Introductory P2 System Manual

Edition 0.1, August 1994

Don Batory
Bart J. Geraci
Jeff Thomas

Edition 0.1
For `P2` Version 0.1
August 1994

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

Preface

This manual documents the use of the P2 system. It also outlines how programmerrs can customize the system for sophisticated applications. First-time users are encouraged to read this manual. Advanced users are encouraged to read '`Advanced P2 System Manual`'.

In addition to the usual index of concepts, this manual provides separate 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 Distribution

The P2 program and manuals can be retrieved via anonymous ftp. See Chapter 3 [Installation], page 7.

To be put on a P2 mailing list, please end an e-mail message with the subject "Add to P2 list" followed by your e-mail address. Send request to `dsb@cs.utexas.edu`

CONTACT INFORMATION:

Don Batory
University of Texas at Austin
Department of Computer Science
Taylor Hall 2.124
Austin TX 78712.

`dsb@cs.utexas.edu`

3 Installation

The p2 distribution consists of two compressed tar files available via anonymous ftp. One tar file contains the p2 system code, the other contains the p2 system manuals.

As a rule, we have tried to make p2 as portable as possible. One exception to this rule are that p2 requires GNU make. *Standard UNIX* make *will not work.* Besides GNU make, the code distribution requires an ANSI C compiler, lex and yacc (or their GNU equivalents). The manuals distribution includes PostScript and dvi versions of the manual, and thus requires only a PostScript or dvi viewer.

To install the code distribution, you should get the file via ftp, uncompress it, un-tar it, cd into it, run configure, and make.

Here is a step-by-step example of how to get the code distribution and make it on your system. This example assumes your login is dsb@cs.utexas.edu, and you wish to install p2 in the directory /u/dsb/foo/p2.

```
% ftp ftp.cs.utexas.edu
Name (ftp.cs.utexas.edu:dsb): anonymous
Password: dsb@cs.utexas.edu
ftp> binary
ftp> cd pub/predator
ftp> get p2-0.1.tar.Z
ftp> bye
% uncompress p2-0.1.tar.Z
% tar xf p2-0.1.tar
% cd p2-0.1
% ./configure --prefix=/u/dsb/foo/p2
% make
% make install
```

To install the manuals distribution, you should get the file via ftp, uncompress it, and un-tar it.

Here is a step-by-step of how to get the manuals distribution. This example assumes your login is dsb@cs.utexas.edu.

```
% ftp ftp.cs.utexas.edu
Name (ftp.cs.utexas.edu:dsb): anonymous
Password: dsb@cs.utexas.edu
ftp> binary
ftp> cd pub/predator
ftp> get p2-manuals-0.1.tar.Z
ftp> bye
% uncompress p2-manuals-0.1.tar.Z
% tar xf p2-manuals-0.1.tar
```

4 Introduction

P2 is a state-of-the-art generator for data structures. It is an extension of ANSI C that allows programmers to interact with complex data structures using high-level and easy-to-use abstractions. With minimal specifications from programmers, P2 replaces program references to these abstractions with C code that implements them. The number of potential implementations of the basic P2 abstractions is already large and is open-ended.

The goals of this manual are (1) to show how powerful programs can be written in terms of P2 abstractions and (2) to show how implementations of P2 abstractions are specified.

4.1 The Conceptual Basis for P2

P2 is among a new breed of generators that rely on software components to synthesize software. P2 is based on the *GenVoca* model of software system construction. In essence, the premise of GenVoca is that fundamental programming abstractions underly all mature software domains. By standardizing abstractions and their implementations, one can realize a software components technology for a domain.

Although the number of fundamental programming abstractions in a domain is rather small, there is a huge number of possible implementations. The GenVoca approach also advocates a layered decomposition of implementations, where each *layer* (or *component*) encapsulates a primitive software building block. The number of primitive building blocks in a domain is generally small (i.e., on order of 100); however the number of ways in which building blocks can be combined is exceedingly large.

The model of data structures that is implemented by P2 relies on a small number of simple but powerful programming abstractions that have been standardized. Moreover, the P2 library consists of over thirty components that encapsulate many of the common data structure building blocks.

In this manual, we will explain in detail the standardized programming abstractions of P2 and the current set of building blocks in the P2 library.

4.2 The Organization of the P2 Generator

P2 consists of a series of three interconnected preprocessors:

p2 - a shell script that converts a .p2 program into a format understandable by ddl.

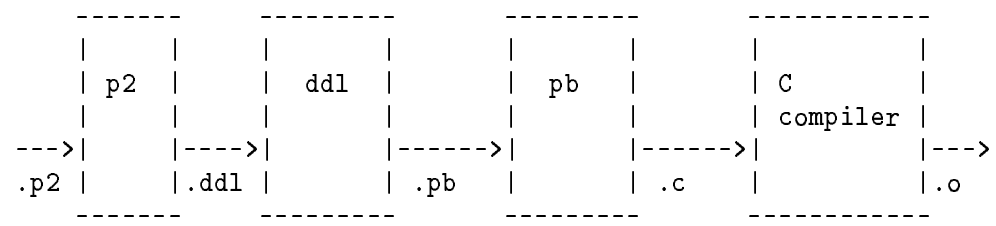
ddl - a preprocessor that repackages implementation specifications for P2 abstractions into a format understandable by pb.

pb - a preprocessor that translates P2 constructs into C code

As a .p2 file is being "compiled", different intermediate formats of the file (e.g., .ddl, .pb) are produced. Errors are detected and reported at all stages of translation, where different classes of errors are detected during each translation phase.

There is a fourth preprocessor that is not part of the .p2 to .c translation pipeline. This is *xp*, a special language/preprocessor that is used to write P2 components. We will not discuss xp further in this manual; readers interested in writing P2 components are urged to read section "xp Manual" in *Advanced P2 System Manual*.

The figure below illustrates the P2 system organization:



To compile the program '`mumble.p2`', simply type:

```
P2 mumble.p2
```

P2 understands several command-line options, which are explained in Section 8.3 [P2 options], page 58. A sample program is listed and dissected in Appendix A [Example P2 program], page 67.

4.3 How to Use this Manual

We assume that the reader is familiar with C, UNIX, and GNU make.

This manual has several chapters. The ones which are worth reading depends on your goals:

For P2 novices:

- Introduction - This chapter.
- P2 Language - The syntax for a '`mumble.p2`' file. See Chapter 5 [P2 Language], page 13.
- Operations - The list of functions in the P2 system to provide the container/cursor operations. See Chapter 6 [P2 Operations], page 37.
- P2 Layers - The different layers that can be used to describe the method elements are organized in containers. See Chapter 7 [P2 Layers], page 41.
- Invoking P2 - How to run P2 and the arguments for the P2 system. See Chapter 8 [Invoking P2], page 53.

People who intend to write layers should first read this manual and then the Advanced P2 System Manual (see section "xp Manual" in *The Advanced P2 System Manual*) which contains information about how to write a layer in xp.

For people who are installing the P2 system, please read the section

> Installation - This chapter provides a brief, but useful, introduction to installing P2. See Chapter 3 [Installation], page 7.

People who are responsible for maintenance of the P2 hierarchy should be familiar with `GNU autoconf` and `make`.

5 P2 Language

P2 is a superset of ANSI C. Thus a P2 source file has the format of a C file with support for the container/cursor abstractions. These will be discussed in detail below. In addition, C++ style comments (where everything to the right of a '//' is ignored) are supported.

5.1 The Container/Cursor Overview

The paradigm used in P2 is the *container/cursor model*. A *container* is a collection of objects called *elements*. Elements are referenced by a structure called a *cursor*. With respect to the container, a cursor can:

- move backwards and forwards through the container
- start at the beginning or the end of the container
- add, delete, swap, and update elements

A *qualified cursor* only points to elements that share some characteristic. Qualified cursors have two properties: a *predicate*, which restricts the cursor to point only to elements that satisfy the predicate, and an *orderby clause*, which specifies the order in which the cursor retrieves elements from the container.

The power of qualified cursors is the ability of the P2 system to optimize operations based on their qualifications. If a cursor is restricted to all even numbers, then the procedure that is generated by P2 for searching that container will have this test embedded. The disadvantage is that to a certain extent, we cannot have dynamically (run-time) cursor predicates that are common in interactive environments.

P2 programs are written in terms of operations on cursors and containers and without regard to how they are implemented. This, in principle, enables different implementations of cursors and containers to be "plugged in" without requiring program modifications.

Ultimately, however, a specification of how cursors and containers are to be implemented must be provided to the P2 compiler, as the compiler replaces cursor and container declarations and operations with their corresponding C implementation. An implementation specification comes in two parts: a type expression and its annotations.

A *type expression* is a composition of P2 building blocks; it specifies a stacking of layers that defines the general characteristics of the data structure that P2 is to generate. Additional information, such as key fields, array sizes, etc., are called *annotations*. The combination of type expressions and annotations is the sole means for specifying data structure implementations. Among the characteristics that can be defined by type expressions and annotations are:

- Whether elements are ordered or not, and if so, under what field is the element ordered.
- If the container uses transient or persistent storage.
- Details of searching and deletion strategies.
- Whether or not the container should contain a maximal number of elements

As mentioned earlier, the power of P2 programs stem from the separation of cursor and container abstractions and their implementations; by altering a P2 program's type expression(s), containers and cursors can be assigned radically different implementations. This significantly simplifies tuning.

