechanisms involved in writing a foreign function with the help of an illustrative example.

The *return* values for the foreign function have to be passed in as arguments of type **LdlObject**. Inside the funtion, the user accesses for reading only the *input* arguments. Similarly, all the *output* arguments are assigned or set in the function. The return value of the function is of type **LdlStatus** which signifes wither a logical *success* of a *failure*.

Since we might have more than one value being returned after a call to a foreign function, this function should be prepared to handle multiple calls. This feature requires the function to have some sort of a state associated with it to distinguish between several calls in the same session (a session here implies the duration from the instant the first result is generated until the last result is generated).

We also need to distinguish between the first time a function is called, with a new set (possibly null) of values, and subsequent calls. This is done typically to create the state structure, initialize any state variables etc., the first time the function is called. To find out if this was the first time the function was being called, we provide a convenience function **ldl_entry_p**0. This function returns a non-zero value if indeed the function was called the first time with a new set of values--it returns a zero otherwise. Various other functions are provided to extract data from LDL++ objects and to create LDL++ objects. Upon a successful execution of the function the user is expected to return the value **LDL_SUCCESS**, otherwise the user is expected to return the value **LDL_FAIL**.

The user is expected to compile this external function file and create a binary file. This binary file has to be specified as part of the import declaration in the LDL++ program. This basically tells the LDL++ system where to pick up the definition for the externally defined predicate.

Consider now the example **all_letters**. This function essentially takes a *string* as an input parameter and assigns each letter of the string and its position in the string to the two output parameters. It returns a value **of LDL_SUCCESS** as long as it can find such a pair, or it returns a value of **LDL_FAIL**. Let us examine each part of this code.

```
#include <ldl_extern.h>
#include <string.h>
```

Here we are including two files. The first file contains all the declaration of the LDL++ accessory functions. It also contains definitions of some LDL++ objects. The second file, string.h, is the standard C include file containing declarations for string manipulation functions.

```
extern "C"
{
```

```
LdlStatus all_letters
(LdlObject str, LdlObject ch, LdlObject pos);
}
```

Here we enclose the function declaration in an extern statement in order to suppress the name mangling performed by most C++ compilers. The function is declared to accept three arguments of type **LdlObject** and is expected to return a value of type **LdlStatus**.

```
typedef struct
{
int position; // Current position
int length; // Length of string
} State;
```

This is the definition of the state structure we intend to use for this function. It contains two state variables, one to store the current position within the string and the other to store the length of the entire string.

```
LdlStatus all_letters (LdlObject str, LdlObject
ch, LdlObject pos)
{
State* state;
LdlStatus status;
char ret_string [2];
char* c_string;

if (ldl_entry_p())
{
// This is the first time around
// Create the state structure
state = (State*)ldl_create_state(sizeof(State))
// Initialize the position
state->position = 0;
if (c_string = ldl_get_string(str))
state->length = strlen(c_string);
else
state->lenth = 0
}
```

In this section of the code, we examine if this was the first time this function was invoked. If that were the case, we created the state structure by using a LDL++ system function, **ldl_create_state**. We pass to this function the size in bytes of the state structure we are intercede in creating. Once we have the state, we initialize the state variable *position*. To initialize the other variable, *length*, we first need to extract the C string from the input parameter, str and compute its *length*. This length is now assigned to the state variable length. This ends the initialization process. Now we are ready to perform the actual processing.

```
// Recover the state from the LDL++ system
state = (State *)ldl_get_state();
if ((c_string = ldl_get_string(str))
&&
(state->position < state->length))
{
```

```
ret_string[0] = c_string[state->position];
ret_string[1] = '\0';


ldl_make_string(ch,
ret_string);
ldl_make_int(pos,
state->position);
state->position++;
status = LDL_SUCCESS;
}
else
status = LDL_FAIL
return(status);
}
```

This section of the code is executed every time this function is called. We first recover the state using the LDL++ system function **ldl_get_state**. We then examine if our current position is less than the length of the string. If true, this implies that we have some more values to return. We extract the "C" string from the input parameter and create a string of one character picked up from the current position. We use the system function, **ldl_make_string**, to create a string and assign it to the output parameter.ch. Similarly, using **ldl_make_int**, we create a integer denoting the corresponding position of the character and assign it to the other output parameter, *pos*. Then we increment the state variable, *position*, to look at the next character (if any) during the next call of this function. Finally we return a value of **LDL_SUCCESS**. Once all the characters have been processed, we return a value of **LDL_FAIL**.

This link shows a complete LDL++ program using the external C++ function described above.

