

# Using Modules and Externals in $\mathcal{LDL}$

Danette Chimenti      Ruben Gamboa      Ravi Krishnamurthy

MCC  
3500 West Balcones Center Drive  
Austin, TX 78759-6509

November 10, 1999

## Abstract

Modules in  $\mathcal{LDL}$  provide the traditional advantages of separate compilation. They also address the difficulty of optimizing large programs and provide a means to eliminate common subexpressions. Externals in  $\mathcal{LDL}$  allow the wealth of procedural routines already in existence to be exploited. In addition, they improve the efficiency and expressiveness of the language. In this paper, new constructs are introduced for partitioning large  $\mathcal{LDL}$  programs into modules. Communication among modules is addressed by providing a means of exporting and importing predicates, termed global predicates. We then show how global predicates could be written in a procedural language, such as C or FORTRAN. We present a library that can be used by these external predicates to interface with  $\mathcal{LDL}$ . Finally, we show how existing functions may be called directly from  $\mathcal{LDL}$ .

## 1 Introduction

In this paper, we describe modules and externals in  $\mathcal{LDL}$ . We do not attempt to provide a complete formal semantics for their use. The formal semantics of  $\mathcal{LDL}$  are presented in [NT89]. In [CGK89], the semantics are extended to include modules and externals.

There are many motivating reasons for introducing *modules* into  $\mathcal{LDL}$ . The most obvious is the traditional advantage of modular decomposition in programming. More specific to  $\mathcal{LDL}$ , modules address the difficulty of optimizing large programs by allowing unrelated components of the program to be optimized locally. For instance, goal ordering may be specified for certain

sections of the program, thus reducing the optimization effort. Modules also provide a means to eliminate common subexpressions. They can be viewed as subroutines that store computed results, thus avoiding recomputation upon subsequent invocations. This reduces the size of the target code generated, and can also reduce the execution time of the program. In this paper, we introduce new constructs for partitioning large  $\mathcal{LDL}$  programs into modules. Communication among modules is addressed by providing a means of exporting and importing predicates, termed *global predicates*.

There are also many reasons for introducing *externals* into  $\mathcal{LDL}$ . Although  $\mathcal{LDL}$  has been shown to be an extremely powerful language, providing elegant problem-solving capabilities [NT89], there are functions that are more naturally expressed in procedural languages, e.g., iterative methods from numerical analysis. Furthermore, there are some functions that cannot be expressed at all in pure  $\mathcal{LDL}$ , e.g., some higher-order functions. We would like the flexibility of writing such functions in other languages and then invoking them from  $\mathcal{LDL}$ . In addition, a sophisticated programmer may be able to rewrite some  $\mathcal{LDL}$  predicates more efficiently in a low-level procedural language. We would like the capability to design and test a large program in  $\mathcal{LDL}$  and then rewrite, say in C, those predicates that are critical from a performance perspective. Finally, a large number of routines have already been written in conventional languages. We would like to take advantage of this wealth from the  $\mathcal{LDL}$  environment.

A procedure that has been coded in a different language can be viewed by  $\mathcal{LDL}$  as an *external predicate*. All the properties of global predicates in  $\mathcal{LDL}$  hold for external predicates as well. In fact,  $\mathcal{LDL}$  makes no distinction between global and external predicates. This means that an external predicate must behave as an  $\mathcal{LDL}$  global predicate, i.e., it must be able to handle all  $\mathcal{LDL}$  data types and return multiple solutions. Since procedural languages traditionally do not provide these capabilities, we introduce a library of routines that can be used by externals to interface with  $\mathcal{LDL}$ .

When an external predicate uses only the simple  $\mathcal{LDL}$  data types, i.e., numbers and atoms, and returns only a single solution, the library routines mentioned above are not necessary. These predicates are viewed by  $\mathcal{LDL}$  as *external functions* and correspond to the procedural notion of a subroutine. External functions are particularly useful, since they allow existing routines to be called directly from  $\mathcal{LDL}$  without modification.

Modules are introduced in section 2. External procedures are discussed in section 3 and external functions in section 4. A complete BNF description of the syntax for modules and externals is presented in Appendix A. A complete description of the library routines can be found in Appendix B.

## 2 Modules

The mechanics of modules are quite simple. A module contains a set of rules which define predicates that are local to the module. Local predicates are visible only within the module in which they are defined. A module can export a query form for any of its local predicates and import query forms that have been exported by other modules. The query form supplies information on how a predicate will be invoked, i.e., which arguments will be variables and which will be constant terms [NT89]. A predicate that is exported by a module is said to be global. As in [CGK89], we do not allow two global predicates which are defined in different modules to be mutually recursive. That is, they may not be defined in terms of each other.

A module *mod\_name* consists of one or more *module components*. Each module component has the following general structure.

```
module mod_name
    import_statements
    export_statements

    rules
end mod_name
```

All rules defining a predicate *p* must belong to the same module component. However, note that the definition need not be completely local in the sense that it can use a predicate *q* defined in a separate component. Moreover, *p* and *q* may be mutually recursive. A predicate local to a module is visible not only in the module component in which it is defined, but also in all other components of that module. Each module component need only import and export those query forms that are relevant to that particular component. However, a predicate that is imported in a module component will be visible in all components of that module, just as with local predicates. A predicate may only be exported in the module component where it is defined. There are no restrictions governing the mapping between modules and files. A file may contain many different module components and the components of

a given module may be split among different files. A module component, however, may not be split among files. Also, there may be at most one module component of a given module in a single file. The end statement of a module component is optional.

The basic form of an import statement is as follows.

```
import [ recomputed ] global_queryform
      [ from mod_name ]
      [ as local_predname ].
```

The use of modules normally assures that results are computed only once and then saved to avoid future recomputation. In some cases, however, recomputing the results is less expensive than the overhead of storing them. The `recomputed` option is used to indicate that results are not retained between invocations of the module and, therefore, must be recomputed. If *global\_queryform* is exported by only one module, then the `from` clause may be omitted; otherwise, the disambiguating *mod\_name* must be specified. Similarly, if *local\_predname* is the same as the predicate name used in *global\_queryform*, the `as` clause may be omitted. The `as` clause provides a means to specify a local name for an imported global predicate that is different from the name under which it was defined. This is most useful when two global predicates must be imported which have been defined with the same name. More than one *global\_queryform* may be imported with a single import statement as indicated by the BNF in appendix A.

The basic form of an export statement is as follows.

```
export [ ename = external_name ] [ compile_options ] local_queryform.
```

The `ename` option is used to assign *external\_name* to the query form. The user is responsible for ensuring the uniqueness of the external name. This option is useful only in conjunction with external predicates (see section 3). The *compile\_options* are enumerated and described in [CG89]. They are used to set parameters within the  $\mathcal{LDL}$  compiler that determine how the exported predicate will be compiled. As with the import statement, more than one *local\_queryform* can be defined in a single export statement.

Any predicates not defined inside a module are implicitly included in a module denoted the *global* module. Predicates are said to be in the same component of the global module if their definitions are in the same file. All restrictions applicable to module components are also applicable to groups of such predicates that are defined in different files. Hence, predicate definitions

cannot be split among files, yet mutually recursive predicates may be defined in different files.

We now present an example using modules. In graph theory, a connected set is defined as a set of nodes such that there is a path from every node to every other node in the set. A component is a connected set that is contained in no other connected set. We define `connected` in a module, and use it to define `component`. We represent the graph as a set of edges stored in the relation `edge`. The `LDL` program is given in program 2.1. The global

---

```

import connected( S ) from mod_connected.

component( S ) ← connected( S ), ~subcomponent( S ).

subcomponent( S ) ← connected( S0 ), subset( S, S0 ), S ≠ S0.

module mod_connected.
export connected( S ).
connected( { N } ) ← node( N ).
connected( S ) ← connected( S0 ), member( N, S0 ),
                path( N, N2 ), union( { N2 }, S0, S ).

node( N ) ← edge( N, _ ).
node( N ) ← edge( _, N ).

path( N1, N2 ) ← edge( N1, N2 ).
path( N1, N2 ) ← edge( N2, N1 ).

end mod_connected.

```

---

**Program 2.1:** Using modules

module contains the predicate definitions for `component` and `subcomponent`. Note the import statement for `connected`. It specifies the query form to be imported and the module where the predicate is defined. Also notice how `connected` is used in two places in the definition of `component`. By placing `connected` in a module, we are guaranteed that the answers are computed only once. The module `mod_connected` exports the query form for `connected`. The other predicates that are defined in the module, i.e.,

`node` and `path`, are local to the module and not visible outside it.

A module may import from another module, thus `node` or `path` could be defined in a separate module and then imported by `mod_connected`. To illustrate this, we present the module `mod_path` which contains definitions for `path` and `node` and show how it can be used from `mod_connected`. The modules are shown in program 2.2. Since `path` and `node` are exported

---

```
module mod_connected.  
import node( N ), path( $N1, N2 ).  
export connected( S ).  
  
connected( { N } ) ← node( N ).  
connected( S ) ← connected( S0 ), member( N, S0 ),  
                path( N, N2 ), union( N2, S0, S ).  
  
end mod_connected.  
  
module mod_path.  
export path( $N1, N2 ), node( N ).  
  
node( N ) ← edge( N, _ ).  
node( N ) ← edge( _, N ).  
  
path( N1, N2 ) ← edge( N1, N2 ).  
path( N1, N2 ) ← edge( N2, N1 ).  
  
end mod_path.
```

---

**Program 2.2:** Nested importing of global predicates

by only a single module, the module name need not be specified in the import statement. Note the specification of more than one query form in single import and export statements. The global module remains as in program 2.1.

Even though two global predicates defined in different modules may not be mutually recursive, predicates (local or global) defined in different module components of the same module may, in fact, be mutually recursive. Program 2.3 illustrates such a case. Placing mutually recursive predicates

---

```

module mod_parts_of.

parts_of( X, { X } ) ← basic_part( X ).
parts_of( X, Y ) ← subparts( X, S ), parts_of_set( S, Y ).

end mod_parts_of.

module mod_parts_of.

parts_of_set( {}, {} ).
parts_of_set( S, Y ) ← member( X, S ),
                    difference( S, {X}, S1 ),
                    parts_of( X, Y1 ),
                    parts_of_set( S1, Y2 ),
                    union( Y1, Y2, Y ).

end mod_parts_of.

```

---

**Program 2.3:** Mutual recursion between module components

in different components of a module may be useful when the rule set defining the recursive predicates is large enough to warrant splitting the module among different files.

### 3 External Predicates

An external predicate is fully integrated with the  $\mathcal{LDC}$  system. As such, it can access all of the  $\mathcal{LDC}$  data types, such as sets, functors, and relations. Also, it is capable of returning multiple solutions for a given input value — a notion almost alien in the procedural world.

The basic form of an import statement for an external predicate is as follows.

```
import [ recomputed ] global_queryform
      from [ language ] epred object_filename
          [ object object_files ]
          [ ename function_names ]
          [ library external_libraries ]
      [ as local_predname ].
```

*Global\_queryform* and *local\_predname* are the same as described in the previous section on modules. The **recomputed** option is also as described in the context of modules. The **from** clause identifies the imported predicate as an external predicate and is used to specify the object code file for the external routine. Any necessary object files,  $\mathcal{LDC}$  functions and libraries used by the external predicate are also specified in the **from** clause. The standard language libraries are linked in automatically if the **language** specification is given. Currently, the language specifications recognized are **C** and **FORTRAN**.

In the remainder of this section, we describe a set of library routines that can be used by an external predicate. For brevity, we describe only the **C** language interface, although the techniques are applicable to any procedural language.

An external predicate accepts two arguments: a temporary relation and a tuple<sup>1</sup>. All input values are initially placed in the tuple, and the output values generated are placed in the tuple and then stored in the temporary relation. The elements of a tuple can be accessed with the functions

---

<sup>1</sup>A relation can be viewed as a table, where each row is a tuple and each column corresponds to one component [U188].

`ldl_get_tuple_arg` and `ldl_put_tuple_arg`. The tuple is added to the relation using the function `ldl_add_tuple`.

Program 3.1 illustrates how these functions are used in an external predicate. The predicate returns both square roots of a real number. If the number is negative, no results are returned; i.e., the predicate *fails*.

---

```
void          square_roots (Rel, Tuple)
ldl_relation  Rel;
ldl_tuple    Tuple;
{
    float x;
    float y;

    x = ldl_get_float (ldl_get_tuple_arg(Tuple,1));
    if (x >= 0) {
        y = sqrt ((double) x);
        ldl_put_tuple_arg (Tuple, 2, ldl_put_float(y));
        ldl_add_tuple (Rel, Tuple);
        ldl_put_tuple_arg (Tuple, 2, ldl_put_float(-y));
        ldl_add_tuple (Rel, Tuple);
    }
}
```

---

**Program 3.1:** External predicate in C

A trivial  $\mathcal{LDL}$  program which uses this external routine is given in program 3.2. Note in particular that the math library must be specified since

---

```
import square_roots( $X, Y )
    from C epred 'square_roots.o' library m.

trivial( Y ) ← square_roots( 4.0, Y ).
```

---

**Program 3.2:** Calling an external predicate

it is not one of the standard C libraries.

We now introduce the routines used to convert objects from their  $\mathcal{LDL}$

representation to a suitable representation in C and vice-versa. A variable of type `ldl_object` can store any  $\mathcal{LDL}$  object, as well as the special object `LDL_NULL`. All functions of type `ldl_object` return `LDL_NULL_OBJECT` to indicate an error condition. The routines `ldl_get_float` and `ldl_put_float` were presented in program 3.1. They are used to convert a real number from and to its  $\mathcal{LDL}$  representation, respectively. The routines `ldl_get_int`, `ldl_put_int`, `ldl_get_atom`, and `ldl_put_atom` are analogous. We present the functions used for complex terms later.