5.2 Operation Usage

For example, suppose we have a container called primes with the elements `2 3 5 7 11 13 17 23 29`. We declare a cursor `c` that ranges over all elements of the container. After we initialize the cursor, we can perform a reset which positions the cursor on the first object:

```
c
2 3 5 7 11 13 17 23 29
```

We can iterate through the list by successive advance and/or reverse operations. Three advances positions the cursor at `7`.

```
      c
2 3 5 7 11 13 17 23 29
```

A reverse and cursor is pointing at `5`.

```
    c
2 3 5 7 11 13 17 23 29
```

Two more reverses and cursor is pointing at `2`.

To test whether a cursor has gone past the end of the container, the operations end_adv and end_rev are provided. end_adv returns true if an advance operation positions the cursor past the end of the container; end_rev does the same for the reverse operation. If another reverse is attempted, then end_rev returns true.

Normally, programmers do not call the advance and end_adv operations directly, but use the P2 foreach(c) loop construct. foreach(c) uses cursor c to iterate over elements of a container. If c is qualified, then only those elements that satisfy c's predicate are examined. P2 expands foreach(c) into calls to advance(c) and end_adv(c). The rofeach(c) loop construct does the same, except it traverses elements of the container in the opposite order, using reverse(c) and end_rev(c) operations.

Using the primes container, a cursor `q` qualified over all numbers ending in '7' will either point to `7` or `17`. Therefore if `q` is initialized and then a reset_start operation is performed on it, `q` will point to `7`. A single advance would point `q` next to `17`.

Cursors are also used to delete objects. In the previous example, if the delete operation is invoked, then `17` would be removed from the container, but the cursor would still point to the location occupied by the `17`. If an advance is performed, the cursor would be positioned on the next qualified element, which in this case would be `NULL`, since there are no elements ending in '7' past `17`.

```
                        q
2 3 5 7 11 13 23 29
```

An insert operation performed with the number `47` would insert the element somewhere in the container. It is up to the particular type expression of the container as how to elements should be added (whether it places it at the head of the list, the tail of the list, or keyed by some field).

```
                                q
2 3 5 7 11 47 13 17 23 29
```

More details will be given in subsequent sections.

5.3 Container Declarations

A container is a collection of elements of *element_type* that is implemented by some type expression. The type expression may need additional information, which is the annotation list. The syntax is:

```
container_declaration : 'container' '<' element_type '>'
                        'stored_as' type_expression_name
                        'with' '{' annotation_list '}'
                        identifier_declarator_list ';'
annotation_list : { annotation ';' }*
```

For example, let us declare two types that we will use in our examples. A type for storing prime numbers called prime_num_type and a type for storing employee data, emp_type.

```
typedef struct {
    int num;
}  prime_num_type

typedef struct EMPLOYEE {
    char name[20];  // last name
    int  age;
} emp_type;
```

Next, we will define a container storing a linked list of prime numbers:

```
container < prime_num_type > stored_as linked_list
with { } prime_container, *pointer_to_prime_container;

// alternate way of defining the above
typedef container < prime_num_type > stored_as linked_list
with { } PRIME_CONTAINER_TYPE;

PRIME_CONTAINER_TYPE prime_container, *pointer_to_prime_container;
```

Now for the employees example:

```
container < struct EMPLOYEE > stored_as linked_list
with { } emp_cont, emp_cont_arr[4];       // array of containers
```

Note that `linked_list` is the name of a type expression (whose definition we will give later on) and that it has no annotations.

5.4 Cursor Declarations

A cursor is a structure that points to elements in a container. A cursor can only point in one container, but a container may have more than one cursor. The syntax for a cursor declaration is:

```
cursor_declaration: 'cursor' '<' container_name '>'
                    [ predicate_specifier ]
                    [ ordering_specifier ]
                    identifier_declarator_list ';'
predicate_specifier:      'where' predicate
ordering_specifier:      'orderby' orderby_clause
```

Cursors have two optional properties: a predicate and an orderby clause.

A cursor predicate specifies a subset of the elements in the container to which the cursor my be bound. A predicate is specified as a double-quote enclosed C boolean expression with the extensions that (1) the dollar sign refers to the cursor object and (2) string constants are enclosed within single quotes. Any expression for the predicate can be specified, but P2 is able to make certain optimizations only on a subset of predicates known as *structured terms*. This subset of predicates is defined by the following grammar:

```
predicate: term
         | predicate ' && ' term // Blanks are important
term:      field relop value          // Structured term
         | value                      // Unstructured term
field:     '$.' field_name          // $ is the cursor alias
relop:     '==' | '>=' | '<=' | '<' | '>' |
           '!='                     // Boolean expressions
value:     non-blank_character_sequence |
           string-literal
```

A predicate is a series of terms joined by '`&&`'. A term is either a relational expression, which is called a *structured term*, or not, which is called an *unstructured term*. Unstructured terms must be written in parentheses and without spaces.

An example of a structured term is '`"$.name == 'Batory, Don' "`' where an example of an unstructured term would be '`"($.age>40||$.age<30)"`'. Anytime a disjunction ('`||`') appears, the expression is unstructured. So whenever a disjunction is used, the term it appears in must be written without blanks and in parenthesis. Also note that predicates can contain function calls such as `"(fn($.name,currname))"`, in which case they are unstructured as well. String literals are enclosed within single quotes (e.g., 'Batory'). A string literal cannot have embedded quotes, single or double.

Here are more examples of structured P2 predicates on an employee record type with a name string field and an integer age field:

```
"$.age > 15"                    // age field greater than 15
"$.age == 15"                   // age field equals 15
"$.name >= 'B' && $.name < 'C' && $.age > 40"
                                // name field begins with letter 'B'
                                // and age field greater than 40
"$.age > x"                     // age field greater than value of variable x
"$.age > x && $.age < y"        // age field falls within range (x,y)
```

Note that spaces are important in the above predicates.

Here are more examples of unstructured P2 predicates:

```
"($.age>15)"                    // enclosed within () and without blanks
"(is_old($.age))"               // function call where age field passed
                                // as parameter
"($.name>='B'&&$.name<'C'&&$.age>40)"
                                // same as third structured example above
"(($.name>='B'&&$.name<'C')||$.age>40)"
                                // name field begins with 'B' or age > 40
```

Here are examples of illegal P2 predicates:

```
"$.age> 15"                     // missing blank before >
"($.age>15"                     // missing right parenthesis
```

An *orderby clause* specifies the field and sort direction for retrieved elements. The syntax for the orderby clause is:

```
orderby_clause: 'ascending' field_name
              | 'descending' field_name
              | field_name
```

The reserved word `ascending` (`descending`) specifies to sort in increasing (decreasing) order. The field `field_name` is the ordering key. Depending on the type of the key, the sorting method is numeric or lexicographic. Sorting on multiple keys are not currently supported. The third case defaults to ascending by the `field_name` key.

For our first example, the following cursor declaration declares a cursor variable over the prime number container. The cursor is qualified to only match numbers ending in '1' and to return the elements in increasing order.

```
cursor < prime_container >
  where "($.num%10) == 1" // Unstructured Predicate
  orderby ascending num // Orderby clause
prime_cursor;           // cursor variable
```

The next example is a cursor for the employees container, which will range over all employee elements whose name begins with an 'M'. In addition, these elements are retrieved in reverse order (larger numbers appear first) by their ages.

```
cursor < emp_cont >
    where "$.name >= 'M' && $.name < 'N'"
    orderby descending age
```

5.5 Type Expressions

5.5.1 Type Expression Declarations

A *type expression* is a composition of layers which represents an implementation for the container. Type expressions are defined using a *typex declaration* which is a sequence of zero or more type assignments. The typex declaration assigns a symbolic name to type expressions. These names may be subsequently referenced to specify the container implementation. Type expression names are ordinary C identifiers which do not end with a digit.

```
typex_declaration :      'typex' '{' typex_assignment* '}' ;
typex_assignment :       typex_name '=' typex_definition ';'
```

For example, the following typex declaration declares two type expressions, named s and t:

```
typex {
  s = conceptual[slist[delflag[array[transient]]]];
  t = conceptual[odlist1[odlist2[malloc[transient]]]];
}
```

Interpreting type expressions requires some background on what they actually mean. A *term* of a type expression is a P2 layer (or component). Every P2 component encapsulates a consistent

data type and operation mapping for cursors, containers, and their elements. A type expression is a composition of layers that defines a sequence of mappings that transforms a P2 program into a C program. Thus, to understand what a type expression means requires understanding the mapping that is performed by each layer of the type expression.

With this as a background, let's analyze these expressions to see what they mean. The full meanings of the individual layers are given in detail in See Chapter 7 [P2 Layers], page 41. For '`s`':

- The `conceptual` layer is actually a built-in composition of many P2 layers that accomplish sorting, loop rewrites etc. It is typically the first layer of every type expression.
- The `slist` layer links together elements of a container onto a singly linked list in order of insertion.
- The `delflag` layer the delflag layer rewrites an element delete operation into an element update that marks an element deleted; the storage space for an element is not reclaimed.
- The `array` layer provides storage for the elements in a preallocated array. The implementation of array ignores deletions, so the `delflag` layer is needed above the `array` layer.
- The `transient` layer stores the element in main memory. This is different than the `persistent` layer which stores the element to disk.