In general, to import an external C/C++ function as a predicate, you need to declare it as follows:

```
import foreign
PredicateName( Argl: Type,
Arg2: Type, ... ) from 'file1.o',
'file2.o', ....,
'lib1.o', 'lib2.o', ... , 'libn.o'.
```

For our example, the declaration would be:

```
import foreign all_letters( $Word: string,
Letter:string, Position: integer_ from
'/usr/local/ldl++/demo/all_letters.o'
'/usr/lib/libc.a'.
```

The C++ file is assumed to be located at **'/usr/local/ldl++/demo/all_letters.o'**. Once we have this declaration, all_letters can be treated as an LDL++ predicate.

Summary of UI Commands

```
pwd
ls               [ < directory> ... ]
cd               [ <directory> ]
popd             [ <directory> ]
pushd            [ <directory> ]
more             [ <filename> ]
unix             [ <unix command string> ]

open             [ <ldl++ schema and/or rules filename> ... ]
load             [ <ldl++ schema and/or rules filename> ... ]
close
initdb           [ <ldl++ facts filename> ... ]
savedb             <ldl++ database filename>
getdb            [ <ldl++ database filename> ]
releasedb
compile          [ <exported query form> ... ]
query            [ <predicate> ]
insert           [ <ground predicate> ]
delete           [ <predicate> ]
display          [ "schema" | "rules" | "facts" |
                   "export" | "import" | "module" |
                    <predicate name>"/"<arity> ]
default          [ "module" | "schema" ] [ <module name> ]
exit

!!
![ <pattern> ]
history
alias            <alias_name> [ <alias_string> ]
unalias          <alias name>
set              [ <option> ] [ <option value> ]
edit             [ "-e" <editor> ] [ <filename> ]
```

Remarks

[ ] means optional.
... means a sequence.

UNIX-BASED COMMANDS

```
pwd
ls               [ <directory> ... ]
cd               [ <directory> ]
popd             [ <directory> ]
pushd            [ <directory> ]
more             [ <filename> ]
unix             [ <unix command string> ]
```

REMARKS

Similar to the corresponding commands on unix shell.

BASIC COMMANDS

```
open            [ <ldl++ schema and/or rules filename> ... ]
close
initdb          [ <ldl++ facts filename> ... ]
savedb          <ldl++ database filename>
getdb           [ <ldl++ database filename> ]
releasedb
compile         [ <exported query form> ... ]
query           [ <predicate> ]
insert          [ <ground predicate> ]
delete          [ <predicate> ]
display         [ "schema" | "rules" | "facts" | "export" | "import" | "module" |
                  <predicate name>"/"<arity> ]
default         [ "module" | "schema" ] [ <module name> ]
exit
```

REMARKS: Main commands and their meaning

- "load" and "open" are the same command and they load schema and rules from a file.
- "initdb" initializes the local database with facts in predicate format from a file.
- "close" removes the current set of schema, rules and facts.
- "savedb" dumps the local database into a file with internal format.
- "releasedb" destroys all data in the local database.
- "getdb" loads the local database from a file with internal format.
- "compile" compiles a sequence of query forms into an interpretive form.
- "query" queries against the local database.
- "insert" adds a tuple into the local database.
- "delete" removes tuple(s) from the local database.
- "display" shows all or part of the current environment of the system.
- "default" shows or changes the default schema and rule module.
- "exit" quits from the system.

Other Commands

```
!!
![ <pattern> | <history index> ]
history
alias           <alias_name> [ <alias_string> ]
unalias         <alias name>
edit            [ "-e" <editor> ] [ <filename> ]
set             [ <option> ] [ <option value> ]
```

REMARKS:

- Command completion is provided and the user is prevented from typing in wrong commands.
  "!!" executes the last command.
  "!<pattern>" executes the past command that has a prefix matching the pattern.
- "!<history index>" executes the past command with that history index.
- "history" shows the history of all entered commands.

- "alias" allows additional naming of commands.
- "unalias" removes an alias.
- "edit" invokes the appropriate editor.
- "set" changes the vlaue of the various options, described below:

OPTION TYPES	OPTION VALUES	DEFAULT VALUES
compile	{ shallow, deep }	shallow
editor	{ emacs, vi, ... }	emacs
duplicate	{ yes, no }	no
reorder	{ yes, no }	no
trace	{ 0, 1, ... 4 }	0
optimizer	{ 0, 1, ... }	0
timer	{ on, off }	off
database_size	?	8388608 (8 Mbytes)

Return to: Table of Contents

Carlo Zaniolo, 1997