Program 3.3 shows how an external predicate can use  $\mathcal{LDL}$  atoms. The predicate has the form `all_letters($X,Y)`. It returns all letters contained in a given atom. We note that this predicate cannot be expressed in pure  $\mathcal{LDL}$ , as defined in [NT89]. The functions `ldl_get_atom` and `ldl_put_atom`

---

```

void          all_letters (Rel, Tuple)
ldl_relation  Rel;
ldl_tuple    Tuple;
{
    char  letter[2];
    char *string;
    char *s;

    string = ldl_get_atom (ldl_get_tuple_arg(Tuple,1));
    letter[1] = '\0';
    for (s=string; *s!='\0'; s++) {
        letter[0] = *s;
        ldl_put_tuple_arg (Tuple, 2, ldl_put_atom(letter));
        ldl_add_tuple (Rel, Tuple);
    }
}

```

---

**Program 3.3:** Manipulating atoms

operate on regular, null-terminated strings, not characters. Hence, a trailing null is needed in `letter` of program 3.3 above.

The function `ldl_type` is used to return the type of an  $\mathcal{LDL}$  object. If the object is a valid  $\mathcal{LDL}$  object, `ldl_type` returns one of the following constants: `LDL_INT`, `LDL_FLOAT`, `LDL_ATOM`, `LDL_FUNCTOR`, `LDL_LIST`, or `LDL_SET`. Otherwise, it returns -1.

Since  $\mathcal{LDL}$  objects are not simple C objects, they cannot be compared using the C relational operators. The following functions must be used instead: `ldl_equal`, `ldl_less`, and `ldl_greater`.

$\mathcal{LDL}$  functors can be manipulated with the functions described below. The name of the functor is returned with `ldl_get_functor_name` and stored with `ldl_put_functor_name`. The name is returned (and taken) in the form of an  $\mathcal{LDL}$  atom, not a C string. Of course, the standard atom-to-string-conversion routines can be used if necessary. The arity of a functor can be accessed by the function `ldl_get_functor_arity`. The functions `ldl_get_functor_arg`, and `ldl_put_functor_arg` are used to access the arguments of a functor. Finally, the functions `ldl_alloc_functor` and `ldl_free_functor` create and destroy functors, respectively. In program 3.4, the arguments of a binary functor are swapped, e.g., if  $f(A,B)$  is given, then  $f(B,A)$  is returned. Note that this procedure cannot be written in pure  $\mathcal{LDL}$ , since functor names must be specified explicitly in  $\mathcal{LDL}$  terms.

Lists can be accessed by using the functions `ldl_head` and `ldl_tail`. These functions accept a list and return its head and tail, respectively. The empty list is denoted by the constant `LDL_EMPTY_LIST`. Lists can be constructed using the function `ldl_cons` which accepts an  $\mathcal{LDL}$  object, X, and a list, L, and returns a list such that X is the head and L is the tail. Program 3.5 gives the C program to reverse an  $\mathcal{LDL}$  list.

We now show how sets can be used from C. The constant `LDL_EMPTY_SET` denotes the empty set. Sets can be constructed by using the function `ldl_scons` which takes in an element, X, and a set, S, and returns the union of  $\{X\}$  and S. The functions `ldl_union`, `ldl_intersection`, and `ldl_difference` take in two sets and return their union, intersection, and set difference, respectively. The function `ldl_cardinality` returns the cardinality of a set. The two functions `ldl_member` and `ldl_subset` can be used to test whether an element belongs to a set, or whether a set is a subset of another, respectively. The function `ldl_get_element` is used to get a given element from a set. It can be used in conjunction with `ldl_cardinality` to get all elements from a set. An example of a C procedure to partition a given  $\mathcal{LDL}$  set into two subsets is given in program 3.6.

There are two distinct types of relations in  $\mathcal{LDL}$ : temporary relations, and base relations. Temporary relations are created by a program, and their temporal scope is the same as the program's. The relations used so far in this paper, namely to get/store results from/to external procedures, are examples of temporary relations. Base relations, on the other hand, are described in the schema and stored in the permanent database; i.e.,

---

```
void          swap_args (Rel, Tuple)
ldl_relation  Rel;
ldl_tuple    Tuple;
{
    ldl_object in_functor;
    ldl_object out_functor;
    ldl_object functor_name;
    ldl_object A;
    ldl_object B;

    in_functor = ldl_get_tuple_arg (Tuple, 1);
    out_functor = ldl_alloc_functor (2);
    functor_name = ldl_get_functor_name (in_functor);
    ldl_put_functor_name (out_functor, functor_name);
    A = ldl_get_functor_arg (in_functor, 1);
    B = ldl_get_functor_arg (in_functor, 2);
    ldl_put_functor_arg (out_functor, 1, B);
    ldl_put_functor_arg (out_functor, 2, A);
    ldl_put_tuple_arg (Tuple, 2, out_functor);
    ldl_add_tuple (Rel, Tuple);
}
```

---

**Program 3.4:** Manipulating functors

---

```

void          reverse (Rel, Tuple)
ldl_relation  Rel;
ldl_tuple    Tuple;
{
    ldl_object  in_list;
    ldl_object  out_list;

    in_list = ldl_get_tuple_arg (Tuple, 1);
    out_list = LDL_EMPTY_LIST;
    while (!ldl_equal(in_list, LDL_EMPTY_LIST)) {
        out_list = ldl_cons (ldl_head(in_list), out_list);
        in_list = ldl_tail (in_list);
    }
    ldl_put_tuple_arg (Tuple, 2, out_list);
    ldl_add_tuple (Rel, Tuple);
}

```

---

**Program 3.5:** Manipulating lists

they outlast program executions. We now describe the primitives used to manipulate relations.

A relation can be accessed by using the function `ldl_get_relation`. This returns a relation with the given name and arity. If no such relation exists, it creates a temporary relation with the given name and arity. A temporary relation is deleted using the function `ldl_del_relation`. There is no way to create or destroy base relations, since they are assumed to exist for the entire duration of the database.

There are two ways to access the tuples in a relation. All tuples in a relation can be read at once, i.e., scanned, or a subset of the tuples can be read through the use of an index. The function `ldl_get_index` returns an index into a relation on the specified columns. It creates an index if one does not already exist. Since `ldl_get_index` can accept a multiple number of arguments (for specifying columns), -1 must be used to terminate the argument sequence. To read the tuples in a relation, a pointer into the relation, i.e., a cursor, must be defined. The function `ldl_get_cursor` returns a cursor that can be used to access a given index in a given relation. The constant `LDL_NULL_INDEX` is used in place of an index when the desired

---

```

void          partition_once (Rel, Tuple)
ldl_relation  Rel;
ldl_tuple    Tuple;
{
    ldl_object elem, set, set1, set2;
    int        i, size;

    set = ldl_get_tuple_arg (Tuple, 1);
    size = ldl_cardinality (set);
    if (size <= 1)
        return;
    set1 = set2 = LDL_EMPTY_SET;
    for (i=1; i<=size/2; i++) {
        elem = ldl_get_element (set, i);
        set1 = ldl_scons (elem, set1);
    }
    for ( ; i<=size; i++) {
        elem = ldl_get_element (set, i);
        set2 = ldl_scons (elem, set2);
    }
    ldl_put_tuple_arg (Tuple, 2, set1);
    ldl_put_tuple_arg (Tuple, 3, set2);
    ldl_add_tuple (Rel, Tuple);
}

```

---

**Program 3.6:** Manipulating sets

access method is scan. Finally, the function `ldl_get_tuple` returns the next tuple from a cursor.

Program 3.7 illustrates the use of these functions. It computes all child/grandparent pairs<sup>2</sup>.

---

```
void          child_grandparent (Rel, Tuple)
ldl_relation  Rel;
ldl_tuple    Tuple;
{
    ldl_object  child, parent, grandparent;
    ldl_tuple   t;
    ldl_relationp;
    ldl_index   idx;
    ldl_cursor  c1, c2;

    p = ldl_get_relation ("parent", 2);
    idx = ldl_get_index (p, 1, -1);
    c1 = ldl_get_cursor (p, LDL_NULL_INDEX);
    while ((t=ldl_get_tuple(c1)) != NULL) {
        child = ldl_get_tuple_arg (t, 1);
        parent = ldl_get_tuple_arg (t, 2);
        c2 = ldl_get_cursor (p, idx, parent);
        while ((t=ldl_get_tuple(c2)) != NULL) {
            grandparent = ldl_get_tuple_arg (t, 2);
            ldl_put_tuple_arg (Tuple, 1, child);
            ldl_put_tuple_arg (Tuple, 2, grandparent);
            ldl_add_tuple (Rel, Tuple);
        }
    }
}
```

---

### Program 3.7: Accessing base relations

We have shown the use of the function `ldl_add_tuple` to add a tuple to a temporary relation. It can also be used to add a tuple to a base relation.

---

<sup>2</sup>We note this can be much more easily expressed in *LDL*. In fact, the equivalent *LDL* program is compiled into code that is very similar and just as efficient as that given.

The function `ldl_del_tuple` may be used to delete a tuple from a base relation. There is no way to delete a tuple from a temporary relation.

*LDL* global predicates can also be called from within C. The calling program simply needs to create a temporary relation for the predicate to store answers, and construct the tuple containing the input arguments. The only functions that need to be introduced are `ldl_alloc_tuple` and `ldl_free_tuple` used for the creation and deletion of tuples, respectively.

Program 3.8 shows how the `square_roots` predicate defined in program 3.1 can be called from C. It uses this predicate to find the solutions to a quadratic equation<sup>3</sup>.

Program 3.9 shows how the external predicate `solve_quadratic` can be used from *LDL*. Since the predicates `solve_quadratic` and `square_roots` are defined in separate files, *LDL* must link both of these files together. The import statement for `solve_quadratic` must identify `square_roots.o` as an external object file in order for the system to be able to link correctly.

Any global predicate written in *LDL* can also be invoked as illustrated above. It is only necessary to know the internal name of the compiled query form. The *LDL* system can be forced to assign a given name to a compiled query form by specifying the name in the `ename` option of the `export` statement. Program 3.10 gives the external predicate `find_root` which uses the `root` predicate defined in program 3.9. `Root` must be exported and given a function name as in the following export statement.

```
export ename = ldl_find_root root(X).
```

Note that the temporary relation names used in external predicates must be unique when used in the same program. In this example, `root` invokes `solve_quadratic` which creates a temporary relation named `temp`. Hence, we must use a different relation name in `find_root`.

## 4 External Functions

We have already seen how an *LDL* predicate can be written in a procedural language and used from an *LDL* program. This involved writing the procedure in a specific manner to interface with *LDL*. We now turn to the class of functions already written in procedural languages.

---

<sup>3</sup>This example is purely for didactic purposes. We do not recommend this as a solution to the problem.

---

```

void          solve_quadratic (Rel, Tuple)
ldl_relation  Rel;
ldl_tuple     Tuple;
{
    float      a, b, c, discr, s_discr, x;
    ldl_cursor cr;
    ldl_relationr;
    ldl_tuple   t;

    a = ldl_get_float (ldl_get_tuple_arg(Tuple,1));
    b = ldl_get_float (ldl_get_tuple_arg(Tuple,2));
    c = ldl_get_float (ldl_get_tuple_arg(Tuple,3));
    discr = b*b - 4*a*c;
    r = ldl_get_relation ("temp", 2);
    t = ldl_alloc_tuple (2);
    ldl_put_tuple_arg (t, 1, ldl_put_float(discr));
    square_roots (r, t);
    cr = ldl_get_cursor (r, LDL_NULL_INDEX);
    while ((t=ldl_get_tuple(cr)) != NULL) {
        s_discr = ldl_get_float (ldl_get_tuple_arg(t,2));
        x = (-b+s_discr)/(2*a);
        ldl_put_tuple_arg (Tuple, 4, ldl_put_float(x));
        ldl_add_tuple (Rel, Tuple);
    }
    ldl_free_tuple (t);
    ldl_del_relation (r);
}

```

---

**Program 3.8:** Calling an external predicate from *C*

---

```

import solve_quadratic( $A, $B, $C, X )
    from C epred 'solve_quadratic.o'
    object 'square_roots.o' library m.

root( X ) ← solve_quadratic( 1.0, 2.0, 1.0, X ).

```

---

**Program 3.9:** Using nested external procedures

---

```

void          find_root (Rel, Tuple)
ldl_relation  Rel;
ldl_tuple    Tuple;
{
    ldl_object a; ldl_cursor c;
    ldl_relation r;
    ldl_tuple t;

    r = ldl_get_relation ("temp1", 1);
    t = ldl_alloc_tuple (1);
    ldl_find_root (r, t);
    c = ldl_get_cursor (r, LDL_NULL_INDEX);
    while ((t=ldl_get_tuple(c)) != NULL) {
        a = ldl_get_tuple_arg(t,1);
        ldl_put_tuple_arg (Tuple, 1, a);
        ldl_add_tuple (Rel, Tuple);
    }
    ldl_free_tuple (t);
    ldl_del_relation (r);
}

```

---

**Program 3.10:** Calling an *LDL* predicate from *C*

There are some important differences between predicates and functions. First of all, functions are expected to produce only one answer for a given input. Functions can return results either as an argument or as the return value of the function itself. An argument to a function can, in fact, be used for both input and output. Moreover, function arguments must be typed, and the only data types that are allowed are the simple  $\mathcal{LDL}$  data types, namely integers, reals, and atoms.

We now introduce notation to describe the input/output properties of a function. A *function form* is analogous to a query form in that it describes how a function is to be invoked. It specifies the typed input and output variables of each argument and the typed return variable, if any, for the function. It can also specify if an argument is to be passed by reference (the default is by value). The *functionform* syntax is as follows.

```
function_name ( [ ref ] in_variable:type [ => out_variable ], ... )
                [ => return_variable:type ]
```

As expected, an import statement is used to declare an external function from within  $\mathcal{LDL}$ . The basic form of an import statement for an external function is as follows.

```
import global_functionform
      from [ language ] [ external object_filename ]
          [ library external_libraries ]
      as local_queryform.
```

As with external predicates, the *language* specification may be used to select the standard language libraries. It is also used to select a default input parameter passing mode, namely, by value for C and by reference for FORTRAN. The output parameters must always be passed by reference. The *object\_filename* may only be omitted when importing directly from a library, either explicitly (e.g. **from library m**) or through the language specification (e.g. **from C**). The *local\_queryform* specifies how the *global\_functionform* is to be accessed from an  $\mathcal{LDL}$  program. Note that the *local\_queryform* may not be omitted since a function form, not a query form, is being imported.

As before, we illustrate with some examples. First of all, we use C to concatenate two atoms. We note that this is a computation that currently cannot be expressed in pure  $\mathcal{LDL}$ . The C routine accepts two input parameters, namely the two atoms to concatenate, and returns an output parameter with the expected result. The routine is given in program 4.1. The input

---

```
concat (a, b, c)
char *a;
char *b;
char **c;
{
    int          length;
    static char  *s = NULL;
    char         *t;

    length = strlen (a) + strlen (b) + 1;
    if (s == NULL)
        s = malloc (length);
    else
        s = realloc (s, length);
    for (t = s ; *a != '\0'; *t++ = *a++)
        ;
    for ( ; *b != '\0'; *t++ = *b++)
        ;
    *t = '\0';
    *c = s;
}
```

---

**Program 4.1:** External function

parameters are of type `char *`. Atoms are passed in as null-terminated strings, as would be expected. The output parameter is defined to be `char **`; that is, an atom passed by reference. Finally, note that the routine allocates a block of memory of the necessary length. `Realloc` is used to reclaim the space from the previous call. (Notice the static declaration.) The return parameter is a null-terminated string. `LDL` copies the string into its own internal space, so `concat` could use a locally allocated array; however, this could create problems if the array does not have sufficient space to store the result. We note here that some string handling functions (such as `strcpy` or `strcat`) expect one of their arguments to be a buffer containing enough space for the result. They then proceed to write over this buffer. Such functions cannot be called directly from within `LDL`. Of course, any external procedure can call them, so they can be easily redefined, if they are needed.

An `LDL` program which uses the `concat` procedure is given in program 4.2.

---

```
import concat( $A: string, $B: string, C: string )
    from C external 'concat.o'
    as concat( $A, $B, C ).

c_concat( X, Y, Z ) ←
    concat( X, Y, Z ).
```

---

**Program 4.2:** Using an external function

The next example shows a random number generator. It changes the value of its input argument, namely the seed. The C routine is given in program 4.3.

---

```
int  random (seed)
int  *seed;
{
    return ((*seed)++);
}
```

---

**Program 4.3:** In/out parameters in external functions

Program 4.4 shows how this function can be used from an  $\mathcal{LDL}$  program. Note that while the C function `random` accepts a single argument, the  $\mathcal{LDL}$

---

```
import random( $Seed1:integer => Seed2 ) => Rand:integer
    from C external 'random.o'
    as random( $Seed1, Seed2, Rand ).

trivial( S, R ) ← random( 3, S, R ).
```

---

**Program 4.4:** Using external functions with in/out parameters

predicate requires two arguments since  $\mathcal{LDL}$  has a single assignment model, i.e., the value of a bound variable cannot be modified. Also, note that since the argument is being used for output, it must be passed by reference.

The third example uses a FORTRAN function to compute the roots of a quadratic equation. The FORTRAN code is presented in program 4.5; the  $\mathcal{LDL}$  code is given in program 4.6.

When linking  $\mathcal{LDL}$  and FORTRAN, there is a common pitfall which we have avoided in our presentation so far.  $\mathcal{LDL}$  passes parameters very much like C. In particular, objects of type real are passed as double precision floating point. However, a pointer to a real object points to the single precision real, not a double precision variable. FORTRAN treats all real objects in the same way. Since parameters are passed only by reference, reals are passed as pointers to single precision objects, hence our declarations are correct. However, the return value of a function is also passed as a single precision object, whereas  $\mathcal{LDL}$  (and for that matter C) expects it to be double precision. Thus, the return value must be declared as double precision or `real*8`.

## 5 Conclusions

We have introduced the concept of modules into  $\mathcal{LDL}$ . The use of modules allows a programmer to partition a large  $\mathcal{LDL}$  program into smaller, more manageable pieces, which can then be debugged separately. We have shown how modules can import and export global predicates. The traditional compilation methods used in  $\mathcal{LDL}$ , such as global optimization, safety analysis, rule rewriting, etc. can be applied to each global predicate.

---

```
integer function roots (a, b, c, x1, x2)
real a, b, c, x1, x2
real discr, t

discr = b*b - 4*a*c
if (discr .lt. 0) then
    roots = 0
    x1 = 0
    x2 = 0
elseif (discr .eq. 0) then
    roots = 1
    x1 = -b/(2*a)
    x2 = 0
else
    roots = 2
    t = sqrt(discr)
    x1 = (-b+t)/(2*a)
    x2 = (-b-t)/(2*a)
endif
return
end
```

---

**Program 4.5:** FORTRAN external function

---

```

import roots( $A: real, $B: real, $C: real, X1: real, X2: real )
      => N: integer
      from FORTRAN external 'roots.o'
      as find_roots( $A, $B, $C, N, X1, X2 ).

roots( A, B, C, X ) ← find_roots( A, B, C, N, X1, X2 ),
      select_roots( N, X1, X2, X ).

select_roots( 1, X, -, X ).
select_roots( 2, X, -, X ).
select_roots( 2, -, X, X ).

```

---

**Program 4.6:** Using a FORTRAN external function

We have also shown how a global predicate can be written in a procedural language; such predicates are referred to as external predicates. In particular, an  $\mathcal{LDL}$  global predicate, critical from a performance perspective, can be rewritten as an external predicate. The external predicate is viewed exactly as the original global predicate by the calling  $\mathcal{LDL}$  program.

Finally, we have shown how external functions can be called from an  $\mathcal{LDL}$  program. The use of external functions allows an  $\mathcal{LDL}$  programmer to take advantage of the many routines that have already been written in procedural languages. For example, many of the routines in standard libraries can now be used directly from  $\mathcal{LDL}$ .

There remain two issues that need to be addressed in later research. In conventional languages, it is common for procedures to perform side-effects when they are called. For example, reading or writing to a file changes the file pointer. To use these procedures from  $\mathcal{LDL}$ , we need to specify the exact location where they are called from the program. When the various  $\mathcal{LDL}$  rewriting techniques (e.g., recursion) and run-time optimizations (e.g., intelligent backtracking) are considered, this seems an impossible task. Naqvi and Krishnamurthy propose a sequencing operator in [NK88]. This is used to defined the semantics of  $\mathcal{LDL}$  in the presence of updates, where many of the same problems are present, and to define the semantics of  $\mathcal{LDL}$  iterative operators. We suspect that a similar approach can be used in the presence of externals, but we have yet to define  $\mathcal{LDL}$  constructs to accomplish this.

The techniques presented in this paper allow an external procedure to call an *LDL* predicate, but only from within the *LDL* environment. A more general solution would allow a program to invoke the *LDL* environment first, and then call an arbitrary *LDL* predicate. For example, this would allow a programmer to design a new user interface for *LDL* using the routines provided by a window manager. Moreover, we would like *LDL* to interact with declarative languages, such as Prolog, or object-oriented languages. These languages generally have their own run-time environments which would have to be invoked, much as with the *LDL* environment, before any “subprograms” could be used. This is most easily accomplished by running the *LDL* and any other necessary environments as separate processes. Such a solution may be proposed in the future.

## A BNF for Language Extensions

In this appendix, we present a BNF description for the new *LDL* constructs introduced in this paper. We use a `typewriter` font to indicate terminal symbols, and *italics* for non-terminals. Optional arguments are enclosed in square brackets (`[]`).

The non-terminals *mod\_name*, *pred\_name*, *library\_name*, *func\_name*, and *file\_name* refer to *LDL* atoms; *return\_variable*, *in\_variable*, and *out\_variable* refer to *LDL* variables; *query\_form*, *local\_queryform*, and *global\_queryform* refer to *LDL* query forms; and *program* refers to an *LDL* program. The BNF for these can be found in [NT89].

```

module           → module_component
                  | module_component module

module_component → module mod_name
                   import_export_stmt
                   program
                   end mod_name .

import_export_stmt → import_stmt
                    | export_stmt
                    | import_stmt import_export_stmt
                    | export_stmt import_export_stmt

import_stmt      → import import_preds .

```

*import\_preds* → *import\_pred*  
| *import\_pred* , *import\_preds*

*import\_pred* → *import\_global\_pred*  
| *import\_ext\_pred*  
| *import\_ext\_function*

*import\_global\_pred* → [ *recomputed* ] *global\_queryform*  
| [ *from mod\_name* ]  
| [ *as pred\_name* ]

*import\_ext\_pred* → [ *recomputed* ] *global\_queryform*  
| *from* [ *language* ] *epred file\_name*  
| [ *object object\_specs* ]  
| [ *ename ename\_specs* ]  
| [ *library library\_specs* ]  
| [ *as pred\_name* ]

*language* → C  
| FORTRAN

*object\_specs* → *file\_name*  
| *file\_name object\_specs*

*ename\_specs* → *external\_name*  
| *external\_name ename\_specs*

*library\_specs* → *library\_name*  
| *library\_name library\_specs*

*import\_ext\_function* → *global\_functionform*  
| *function\_from\_stmt*  
| *as local\_queryform*

*function\_from\_stmt* → *from* [ *language* ] *external file\_name*  
| [ *library library\_specs* ]  
| *from language* [ *library library\_specs* ]

```

| from library library_specs

functionform    → func_name ( argument_specs )
                  [ => return_variable:type ]

argument_specs  →
| argument , argument_specs

argument        → [ ref ] $in_variable:type [ => out_variable ]
| out_variable:type

export_stmt     → export [ option_list ] export_preds .

option_list     → [ ename = external_name
|   [ cycles = yes|no ]
|   [ duplicates = yes|no ]
|   [ memory = yes|no ]
|   [ optimizer = yes|no ]
|   [ reorder = yes|no ]

export_preds    → queryform
| queryform , export_preds

```

## B External Interface Library

In this appendix, we describe the `ldl` library that can be used by C programs to access `LDL` data structures. The functions in this library maintain a uniform error handling mechanism, similar in style to the Unix system call interface. To indicate an error, they return a special value, consistent with the type of the function. For example, `-1` is returned by functions of type `int`, `NULL` by functions of pointer types, etc. An error code is made available in the external variable `errno`. The header file `ldl.h` contains the definition for the possible types of error cases. This file must be included by any external C programs.

## B.1 Data Types

### B.1.1 `ldl_object`

All valid  $\mathcal{LDL}$  objects, i.e., atoms, integers, reals, functors, lists, and sets, are of this type. The constants `LDL_EMPTY_LIST` and `LDL_EMPTY_SET` are used to denote the nil list and empty set, respectively. The special constant `LDL_NULL_OBJECT` is of type `ldl_object` and is used to denote an invalid object.

### B.1.2 `ldl_relation`

This type is used for both base and temporary relations.

### B.1.3 `ldl_tuple`

This is the type of a tuple in a relation.

### B.1.4 `ldl_index`

This is the type of an index into a relation. The constant `LDL_NULL_INDEX` is of this type and is used to denote no index, in places where an index would normally be used.

### B.1.5 `ldl_cursor`

This is the type for a pointer into a relation. The pointer is used to hold the next tuple to read from the relation. More than one cursor may point into the same relation.

## B.2 Comparison Routines

### B.2.1 `ldl_type`

```
int          ldl_type (Obj)
ldl_object   Obj;
```

This function returns the type of `Obj`. The value returned is one of `LDL_INT`, `LDL_FLOAT`, `LDL_ATOM`, `LDL_FUNCTOR`, `LDL_LIST`, or `LDL_SET`. If `Obj` is not a valid  $\mathcal{LDL}$  object, `ldl_type` returns -1, and the global variable `errno` is set to `EINVAL`.

### B.2.2 `ldl_equal`

```
int          ldl_equal (Obj1, Obj2)
ldl_object   Obj1, Obj2;
```

This function is used to compare two *LDL* objects. It returns 1 if `Obj1` is equal to `Obj2`, and 0 otherwise.

### B.2.3 `ldl_less`

```
int          ldl_less (Obj1, Obj2)
ldl_object   Obj1, Obj2;
```

This function is used to compare two *LDL* objects. It returns 1 if `Obj1` is less than `Obj2`, and 0 otherwise.

### B.2.4 `ldl_greater`

```
int          ldl_greater (Obj1, Obj2)
ldl_object   Obj1, Obj2;
```

This function is used to compare two *LDL* objects. It returns 1 if `Obj1` is greater than `Obj2`, and 0 otherwise.

## B.3 Simple Data Types

### B.3.1 `ldl_get_int`

```
int          ldl_get_int (Obj)
ldl_object   Obj;
```

This function is used to convert the *LDL* object `Obj` into a C integer. If `Obj` is not of type `LDL_INT`, the function returns -1 and places `EINVAL` into `errno`.

### B.3.2 `ldl_put_int`

```
ldl_object   ldl_put_int (n)
int          n;
```

This function is used to convert the integer `n` into its *LDL* representation.

### B.3.3 `ldl_get_float`

```
float          ldl_get_float (Obj)
ldl_object     Obj;
```

This function is used to convert the *LDL* object `Obj` into a C float. If `Obj` is not of type `LDL_FLOAT`, the function returns -1 and places `EINVAL` into `errno`.

### B.3.4 `ldl_put_float`

```
ldl_object     ldl_put_float (x)
int            x;
```

This function is used to convert the float `x` into its *LDL* representation.

### B.3.5 `ldl_get_atom`

```
char           *ldl_get_atom (Obj)
ldl_object     Obj;
```

This function is used to convert the *LDL* object `Obj` into a C string. If `Obj` is not of type `LDL_ATOM`, the function returns `NULL` and places `EINVAL` into `errno`.

### B.3.6 `ldl_put_atom`

```
ldl_object     ldl_put_atom (s)
char           *s;
```

This function is used to convert the C string `s` into its *LDL* representation. If `s` is a `NULL` pointer, the function returns `LDL_NULL_OBJECT` and places `EINVAL` into `errno`.

## B.4 Functions

### B.4.1 `ldl_alloc_funcutor`

```
ldl_object     ldl_alloc_funcutor (arity)
int            arity;
```

This function creates a new functor with `arity` number of arguments. If there is not enough memory to allocate the new functor, `ldl_alloc_functor` returns `LDL_NULL_OBJECT` and puts the value `ENOMEM` into the global variable `errno`. Similarly, if `arity` is less than zero, `EINVAL` is placed into `errno`.

#### B.4.2 `ldl_free_functor`

```
int          ldl_free_functor (Obj)
ldl_object   Obj;
```

This function frees the memory allocated for the functor `Obj`. If `Obj` is not a valid `LDL` object, the function returns -1 and puts `EINVAL` into `errno`.

#### B.4.3 `ldl_get_functor_name`

```
ldl_object   ldl_get_functor_name (Obj)
ldl_object   Obj
```

This function returns the name of the functor `Obj`. If `Obj` is not a valid functor, the function returns `LDL_NULL_OBJECT` and places `EINVAL` into `errno`.

#### B.4.4 `ldl_put_functor_name`

```
int          ldl_put_functor_name (Obj, Name)
ldl_object   Obj, Name;
```

This function makes `Name` the new functor name of `Obj`. If `Name` is not of type `LDL_ATOM` or `Obj` is not of type `LDL_FUNCTOR`, the function returns -1 and places `EINVAL` into `errno`.

#### B.4.5 `ldl_get_functor_arity`

```
int          ldl_get_functor_arity (Obj)
ldl_object   Obj;
```

This function returns the number of arguments of functor `Obj`. If `Obj` is not of type `LDL_FUNCTOR`, the function returns -1 and places `EINVAL` into `errno`.

#### B.4.6 `ldl_get_functor_arg`

```
ldl_object      ldl_get_functor_arg (Obj, i)
ldl_object      Obj;
int             i;
```

This function returns argument `i` of functor `Obj`. If `Obj` is not of type `LDL_FUNCTOR` or `i` is outside the range of the functor, the function returns `LDL_NULL_OBJECT` and places `EINVAL` into `errno`.

#### B.4.7 `ldl_put_functor_arg`

```
int             ldl_put_functor_arg (Obj, i, Arg)
ldl_object      Obj, Arg;
int             i;
```

This function places `Arg` into argument `i` of functor `Obj`. If `Obj` is not of type `LDL_FUNCTOR` or `i` is outside the range of the functor or `Arg` is not a valid *LDL* object, the function returns `-1` and places `EINVAL` into `errno`.

### B.5 Lists

#### B.5.1 `ldl_cons`

```
ldl_object      ldl_cons (X, List)
ldl_object      X, List;
```

This function returns a list, whose first element is `X` and whose remainder is `List`. The special constant `LDL_EMPTY_LIST` denotes a list with no elements. It can be used with `ldl_cons` to build longer lists. If `X` is not a valid *LDL* object, or `List` is not of type `LDL_LIST`, the function returns `LDL_NULL_OBJECT` and places `EINVAL` into `errno`.

#### B.5.2 `ldl_head`

```
ldl_object      ldl_head (List)
ldl_object      List;
```

This function returns the first element of `List`. If `List` is not of type `LDL_LIST`, the function returns `LDL_NULL_OBJECT` and places `EINVAL` into `errno`. If `List` is the empty list, the function returns `LDL_NULL_OBJECT` and places `ERANGE` into `errno`.

### B.5.3 `ldl_tail`

```
ldl_object      ldl_tail (List)
ldl_object      List;
```

This function returns the remainder of `List`. If `List` is not of type `LDL_LIST`, the function returns `LDL_NULL_OBJECT` and places `EINVAL` into `errno`. If `List` is the empty list, the function returns `LDL_NULL_OBJECT` and places `ERANGE` into `errno`.

## B.6 Sets

### B.6.1 `ldl_scons`

```
ldl_object      ldl_scons (X, Set)
ldl_object      X, Set;
```

This function returns the union of `Set` and `X`. It can be used to construct arbitrary sets. The special constant `LDL_EMPTY_SET` is used to denote the empty set. If `X` is not a valid `LDL` object or `Set` is not of type `LDL_SET`, it returns `LDL_NULL_OBJECT` and places `EINVAL` into `errno`. If there is not enough memory to hold the new set, it returns `LDL_NULL_OBJECT` and places `ENOMEM` into `errno`.

### B.6.2 `ldl_get_element`

```
ldl_object      ldl_get_element (Set, i)
ldl_object      Set;
int             i;
```

This function returns element `i` from `Set`. By placing it within a `for` loop, all elements from `Set` can be selected. If `Set` is not of type `LDL_SET`, the function returns `LDL_NULL_OBJECT` and places `EINVAL` into `errno`. If `Set` does not have an element `i`, i.e., `i` is less than 1 or greater than the cardinality of `Set`, `LDL_NULL_OBJECT` is returned and `ERANGE` is placed into `errno`.

### B.6.3 `ldl_cardinality`

```
int             ldl_cardinality (Set)
ldl_object      Set;
```

This function returns the number of elements of `Set`. If `Set` is not of type `LDL_SET`, the function returns -1 and places `EINVAL` into `errno`.

#### B.6.4 `ldl_union`

```
ldl_object      ldl_union (Set1, Set2)
ldl_object      Obj1, Obj2;
```

This function returns the union of `Set1` and `Set2`. If either argument is not of type `LDL_SET`, the function returns `LDL_NULL_OBJECT` and places `EINVAL` into `errno`. If there is not enough memory to hold the new set, it returns `LDL_NULL_OBJECT` and places `ENOMEM` into `errno`.

#### B.6.5 `ldl_intersection`

```
ldl_object      ldl_intersection (Set1, Set2)
ldl_object      Obj1, Obj2;
```

This function returns the intersection of `Set1` and `Set2`. If either argument is not of type `LDL_SET`, the function returns `LDL_NULL_OBJECT` and places `EINVAL` into `errno`. If there is not enough memory to hold the new set, it returns `LDL_NULL_OBJECT` and places `ENOMEM` into `errno`.

#### B.6.6 `ldl_difference`

```
ldl_object      ldl_difference (Set1, Set2)
ldl_object      Obj1, Obj2;
```

This function returns the set difference of `Set1` and `Set2`. If either argument is not of type `LDL_SET`, the function returns `LDL_NULL_OBJECT` and places `EINVAL` into `errno`. If there is not enough memory to hold the new set, it returns `LDL_NULL_OBJECT` and places `ENOMEM` into `errno`.

#### B.6.7 `ldl_member`

```
int             ldl_member (X, Set)
ldl_object      X, Set;
```

This function returns 1 if `X` is an element of `Set` and 0 otherwise. If `X` is not a valid `LDL` object, or `Set` is not of type `LDL_SET`, it returns -1 and places `EINVAL` into `errno`.

### B.6.8 `ldl_subset`

```
int          ldl_subset (Set1, Set2)
ldl_object   Set1, Set2;
```

This function returns 1 if `Set1` is a subset of `Set2` and 0 otherwise. If either argument is not of type `LDL_SET`, the function returns -1 and places `EINVAL` into `errno`.

## B.7 Relations

### B.7.1 `ldl_get_relation`

```
ldl_relation ldl_get_relation (name, arity)
char         *name;
int         arity;
```

This routine returns the relation with name `name` and `arity` number of columns. If no such relation exists, a temporary relation with the given `name` and `arity` is created. If there is not enough memory to store the new relation, `NULL` is returned and `ENOMEM` is placed into `errno`.

### B.7.2 `ldl_del_relation`

```
int          ldl_del_relation (Rel)
ldl_relation Rel
```

This routine is used to destroy the temporary relation `Rel`. If `Rel` is not a valid relation, the function returns -1 and places `EINVAL` into `errno`. If `Rel` is a base relation, the function returns -1 and places `EBASE` into `errno`.

### B.7.3 `ldl_get_index`

```
ldl_index    ldl_get_index (Rel, col1, ..., -1)
ldl_relation Rel;
int         col1, ...;
```

This function returns an index into `Rel` on the given columns. Up to five columns may be spanned by a single index. The -1 is used to indicate the end of the parameter sequence. If the index specified does not exist, it is automatically created. If `Rel` is not a valid relation, the function returns

LDL\_NULL\_INDEX and places EINVAL into `errno`. Similarly, if one of the columns specified is outside the range of `Rel`, ERANGE is placed into `errno`. If there is not enough memory to hold the index, ENOMEM is placed in `errno`.

#### B.7.4 `ldl_get_cursor`

```
ldl_cursor      ldl_get_cursor (Rel, Index, Obj1, ...)  
ldl_relation    Rel;  
ldl_index       Index;  
ldl_object      Obj1, ...;
```

This function is used to return a pointer (i.e., cursor) into a subset of `Rel`. This pointer can be used to read the tuples in the specified subset. If all the tuples in the relation are to be specified (i.e., scan access method), the constant LDL\_NULL\_INDEX is used in place of `Index`. The objects `Obj1, ...`, correspond to the columns specified on the `Index`. If any of the objects is not of the valid type, the function returns NULL and places EINVAL into `errno`. Similarly, if there is not enough memory to hold the cursor, ENOMEM is placed into `errno`.

#### B.7.5 `ldl_get_tuple`

```
ldl_tuple      ldl_get_tuple (Cursor)  
ldl_cursor     Cursor;
```

This function is used to return the next tuple in the subset of a relation specified by `Cursor`. If there are no more tuples, the function returns NULL. If `Cursor` is not a valid *LDL* cursor, the function returns NULL and places EINVAL into `errno`.

#### B.7.6 `ldl_add_tuple`

```
int            ldl_add_tuple (Rel, Tuple)  
ldl_relation   Rel;  
ldl_tuple      Tuple;
```

This function is used to add `Tuple` into `Rel`. If there is not enough memory to add the new tuple, the function returns -1 and places ENOMEM into `errno`. Similarly, if `Rel` is not a valid relation or `Tuple` is not a valid tuple, it places EINVAL into `errno`.

### B.7.7 `ldl_del_tuple`

```
int          ldl_del_tuple (Rel, Tuple)
ldl_relation Rel;
ldl_tuple    Tuple;
```

This function is used to delete `Tuple` from the base relation `Rel`. `Tuple` must be a tuple accessed from `Rel` using `ldl_get_tuple`. If `Rel` is not a valid relation or `Tuple` is not a valid tuple from `Rel`, the function returns -1 and places `EINVAL` into `errno`. If `Rel` is a temporary relation, `ETEMP` is placed into `errno`.

### B.7.8 `ldl_alloc_tuple`

```
ldl_tuple    ldl_alloc_tuple (arity)
int          arity;
```

This function returns a tuple of the specified `arity`. If there is not enough memory to store the new tuple, the function returns `NULL` and places `ENOMEM` into `errno`. If `arity` is negative, the function places `EINVAL` into `errno`.

### B.7.9 `ldl_free_tuple`

```
int          ldl_free_tuple (Tuple)
ldl_tuple    Tuple;
```

This function destroys `Tuple`. If `Tuple` is not a valid tuple, the function returns -1 and places `EINVAL` into `errno`.

### B.7.10 `ldl_get_tuple_arg`

```
ldl_object    ldl_get_tuple_arg (Tuple, i)
ldl_tuple    Tuple;
int          i;
```

This function returns argument `i` of `Tuple`. If `i` is outside the range of `Tuple`, the function returns `LDL_NULL_OBJECT` and places `ERANGE` into `errno`. Similarly, if `Tuple` is not a valid tuple, `EINVAL` is placed into `errno`.

### B.7.11 `ldl_put_tuple_arg`

```
int          ldl_put_tuple_arg (Tuple, i, Obj)
ldl_tuple    Tuple;
int          i;
ldl_object   Obj;
```

This function places `Obj` into argument `i` of `Tuple`. If `i` is outside the range of `Tuple`, the function returns `LDL_NULL_OBJECT` and places `ERANGE` into `errno`. Similarly, if `Tuple` is not a valid tuple or `Obj` is not a valid *LDL* object, `EINVAL` is placed into `errno`.

## References

- [CG89] Danette Chimenti and Ruben Gamboa. The salad cookbook: A user/programmer's guide. Technical Report ACA-ST-???-89, MCC, 1989.
- [CGK89] Danette Chimenti, Ruben Gamboa, and Ravi Krishnamurthy. Towards an open architecture in *LDL*. In *Proceedings of the Fifteenth Conference on Very Large Data Bases*, 1989.
- [NK88] Shamim Naqvi and Ravi Krishnamurthy. Database updates in logic programming. In *Proceedings of the Seventh Annual Association of Computing Machinery Symposium on Principles of Database Systems*, 1988.
- [NT89] Shamim Naqvi and Shalom Tsur. *A Logic Language for Data and Knowledge Bases*. W.H. Freeman, 1989.
- [Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume I. Computer Science Press, 1988.