In summary, '`s`' is a layer which stores its elements in an array (which are also linked in a list) in main memory. Let us analyze '`t`' similarly:

- The `conceptual` layer is actually a built-in composition of many P2 layers that accomplish sorting, loop rewrites etc. It is typically the first layer of every type expressions.
- The `odlist1` layer links together elements of a container via an ordered doubly-linked list. The ordering for this layer depends on the annotation.
- The `odlist2` layer is exactly like `odlist1`, though the ordering key may be different.
- The `malloc` layer provides storage from the heap for each new element allocation.
- The `transient` layer stores the element in main memory.

So '`t`' allocates space for the elements on demand and links the elements together using two keys.

5.5.2 Type Expression Annotations

Besides the layer parameters, a layer may have additional parameters called annotations. The only layer of type expression `s` that has an annotation is `array`. The '`array`' needs the size of the

array to allocate. If we wish to set that value to 200, then we could declare a container that can store up to 200 primes as:

```
typex {
     s = conceptual[slist[delflag[array[transient]]]];
}

container < prime_num_type > stored_as s with {
    array size is 200;
} prime_container;
```

In expression `t`, only the `odlist` layer has an annotation. This layer stores elements onto an ordered, doubly-linked list. The key or sort field is the annotation of `odlist`. Note that `odlist` appears twice in `t`, and each instance can have its own distinct key. To ensure that the proper annotation is associated with the proper instance of `odlist`, layer names are followed by a unique digit. Therefore a layer can appear at most 10 times in a type expression (there is a way around this limit in Section 5.5.3 [Automatic Repetition], page 21). Using the employee example, we can declare:

```
typex {
     t = conceptual[odlist1[odlist2[malloc[transient]]]];
}

container < emp_type > stored_as t with {
    odlist1 key is name;
    odlist2 key is age;
} emp_cont1;

cursor < emp_cont1 > orderby age emp_curs1;
cursor < emp_cont1 > orderby name emp_curs2;
```

5.5.3 Automatic Repetition

One more feature in type expressions is *automatic repetition*. If a layer takes a single annotation, and two annotations are given, then the layer is automatically repeated. If no annotations are given, then the layer is automatically deleted.

For instance, let us define type expression '`u`':

```
typex {
    u = conceptual[bintree[odlist[malloc[transient]]]];
}
```

The actual type expression will change with respect to these container declarations:

```
container < emp_type > stored_as u with {
    bintree key is name;
    odlist  key is age;
} emp_cont1;

container < emp_type > stored_as u with {
    bintree key is name;
    bintree key is age;
} emp_cont2;

container < emp_type > stored_as u with {
    odlist key is name;
    odlist key is age;
} emp_cont3;
```

In '`emp_cont1`', the annotations will preserve the type expression '`u`' as before. However, '`emp_cont2`' will cause the '`bintree`' layer to be duplicated and the '`odlist`' layer to disappear since there are two '`bintree`' annotations and no '`odlist`' annotations. Therefore the type expression is changed to the equivalent of: '`conceptual[bintree[bintree[malloc[transient]]]]`'. The last container declaration has no '`bintree`' annotations but two '`odlist`' annotations, therefore the type expression is equivalent to: '`conceptual[bintree[bintree[malloc[transient]]]]`'.

Automatic repetition can be combined with the method of distinguishing layer instances by their last digit, so declarations like the one below are legal:

```
typex {
    u = conceptual[bintree1[odlist[bintree2[malloc[transient]]]]];
}
container < emp_type > stored_as u with {
    bintree1 key is name;
    bintree1 key is age;
    bintree2 key is name;
} emp_cont1;
```

The above example means that for container '`emp_cont1`', the first '`bintree`' layer is duplicated, the '`odlist`' layer is deleted, and the second '`bintree`' layer appears only once.

5.6 Generic Containers/Cursors

A *generic container* is a proxy for a *concrete* (i.e. non-generic) container. The motivation for generic containers is to enable the defintion of procedures that operate over several containers that share the same element type.

The syntax for the declaration is:

```
generic_container_declaration: 'generic_container'
                '<' element_type_specifier '>'
```

Suppose we wish to write a `print_size` function which returns the number of elements in the container. Additionally, we have several different containers of emp_types; each with a different type expression. This means that the C struct type for each container would be different, thus causing one `print_size()` function to be written for each container type. This is awkward.

Generic containers were introduced to eliminate this problem. One `print_size()` function can be written, which takes *any* container of `prime_num_type` elements as an argument. Here is how such a container would be written:

```
container < emp_type > stored_as linked_list with {
} emp_cont

typedef generic_container < emp_type > GK;

void print_size( GK cont, char *name)
{
    printf("Size of container %s = %d",
        name, getsize(cont));
}

main()
{
    ...
    print_size((GK) &emp_cont,"Employee #1 Container");
}
```

will declare a type `GK` that can be used in the procedure `print_size`. This procedure will work for all containers based on the `emp_type`. The `getsize` procedure is one of P2's special container operations.

A *generic cursor* is a proxy for a concrete cursor in the same way as a generic container is a proxy for a concrete container. The syntax is similar as well:

```
generic_cursor_declaration: 'generic_cursor'
                              '<' element_type_specifier '>'
                            identifier_declarator_list ';'
```

Notice that generic cursors have neither a predicate nor an orderby clause. Also note that generic cursors are based on the element type and not on the container name like ordinary cursors.

Suppose we have two cursors for our prime container: one points to elements ending in '1' while the other one points to elements ending in '7'. Generic cursors allow us to write one procedure that will print out the full list of elements that are qualified by these cursors.

```
cursor < prime_container > where "($.num%10==1)" prime_one;
cursor < prime_container > where "($.num%10==7)" prime_seven;

typedef generic_cursor < prime_num_type > GC;

void print_primes(GC gc)
{
  foreach(gc)
    printf("%d\n",gc.num); // print prime number
}

main()
{
    ...
    print_primes ( (GC) &prime_one);
    print_primes ( (GC) &prime_seven);
}
```

The `print_primes` procedure will work for all cursors whose containers are of type `prime_num_type`, regardless of the concrete cursor's ordering or qualification.

5.7 Element Declaration

The element declaration will return the the transformed element type of the container or cursor name *k_or_c_name*. This is useful when we need to know the size of the element after it has been transformed by P2.

```
'element' '<' k_or_c_name '>'
```

For instance, if we want to find out the overhead for storing the prime number elements in the container prime_container, we can measure this using:

```
element < prime_cursor > e_prime;

int orig = sizeof(prime_num_type);
int new  = sizeof(e_prime);

printf("Size of prime_num_type:             %d\n",orig);
printf("Size of element in prime container: %d\n",new );
printf("Overhead for storing primes:        %d\n",new-orig);
```

In addition, if we want to assign values directly from the cursor objects, we can use 'element' to do this:

```
{
    ...
    element < emp_curs1 > *c1;
    c1 = emp_curs1.obj;
    printf("%s %d\n",c1->name, c1->age);
}
```

Here, c1 is declared as a pointer to the object type of emp_curs1. Once the assignment has been made in the second line, c1 can reference the fields of emp_type.

5.8 Comprehensive Example

Let us see how everything fits so far with an example involving the previous sections. The program is called 'prime.p2'.

```
// PART I

#define LIMIT 10

typedef struct {
    int num;
    } prime_num_type;
```

```
typex {
    ta = conceptual[sizeof[dlist[malloc[transient]]]];
    tb = conceptual[sizeof[odlist[array[transient]]]];
}

// PART II

container < prime_num_type > stored_as ta with {
} prime_cont1;

container < prime_num_type > stored_as tb with {
    odlist key is num;
    array size is LIMIT;
} prime_cont2;

typedef generic_container < prime_num_type > GK;

// PART III

cursor < prime_cont1 > orderby ascending num c11;
cursor < prime_cont1 > where "($.num%10) == 1" orderby ascending num c12;

cursor < prime_cont2 > orderby descending num c21;
cursor < prime_cont2 > where "($.num%10) == 1" orderby descending num c22;

typedef generic_cursor < prime_num_type > GC;

// PART IV

element < prime_cont1 > prime_cont1_type;
element < prime_cont2 > prime_cont2_type;

// PART V

int maxnum; // maximum number.

// PART VI

void print( GC gc, char *name )
{
    printf("\nContainer %s:\n",name);
    foreach(gc) {
        printf("%3d ",gc.num);
    }
    printf("\n");
}

void getsize_k( GK gk, char *title )
```

```
{   printf("size of %s container =  %d\n",title,getsize(gk)); }


void init_primes(GC gc)
{
    int i,j;
    int div;
    prime_num_type node;

    for(i=3;i<maxnum;i+=2) {
        div = 0;
        for(j=2;j<(i/2) && !div;j++)
            if ( i%j == 0 )
                div=1;
        if ( !div ) {
            node.num = i;
            insert(gc, node);
        }
    }
    node.num = 2;
    insert(gc, node);
}

// PART VII

main(void)
{
    open(prime_cont1);
    open(prime_cont2);

    init_curs(c11);
    init_curs(c12);
    init_curs(c21);
    init_curs(c22);

    printf("Enter maximum number: ");
    scanf("%d",&maxnum);

    init_primes((GC)&c11);

    // insert all primes ending in 1 into the 2nd container.
    foreach(c12) {
        prime_num_type p;
```

```
        if ( overflow(prime_cont2)) {
            printf("reached container 2 capacity\n");
            break;
        }
        p.num = c12.num;
        insert(c21,p);
    }

    printf("size of container #1 structure is: %d\n",
           sizeof(prime_cont1_type));
    printf("size of container #2 structure is: %d\n",
           sizeof(prime_cont2_type));
    printf("size of elements are: %d\n",sizeof(prime_num_type));


    print((GC)&c11, "#1");
    getsize_k((GK)&prime_cont1, "all primes");

    print((GC)&c22, "#2 - 1s only");
    getsize_k((GK)&prime_cont2, "primes ending in 1");
}
```

The first part of the program defines the prime element structure, which is just a single integer. Next comes the two type definitions. Both of these type expressions includes the layer `sizeof` because it is this layer that will define the `getsize` operation that will be used in `getsize_k`.

The second part defines two containers. The first one will be used to hold all the prime numbers while the second one will be used to hold all prime numbers ending in '`1`'. However, the second container is constrained to only hold 10 primes due to the '`array size is LIMIT`' statement. A generic container is defined and will be used in the `getsize_k` procedure.

The third part defines several cursors. The first two cursors belong to the first container and the second two belong to the second container. Notice that the second container has the values sorted in decreasing order. Also in these container definitions, the expression '`$.num % 10`' has to be written without spaces and within parentheses because it is an unstructured term (what made it unstructured was the arithmetic operation '`%`'). Finally a generic cursor is declared.

The fourth part defines two element types. The '`prime_cont1_type`' will be of type whatever the element of '`prime_cont1`' is.

The fifth part declares a global int called '`maxnum`'. This is the maximum value of the number to search for primes.

The sixth part declares three generic-based procedures. The procedure `print` will take any cursor belonging to a container and print out the list of primes that are accessible by the cursor. For instance, the cursor '`c12`' will print out only those primes that end in '`1`', even though it belongs to the container which stores all primes.

The `getsike_k` takes a container for an argument and prints out the size of the container.

The `init_primes` takes a container and inserts all primes up to but not including the maximum number. At the end, it inserts the number '`2`' into the container. Insertions are done via a record of the original type. This record, called `node`, is allocated one and the values are overwritten per iteration of the loop. Insertions work by copying the record fields and allocating new memory to store these fields.

The seventh part is the main program. The first eight lines initializes the containers and cursors and prompts for the maximum value to halt the search for primes.

To call the `init_primes` procedure to insert elements into the first container, the address of *any* cursor belonging to the first container is passed and it is cast to the generic procedure type. While the syntax may look clumsy, it works.

Once all the primes are inserted into the first container, we wish to insert all primes ending in '`1`' into the second container. The cursor `c12` points to the only numbers we want and is a member of the first container. Therefore this is the cursor to use in the '`foreach`' statement.

Within the '`foreach`' body, we make a test to see if there are already 10 elements in the second container with the operation '`overflow`'. If the value is true, then the capacity of the container has been reached meaning it is unable to accept more elements. Therefore, this condition will break the loop, just like any other C '`for`' loop. But as long as it is possible, the number from `c12` is copied to a record and that record is inserted into the second container.

The program next prints the size of some types and then the program prints both the content and the size of the two containers.

The execution of the program resulted in these output:

```
%prime
Enter maximum number: 100
size of container #1 structure is: 16
size of container #2 structure is: 16
```

```
size of elements are: 4

Container #1:
  2   3   5   7  11  13  17  19  23  29  31  37  41  43  47
  53  59  61  67  71  73  79  83  89  97
size of all primes container =  25

Container #2 - 1s only:
 71  61  41  31  11
size of primes ending in 1 container =  5
%
% prime
Enter maximum number: 250
reached container 2 capacity
size of container #1 structure is: 16
size of container #2 structure is: 16
size of elements are: 4

Container #1:
  2   3   5   7  11  13  17  19  23  29  31  37  41  43  47
 53  59  61  67  71  73  79  83  89  97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173 179 181 191 193 197
199 211 223 227 229 233 239 241
size of all primes container =  53

Container #2 - 1s only:
191 181 151 131 101  71  61  41  31  11
size of primes ending in 1 container =  10
```

Notice how in the second run, the second container reached its capacity. Also notice that the second container orders the numbers backward.

5.9 Type Expression Constraints

For a P2 system equipped with 40 layers, the number of type expressions consisting of exactly 5 layers are approximately (1*37^3*2), representing '`conceptual [` *3 layers* `[ (transient | persistent)]]`', which is over 100,000 expressions. There is one test suite case in P2 where a type expression is composed of 60 layers, so there are a tremendous number of combinations for type expressions.

However, not all type expressions make sense; only a very small subset does. Type expressions are constrained by three factors:

- Interface connections
- Semantic constraints
- Layer Ordering

5.9.1 Interface Constraints

For a type expression to have the correct interface connection, the interfaces imported by a layer must match the interfaces exported by its arguments. For example, the above type expression `t = conceptual[odlist[array[transient]]]` is syntactically correct, because:

- the parameter of '`conceptual`' is the same type as the interface exported by '`odlist`',
- the parameter of '`odlist`' is the same type as the interface exported by '`array`',
- the parameter of '`array`' is the same type as the interface exported by '`transient`',
- and the '`transient`' layer has no parameters.

P2 performs simple compositional transformations at compile time. Static composition trades the disadvantage that implementations cannot be altered (a la schema evolution) at run time, for the advantage that composition has zero runtime cost. P2 generates code on a per special operation (see Chapter 6 [P2 Operations], page 37) basis. Specifically, P2 composes code fragments contributed by each layer in the type expression in order down from the top of the type expression to the bottom, and then back up from the bottom to the top.

5.9.2 Semantic Constraints

Once a type expression has the proper interface connections, the next check is for proper semantic constraints. For example, one cannot perform a retrieval operation on a cursor that is bound to a container with type expression `conceptual[malloc[persistent]]`, since none of the components in this type expression implement retrieval operations. The purpose of P2's built-in design rule checker is to evaluate additional rules (beyond interface matching) to assure that the type equations are semantically meaningful. In addition, the built-in checker will explain in some detail what is wrong with an incorrect type expression.

For further details on the design rule checking algorithm, see section "Layer Composition Checks" in *Advanced P2 System Manual*.

5.9.3 Ordering Constraints

Finally, a type expression that satisfies the semantic constraints may work, but the ordering of the layers in the expression may be crucial. The placement of a layer in a type expression can have a tremendous effect on the algorithms (and consequently, the running times). Consider the two type expressions below:

```
typex {
    t = conceptual[delflag[odlist[array[transient]]]];
    s = conceptual[odlist[delflag[array[transient]]]];
}
```

One of the routines that is affected is the `delete` operation. This operation in '`delflag`' marks the element and does not execute the operation in the next layer in the type expression. So in expression '`t`', the `delete` operation for '`odlist`' is not called whereas in '`s`' it is.

So how important is this? Well, it depends on the cost for testing every element in every operation to determine if it has been deleted versus the cost for doing the deletion on an element.

5.10 Composite Cursors

Complex data structures consist of multiple containers whose elements are interconnected by pointers. A relationship among containers C*1*, C*2*, ..., C*n* is a set of n-tuples <e*1*, e*2*, , e*n*> where element e*i* is a member of container C*i*. Let us assume that for three containers A, B, and C, the relationship among containers over specific elements are:

```
{(a3,b1,c1), (a3,b1,c3), (a1,b2,c4), (a2,b3,c2), (a2,b3,c4)}
```

A *composite cursor* enumerates the n-tuples of a relationship. More specifically, a composite cursor '`k`' is an n-tuple of cursors, one cursor per container of a relationship. A particular n-tuple <e*1*, e*2*, ..., e*n*> of a relationship is encoded by having the *i*th cursor of '`k`' positioned on element e*i*. By advancing '`k`', successive n-tuples of a relationship are retrieved.

A composite cursor is declared with a list of cursor-container pairs which are the name of the cursor internal to this one and the name of the container to range over. A *given* clause specifies which of the internal cursors is already bound, a predicate restricts the elements satisfying the tuple, and a *valid* clause tests the next tuple before advances. A composite cursor has the syntactical form:

```
composite-cursor-declaration :  'compcurs'
    '<' cursor-container-pair-list '>'
    [ 'given' '<' cursor-list '>' ]
    [ 'where' predicate ]
    [ 'valid' valid-string ]
cursor-container-pair-list : cc-pair
    | cursor-container-pair-list ',' cc-pair
cc-pair : internal_cursor_name container_name
```

There are two new operations which are special to composite cursors. *initk(c_curs)* is the operation which initializes composite cursors and *foreachk(c_curs)* is the looping construct for composite cursors.

The type expression of the containers used in composite cursors require a link layer to implement the joins between containers. If a link layer is not declared, the nested loop link layer ('`nloops`') will be automatically provided by P2. This example declares a composite cursor `goldbach` to determine prime numbers that differ by 2.

```
prime_list = top2ds[inbetween[qualify[delflag[array[transient]]]]];

container < prime_num_type > stored_as prime_list with {
    array size is 200;
} prime_container;

compcurs < a prime_container, b prime_container >
    where "($a.num+2==$b.num)"
    goldbach;
```

The composite cursor '`goldbach`' is derived from two internal cursors, '`a`' and '`b`', both belonging to the same container, '`prime_container`'. The where clause is an unstructured predicate because it uses the '`+`' operation. Therefore, it is expressed without spaces and in parentheses.

The main program can be written as:

```
main()
{

    ... // Initialize the container with primes

    initk(goldbach);
    foreachk(goldbach) {
        printf("(%d, %d) ",goldbach.a.num, goldbach.b.num);
    }
}
```

After 'prime_container' has been filled with primes (generated by a procedure not shown), the composite cursor 'goldbach' is initialized and the looping construct begins. The output will look like this:

```
(3, 5) (5, 7) (11, 13) (17, 19) ...
```

The next example uses the given clause. Assuming that the first element of the composite cursor is already bound to a prime, find all the primes such that they are within 10 of the first number. This is used in conjunction with a cursor find_num positioned on an element in the prime container.

```
cursor < prime_container > where "$.num == input_value" find_num;

compcurs < a prime_container, b prime_container >
    given < a >
    where "($a.num+10>=$b.num) && ($a.num-10<=$b.num)"
    range;

main()
{
    ...
    init_curs(find_num);
    initk(range);

    printf("Enter a prime number:");
    scanf("%d",&input_value);

    // position the find_num cursor on <input_value> in the container
    reset_start(find_num);

    // position the internal cursor a to point to find_num.
    range.a = find_num;

    printf("%d: ",range.a.num);
    foreachk(range) {
        printf("%d ", range.b.num);
    }

}
```

Because the internal cursor 'a' is used in the given clause, the program must set 'a' to some specific value before the 'foreachk' operation. This is because the 'foreachk' operation understands that cursors in the given clause are already bound to some value and it will try make some optimizations based on that information. If the user entered the number '17', then the output would be:

```
17: 7 11 13 17 19 23
```

Now before we demonstrate an example using the `valid` clause, we need to discuss its operation in detail.

On every iteration of a '`foreachk`', a new n-tuple of elements will be produced. If no element is updated or deleted, things remain simple. However, if an element of an n-tuple is updated, then the next n-tuple that is to be retrieved may be different from the n-tuple that would have been retrieved had there been no update.

Suppose there is a container of employees and a container of departments. The composite cursor below defines ordered pairs (i.e., 2-tuples) of department and employee objects that are related (by sharing the same '`deptno`' value):

```
compcurs < d department_container, e employee_container >
where "$d.deptno == $e.deptno" cc;

foreachk( cc )
    {   printf("(%s, %s)\n", cc.d.name, cc.e.name );   }
```

What the above '`foreachk`' does is to loop over each (department, employee) pair that satisfy the join predicate "$d.deptno == $e.deptno". Suppose this sequences of ordered pairs is:

```
(d1,e1), (d1,e2), (d1,e3), (d2,e4), (d2, e5), (d3, e6)
```

Now suppose the following '`foreachk()`' is executed:

```
foreachk( cc ) { delete( cc.d ); }
```

Note that the department object '`d1`' is first to be deleted; this impacts the sequence of ordered pairs in the following way:

```
(d1,e1), (d2,e4), (d3,e6)
```

Note that the tuples '`(d1,e2), (d1,e3), (d2,e5)`' are not produced. The reason is that that once d1 is deleted, all subsequent tuples in which it appeared should be deleted as well.

The main problem is that modifications to elements of an n-tuple identified by a composite cursor will alter the sequence of n-tuples produced. A valid query is a predicate which is used to

determine the validity of elements of an n-tuple. This predicate is generally *not* the selection predicate; rather, it is a predicate that merely tests to see if an element has been modified. If the valid predicate is false, then a new n-tuple will need to be generated on a composite cursor advancement that *does not* include the element whose valid test has failed.

Generally, valid predicates are usually limited to testing for deletions (e.g., !is_deleted($e)) or to updates of join fields (e.g., join_field_valid(&$d,&$e)). Our composite cursor should now look like this:

```
compcurs < d department_container, e employee_container >
    where "$d.deptno == $e.deptno"
    valid "!is_deleted($d)" cc;
```

6 P2 Operations

This is the list of operations understood by P2. An operation is *cursor-based* if the first argument is a cursor. An operation is *container-based* if the first argument is a container.

6.1 Container Operations

open (*container*) Function
close (*container*) Function

The `open` procedure performs two tasks:

- Creates the record for the container if this has not already been done. (Persistent containers would create such a record on open time).
- Initializes the container if it has not already been done.

The `close` procedure is the counterpart to open. It destroys the container. Most of the current layers do not do anything for `close`, but `persistent` does.

overflow (*container*) Function

This operation returns a conditional expression representing the test for a completely filled container. This operation is required only in layers, such as array, which allocate a fixed number of elements.

getsize (*container*) Function

This operation returns the size of the container. This operation may not be provided by all layers.

6.2 Cursor Operations

6.2.1 Element Retrieval Operations

A retrieval operation is one that either moves the cursor over the elements in a container or determines if there are no more elements in the container.

reset_start (*cursor*) Function
reset_end (*cursor*) Function

The `reset_start` (`reset_end`) operation positions the cursor on the first (last) object in the container. If there are no elements, the operation will point to a value defined in the component (usually '`NULL`').

adv (*cursor*) Function
rev (*cursor*) Function

The `adv` (`rev`) operation moves the cursor to the next (previous) object . If there is no next (previous) record, `cursor.obj` is set to some layer-defined value so that the `end_adv` (`end_rev`) function can recognize that no more advances (reverses) can be made.

end_adv (*cursor*) Function
end_rev (*cursor*) Function

The `end_adv` (`end_rev`) operation generates a boolean expression which determines whether or not the cursor has been advanced (reversed) past the end of the container.

foreach (*cursor*) { *code* } Function
rofeach (*cursor*) { *code* } Function

These two operations are looping constructs. These operation will move the cursor to the first (last) element in the container, execute the body of code, and iterate over the collection forwards (backwards) until the cursor reaches the end (beginning) of the container.

If a cursor is qualified to range over a subset of the elements, then these operations only apply to the qualified elements. For instance, if the cursor is restricted to point to all primes ending in '`1`', then `reset_start` will move the cursor to the first element ending in '`1`', `adv` will move the cursor to the next element ending in '`1`', and so on.

6.2.2 Element Update Operations

delete (*cursor*) Function

This operation removes the element on which the cursor is positioned. Ideally, there should be two delete operations: both delete a record, but one positions the cursor for subsequent advancing, and the other for subsequent reversing. Currently, the semantics

of delete is the former. However, calling `delete` within a `rofeach` will do the latter, which is correct.

insert (*cursor, record*) Function

This operation adds a new element into the cursor's container. If the container maintains an ordering, the operation will place it in the proper position.

upd (*cursor, field, expr*) Function
cursor.field = expr Function

This is the update function, which is equivalent to the expression `cursor.field = expr`. Some ordered layers will generate an error if the operation is performed over the ordering field. This is a result of the famous "Halloween problem"[1]. See '`bintree.xp`' layers for an example.

6.2.3 Composite Cursor Operations

initk (*compcurs*) Function

This operation initializes composite cursors.

foreachk (*compcurs*) `{` *code* `}` Function

This looping construct is similar to the `foreach` operation except the argument is a composite cursor and iterates over each composite tuple.

6.2.4 Miscellaneous Cursor Operations

getrec (*cursor, record*) Function

This operation copies the data from the cursor into the record variable.

[1] The *Halloween Problem* arises when the list of elements that are being updated must be kept in order. Suppose we have a collection '`10 20 30`' and we wish to add '`30`' to each element, while still maintaining the order. After changing '`10`' to '`40`', the list looks like this: '`20 30 40`'. Now '`30`' is no longer the last element in the collection, and in fact, this process will not terminate as the three elements will be continuously updated. P2 will catach the error at runtime (actually, it should be caught at compile-time).

swap (*cursor0, cursor1*) Function

This operation swaps the elements referenced by the two cursor arguments. Only layers that do not maintain an ordering can implement this operation. All layers maintaining an order report an error at code generation time.

init_curs (*cursor*) Function

This operation initializes the cursor fields. This operation should be called before any other operation is performed on the cursor.

gettime (*cursor*) Function

This operation returns the timestamp of the element pointed at by the cursor. This operation may not be provided by all layers.

is_deleted (*cursor*) Function

This operations returns true if the current element the cursor is pointing to has been marked as deleted. This operation may not be provided by all layers.

7 P2 Layers

7.1 Layer Format

The P2 layers are *vertically parameterized*. Each component imports zero or more standardized interfaces and exports a standardized interface. A standardized interface is one of these five types called a *realm*:

'`ds`'	data structures realm - usually things like linked lists, arrays, binary trees.
'`top`'	the conceptual layer - the topmost layers. Usually this is the layer that defines `foreach` and `rofeach` in terms of other operations.
'`mem`'	memory allocation - the layers responsible for memory allocation.
'`lnk`'	link layers - layers to do link processing.
'`toplnk`'	conceptual layer for links - the topmost layer in the link sub-realm.

This is the order that they appear in the file '`op-tab.h`'. This is important if we wish to add another operation or another realm.

Below is the list of layer options to describe certain characteristics of the layer. Through the layer options, the syntactic complexity of the xp file is reduced.

no annotation Layer option
annotation Layer option
optional annotation Layer option
multiple annotation Layer option
optional multiple annotations Layer option

These mutually exclusive options describe the number of annotations a layer expects. They consecutively represent: zero, one, zero or one, one or more, zero or more. If none of these options are present, then "no annotation" is assumed.

stable Layer option
unstable Layer option

These mutually exclusive options describe the relationship between the delete operation and the current cursor. A `stable` option will not move the cursor after performing a

delete operation. An `unstable` option will move the cursor to the first element after the deleted one.
If neither option is present, then "stable" is assumed.

retrieval_always Layer option
retrieval_never Layer option
retrieval_sometimes Layer option

These three mutually exclusive options describe the status of the retrieval operations (`adv`, `rev`, `reset_start`, etc.) in the layer. A `retrieval_always` option means the retrieval operations are always executed, even if the layer is not chosen as the retrieval layer. A `retrieval_never` option means that retrieval options are not present in this layer. A `retrieval_sometimes` option means that the retrieval operations are performed only if the layer is chosen as the retrieval layer.
If none of these options are present, then "retrieval_never" is assumed.

curs_state Layer option
cont_state Layer option

These options describes if the layer needs to maintain cursor state and/or container state information.

d2u Layer option

This option means that the layer implements the delete operation by using updates. For instance, the '`delflag`' layer "deletes" an element by updating the element's "deleted-field" from 0 to 1.

indirect_only Layer option

This option is presently used to indicate whether a layer provides persistent storage or not. The name comes from the fact that persistent containers cannot be declared directly, but only indirectly.

To recap, a layer without any options expects no annotations, does not move the cursor on deletions, has no retrieval operations, does not maintain cursor or container state information, does not map deletions onto updates, and can declare containers directly.

Finally, to interpret the layer definition:

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

the format is:

- The realm of the layer. (‘`ds`’)
- The name of the layer. (‘`array`’)
- The parameters of the layer surrounded by ‘’`[` ’`]`’’. There may be more than one parameter. Each parameter is the the name of a realm, which can be instantiated with a layer from that realm. (‘`[ mem ]`’)
- The remaining elements are the layer options that were discussed in the previous subsection. (‘`stable`’ ‘`annotation`’ ‘`retrieval_sometimes`’)

7.2 P2 Layer Specifications

These are the layers implemented in P2 so far. New layers are likely to be developed, so this list may change.

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

Annotation: `array size is` *size*

Array allocates a linear array of *size* elements. Space occupied by deleted elements is not reused. An error is raised if there is an attempt to add more than *size* elements to the container.

`ds` **avail** [`ds`] `stable no annotation d2u retrieval_never` layer

Avail keeps a list of all elements that have been deleted so that the space can be reused for subsequent insertions.

`ds` **avl** [`ds`] `unstable annotation curs_state retrieval_sometimes` layer

Annotation: `avl key is` *field*.

This layer implements AVL trees ordered on the field *field*. AVL trees are height-balanced binary trees, meaning the maximum height for the tree with N nodes is log2(N).

`ds` **bintree** [`ds`] `unstable annotation curs_state retrieval_sometimes` layer

Annotation: `bintree key is` *field*

Binary tree ordered on key *field*.

`top` **ccbus** [`top`] `stable no annotation retrieval_always` layer

This layer is only used internally by P2 to link containers of different implementations. This layer will first re-route the type expression based on the information in the container (which has references to all the lower-level type expressions).

`ds` **delflag** [`ds`] `stable retrieval_always no annotation d2u` layer

Delflag marks deleted elements instead of actually deleting them. Most often used on top of array.

`top` **conceptual** [`ds`] layer

This layer does not really exist. It is a layer name that is understood by P2 to expand into a series of layers. The current definition of conceptual is:

```
conceptual[X] ==
    top2ds[init_generic[generic[generic_funcall[orderby[
        inbetween[qualify[X]],
        top2ds[inbetween[dlist[malloc[transient]]]]
        ]]]]]
```

`ds` **deque_dlist** [`ds`] `stable no annotation retrieval_sometimes` layer

deque_deque_dlist is an ugly hack of '`dlist`'; it assumes a global integer variable "ugly_hack", which has the values zero and nonzero. A zero value means that records are inserted at the head of the dlist. A nonzero value means that records are inserted at the tail of the dlist. Note: "ugly_hack" is reset to zero upon every insertion.

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

Doubly-linked list.

`mem` **fasttransient** [ ] `stable no annotations` layer

Like '`transient`', but it calls `fastmalloc()`, (which is built into the P2 runtime system) for faster memory allocations.

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

This layer will reroute operations from being inline to calling a procedure through the operation vector of a container (or a cursor) if the container is declared as a generic container. If the container is not a generic container, then this layer will not affect the operations. The layer '`init_generic`' has to come after '`generic`' in the type expression.

`ds` **generic_funcall** [`ds`] `stable` `curs_state` `no annotation` `retrieval_always` layer

If the operations are marked as "function expansion", then the calls to the operations are made through the operation vector of the container (cursor). This layer is remarkably similar to the `generic` layer. This layer usually comes after `generic`.

`ds` **hash** [`ds`] `unstable` `curs_state` `annotation` `retrieval_sometimes` layer

Annotation: `hash key is` *fieldname* `with size` *size*

Performs hashing. The number of buckets is *size* and the field to hash on is *fieldname*.

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

Annotation: `hashcmp field` *field*.

String equality comparisons are slow operations. This layer will speed that up by storing the hash value of a string field alongside the string field. Equality tests between strings are transformed into an equality test between two hash values—only if the values are equal will the string comparison be performed.

`ds` **hlist** [`ds`] `stable` `curs_state` `annotation` `retrieval_sometimes` layer

Annotation: `hlist timestamp is` *timestamp˙field* `key is` *field* `with size` *hashsize*.

This layer implements a time-stamp ordered hash-list. Timestamps are assigned increasing values and stored in the field *timestamp_field*. Inserted elements are placed at the head of the list. Updates are modelled as insertions followed by deletions. The size of the hash table is *hashsize*. The hashing field is *field*.

Note: this layer was used in the implementation of LEAPS. It is not clear if it has any other usage.

`ds` **hpredindx** [`ds`] `stable annotation retrieval_sometimes curs_state` layer

Annotation: `hpredindx timestamp is` *timestamp˙field* `key is` *field* `with size` *hash˙size* `predicate` *pred* `with` *empty˙proc* `with` *nonempty˙proc*

This is a timestamp ordered container of qualified elements that are hashed into an array of buckets. The timestamp field is *timestamp_field*. The key to hash the elements on is *field* and the size of the hash array is *hash_size*. The predicate *pred* is used to only allow all elements satisfying a particular predicate to be in this container. The procedure *empty_proc* is called when the last element is deleted from the container whereas the procedure *nonempty_proc* is called when the first element is added to the container. Either one or both procedures can be the string "null" which means no function is called.

Note: this layer was used in the implementation of LEAPS. It is not clear if it has any other usage.

`ds` **inbetween** [`ds`] `stable curs_state no annotation retrieval_always` layer

This layer is used to point the cursor to the next object after a deletion for unstable layers. This layer must be used if there are unstable layers in the type expression and this layer must be above all the unstable layers.

`ds` **init_generic** [`ds`] `stable no annotation retrieval_never` layer

This layer initializes the operation vector (of the list of procedures) associated with a container or a cursor. This layer comes after the `generic` layer.

`lnk` **linkterm** [`top`] `stable no annotation retrieval_never` layer

The bottommost link layer. The transition from the link realm to the top realm.

`ds` **llist** [`ds`] `stable curs_state annotation retrieval_sometimes` layer

Annotation: `llist timestamp is` *timestamp˙field*.

This layer implements a time-stamp ordered hash-list. Timestamps are assigned increasing values and stored in the field *timestamp_field*. Inserted elements are placed at the head of the list. Updates are modelled as insertions followed by deletions.

Note: this layer was used in the implementation of LEAPS. It is not clear if it has any other usage.

`ds` **lpredindx** [`ds`] `stable annotation retrieval_sometimes curs_state` layer

Annotation: `lpredindx timestamp is` *timestamp˙field* `predicate` *pred* `with` *empty˙proc* `with` *nonempty˙proc*

This is a timestamp ordered list of predicate qualified elements. The timestamp field is *timestamp_field*. The predicate *pred* is used to only allow all elements satisfying a particular predicate to be in this container. The procedure *empty_proc* is called when the last element is deleted from the container whereas the procedure *nonempty_proc* is called when the first element is added to the container. Either one or both procedures can be the string "null" which means no function is called.

Note: this layer was used in the implementation of LEAPS. It is not clear if it has any other usage.

`ds` **malloc** [`mem`] `stable no annotations` layer

Allocates space dynamically.

`ds` **mlist** [`ds, top`] `unstable curs_state cont_state multiple annotation retrieval_sometimes` layer

Annotation: `mlist on` *fieldname*.

This is the multi-list indexing layer. This layer accepts any number of fields and the container is indexed over all these fields. The first parameter is the type expression of how the elements are stored and the second parameter is the type expression of how the index objects are stored.

Note that this layer accepts multiple annotations. That is, several indices can be performed over the same container and handled by only one invocation of the mlist layer.

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

Annotation: `multimalloc size is` *size*.
If no annotation is specified, the default value of *size* is 100.

This layer works like malloc, but it allocates *size* objects at once and keeps track of the next available location via caching.

`ds` **named_funcall** [`ds`] `stable` `curs_state` `no` `annotation` `retrieval_always` layer

A type expression without `named_funcall` inlines the code at the point of invocation. With this layer, a procedure is generated (based on the name of the container) and the invocation has been replaced by a procedure call. These are done if the operations are tagged as "function expansion".
Unlike the `generic_funcall` layer, there is no operation vector associated with the container (cursor).

`lnk` **nloops** [`lnk`] `stable` `retrieval_never` `optional` `multiple` `annotations` layer

Annotation: `nloop` `link` *linkname* `on` *pcard p˙k˙name* `to` *ccard c˙k˙name* `where` *link˙pred*.
Note that the annotation is optional, in which case the default would be used.

This layer implements link traversals as a series of nested loops. The name of the link is *linkname*. The predicate for the link, called *link_pred*, determines how the parent elements and child elements are connected. The names of the parent and child containers are *p_k_name* and *c_k_name*, respectively. The cardinality relationship between the parent and the child is *pcard*:*ccard*.

`ds` **odlist** [`ds`] `unstable` `annotation` `retrieval_sometimes` layer

Doubly-linked list ordered by field *field*.

Annotation: `odlist` `key` `is` *field*.

`ds` **orderby** [`ds` , `top`] `stable` `curs_state` `no` `annotation` `retrieval_always` layer

This layer is needed if cursors with orderby clauses are defined. This layer is used when a cursor is ordered by a field which no other layer uses for ordering. The first parameter is the continuation of the type expression of the base type. The second parameter is the type expression of the container of pointers which maintain the ordering specified in the cursor declaration.

`ds` **qualify** [`ds`] `stable` `curs_state` `no` `annotation` `retrieval_always` layer

This layer modifies retrieval operations to advance to the next qualified object. The qualification is determined by the cursor predicate.

`ds` **part** [`top, top`] `stable curs_state cont_state annotation retrieval_always` layer

Annotation: `part at field` *fieldname*.

This layer partitions the element into two structures. All fields in the original element data type which appear before (and including) *fieldname* are added to the second data structure and all other types are listed in the first data structure.

`mem` **persistent** [ ] `stable indirect_only` layer

Annotation: `persistent file is` *filename* `with size` *size*.

This is a layer where the memory is mapped to disk. This layer is highly machine-dependent. For instance, it does not work on the ULTRIX, but it does work on the SunOS. The file to do the mapping is called *filename* and the maximum character size of the file is *size*.

`ds` **predindx** [`ds`] `annotation curs_state retrieval_sometimes` layer

Annotation: `predindx predicate` *pred*.

This layer maintains a list of all elements satisfying predicate *pred* in change order (inserted and updated elements are placed at the head of the list).

`ds` **qsort** [`mem`] `stable annotation retrieval_sometimes` layer

Annotation: `qsort key is` *field* `with size` *size*.

This layer maintains the elements in a fixed size array (at most *size* elements) ordered by the field *field*. Actually, the array is in sorted order only after a call to `reset_start` or `reset_end`. An error is raised if there is an attempt to add more than *size* elements to the container.

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

This layer adds the adhoc operation *sizeof* which returns the number of elements in the container.

`ds` **slist** [`ds`] `unstable no annotation retrieval_sometimes` layer

This is the unordered singly-linked list layer.

`ds` **slow_hash** [`ds`] `unstable annotation retrieval_sometimes` layer

Annotation: `slow_hash key is` *field* `with size` *size*.

This layer implements a hash function over *size* buckets on the field *field*. This is a slow version because the current bucket is recomputed for each operation (instead of '`hash`' which attempts to cache the value of the current bucket.

`ds` **timestamp** [`ds`] `stable annotation` layer

Annotation: `timestamp on` *counter*

This layer adds a field *counter* to the element type which the main program has to initialize. This field is incremented for each insert and update. This layer also adds the adhoc operation `gettime` which returns the value of the timestamp of the element of the current cursor.

`ds` **tlist** [`ds`] `stable annotation retrieval_sometimes` layer

Annotation: `tlist key is` *field*.

This layer maintains a list of elements in timestamp ordering using the field *field*.

`top` **top2ds** [`ds`] `stable no annotation retrieval_never` layer

This is the interface between the top realm and the data structure realm. The definitions of `foreach` and `rofeach` are specified here.

`top` **top2ds_qualify** [`ds`] `stable no annotation retrieval_always` layer

The union between '`top2ds`' and '`qualify`' layers.

`toplnk` **top2link** [`lnk`] `stable no annotation retrieval_never` layer

The layer which sits above all link layers. It gathers information such as which layer will process the link.

`ds` **tpredindx** [`ds`] `stable annotation retrieval_sometimes curs_state` layer

Annotation: `tpredindx predicate` *pred* `with` *empty˙proc* `with` *nonempty˙proc*

This is a timestamp ordered list of predicate qualified elements. The predicate *pred* is used to only allow all elements satisfying a particular predicate to be in this list. The procedure *empty_proc* is called when the last element is deleted from the list whereas

the procedure *nonempty_proc* is called when the first element is added to the list. Either one or both procedures can be the string "null" which means no function is called.

Note: this layer was used in the implementation of LEAPS. It is not clear if it has any other usage.

`mem` **transient** [ ] `stable no annotations` layer

This layer stores elements in memory.

`ds` **vtimestamp** [`ds`] `stable annotation` layer

Annotation: `vtimestamp field is` *field* `counter is` *countername*.

This layer updates the timestamp field *field*, which is already defined in the base type, with the value *countername*, which is already defined in the main program.

8 Invoking P2

8.1 Writing P2 programs

Writing a P2 program is like writing a C program, but there are differences. We will look at '`sample.p2`' bit by bit to analyze what's going on and why are things done that way. The program reads in employee data into a container and displays them if the meet some propery. The full program is in Appendix A [Example P2 program], page 67.

8.1.1 p2.sample - Declaring Types

The first part of the file is where the types are declared. Here we declare an employee structure, `E`, which contains an employee number, age, temperature, department name, and finally, the employee name.

```
#include <stdio.h>

// Element.

typedef struct {
  int   empno;
  int   age;
  float temp;
  char  *dept_name;
  char  name[20];
} E;
```

8.1.2 p2.sample - Containers

This portion of the code declares the containers and the types they are based on. Container `k` orders the elements by age and allocates space for only 10 elements: attempts to insert an 11th element will result in a overflow warning. The other container, `pk`, allocates 1000 bytes in the file '`~/foo`' for persistent storage. This container orders elements both by age and by name.

```
// Type expressions.

typex {
  p = conceptual[odlist1[odlist2[malloc[persistent]]]];
  t = conceptual[odlist[delflag[array[transient]]]];
}

// Containers.

container <E> stored_as t with {
  odlist key is age;
  array size is 10;
} k;

container <E> stored_as p with {
  odlist1 key is age;
  odlist2 key is name;
  persistent file is "~/foo" with size 1000;
} pk;
```

8.1.3 p2.sample - Cursor Declarations

This example shows that cursor `c` and pointer to cursor `pc` are declared over the container `k`. Both cursors will select only elements where the temperature field is greater or equal to 98.6. Both cursors will also retrieve elements in alphabetical order over the element's name field.

```
// Cursors.

cursor <k>
  where "$.temp >= 98.6"  // Predicate.
  orderby ascending name  // Orderby clause.
c,                        // c is a cursor variable.
*pc;                      // pc is a pointer to cursor
```

In the second cursor example below, the cursor structure is used as part of a `typedef`. The third example below shows that cursors can match exactly those elements with the department name "Computer Sciences". The last cursor portion is the declaration of generic cursor, and its use in typedefs. Generic cursors have neither an ordering nor a predicate. See Section 5.6 [Generic Containers/Cursors], page 23.

```
typedef cursor <k>
  where "$.temp >= 98.6"
  orderby ascending name
C;
C v;

cursor <k>
  where "$.dept_name == 'Computer Sciences'"
  orderby descending empno
cs;

// Generic cursors.

generic_cursor <E> gc;
typedef generic_cursor <E> GC;
GC gv;
```

8.1.4 p2.sample - Functions and Data

The macro `F` iterates over a cursor `X`, which will point to successive elements in the container. The next function `f` uses specific cursors, whereas the last function `gf` uses generic cursors.

```
// Function body.

#define F( X ) \
{ \
  foreach( X ) { \
    printf( "%d, %d, %.1f, \"%s\", \"%s\"\n", \
      X.empno, X.age, X.temp, X.dept_name, X.name ); \
  } \
}

// Function with a non-generic formal parameter.

int f( C x )
{
  F( x )
}

// Function with a generic formal parameter.

int gf( GC gx )
{
  F( gx )
}
```

The employee data used in the program.

```
// Employee data.

E rawdata[] = {
   { 10000, 60, 99.5,           "English",    "Akers, Mark" },
   { 10070, 22, 99.4,           "Physics",   "Andrews, Kay" },
   { 10020, 18, 99.0,           "History",     "Aaron, Bob" },
   { 10040, 42, 98.5, "Computer Sciences", "Singhal, Vivek" },
   { 10010, 40, 98.7, "Computer Sciences",    "Batory, Don" },
   { 10040, 53, 96.3,        "Accounting",  "Akerson, Mary" },
   { 10060, 65, 98.8,         "Nutrition", "Zacks, William" },
   { 10050, 23, 96.1, "Computer Sciences",   "Thomas, Jeff" },
   { 10080, 31, 98.7,     "Culinary Arts",   "Geraci, Bart" },
   { -1 }
};
```

8.1.5 p2.sample - Main Program

First, the container `k` is opened, two cursors are initialized, and the elements in the data are inserted into the container. And any cursor, qualified or not, can be used for the `insert` operation.

```
// Main.

main()
{
  int i;
  E   *e;

  open( k );
  init_curs( c );
  init_curs( cs );

  for (i=0, e=rawdata; e->empno != -1; i++, e++) {
    insert( c, *e );
  }
```

These are the examples of legal and illegal assignments. Recall that `f(c)` will print all those elements that cursor `c` can point to, namely, elements where the temp >= 98.6. In addition, the elements will be printed in alphabetical order, based on name.

```
  // You may pass c as a actual to formal c
  // and assign c to temporary cursor variable v:

  printf( "f( c ):\n" );
  f( c ); // Legal.
  v = c;  // Legal.

  // You may not pass cs as an actual to formal c,
  // nor assign cs to temporary cursor variable v:

#if 0
  printf( "\n" );
  printf( "f( cs ):\n" );

  f( cs ); // Not legal.
  v = cs;  // Not legal.
#endif
```

These are more example of legal assignments. The code `gf( (GC) &c );` demonstrates that a generic cursor can take on any cursor and therefore procedures such as `gf()` can be written to apply to any cursor.

```
  // You may pass c and cs as actuals to generic formal gx,
  // and assign c and cs to generic temporary cursor variable gv:

  printf( "\n" );
  printf( "gf( c ):\n" );

  gf( (GC) &c );  // Legal.

  printf( "\n" );
  printf( "gf( cs ):\n" );

  gf( (GC) &cs ); // Legal.

  gv = (GC) &c;   // Legal.
  gv = (GC) &cs;  // Legal.

  close( k );
  exit( 0 );
}
```

8.2 Executing P2

Just type `P2` *filename*`.p2` like in the example below.

```
ahhnold% P2 sample.p2
liner ... done
ddl ... done
pb ... done
cat ... done
deliner ... done
compile ... done
link ... done
clean-up ... done
ahhnold% sample

f( c ):
10020, 18, 99.0, "History", "Aaron, Bob"
10000, 60, 99.5, "English", "Akers, Mark"
10070, 22, 99.4, "Physics", "Andrews, Kay"
10010, 40, 98.7, "Computer Sciences", "Batory, Don"
10060, 65, 98.8, "Nutrition", "Zacks, William"
10080, 31, 98.7, "Culinary Arts", "Geraci, Bart" },

gf( c ):
10020, 18, 99.0, "History", "Aaron, Bob"
10000, 60, 99.5, "English", "Akers, Mark"
10070, 22, 99.4, "Physics", "Andrews, Kay"
10010, 40, 98.7, "Computer Sciences", "Batory, Don"
10060, 65, 98.8, "Nutrition", "Zacks, William"
10080, 31, 98.7, "Culinary Arts", "Geraci, Bart"

gf( cs ):
10050, 23, 96.1, "Computer Sciences", "Thomas, Jeff"
10040, 42, 98.5, "Computer Sciences", "Singhal, Vivek"
10010, 40, 98.7, "Computer Sciences", "Batory, Don"
```

8.3 P2 options

These are the options that can be set for running P2. The environment variable `P2_FLAGS` is read for option values before the command line is read (i.e. later values of flags overwrite earlier values).

```
‘P2’ [ P2_options ] filename‘.p2’
```

'`--help`' P2 Option

Print the list of options and exit.

'`--[no-]keep`' P2 Option

Do [not] keep intermediate files. Not keeping the intermediate files will save disk storage space.

'`--[no-]drc`' P2 Option

Do [not] perform design rule checking.

'`--[no-]verbose`' P2 Option

Do [not] print stages of processing. If '`--no-verbose`' is selected, then the only thing the system will display is:

```
ahhnold% P2 mumble
ahhnold%
```

'`--[no-]lines`' P2 Option

Do [not] keep original line numbers. If '`--lines`' is chosen, then errors caught by the C compiler will refer to the original '`mumble.p2`' line numbers. If '`--no-lines`' is chosen, the errors caught by the C compiler will refer to the '`mumble.c`' generated code.

'`--[no-]indent`' P2 Option

Do [not] indent the '`mumble.c`' generated code.

'`--libdir =` *directory*' P2 Option

Specify explicitly the location of the directory containing '`libp2.a`', '`ddl`', and '`pb`'.

'`--datadir =` *directory*' P2 Option

Location of .h files: '`persist.h`' and '`p2.h`'.

'`--cflags =` *list*' P2 Option

Set the C compiler flags to *list*.

'`--cppflags =` *list*' P2 Option

Set the C preprocessor flags to *list*.

‘`--lflags =` *list*’ P2 Option

Set the linker flags to *list*.

The default flags are: ‘`--keep`’, ‘`--verbose`’, ‘`--lines`’, ‘`--indent`’, ‘`--drc`’.

9 P2 Bibliography

These are a few of the papers which describe the P2 system.

- D. Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas, "Scalable Software Libraries", *Proceedings of ACM SIGSOFT '93: Symposium on the Foundations of Software Engineering*, Los Angeles, California, 7-10 December, 1993.
- Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. "Achieving Reuse with Software System Generators", *IEEE Software*, September 1994.
- M. Sirkin, D. Batory, and V. singhal, "Software Components in a Data Structure Precompiler", *Proc. 15th Internnational Conference on Software Engineering*, May 1993.
- M. Sirkin, *A Software System Generator for Data Structures*, Ph.D. dissertation, Department of Computer Science, University of Washington, Seattle, Washington, 1994.
- D. Batory and S. O'Malley. "The Design and Implementation of Hierarchical Software Systens with Reusable Componets, *ACM Transactions on Software Engineering and Methodology*, October 1992.
- D. Batory, J. Thomas. and M. Sirkin, *Reengineering a Complex Application Using a Scalable Data Strcutre Compiler*, Department of Computer Science, University of Texas at Austin, May 1994.

Concept Index

R

S

T

U

V

X

Functions and Variables Index

Appendix A Example P2 program

Below is the complete P2 program, '`sample.p2`'.

```
#include <stdio.h>

// Element.

typedef struct {
  int   empno;
  int   age;
  float temp;
  char  *dept_name;
  char  name[20];
} E;

// Type expressions.

typex {
  p = conceptual[odlist1[odlist2[malloc[persistent]]]];
  t = conceptual[odlist[delflag[array[transient]]]];
}

// Containers.

container <E> stored_as t with {
  odlist key is age;
  array size is 10;
} k;

container <E> stored_as p with {
  odlist1 key is age;
  odlist2 key is name;
  persistent file is "~/foo" with size 1000;
} pk;

// Cursors.

cursor <k>
  where "$.temp >= 98.6"  // Predicate.
  orderby ascending name  // Orderby clause.
c,                        // c is a cursor variable.
*pc;                      // pc is a pointer to cursor

typedef cursor <k>
  where "$.temp >= 98.6"
  orderby ascending name
```

```
C;
C v;

cursor <k>
  where "$.dept_name == 'Computer Sciences'"
  orderby descending empno
cs;

// Generic cursors.

generic_cursor <E> gc;
typedef generic_cursor <E> GC;
GC gv;

// Function body.

#define F( X ) \
{ \
  foreach( X ) { \
    printf( "%d, %d, %.1f, \"%s\", \"%s\"\n", \
      X.empno, X.age, X.temp, X.dept_name, X.name ); \
  } \
}

// Function with a non-generic formal parameter.

int f( C x )
{
  F( x )
}

// Function with a generic formal parameter.

int gf( GC gx )
{
  F( gx )
}

// Employee data.
```

```
E rawdata[] = {
   { 10000, 60, 99.5,           "English",    "Akers, Mark" },
   { 10070, 22, 99.4,           "Physics",   "Andrews, Kay" },
   { 10020, 18, 99.0,           "History",     "Aaron, Bob" },
   { 10040, 42, 98.5, "Computer Sciences", "Singhal, Vivek" },
   { 10010, 40, 98.7, "Computer Sciences",    "Batory, