Advanced P2 System Manual

Edition 0.1, July 1994

Don Batory
Bart J. Geraci
Jeff Thomas

Edition 0.1
For `P2` Version 0.1
July 1994

For information, questions, and to report inaccuracies, please contact: *dsb@cs.utexas.edu*

Preface

This manual documents advanced usage of the P2 system and is the companion document to '`Introductory P2 System Manual`'. It also outlines how experts can customize the system for sophisticated applications.

In addition to the usual index of concepts, this manual provides indices of all functions and variables.

This manual is available in both a printed format and on on-line format. The on-line format can be browsed by the using the GNU "info" program or the GNU emacs "info" command.

1 Agreement

By using this software, you, the *user*, indicate you have read, understood, and will comply with the following:

1. Nonexclusive permission to use, copy and/or modify this software for internal, noncommercial, research purposes is granted. Any distribution, including commercial sale, of this software, copies, associated documentation and/or modifications is strictly prohibited without the prior written consent of *UTA*. Appropriate copyright notice shall be placed on software copies, and a full copy of this license in associated documentation. No right is granted to use in advertising, publicity or otherwise any trademark of *UTA*. Any software and/or associated documentation identified as "confidential" will be protected from unauthorized use/disclosure with the same degree of care *user* regularly employs to safeguard its own such information.

2. This software is provided "as is", and *UTA* makes no representations or warranties, express or implied, including those of merchantability or fitness for a particular purpose, or that use of the software, modifications, or associated documentation will not infringe on any patents, copyrights, trademarks or other rights. *UTA* shall not be held liable for any liability nor for any direct, indirect or consequential damages with respect to any claim by *user* or any third party on account of or arising from this Agreement.

2 Introduction

This is the manual for doing advanced things to P2 like:

- Interpreting and writing xp programs. See Chapter 3 [xp Manual], page 7 and also see Chapter 4 [xp Operations], page 47.
- Setting the option flags for xp, ddl, and pb. See Chapter 5 [Invoking the P2 Programs], page 57.
- Verifying the semantics of a type expression. See Chapter 6 [Layer Composition Checks], page 61.

It is assumed that the user has first read the *Introductory P2 System Manual*.

2.1 Overview of programs

This section talks about the three programs: xp, ddl, and pb in some detail.

The *xp* program translates an xp file into a C file that is to be compiled and linked with the pb program.

The *ddl* program parses a p2 file and rewrites the type expressions and the annotations into a format understood by pb. This program is separate from pb because of it's mutability: new layers are likely to require new annotation formats, which have to be added to the ddl grammar.

The *pb* program is the workhorse of the system. It recognizes containers, cursors, and special operations and converts them into C code.

3 xp Manual

The xp program is a preprocessor for specifying components (also called layers) in P2. It translates an .xp file into a .c file which can then linked into pb.

The xp language is fairly complex; one of the goals in its evolution is to simplify the structure of the language.

3.1 xp BNF

This is an abbreviated BNF for an uncommented xp file:

```
layer           : layer_head operations*
layer_head      : 'layer' realm identifier '[' realm* ']' layer_options*
realm           : 'ds' | 'mem' | 'top' | 'lnk' | 'toplnk'
```

The '`layer_options`' represent all the options specified in section "P2 Layers" in *Introductory P2 System Manual*. The '`operations`' are all operations defined in both Chapter 4 [xp Operations], page 47 and section "P2 Operations" in *Introductory P2 System Manual*.

Note: future versions of xp (and P2) will only export realms '`ds`', '`mem`', and '`lnk`'. The realms '`toplnk`' and '`lnk`' will be unified, and so will '`top`' and '`ds`'.

3.2 xp Syntax

3.2.1 xp Special Characters

An xp file looks like a C file, but there are several special characters. Here are their interpretations:

'`%{`' The beginning of generated code. See Section 3.3 [Generated versus Executed Code], page 9.

'`%}`' The end of generated code. See Section 3.3 [Generated versus Executed Code], page 9.

'`%a`' '`%a`'.field is the value of field in the annotation structure. See Section 3.7.2 [Annotation Structure], page 12.

'`%c`' '`%c`'.field is the value of field in the cursor state structure. See Section 3.7.3 [Cursor State Structure], page 13.

'`%k`' '`%k`'.field is the value of field in the container state structure. See Section 3.7.4 [Container State Structure], page 14.

'`%v`' '`%v`'.field is the value of field in the variable structure. See Section 3.7.1 [Variable Structure], page 12.

'`!`' '`!`' copies the rest of the line verbatim to the xp generated code output.

'`%ln`' The run-time value of the current layer.

'`%%`' The literal '`%`'.

'`#`' When used literally at the end of an operation name, the operation will be called in the current layer rather than the lower layer. For instance, `adv#(cursor);`.

'`//`' C++ comments. All characters between the '`//`' and the end of the line are ignored, including the '`//`'.

3.2.2 Cursor & Container Expressions

The table below lists keywords for cursors and containers. All these keywords can be followed by a single digit 0-9. If the keyword is not followed by a digit, it is equivalent to the digit 0 (i.e. '`cursor`' ≡ '`cursor0`'). Therefore '`predicate`' is the value of the predicate field of cursor (also called '`cursor0`') and '`layer5`' is the retrieval layer for '`cursor5`'.

'`cont_expr_const`'
Boolean variable to reveal if the container expression is a constant.

'`container`'
Expression for container.

'`container_id`'
P2 internally generated integer identifier of container type.

'`container_type`'
Name of the container structure.

'`cost`' Integer cost estimate of processing query using layer.

'`cursor`' Expression for cursor.

'`cursor_id`'
: P2 internally generated integer identifier for cursor.

'`cursor_type`'
: Name of the cursor structure.

'`layer`'
: Integer identifier of layer that is to be the retrieval layer.

'`obj_type`'
: Name of the transformed structure for container or cursor. The structure is transformed by the addition of fields that are added through the xform operation (see Section 4.5 [xform], page 49).

'`orderby_direction`'
: Requested orderby retrieval direction: -1 descending, 0 unsorted, `+`1 ascending.

'`orderby_field`'
: Name of requested orderby element sort key/attribute.

'`orig_type`'
: Name of the untransformed structure for the container or cursor.

'`predicate`'
: Predicate of cursor.

'`retrieval_direction`'
: Generated retrieval direction: -1 descending, 0 unsorted, `+`1 ascending.

3.3 Generated versus Executed Code

A central idea behind xp is the distinction between code that is to be executed by the xp compiler (*xp-executed code*) versus code that is to be generated by xp (*xp-generated code*) and executed at pb run-time. The distinction is made through the use of `%{` and `%}` braces: code within `%{` and `%}` is xp-generated code. Consider the code below:

```
if (test)
 %{ a = b;    %}
else
 %{ a = b+1;  %}
```

When the xp compiler parses this code, if `test` is true, then a '`printf("a=b;");`'[1] is generated, otherwise '`printf("a=b+1;");`' is generated. When this layer is compiled and integrated into the

[1] Actually, it's another procedure altogether, but you get the idea.

'`pb`' program, the '`printf`' procedures will be executed and those strings will be added to the pb-generated C output.

Nesting of '`%{`' and '`%}`' is not permitted.

3.4 Code Generation

In order to generate the object code for an operation, P2 walks through the tree of layers in the type expression, composing the object code for the operation from bits and pieces of object code added by the various layers that provide the operation.

In the case of normal operations such as `xform` (which adds fields to structures) or `delete` (which removes a record from a container), every layer generates code to provide the operation. In the case of retrieval operations such as `reset_start` (which finds the first qualified tuple) or `adv` (which finds the next qualified tuple), however, only one layer needs to generate the code to process the query. The layer that retrieves the object is called the *retrieval layer*.

Thus, in the case of normal operations, a layer specifies the same object code regardless of what other layers appear above or below it in the type expression—that is, local information is sufficient. In the case of retrieval operations, however, a layer generates different code depending on whether or not it is the retrieval layer. Whether or not an operation is the retrieval layer depends in turn on what other layers appear above or below it in the type expression—that is, global information is necessary.

Every single-container query has a unique retrieval layer. During the processing of the `optimize()` operation, every layer is asked to return an estimate of the cost for that layer to process the query. (Processing a query involves traversing the data structures of that layer to extract all elements that satisfy the query). During the process of `optimize`, each layer is returned an estimate of the cheapest cost of processing the query by layers beneath it. (Actually, what is returned from lower layers is a cost estimate, the identity of the layer that produced this estimate, and an indicator about the sort order in which elements will be returned). If the current layer can perform the retrieval more efficiently, it will return its cost estimate, use its id as the "retrieval" layer id, and will set the sort-order indicator. The layer that returns the cheapest estimate for processing the query will therefore be known upon the completion of an `optimize()` operation.

It should be known that there is a definite protocol for selecting cost estimates by layers. Part of the design of P2 is a tabulation of the different classes of queries that will be encountered, plus a predefined linear ranking of all layers from most efficient to least efficient for each query class.

Thus, the processing of `optimize()` merely has each layer determine the classes of the query (note: a query can belong to multiple classes) and to return its predetermined ranking. Readers should consult '`common/qopt.h`' and Section 3.12 [Query Optimization], page 25 for the specifics actually used.

3.5 Programming Hints

The xp program is a preprocessor that does not have access to the pb symbol table. Consequently, xp has rather strange limitations.

First, anywhere within a '`mumble.xp`' file:

- Field names cannot end in a digit because xp appends the number of the layer to the field during code generation. Thus:

```
cursor.field10   // incorrect as field ends in digits
cursor.field_ten  // OK
```

- There is a limit of translating references. The expression `container->field_a->field_b->field_c` is valid only if the field names all refer to elements that point to the original container.

Second, anywhere within the xp-generated code area (between '`%{`' and '`%}`'):

- xp cannot handle nested assignments: `x = y = z;`
- The `xp` program relies on "`{`", "`}`", and "`;`" to separate statements. The program cannot properly parse expressions of the form `if (test) statement;`. It needs to be written as `if (test) { statement; }`.
- Assignment in conditions `if ((t=get_data()) != 0)` are not properly translated.
- If xp crashes on a dollarword expression, try adding a "`;`" after the dollarword. See Section 4.11 [Dollarwords], page 53.

Additionally, there are other things to be aware of:

- For all operations that generate an expression, do not use "`;`" within the xp-generated code. For instance, the procedure `end_adv()` in '`dlist.xp`' is defined as '`%{ (cursor.obj == NULL) %}`'. Note the absence of the semicolon.

3.6 Field References

Fields can be added to the cursor. The format is `cursor.field` or `cursor.field[index]`. Fields and indices can either be the actual name (or number) or a variable such as '`%v.id`', '`%c.id`', etc.

3.7 xp Structures

Annotation, cursor state and container state structures are found in the '`name.h`' file, where '`name.xp`' is the name of the layer.

3.7.1 Variable Structure

In each operations, we can define a variable structure which stores a value that is evaluated in xp generated code. The form is `%v.`*field*, where *field* is the name of the variable. For instance:

```
query( cursor )
{
    char *p = bind_predicate( cursor, obj_type, predicate );
    %{
        %v.p
    %}
    free( p );
}
```

The `bind_predicate` returns a string which we want to include in the final C code. The '`%{ %v.p %}`' prints the value of string '`p`' in the generated code; free(p) frees the allocated string.

3.7.2 Annotation Structure

This is the structure for annotations:

```
struct <layer>_annote {
    struct <layer>_annote *next;
    /* user fields */
    ...
} ;
```

See Section 3.10.1 [Simple Annotations], page 22 for an example using the annotation structure.

In the user fields are data to store the annotations and data that needs to be kept around for layer processing (which are neither cursor nor container specific). In order to use this structure, the layer must have the `annotation` option in its heading.

3.7.3 Cursor State Structure

A cursor state is a structure that is replicated per cursor.

Fields that are added to the cursor via the '`xform`' operation are permanent; they become part of the cursor in the C file generated by pb. Fields added to cursor state structures are temporary. They do not exist in the pb-generated C file.

```
struct <layer>_cstate {        /* cursor state */
    int id;                     /* identifier of cursor type */
    int layerno;                /* layer number */
    struct <layer>_cstate *next;/* next record of this type */
    /* user fields*/
    ...
};
```

In order to use this structure, the layer must have the `curs_state` option in its heading. Usually, the cursor state user fields are initialized in `xform`, or `optimize` and used in the other operations. For instance, in '`hash.h`', the cursor state was declared as:

```
struct hash_cstate {
  int                   id;
  int                   layerno;
  struct hash_cstate    *next;

  char                  num_bits[20];     /* ceiling of log base 2 of size */
};
```

The new field is `num_bits`, which is a string. In the file '`hash.xp`', the field is initialized in `optimize` and used in `insert` (among other procedures) to use in the hashing function:

```
i = %v.hash_func( (record).%a.ofield, %c.num_bits )  %% %a.size ;
```

This is an interesting example. The name of the function is stored in the variable `hash_func`. The name of the field to hash on was passed in as an annotation and stored in the annotation structure field `ofield` (for ordering field). The number of bits was stored in a cursor state structure. The '`%%`' stands for the modulo C operation, and the `size` is the size of the hash table, also passed in as an annotation.

3.7.4 Container State Structure

A container state is the structure which is replicated per cursor. Like cursor state structures, fields added to container state structures are not present in the pb-generated C file.

```
struct <layer>_kstate {
    int id;                      /* identifier of cont. type */
    int layerno;                 /* layer number */
    struct <layer>_kstate *next; /* next record of this type */
    /* user fields */
    ...
};
```

In order to use this structure, the layer must have the `cont_state` option in its heading. Files having container state structures include '`part.xp`' and '`mlist.xp`'.

3.8 Type Extension Details

The procedure `xform` augments the elements, cursors, and containers with additional fields.

The syntax for adding fields:

```
add : 'add' struct ':' var-declaration ';'
    | 'add' struct ':' 'mcurs' variable ';'
    | 'add' struct ':' 'mcurs' old-var new-var ';'
    ;

struct : 'element'digit
       | 'container'digit
       | 'cursor'digit
       | identifier
       ;
```

The latter two cases of "add" are used for member cursor manipulations. The second case copies the name of the variable. The third case copies the variable and gives it a new name.

A member cursor is a special cursor that is used to traverse the list of fields of a previously defined record type. A member cursor can thus reference a particular field of a given record type. Given this ability, one may want to copy the referenced field into another record, extract the type of the referenced field, etc. See the examples below and Section 3.14.1 [Member Cursor Details], page 33 for more details.

Some examples of augmentation are:

```
// add an integer field "timestamp" to the cursor.
   add cursor    : int timestamp;

// add an array "elements" of elements to the container.
// The size of the array is determined by an annotation variable
   add container : struct element elements[%a.size];

// add to the structure "s" a copy of the member cursor "m"
   add s         : mcurs m;

// add to the structure "x" a copy of the member cursor "m"
// which has been renamed "val"
   add x         : mcurs  m       val;
```

The augmentation method involves adding the layer number to the end of the field. This is why fields cannot end in digits.

3.9 Example - dlist.xp

In this section, the file '`dlist.xp`' will be analyzed a piece at a time. The complete file is in Appendix A [dlist.xp program], page 69.

3.9.1 dlist - Parameters

This line describes dlist as a layer with the following characteristics:

```
layer ds dlist [ ds ] stable no annotation retrieval_sometimes
```

- The interface exported by dlist is the `ds` realm, which is the realm of data structures.
- The parameter to dlist is one argument: `ds`. Therefore this parameter can be instantiated with a layer whose interface exports the ds realm.

The remaining items on the line are layer options which are defined in section "Layer Format" in *Introductory P2 System Manual*.

- dlist is `stable`. That means after a delete operation is performed, the cursor points to the deleted object. If instead the cursor pointed to the next undeleted object, then the layer would be `unstable`.
- dlist has no annotations.
- dlist is a `retrieval_sometimes` layer. The retrieval operations will only be executed if this layer is chosen as the retrieval layer. If the retrieval operations are executed regardless of which layer is the retrieval layer, then the layer would be marked as `retrieval_always`. If the layer has no retrieval operations, then the layer would be marked as `retrieval_never`. If the layer has no retrieval-based parameter, then `retrieval_never` is default.

3.9.2 dlist - File Inclusion

The line beginning with '`!`' means to copy the line '`#include "qopt.h"`' directly to the generated '`dlist.c`' file. This particular line retrieves the optimization constants used for the `optimize` procedure. See Section 3.12 [Query Optimization], page 25.

```
!#include "qopt.h"
```

3.9.3 dlist - xform

This procedure augments the element and container structures. To the base element type, two pointers to elements are added: `next` and `prev`. To the container, two pointers are added: `first` and `last`. The call to `xform` in the last line actually means to call the `xform` procedure in the next lower layer. Some P2 operations call the next layer first, some of them call it last.

```
xform( element, container, cursor )
{
    add element : struct element *next;
    add element : struct element *prev;
    add container : struct element *first;
    add container : struct element *last;
    xform( element, container, cursor );
}
```

Here's a concrete example. Suppose that the element type `Person` is the following structure untranslated by layers:

```
typedef struct e {
    char name[30];
    int  age;
} Person;
```

The `Person` type is the base element type. After the '`dlist`' transformation, the new type would be:

```
typedef struct e_transformed {
    char name[30];
    int  age;
    struct e_transformed *nextln;
    struct e_transformed *prevln;
} Person_transformed;
```

Where *ln* is the layer number of the '`dlist`' layer.

3.9.4 dlist - optimize

For a given cursor, this procedure determines if this layer can provide retrieval operations for the lowest cost. Using the example below, this procedure first calls `optimize` for the lower layers. If the lowest cost of the lower layers is greater than the constant `COST`, then (1) the lowest cost value `cost` is updated, (2) the lowest cost layer `layer` is updated, and (3) because dlist does not order its elements, the retrieval_direction is marked as 0. For a more detailed analysis of what `optimize` can do, see See Section 3.12 [Query Optimization], page 25.

```
#define COST    LINEAR_TIME_QOPT_COST*PTR_FOLLOWING_QOPT_FACTOR

optimize( cursor )
{
    optimize( cursor );
    if (cost>COST) {
        cost = COST;
        layer = %ln;
        retrieval_direction = 0;
    }
}
```

3.9.5 dlist - retrieval operations

The '`query`' operation returns the predicate to be used by the cursor. The first line generates a string based on the cursor, the type of the base element, and the current predicate. The value of `predicate` may have been changed in previous layers.

The next line has a `%{` which begins the portion of the code to be generated by '`dlist.c`' (within the pb file). There is just one line, `%v.p`, before the closing symbol `%}`. The last line just frees the memory allocated to the pointer p.

```
query( cursor )
{
    char *p = bind_predicate( cursor, obj_type, predicate );
    %{
        %v.p
    %}
    xfree( p );
}
```

The expression `%v.p` stands for the value of the variable `p`. In this case, if the predicate is '`"$.age>30"`' and the cursor's name was '`"foo"`' and the orig_type is verified to have a structure element called '`age`' of type integer, then the `bind_predicate` routine would generate '`"foo.age > 30"`'. Note that since `query` returns an expression, there is no '`;`' after the code `%v.p`.

These two operations move the object pointer of the cursor to the next and previous element in the list. The expression `cursor.next` is xp shorthand for `cursor.obj.next`.

```
adv(cursor)
%{
    cursor.obj = cursor.next;
%}

rev(cursor)
%{
    cursor.obj = cursor.prev;
%}
```

These two operations return the boundary cases for the list. Both operations return true if the cursor is pointing to NULL. Since these two operations return expressions, they do not end with a semicolon (';').

```
end_adv( cursor )
%{
    ( cursor.obj == 0 )
%}

end_rev( cursor )
%{
    ( cursor.obj == 0 )
%}
```

The `reset_start` function resets the cursor to point to the start of the list; the `reset_end` function resets the cursor to point to the end of the list.

```
reset_start( cursor )
%{
    cursor.obj = container.first;
%}

reset_end( cursor )
%{
    cursor.obj = container.last;
%}
```

3.9.6 dlist - initializing container

Because the container was augmented by `xform` to include the fields '`first`' and '`last`', these two fields must be initialized in the procedure `init_cont`.

```
init_cont( container )
%{
    init_cont ( container );

    container.first = 0;
    container.last = 0;
%}
```

3.9.7 dlist - insert

In the '`insert`' subroutine below, elements will be added to the front of the list. Also note that the first line is `insert(container, record)` which actually means to call the `insert` operation of the next lower layer.

```
insert( cursor, record )
%{
    insert( cursor, record );
    if (!container.first) {
        // EMPTY list.
        container.first = cursor.obj;
        container.last = cursor.obj;
        cursor.next = 0;
        cursor.prev = 0;
    } else {
        // NON-empty list.
        // Add record to beginning of list.
        cursor.next= container.first;
        cursor.prev= 0;
        cursor.next->prev = cursor.obj;
        container.first= cursor.obj;
    }
%}
```

Note that calling the `insert()` operation of the lower layer first is important, as this operation will allocate space for the element and copy the record into the allocated space. Only after this is done is it meaningful to initialize the element and container augmented fields '`first`', '`last`', '`next`', and '`prev`'.

3.9.8 dlist - delete

The lower layer deletions are done after this layer's deletions. In addition, the value of '`cursor.obj`' is unchanged; this constancy is what makes this layer "stable".

```
delete( cursor )
%{

    if ( cursor.next ) {
        cursor.next->prev = cursor.prev;
    } else {
        container.last = cursor.prev;
    }
    if ( cursor.prev ) {
        cursor.prev->next = cursor.next;
    } else {
        container.first = cursor.next;
    }

    delete(cursor);
%}
```

3.9.9 dlist - default operations

If the only thing an operation does is to just call down to the next layer, then the operation does not have to be made explicit. This is the concept of *default operations*. This is different from an operation that does nothing like the example below:

```
init_cont ( container )
{
}
```

If an operation does nothing, not even call the next lower layer, then that operation has to be made explicit. An exception to the rule is `l_verbatim`, which xp knows does not ever call down to the next layer. In this case, the default is to, in fact, do nothing.

There is another exception to this rule. If the layer is a *transitional layer*, that is, the realm type of the layer does not match the realm type of it's first parameter export, then the operation must be defined. In other words, if the operation is allowed in the current realm, and the next layer down belongs to a different realm where the operation is not allowed, then the operation must be defined in the current layer. The allowable realms for the operations are listed in the operations table, whose format is detailed in Section 3.16.2 [Altering the Operations Table], page 43.

For instance, the '`array`' layer is a transitional layer going from realm '`ds`' to realm '`mem`'. The operation `getrec` is valid for the '`ds`' realm, but not for '`mem`'. Therefore, an implementation for `getrec` must be provided in the '`array`' layer.

There are three operations that are exceptions to this exception: `ref`, `iref`, and `pos`. These operations do not have to be defined for transitional layers because xp knows what the defaults should be and can generate the code itself.

Later on in Section 3.13 [Generic Layers], page 30, it will be shown how to change the default for an entire layer.

3.10 Adding Annotations

3.10.1 Simple Annotations

Annotations are used for layers that need some additional data provided by the user in order to process the layer. For instance, the array layer uses an annotation to specify the size of the array. Using '`array.xp`' as our sample code:

```
layer ds array[ mem ] stable annotation retrieval_sometimes

!#include "qopt.h"
!#include "array.h"

ddlhint( argc, argv )
{
    limit 1 argument;
    getarg( %a.size, argv[0] );
}
```

Note that the layer heading contains the keyword `annotation`.

The first line in `ddlhint` says that it only expects one argument to the annotation. The next line copies into the annotation structure field `size` the first argument, `argv[0]`. This annotation structure is defined in '`array.h`', which is listed below:

```
struct array_annote {
    char size[20];
    struct array_annote *next;
};
```

This is the specification of annotations. The first line is the new argument string. The second line is used to link annotations together. Even though `size` is an integer, we need to keep the value as a string due to the way the xp program generates code.

See Section 3.11 [Parsing Annotations Details], page 25 for functions within `ddlhint`.

3.10.2 Optional Annotations

A layer can have optional annotations. One variant of '`malloc`', called '`multimalloc`', determines if an annotation is present. If so, the layer would allocate N objects at a time (where N is the annotation). If there was no annotation, a default of 100 objects would be allocated. The relevant code is below:

```
layer ds multimalloc [ mem ] stable optional annotation retrieval_never

ddlhint( argc, argv )
{
    limit 1 argument;

    /* the test for optional annotations! */
    if ( argc == 0 ) {
        strcpy(%a.size,"100");
    } else {
        getarg( %a.size, argv[0]);
    }
}
```

In a layer marked as `optional annotation`, the code `limit 1 argument` will check to see if there are either zero or one annotations.

3.10.3 Multiple Annotations

Some layers can process many annotations at once (as opposed to replicating the layer once per annotation). These layers are marked '`multiple annotations`' if they require at least one annotation or '`optional multiple annotations`' if they do not require any annotations. Here is an example from '`hashcmp.xp`':

```
layer ds hashcmp [ ds ] stable multiple annotations

!#include "qopt.h"
!#include "hashcmp.h"

ddlhint( argc, argv )
{
    limit 1 arguments;
    getarg( %a.ofield, argv[0] );
}

xform( element, container, cursor)
{
    OP_TAB *t;
    char fieldname[30];

    foreach_annotation {

        // note that the name of the int field that is augmented
        // has the layer number on it, to distinguish it from
        // the original field

        add element : short %a.ofield ;

        sprintf(%a.hfield,"%s%d",%a.ofield,%ln);

        t = find_data_type(element, %a.ofield);
        %a.hashcmp   = t->hashcmp;
        %a.is_literal = t->is_literal;
        strcpy(%a.hashcmp_name, t->hashcmp_name);
    }
    xform( element, container, cursor );
}
```

The ddlhint procedure takes on one annotation at a time. Since the layer is marked with '`multiple annotations`', the annotations are added one at a time to an annotation list. Then, within `xform`, the macro `foreach_annotation` iterates over this list and adds each of the fields to the element.

Layers marked as '`multiple annotation`' are more complicated to write than single annotation layers. Another way to achieve this capability, for the most part, is the *automatic layer replication* mechanism of the pb program, mentioned in section "Automatic Repetition" in *Introductory P2 System Manual*. Briefly stated, a layer in a type expression that accepts a single annotation will be automatically replicated once per annotation in a container declaration. If a container declaration has three annotations, then there are three copies of the layer in the type expression. If the container declaration has no annotations, then the layer is removed from the type expression.

3.11 Parsing Annotations Details

The annotations are sent to a layer by calling the layer's `ddlhint` function. Since there can be an arbitrary number of annotated values, the argc/argv format from C programming is used. There are two macros defined for processing the ddlhint argument vector. They are:

getarg (*a*, *b*) ddlhint

Convert argument *b* into string *a*.

One example of its usage is from '`bintree.xp`',

```
ddlhint( argc, argv )
{
    char   ofield_type[30];
    OP_TAB *t;

    limit 1 argument;
    getarg( %a.ofield, argv[0] );
}
```

Note that the `limit 1 argument` statement makes sure that precisely one argument is passed to `ddlhint`: the name of the ordering field. The `getarg` line converts the string provided through the '`argv`' vector into the annotation field `ofield`.

findtype (*t*, *n*) ddlhint

The argument *n* is a string containing the name of the data type. This function finds *t*, which is a pointer to *n*'s type table entry.

For instance, the code:

```
findtype(t,"int");
```

will have '`t`' point to a table associated with the integer type.

3.12 Query Optimization

This section details advanced features about the `optimize` procedure.

3.12.1 Query Optimization Introduction

First, The current values of the optimization constants.

'CONSTANT_TIME_QOPT_COST'
≡ 1
Cost of primary key retrieval O(1)

'LOG_TIME_QOPT_COST'
≡ 10
Cost of binary search O(log n)

'LINEAR_TIME_QOPT_COST'
≡ 1000
Cost of scanning O(n)

'QSORT_QOPT_COST'
≡ 2500
Cost of quicksorting O(n log n)

'MAX_QOPT_COST'
≡ 100000
Maximum cost O(infinity)

'RANGE_QOPT_SEL'
≡ 0.25
Selectivity of range predicate

'PTR_FOLLOWING_QOPT_FACTOR'
≡ 1.1
Cost of following a pointer

'ARRAY_QOPT_FACTOR'
≡ 1.2
Cost of array retrieval

'HASH_COMPUTATION_QOPT_FACTOR'
≡ 2.0
Cost of computing a hash function

'HASH_BUCKET_QOPT_FACTOR'
≡ 1.5
Ratio empty to non-empty buckets

'ZERO' ≡ 0
The designated layer will process the join

'NSQUARED'
≡ 5000
Nested loops

If the layer (such as hash) has a key field, then this table below of sample layers lists three cost functions that apply respectively to each of the following three cases, corresponding to the three cases P2 distinguishes via the `use_layer()` procedure:

- case ≡ 0
  No qualification. All elements in the container has to be inspected on searches.
- case ≡ 1
  Range qualification. Only a range of elements in a container have to be inspected during searches.
- case ≡ 2
  Point qualification. A search is looking for an element matching a single value.

Essentially, (cost of case 0) >= (cost of case 1) >= (cost of case 2).

If the layer (such as array) does not have a key field, then this table either lists a single cost function that applies to all cases, "call down" if the layer does not affect the retrieval layer, or "do nothing" if it is a bottom layer, meaning it cannot handle retrievals.

array	LINEAR_TIME_QOPT_COST
avail	call down
bintree	(0) LINEAR_TIME_QOPT_COST (1) LINEAR_TIME_QOPT_COST * RANGE_QOPT_SEL * PTR_FOLLOWING_QOPT_FACTOR (2) LOG_TIME_QOPT_COST*PTR_FOLLOWING_QOPT_FACTOR
conceptual	call down
delflag	call down
dlist	LINEAR_TIME_QOPT_COST * PTR_FOLLOWING_QOPT_FACTOR
hash	(0) & (1) LINEAR_TIME_QOPT_COST * HASH_BUCKET_QOPT_FACTOR (2) CONSTANT_TIME_QOPT_COST * HASH_COMPUTATION_QOPT_FACTOR
hashcmp	call down
indx	call down * PTR_FOLLOWING_QOPT_FACTOR

malloc	LINEAR_TIME_QOPT_COST * PTR_FOLLOWING_QOPT_FACTOR
mlist	cost of retrieving index with lowest retrieval cost * PTR_FOLLOWING_QOPT_FACTOR
odlist	(0) LINEAR_TIME_QOPT_COST * PTR_FOLLOWING_QOPT_FACTOR (1) LINEAR_TIME_QOPT_COST * PTR_FOLLOWING_QOPT_FACTOR*(3/4) (2) LINEAR_TIME_QOPT_COST * PTR_FOLLOWING_QOPT_FACTOR*(1/2)
part	If predicate involves both primary and secondary containers: cost of retrieving secondary container * PTR_FOLLOWING_QOPT_FACTOR Otherwise: cost of retrieving primary or secondary container (whichever the predicate involves)
slist	LINEAR_TIME_QOPT_COST * PTR_FOLLOWING_QOPT_FACTOR
transient	do nothing

Note how the cost for the '`part`' layer is computed. If the predicate involves both the primary and secondary containers, then the cost reflects the cost of retrieving the secondary container only. This is because the algorithm searches the secondary container and follows links to get to the primary container. Thus, this cost does not consider or handle the case in which the cost of retrieving the primary container is less than that of retrieving the secondary container.

3.12.2 Optimize Example

If a retrieval layer has a key field, it must deal with the three return values of `use_layer`. The example code below is adapted from the file '`bintree.xp`'.

```
#define COST0   LINEAR_TIME_QOPT_COST * PTR_FOLLOWING_QOPT_FACTOR * 2
#define COST1   LINEAR_TIME_QOPT_COST * RANGE_QOPT_SEL *
                    PTR_FOLLOWING_QOPT_FACTOR * 2
#define COST2   LOG_TIME_QOPT_COST * PTR_FOLLOWING_QOPT_FACTOR

optimize( cursor )
{

   optimize( cursor );
   switch( use_layer(&(%c.b), predicate, %a.ofield, %a.bi) ) {

   case 0:  // whole container search
            if (cost>COST0) {
                cost = COST0;
                layer = %ln;
                if (strcmp( %a.ofield, orderby_field ) == 0)
                  retrieval_direction = 1;
                else
                  retrieval_direction = 0;
            }
            break;

   case 1:  // range search
            if (cost > COST1) ....

   case 2:  // point search
            if (cost > COST2) ....

   default: fatal_err( "use_layer failed" );
   }
}
```

The constants used in the definitions of the '`COST0`', '`COST1`', and '`COST2`' are defined in Section 3.12 [Query Optimization], page 25. The `use_layer` function takes four arguments: the structure which describes the range of the query, the cursor predicate, the key field `ofield`, and the set of comparison functions. Based on the `use_layer` return value, different cost assessments are made.

3.12.3 Query modification in optimize

Another capability of the optimize procedure is to modify cursor predicates. The example is excerpted from '`orderby.xp`'.

```
xform( element, container, cursor )
{
    ....
    add element : int odf;
    ....
}

optimize( cursor )
{
    char q[40];
    sprintf(q,"$.odf%d == 0", %ln);
    post_and( predicate, q );
    optimize( cursor );
    ....
}
```

First, field `odf` is added to the element. If at run-time an element's `odf` field is 1, then the element has been deleted. Since it does not make sense to retrieve deleted elements, the query is modified by post-appending the current value of predicate with the test `$.odf`*ln* `== 0` where *ln* is the layer number of the current layer and '`$`' represents the cursor's name. So if the value of the predicate was `"cursor.age > 50"` before this transformation, and orderby is the 5th layer in the type expression, the value afterwards would be `"cursor.age > 50 && cursor.odf5 == 0"`.

3.13 Generic Layers

The motivation for generic layers was that there were a few things that we wanted to apply to every operation. Rather than list them one by one (and change the list whenever we add adhoc operations), we developed a shorthand notation for generic operation manipulations. The example below is based on the file called '`generic.xp`'.

```
layer ds generic[ ds ] stable no annotation retrieval_always

!#include "qopt.h"

ref( cursor, field )
%{
    ref( cursor, field )
%}


def_cursor_func :
$HEADER
{
    if (cursor_id == -1) {
        char func[200];

        sprintf(func,"/* %s proceduralized by generic */(%s).op_vec[%d] ",
                $STRNAME, cursor, op_list_match( op_list, $STRNAME, 0));
        sprintf( container, "*(%s).con", cursor);
        %{
            %v.func $GENERICARGS
        %}
    } else {
        $CALLDOWN ;
    }
    $GENSEMI ; /* add semicolon to procedures */
}

def_container_func :
$HEADER
{
    if (container_id == -1) {
        char func[200];
        sprintf(func,"/* %s proceduralized by generic */(%s).op_vec[%d] ",
                $STRNAME, container, op_list_match( op_list, $STRNAME, 0));
        %{
            %v.func $GENERICARGS
        %}
    } else {
        $CALLDOWN ;
    }
    $GENSEMI ; /* add semicolon to procedures */
}
```

The `def_cursor_func` is used to generate the body of all cursor-based operations whereas `def_container_func` is used to generate the body of all container-based operations. Operations which

are neither (such as `ddlhint` and `xform` are not automatically generated. These two def-based procedures are placed at the end of the xp file.

All procedures that are declared between the layer declaration and the `def_` declarations do not take on the new default values. In the example above, the `ref` operation does not go through the transformation; rather it takes on the traditional default value of calling down to the next layer.

There are special words called *dollarwords* which begins with a dollar sign and are all caps (see Section 4.11 [Dollarwords], page 53). Dollarwords are generally treated as if they were surrounded by `%{ %}`s. The dollarwords that appear in the example above (with their interpretations) are:

'`$GENSEMI`'
: Generates a semicolon if the operation is a procedure. If the operation is an expression, like `end_adv` or `gettime`, then the semicolon is not generated.

'`$STRNAME`'
: Returns the quoted name of the operation.

'`$HEADER`' Generates the header of the operation.

'`$GENERICARGS`'
: Generates argument list for generic procedures.

'`$CALLDOWN`'
: Generates the code to call down to the next layer.

For instance, applying the new default for container operations to the `open` function would result in this equivalent code:

```
open (container)
{
    if (container_id == -1) {
        char func[200];
        sprintf(func,"/* %s proceduralized by generic */(%s).op_vec[%d] ",
                "open", container, op_list_match( op_list, "open", 0));
        %{
            %v.func (&container) ;
        %}
    } else {
        %{
            open(container) ;
        %}
    }
}
```

3.14 Multiple Containers and Cursors

The xp program doesn't gracefully handle layers with multiple container and cursor types. The problem is that xp is a preprocessor, not a full-fledged compiler[2]. Recall that xp can recognize up to 10 cursors (called '`cursor`' or '`cursor0`', '`cursor1`', ... '`cursor9`'), containers, etc. For xp to do the right translation, two things must occur: (1) there must be a binding of a '`cursor`*n*' to a particular parameter of a layer and (2) the strings associated with all the keywords of '`cursor`*n*' must be properly initialized.

The first problem is dealt with inside the `xform` procedure of a layer. This procedure performs the type transformations - i.e., the mapping of the input record type to the output record types. It is in this procedure that new container, element, and cursor types are created. The structures to aid the bindings are called *member cursors*.

3.14.1 Member Cursor Details

Below are listed some utility functions for creating new element types and for enumerating members of structures (element types).

new_struct (*name, flag*) Member Cursor Function

Returns a handle to a new structure with the name *name*. If *flag* is true, add the structure to the symbol table of pb.

mcursor (*i*) Member Cursor Function

Returns a pointer to a member cursor for the *i* structure.

init_mcursor (*c, i*) Member Cursor Function

Initialize the member cursor *c* to the *i* structure.

mcursor_adv (*c*) Member Cursor Function

Advance the member cursor *c* to the next member.

mcursor_res (*c*) Member Cursor Function

Reset the member cursor *c* to first member.

[2] Specifically, `xp` has neither a full-fledged symbol table nor notion of lexical scoping. xp only recognizes a predefined set of keywords and leaves uninterpreted everything else.

mcursor_eof (*c*) Member Cursor Function

Returns true if member cursor *c* is positioned past the last member.

name_of (*c*) Member Cursor Function

Returns the name (as a string pointer) of the member referenced by member cursor *c*.

ctype_of (*c*) Member Cursor Function

Returns the ctype structure of the member referenced by member cursor *c*.

type_of (*c*) Member Cursor Function

Returns the name of the type (as a string pointer) referenced by member cursor *c*.

position_member (*c, field*) Member Cursor Function

The *field* argument is a string containing the name of a field. This operation positions the member cursor *c* on this field.

foreach_member (*c*) `{` *code* `}` Member Cursor Function

This operation executes *code* for each member in member cursor c.

replicate_cursor*digit1* **as_cursor***digit2*. Member Cursor Declaration

This declares keywords such as '`container`', '`cursor`', '`cursor_type`', etc. which end in *digit2*, to have the same values as the corresponding keywords which end in *digit1*.

new_container_type*digit1 name* **with_type** *name2* Member Cursor Declaration
mapped_via_param *digit2*

Declare a new container `container`*digit1*. The *name* is used to generate the container, cursor, and element types. The type of the new container is *name2*. The container will be implemented via the type expression in parameter *digit2*.

3.14.2 Member Cursor Example

The `xform` procedure in '`part.xp`' partitions the fields of an abstract record into two sets where each set is stored in a new record type. If the original record was:

```
struct a {
    char name[30];
    int age;
    int dept_no;
    char dept[30];
}
```

and the partition argument was '`dept_no`', then these two structures are created:

```
struct a_prim {
    char name[30];
    int age;
}

struct a_sec {
    int dept_no;
    char dept[30];
}
```

This is accomplished by the following code:

```
xform( element, container, cursor )
{
MCURSOR *m;
BOOLEAN sec;
IDENT   *p;
IDENT   *s;
char    name[100];

   // Step 1:  create the primary and secondary element types
   //          initially they will have no fields

   sprintf(name,"%s_prim", obj_type);
   p = new_struct(name, TRUE);

   sprintf(name,"%s_sec", obj_type);
   s = new_struct(name, TRUE);

   add_field_to_sec = TRUE;

   m = mcursor( element );
```

The procedure `new_struct` takes the name of a structure and the constant '`TRUE`' (which means to add the structure to the symbol table) and returns an identifier type. The `xform` procedure uses this to create a pair of record types: a primary type and a secondary type, each containing only

a portion of the base element types. The `mcursor` call creates a member cursor object `m` from the fields in the element. At this point, note that `element` is possibly transformed.

```
    // cycle through each field of element; if a field appears
    // before %a.pfield, put it in the secondary element type

    foreach_member( m ) {
       if (add_field_to_sec) {
         add s : mcurs m;
         if (strcmp( name_of( m ), %a.part_at_field ) == 0)
           add_field_to_sec = FALSE;
       } else {
         add p : mcurs m;
       }
    }
    ...
}
```

The code within the `foreach_member` construct takes one field at a time from member cursor `m` and adds it to either structure `s` or `p`. The value of the annotation `%a.part_at_field` is the name of the last field to add to the secondary type. All fields after this annotated value are inserted into the primary type.

Once a record type has been created, the next step is to create a container of its instances.

```
// Step 2:  create primary and secondary container types
//          this initializes variables element3, container3,
//          cursor3 and element5, container5, cursor5.
//          NOTE: calling new_container_type automatically
//          calls xform().

// the 2+ merely skips over the "__" characters that
// are appended by pb to begin with, and generally shortens
// names

sprintf(name,"%s_prim", 2+ container_type );
new_container_type3 name with_type p mapped_via_param 0;

sprintf(name,"%s_sec", 2+ container_type );
new_container_type5 name with_type s mapped_via_param 1;
```

The first two statements will generate a new container type, `container3`, which will have as its element the fields that were added to the `p` structure. The instantiation of the container (and

element and cursor) will be done through the layers called down using parameter 0. The next two statements declare `container5` similarly, which uses the secondary record structure `s` and is instantiated through parameter 1.

Finally, we need to create the actual containers and have the two record types point to each other. Here's the remaining code:

```
// Step 3:  Add pointers to corresponding segment.

add element3 : struct element5  *sec;   // Add ptr to secondary segment.
add element5 : struct element3  *prim;  // Add ptr to primary segment.

// Step 4:  Create primary and secondary container instances

add container : struct container3 prim;
add container : struct container5 sec;

// Step 5:  Add to the abstract cursor, cursors over the primary and
//          secondary containers.

add cursor : struct cursor3 p;
add cursor : struct cursor5 s;

// Step 6:  Remember handles to each element type.

%k.p = p;
%k.s = s;
%k.a = element;
```

Step 3 adds a pointer to each element type which points to the other element type. Step 4 creates two containers, `prim` and `sec`. Step 5 adds a cursor for each new container. Step 6 stores cursor and element types in the container state.

3.14.3 Many Cursors in One Container

Occasionally, one needs to declare multiple cursor types over a single container within a layer. This occurs, for example, when different cursors over the same container have different predicates. The '`mlist.xp`' layer is an example where different cursor types are used to search a container than those that are used to update an index container. To accomplish this, let the `new_container_type5` statement create the container type. Keywords, such as `container5` and `cursor5`, have specific meanings. However, `cursor5` will be assigned to be one cursor type; we still need another set of keywords to reference another cursor type. The xp statement used to accomplish this is:

`replicate_cursor`*digit1* `as_cursor`*digit2*

In '`mlist.xp`', the actual code used is:

```
replicate_cursor5 as_cursor7;
```

That is, keywords `container7`, `cursor7`, etc. are now legal to use. Associated with a cursor or container is a set of strings to define its type, cursor expression, etc. This statement tells xp that `cursor5` and `cursor7` are mapped via the same type expression, that they have the same container type, element type, and (usually a different) cursor type. Thus, whenever xp sees the keyword `cursor7`, it knows to map it via whatever the parameter was for `cursor5` (parameter 1 in this example).

3.14.4 Assigning Values to Cursors

Finally, there needs to be a way to assign values to the keywords `cursor5`, `container5`, etc. One way to do this is by using a `cursor_func` to define these strings. Here's another example from '`part.xp`':

```
cursor_func cursorbind3( cursor )
{
    sprintf( cursor3,         "(%s.p%d)", cursor, %ln );
    sprintf( cursor_type3,    "%s_prim_curs", container_type );
    sprintf( obj_type3,       "%s_prim_xf", container_type );
    sprintf( container3,      "(%s.prim%d)", container, %ln );
    sprintf( container_type3, "%s_prim_cont", container_type );
    sprintf( orig_type3,      "%s_prim", obj_type );
    strcpy(  predicate3,      %c.prim_p );
    cursor_id3 = %c.id3;
    cost3  = %c.retrieval_cost3;
    layer3 = %c.retrieval_layer3;
    funbody3 = -1;
}
```

When this procedure is called, the value of `cursor3` gets whatever the current cursor expression is plus the field `.p`*ln* so `cursor3` points at the new primary structure. Note that the strings for `cursor3`, `container3`, etc. is based on the current value (`cursor0`). Notice that some of these values have been precomputed and stored in a cursor state variable. Now, whenever xp sees `cursor3`, it will have string that was given to it in this procedure. Check the '`mlist.xp`' for other examples of cursor bindings.

3.15 Link Layers

This section of the documentation is unstable. Currently there is only one link layer implementation. When more link layers are implemented, some of this material may change.

3.15.1 Link Introduction

A *link* is a named relationship between objects in two containers. The relationship is usually specified by a join predicate that relates objects in one container (the *parent*) to objects of the second (the *child*). It is possible for no predicate to be used to connect parent records to child records. In such cases, child-parent pairings are specified manually. However, our primary concern will be on links that have predicates.

Link cardinalities are important ways of understanding limitations on relationships, as well as means for optimization. Cardinalities are expressed in terms of designators: one (`one`), zero-or-more (`many`), or at-most-one (`at_most_one`). Parent records can have one, many, or at-most-one child records and vice versa. A cardinality is expressed as a pairing of designators `p:c`, where `c` is the cardinality designator of child records for each parent record, and `p` is the cardinality designator of parent records for each child record. Thus, the cardinality `at_most_one:many` means that each parent record is associated with any number of child records, but child records will have at most one parent record. Similarly, `one:many` means that parents can have any number of children, but each child is required to have precisely one parent.

In principle, there can be up to nine different pairings of cardinality designators. By convention, parent labels are assigned to containers so that their cardinalities `p:c` is such that `p <= c`. That is, a cardinality `many:one` is really a `one:many` relationship where the roles of parent and child are reversed. This is the convention used.

3.15.2 Link Expressions

These keywords are used only in link layers. Like container and cursor expressions, these keywords can be optionally followed by a digit, 0-9. In the absence of a digit, 0 is assumed.

'`apply`' This variable is bound by the `linkpartition` operation to all predicates within *lpredicate* which involves neither the parent nor the child cursors.

'`bound`' True if the cursor is bound.

'ccard' The child cardinality.

'cresidual'
The residual predicate from the child container.

'linkannote'
Holds the current link annotation.

'linkcost'
Holds the current lowest cost for link layer processing.

'linklayer'
Holds the layer name with the lowest link layer processing.

'linkname'
The name of the link that will be used as the link retrieval layer.

'linkpredicate'
The predicate for the link layers.

'lstate' The link state variable associated with the link. It contains fields that are needed for processing that are neither in the parent cursor nor the child cursor.

'pcard' The parent cardinality.

'presidual'
The residual predicate from the parent container.

'schema' The schema associated with the container.

'special_foreachl'
This variable affects the `foreachl` operation when neither parent nor child cursor is already bound. If this variable is true, then the retrieval layer determines which of the two cursors to bind first. If this variable is false, then the parent is bound and iteration is done over the child cursor.

3.15.3 Link Declaration

A link is declared in a p2 file by the following annotation:

```
'link' linkname 'on' c-card parent-container
                     'to' p-card child-container
    [ 'using' layername ] [ 'where' link-predicate ]
```

The '`using`' layername clause is optional; if it is not specified, any link layer that implements a join algorithm will be used by P2 to implement the link. If specified, that layer is given preference.

(Links that are implemented by pointers must have explicitly specified layer names. If such names were not made explicit, P2 wouldn't know what pointers to augment to object types, etc.).

Likewise the '`where`' clause is optional. A *link-predicate* defines the relationship between parent objects and child objects. The syntax `$p` references parent objects and `$c` references child objects. As an example, consider the `works_in` link, which relates '`Department`' objects to '`Employee`' objects. This link might be expressed as:

```
link works_in on one Department to many Employee using ringlist
    where "$p.deptno == $c.deptno";
```

The cardinality indicators mean that each employee works in exactly one department and that a department can have any number of employees. The '`using`' clause specifies using the layer '`ringlist.xp`' to implement the links. The where clause restrict pairings to the department that the employee belongs to.

Note that not all link layers support links of all cardinalities. A ring list implementation and a pointer array implementation do not support `many:many` links. Thus, cardinalities can be used to verify that the selected layers are in fact able to implement the declared relationship.

A *self-referential* link is a link that connects objects within the same container. The relationship of people who work in the same department is captured by the `same_dept` link:

```
link same_dept on many Employee to many Employee
    where "$p.deptno == $c.deptno";
```

3.15.4 Link Helper Functions

card (*name* , *layername*) Link Functions

Returns the value `ONE`, `MANY`, or `AT-MOST-ONE`.

linkpartition (*lpredicate, presidual, cresidual, rest*) Function

This function splits the link predicate *lpredicate* into 3 predicates: the parent residual predicate, *presidual*, which are all predicates involving the parent and not the child, the child residual predicate, *cresidual*, which are all predicates involving the child but not the parent, and *rest*, which are all the predicates involving neither parent nor child.

3.15.5 Link Layer Details

A link layer exports all operations in the data structure realm, plus the operations on link cursors (see Section 4.9 [Link Operations], page 51). To explain what a link layer does, let us define a *schema* to be a collection of containers. A link is a relationship between pairs of containers. A link layer is a transformation on schemas; it maps schemas with links of a given variety (ring list, pointer array, etc.) to schemas without such links. The stacking of link layers defines a transformation that progressively removes links of the varieties that were represented in the layers. Ultimately, the resulting schema is simply a collection of non-linked containers, which are implemented via their ds type expressions.

There is a realm called '`lnk`', which has two layers: '`nloops`', which implements joins using nested loops and '`linkterm`', which makes the transition from the link realm to the data structures realm. Each layer in the '`lnk`' realm transform each '`ds`' operation accordingly. For example, on `insert`, a ring list layer implementation will automatically connect the new record to its corresponding parents and children; `nloops`, since it doesn't use pointers to make connections, will do nothing. The same holds for other '`ds`' operations. Each '`lnk`' layer will transform each link cursor operation by either passing it to the next lower '`lnk`' layer or by processing the operation (possibly by converting it into '`ds`' operations). A '`lnk`' layer processes a link operation if that layer implements that link. Thus, the a ringlist layer will process all operations on ringlist links, the `nloops` layer will process operations on join algorithm links, etc. The `linkterm` layer transmits '`ds`' operations as is; if it receives any link operations, it means that no '`lnk`' layer above it could process the link operation. This is a fatal error and should not occur; thus `linkterm` is a safety net.

Links are relationships among containers, not container types. As mentioned earlier, links are declared in the context of schemas. The mapping of a schema is specified by a type expression and is done one container at a time. For example, suppose a schema had an employee container and a department container and the link `works_in` related the two implemented using `nloops`. When the employee container is mapped, the `nloops` layer would augment child fields (as employees are the children of the `works_in` link); when the department container is mapped, the `nloops` layer would augment parent fields.

In general, a link layer will have multiple annotations; one annotation for each link declaration. When a container is being mapped, each link layer will examine its links and determine if the container that is being transformed participates as a child or parent of any of these links. If so, the appropriate fields will be added for each link. That is, if a container is the child of one link and the parent of a second, the link layer will add both parent and child fields to the objects of that container. All of this mapping is done by the '`xform`' operation.

3.16 Adding New Operations to xp

Adding a new operation to the xp layers requires writing the operation for one or maore layers and changing the operations table.

3.16.1 Adding the Operation

Adding a new operation to the P2 system is done via the a*dhoc operation*, which are operations not explicitly defined in the xp grammar. Adding a basic operation would entail rewriting large parts of xp and is generally not worth the effort. The only difference between an adhoc operation and a basic operation is that adhoc operations names must be preceded by the word '`adhoc`' in both its definition and usage.

The steps to adding an adhoc operation:

1. Make sure that the first argument to the adhoc operation is either a cursor or a container.
2. Add the operation to all layer files that need its definition. For instance, `gettime` was added to the layer '`timestamp.xp`'.
3. Determine the transitional layers that need explicit default values for the operation. This is achieved by (1) Determining the lowest realm the operation can belong to. (2) Find all layers that transition from that realm to another realm.

   For instance, the operation `gettime` belongs to the ds realm. There are four layers, '`array.xp`', '`qsort.xp`', '`malloc.xp`', and '`multimalloc.xp`' which makes the transition from the ds realm to the mem realm. Therefore `gettime` has to be explicitly defined for these four layers.
4. To the file '`op-tab.h`' in the '`common`' subdirectory, add the name of the operation, in all caps, appended with '`_OP`' to the enumerated list `ADHOCS_OP`. If this is inserted after '`IS_DELETED_OP`', then alter the constant '`HIGHEST_ADHOC_OP`'.
5. To the file '`op-tab.c`' in the same subdirectory, add the operation information in the `op_tab` table. The adhoc definitions are at the end (the order within the adhoc operation definitions doesn't matter—however, adhoc operations appear after all other operations).

3.16.2 Altering the Operations Table

The table that is stored in '`op-tab.c`' contains information about each operation in xp. Each operation is of the form:

```
{ "k_verbatim", 0, 0, 1, 0, 1, 1, 1, 1, 0,
    { 1, 1, 1, 1, 1 },
    1, { CON },
    K_VERBATIM_OP, "K_VERBATIM_OP",
    "NODE *k_verbatim_%s(TE_PTR te, CONT_ARG *container0)",
    "1, container0"
    }
```

The first entry is the name of the operation. The next nine are boolean variables. They represent, in order:

- Is the operation available to the user?
- Does the operation return type void?
- Can the operation be converted into an expression?
- Can the operation be converted into a procedure?
- Is the operation a verbatim operation?
- Does the operation have a container for its first argument?
- Does the operation have to print local variables?
- Does the operation have to print the return statement?
- Is the operation a delete operation?

The next structure is a list of booleans signifying if the operation is part of a realm. The order of the realms is the same order as specified in section "Layer Format" in *Introductory P2 System Manual*, i.e. '`ds`', '`top`', '`mem`', '`lnk`', and '`toplnk`'.

Next is the number of operation arguments. Then in curly braces, the list of argument types. Then the enumerated constant for the operation, the string for this enumerated constant, the header for the procedure declaration of the operation and the string used to make generic arguments.

The enumeration constants are found in '`op-tab.h`'. Basic operations are grouped in one enumerated class, adhoc operations are grouped in another enumerated class.

3.16.3 Adhoc Operation Example

This example below is the file '`sizeof.xp`'. The new operation added to xp is the '`sizeof`' function.

```
layer ds sizeof[ ds ] stable no annotation

xform( element, container, cursor)
{
    add container : int size;
    xform( element, container, cursor);
}

insert( cursor, record )
%{
    (container.size)++ ;
    insert(cursor, record)
%}

delete( cursor )
%{
    (container.size)--;
    delete(cursor)
%}

init_cont( container )
%{
    container.size = 0;
    init_cont( container )
%}

adhoc getsize ( container )
%{
    (container.size)
%}
```

In this file, the size of the container is monitored through insertions and deletions on the container. The adhoc operation is the function `getsize`, which returns the value of the size of container. Note that `getsize` evaluates into an expression rather than a statement, so there is no ";" in its definition. An adhoc operation can have either `container` or `cursor` as its first argument. To call the getsize operation from another layer (another '`mumble.xp`' file), the code is `adhoc getsize( cont1 )`. To call `getsize` from a '`mumble.p2`' file, the code is `getsize(cont1)`: the "adhoc" keyword is not used.

For this file, we would add before the terminating operation definition in '`op-tab.c`':

```
{ "getsize", 1, 0, 1, 1, 0, 1, 1, 1, 0,
    { 1, 1, 0, 1, 1 }, 1, { CON },
    GETSIZE_OP, "GETSIZE_OP",
    "NODE *getsize_%s(TE_PTR te, CONT_ARG *container0)",
    "1, container0"
    }
```

3.17 Adding a New Layer

To add a new layer '`new`' to the P2 system:

- Create the files '`new.xp`' (and optionally, '`new.h`').
- Edit the files '`layers`' and '`layers.all`' in the top-most subdirectory and add the name '`new`' to it.
- If the layer uses an annotation that is not already recognized, alter '`ddl-gram.y`' to add the new annotation grammar and, if necessary, add some new keywords in '`ddl-lex.l`'.
- In the file '`attr.txt`' in the '`lib`' subdirectory, add the layer and its attributes used in layer composition checking. See Chapter 6 [Layer Composition Checks], page 61.
- At the top level, run `make`. All the necessary files will be made automatically.

4 xp Operations

This is the list of additional operations understood by xp; this is to complement the list presented in section "P2 Operations" in *Introductory P2 System Manual*.

The operations `initk` and `foreachk` (defined in see section "Composite Cursor Operations" in *Introductory P2 System Manual*) are *not* available in xp files.

4.1 Reference Operations

A reference operation is an operation that points to a particular object. For most layers, they take on default values: to reference an age field of an employee data type, use `employee.age`. If the data structure is split (as it is for '`part.xp`') then, these operations are used to get the field: our example becomes `ref(employee,age)`. Since xp is able to deal with split data structures, it has to interpret references in terms of operations.

ref (*cursor, field*) Function
iref (*cursor, field*) Function

The `ref` operation returns field `field` of the object that is referenced by `cursor`.

The `iref` operation is like `ref`, but the first argument to `iref` is really the result of a call from `ref` (or another `iref`).

pos (*cursor, expr*) Function

This operation repositions the given cursor to the address `expr`, which can be NULL.

The above three operations are transparent to the user. The code `cursor.next` is interpreted as a call to `ref(cursor,next)`. For an expression with additional indirection, the code `cursor.next->a->b` will be automatically translated to `iref(iref(ref(cursor,next),a),b)`. For positions, code of the form `cursor.obj = c` will be translated to `pos(cursor, c)`.

For most layers, these three operations take on their defaults:

- `ref(c,f)` is `c.obj->f`
- `iref(c,f)` is `c->f`

- `pos(c,e)` is `c.obj = e`.

For an example of a non-default `ref` and `iref`, see the file '`part.xp`'.

4.2 alloc

alloc (*cursor*) Function

This is a memory (mem realm) operation. It allocates space for the record type associated with the cursor argument, and returns a pointer to this space.

4.3 init_cont

init_cont (*container*) Function

This operation initializes the container fields and fields associated with the layer itself. Whenever container (and cursor) fields are referenced, they are augmented with their layer numbers automatically by xp. This operation is called by the `open` operation and is not available in p2 programs.

4.4 Adhoc Operations

The *adhoc operations* are those operations that have been added to the base set of operations.

adhoc *name* (*arguments, ...*) Function

Declare a procedure *name* which takes several arguments. *name* is not a built-in function of xp, hence it needs to preceded with the keyword "adhoc".

The only distinction between an adhoc operation and the other predefined operations is that adhoc operations need to be preceded by the keyword '`adhoc`' in xp layer files. An example from the file '`part.xp`' shows the use of the adhoc keyword, both in definition and procedure call.

```
adhoc gettime(cursor)
{
    call bindc(cursor);
    %{
        (((adhoc gettime(cursor3)) == (adhoc gettime(cursor5))) ?
         (adhoc gettime(cursor3)) :
         (__runtime_error("Error in part/gettime values"),0))
    %}
}
```

Adhoc operations are used in p2 files without the `adhoc` keyword.

The current set of adhoc operations are:

getsize (*container*) Function

This operation, defined in '`sizeof.xp`', returns the size of the container.

gettime (*cursor*) Function

This operation, defined in '`timestamp.xp`', returns the timestamp of the element pointed at by the cursor.

is_deleted (*cursor*) Function

This operation, defined in '`delflag.xp`', returns true if the current element the cursor is pointing to has been marked as deleted.

4.5 xform

xform (*element, container, cursor*) Function

This operation performs type transformations on elements, containers, and cursors by augmenting their fields. For instance, `add cursor : int odf;` will add a new field to the cursor called `odf` of type integer.

For more details, see Section 3.8 [Type Extension Details], page 14.

4.6 ddlhint

ddlhint (*argc, argv*) Function

This procedure extracts the annotations passed to the layer and stores their values (usually) in an annotation field. For further details, see Section 3.11 [Parsing Annotations Details], page 25.

4.7 optimize

optimize (*cursor*) Function

This operation compares this layer's retrieval complexity against all other layers below it and returns the fastest layer for the cursor. In addition, this operation may alter the predicate. For more details, see Section 3.12 [Query Optimization], page 25.

4.8 Verbatim Operations

Verbatim operations allow generation of definitions and operations that are needed by the layers, but are neither basic nor adhoc operations.

l_verbatim ( ) Function

Any layer-specific code that is to be generated is placed inside this procedure. Most layers have empty `l_verbatim` procedures. Unlike all other operations, `l_verbatim` never calls down to lower layers. This exception is handled properly by xp, so there is no need to explicitly specify a null `l_verbatim` in a layer.

k_verbatim (*container*) Function

Any container-specific code that is to be generated is placed inside this procedure. In '`qsort.xp`', the name of the sort field is passed in as part of the annotation. A comparison function is created for each container because each container can be sorted on a different field.

c_verbatim (*cursor*) Function

Any cursor-specific code that is to be generated is placed inside this procedure. For instance, in '`orderby.xp`' every cursor must have its own comparison function (since

each cursor can be ordered over a different field). In the `c_verbatim` function, a sort function and a comparison function are defined which will be instantiated for each cursor created.

4.9 Link Operations

Briefly, a *link* is a connection between elements from two different containers. Within the link, one of the containers is the *parent*, another container is the *child*, and the name of the link connecting the elements from the containers is the *linkname*. For much more detail, see Section 3.15 [Link Layers], page 39.

These operations are added to the link layer. In all these operations:

cursor0 is the cursor in the parent container of the link

cursor1 is the cursor in the child container of the link

lstate is local information about the link that is not specified in either the parent or child cursor. Every link layer defines what belongs in this structure.

loptimize (*cursor0, cursor1, lstate*) Function

The link analogy to the `optimize` operation. The link retrieval layer is chosen on the basis of the link layer that can process the links in the lowest cost. Eventually, the link retrieval layer will convert the link operation into ds realm operations, which in turn will select a ds realm layer to be the retrieval layer.

resetc (*cursor0, cursor1, lstate*) Function

Position the child cursor to the first child of the parent.

advc (*cursor0, cursor1, lstate*) Function

Go to the next child cursor of the parent.

endc (*cursor0, cursor1, lstate*) Function

Return true if there are no more children for the parent.

foreachc (*cursor0, cursor1, lstate, expr*) Function

Iterate over all the children for the parent.

resetp (*cursor0, cursor1, lstate*) Function
Position the parent cursor to the first parent of the child.

advp (*cursor0, cursor1, lstate*) Function
Go to the next parent cursor of the child.

endp (*cursor0, cursor1, lstate*) Function
Return true if there are no more parents of the child.

foreachp (*cursor0, cursor1, lstate, expr*) Function
Iterate over the parents of the child.

foreachl (*cursor0, cursor1, lstate, expr*) Function
Iterate over each child-parent pair.

disconnect (*cursor0, cursor1, lstate*) Function
Disconnect the link between a parent and child for the linkname.

connect (*cursor0, cursor1, lstate*) Function
Connect the child and parent into a link for the given lstate linkname.

related (*cursor0, cursor1, lstate*) Function
Return true if the child and parent are related by the link in lstate.

4.10 Meta-Ops

These constructs are not true operations but templates for operations.

cursor_func *name* (*cursor, ...*) Function
container_func *name* (*container, ...*) Function
The first function provides a way to create functions that will be called by cursor-based operations (operations where the first argument is a cursor). The second operation is for container-based operations.

For instance,

```
cursor_func common( cursor, "int value" )
%{ cursor.alpha = 0;
   cursor.beta  = %v.value;
   cursor.gamma = 7;
%}

delete( cursor )
%{ call common( cursor, "5");
   delete( cursor );
%}
```

Note that the keyword `call` must precede the call to `common`.

default_cursor_def *':' body* Function
default_container_def *':' body* Function

For every cursor (container) definition not already explicitly defined, the definition is based on this procedure definition.

The body of this procedure contains all of the normal xp syntax plus some special keywords called *Dollarwords* (because they begin with a `$`). The first keyword in the body is a `$HEADER`. For details, See Section 4.11 [Dollarwords], page 53.

4.11 Dollarwords

The dollarwords are tokens beginning with a dollar sign and are used only in default operation specifications. Their meanings are listed below along with an example

$NAME Dollarword

Returns the name of the operation.
'`open`'

$OP Dollarword

Returns the index number of the operation.
'`14`'

$GENSEMI Dollarword

Generates a semicolon if the operation is a procedure. If the operation is an expression, like `end_adv` or `gettime`, then the semicolon is not generated.
semicolon generated

$STRNAME Dollarword

Returns the quoted name of the operation.
‘`"open"`’

$HEADER Dollarword

Generates the header of the operation.
‘`open_generic (container)`’

$GENERICARGS Dollarword

Generates the argument list for generic procedures.
‘`(&container)`’

$CALLDOWN Dollarword

Generates the code to call down to the next layer.
‘`open(container)`’

$CALLDOWN*n* Dollarword

Generates the code to call down to the next layer using *n* as the number of the cursor (or container).
‘`open(container5)`’ (for `$CALLDOWN5`)

$CALL_LEVEL Dollarword

Generates the code to call the procedure at this level. ‘`open#(container)`’

Most dollarwords are treated as if they were surrounded by `%{ %}`s. Additional dollarwords will be added as they are needed. Since these defaults replace the standard default operation (calling down to the next layer), any procedure that is not supposed to do the default operations will have to be explicitly defined.

A synchronization example would be defined thusly:

```
def_cursor_func :
$HEADER
{
    %{ {wait(s); $CALLDOWN ; send(s)} %}
    $GENSEMI ;
}
```

5 Invoking the P2 Programs

The programs '`ddl`', '`xp`', and '`pb`' all take short and long option forms (one hyphen or two). All of them can display the current version using '`-v`' or '`--version`'. All of them can display the list of command line options using '`-h`' or '`--help`'.

5.1 ddl options

'`ddl`' *options* `filename['.ddl']`

'`-d`' ddl Option
'`--debug-yacc`' ddl Option
Display the tokens parsed by ddl.

'`-h`' ddl Option
'`--help`' ddl Option
Display the list of valid command line options and quit.

'`-w`' ddl Option
'`--warnings`' ddl Option
Display warnings.

'`-v`' ddl Option
'`--version`' ddl Option
Display the version number of the program and quit.

5.2 xp options

'`xp`' *options* `filename['.xp']`

'`-h`' xp Option
'`--help`' xp Options
Display the command line options for xp and quit.

‘-c’ xp Option
‘--hierarchy-comments’ xp Options
For every procedure in the layer, write out a comment string containing the name of the procedure at the beginning and the end of the procedure. This is known as a *hierarchical comment*. Used in conjunction with the `-c` flag below for debugging.

‘-v’ xp Option
‘--version’ xp Options
Display the version number of xp and quit.

‘-w’ xp Option
‘--warnings’ xp Options
Display all warnings. The default is not to display any warnings.

‘-d’ xp Option
‘--debug-yacc’ xp Options
Display the tokens parsed by xp.

5.3 pb options

‘pb’ *options* `filename[‘.pb’]`

‘-h’ pb Option
‘--help’ pb Option
Display the options for pb and quit.

‘-a’ pb Option
‘--attribute-file’ pb Option
Read the attribute file for design rule checking from *filename*.

‘-v’ pb Option
‘--version’ pb Option
Display the version number of pb and quit.

`-d` pb Option
`--debug-yacc` pb Option
Display the tokens parsed by pb.

`-w` pb Option
`--warnings` pb Option
Display all warnings. The default is not to display any warnings.

`-c` pb Option
`--hierarchical-comments` pb Option
If a layer is compiled with hierarchical comments (see `-c (xp)` above), then this flag will allow the hierarchical comments to pass through to the generated C code.

6 Layer Composition Checks

6.1 DRC Introduction

There are compositions of layers that are syntactically correct, but are semantically incorrect. The purpose of pb's built-in design rule checking is to evaluate additional rules (beyond matching type signatures) to assure P2 users that their type equations are semantically meaningful.

6.2 Preliminaries

A *subsystem* (or type expression) is a composition of one or more layers. A *property* is an attribute of a layer or subsystem that is rooted at that layer. Two example properties are:

- *stability* - is layer or the subsystem that it roots stable? That is, can cursors point to holes in a container?
- *has_inbetween* - does layer or the subsystem that it roots have the inbetween capability/layer that guarantees stability of cursors?

A *property value* is a value that can be assigned to a property. In the design rule checker (DRC), there are only three possible values that can be assigned to a property P:

asserted	property P is asserted.
unset	no information about P is known.
negated	property P is negated.

As an example, consider the '`stability`' and '`has_inbetween`' properties. If a subsystem has no unstable layers, stability property should have the value 1. If a subsystem does not have an inbetween layer, has_inbetween will have the value 0. On the other hand, it is possible for has_inbetween to have the value 1, yet the subsystem is unstable. This is possible if an unstable layer is placed *above* the inbetween layer.

A *requirement* is a constraint that is imposed on the value of a property. There are six possible requirements in our model for a property P:

assert property P must have the value

unset no constraint is imposed on property P

negate property P must not have the value

may negate
property P may exhibit the values unset or negate

may assert
property P may exhibit the values unset or assert

set property P may exhibit the values assert or negate

Design rules for every layer are expressed in terms of preconditions, postconditions, and restrictions. A *postcondition* is a set of asserted or negated properties. A *precondition* is a constraint that is imposed on selected properties to regulate the usage of a layer. If preconditions are not satisfied, the layer cannot be used. In general, the precondition of a layer must satisfy the postconditions of higher-level layers. A *restriction* is a constraint on a subsystem that instantiates the parameters of the layer. In general, the postconditions that are exported by layers of a subsystem must satisfy the restrictions of a layer's parameter for a correct instantiation.

The model of DRC is to make sure that for every layer X:

- The postcondition of the subsystem (below layer X) satisfies the restrictions of the parameter of the layer X.
- Layer X satisfies the precondition for the layer above X.

If a precondition or a restriction is not satisfied, the checking is usually halted. However, these two conditionals can be flagged with the word "warning" which will continue processing.

6.3 DRC BNF

Below is the BNF for the uncommented attribute table (currently found in the '`p2/tools/attr.txt`') for design rule checking on the layers in P2. C-style comments '`/* comment */`' can appear anywhere. All tokens must be separated by whitespace, including '`/*`' and '`*/`'.

```
restfile   :== attr-list layerdefs*
attr-list  :== 'attributes' '=' '{' attr-pair* '}'
attr-pair  :== layername layer_descr
layerdefs  :== layername+ : '{' cond* '}'
cond       :==  'preconditions' [ 'warnings' ] list*
              | 'postconditions' list2*
              | 'restrictions' [ 'warnings' ] list*
              | 'restrictions-'num [ 'warnings' ] list*
list       :== 'assert' attr* | 'negate' attr* | 'set' attr* |
               'unset' attr* | 'may negate' attr* | 'may assert' attr*
list2      :== 'assert' attr* | 'negate' attr*

layername  :== string representing name of layer
layer_desc :== double quoted string representing layer description
attr       :== string representing attribute name
num        :== number representing nth parameter
```

6.4 Examples

First, the attribute list looks something like this:

```
attributes = {
        in_between      "an inbetween flag layer"
        unstable        "an unstable layer"
        retrval         "a retrieval layer"
        always          "a layer always used in retrievals"
        pred            "a predicate-ordered layer"
        qual_present    "a qualification layer"
}
```

The description string for each attribute name is also used in error messages, so it is important to follow the format '"`a` *something* `layer`"'.

Now let us look at some data from the attributes file to determine how the layer design rule checks work.

```
avl bintree odlist : {
        preconditions       assert in_between
        postconditions      assert retrval pred unstable
}
```

This rule says that the preconditions and postconditions apply to three layers: '`avl`', '`bintree`', and '`odlist`'. The precondition states that before this layer is used, the layer above it must have

asserted the `in_between` property. In addition, these layers assert the properties `retrval` (because the layers are retrieval_sometimes layers), `pred` (because the layers are ordered over a predicate), and `unstable` (because the cursor points to the next element after deletion).

```
avail : {
        restrictions may negate retrval
}
```

This example shows that the '`avail`' layer's parameter 0 (its only parameter) can be instantiated with a subsystem that either explicitly negates the `retrval` property or doesn't mention it at all. This is the effect of `may negate`. If the property had to be explicitly negated, then it would appear as `negate`.

```
orderby : {
        postconditions assert always
        restrictions-1 negate retrval
        restrictions-0 assert retrval
}
```

In this example '`orderby`' has two parameters, 0 and 1 (recall the type of this layer is `ds orderby [ ds, top ]`. Parameter 0 has to be instantiated with a subsystem that has the '`retrval`' property where parameter 1 has to be instantiated with a subsystem that explicitly negates the '`retrval`' property.

```
top2ds_qualify : {
        postconditions assert qual_present
        restrictions   warnings assert qual_present
}
```

This example has the layer '`top2ds_qualify`', which is a combination of the layer '`top2ds`' and '`qualify`'. Because of this, the `qual_present` property is asserted (as well as in the layer '`qualify`'). If the '`qualify`' layer appears underneath this one, then a warning is generated because even though this is a wrong combination, there's no harm in the combination.

Concept Index

P

Q

R

S

T

U

V

W

X

Functions and Variables Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

X

Z

Appendix A dlist.xp program

Below is the complete '`dlist.xp`' program, which was dissected in Section 3.9 [Example - dlist.xp], page 15.

```
layer ds dlist[ ds ] stable no annotation retrieval_sometimes

!#include "qopt.h"

// dlist maintains a doubly-linked list of chronologically-insert-
// ordered elements


// xform - dlist adds next, prev, mark to elements; first, last to
container

xform( element, container, cursor )
{
    add element : struct element *next;
    add element : struct element *prev;
    add container : struct element *first;
    add container : struct element *last;
    xform( element, container, cursor );
}

// optimize

#define COST    LINEAR_TIME_QOPT_COST*PTR_FOLLOWING_QOPT_FACTOR

optimize( cursor )
{
    optimize( cursor );
    if (cost>COST) {
        cost = COST;
        layer = %ln;
     retrieval_direction = 0;
    }
}

// retrieval operations
// query, adv, rev, reset_start, reset_end, end_adv, end_rev

query( cursor )
```

```
{
    char *p = bind_predicate( cursor, obj_type, predicate );
    %{
        %v.p
    %}
    xfree( p );
}

adv(cursor)
%{
    cursor.obj = cursor.next;
%}

rev(cursor)
%{
    cursor.obj = cursor.prev;
%}

end_adv( cursor )
%{
    ( cursor.obj == 0 )
%}

end_rev( cursor )
%{
    ( cursor.obj == 0 )
%}

reset_start( cursor )
%{
    cursor.obj = container.first;
%}

reset_end( cursor )
%{
    cursor.obj = container.last;
%}

// non-retrieval operations
// init_cont, insert, delete

init_cont( container )
%{
    init_cont ( container );

    container.first = 0;
    container.last = 0;
%}
```

```
insert( cursor, record )
%{
    insert( cursor, record );
    if (!container.first) {
        // EMPTY list.
        container.first = cursor.obj;
        container.last = cursor.obj;
        cursor.next = 0;
        cursor.prev = 0;
    } else {
        // NON-empty list.
        // Add record to beginning of list.
        cursor.next= container.first;
        cursor.prev= 0;
        cursor.next->prev = cursor.obj;
        container.first= cursor.obj;
    }
%}

delete( cursor )
%{

    if ( cursor.next ) {
        cursor.next->prev = cursor.prev;
    } else {
        container.last = cursor.prev;
    }
    if ( cursor.prev ) {
        cursor.prev->next = cursor.next;
    } else {
        container.first = cursor.next;
    }

    delete(cursor);
%}
```

Table of Contents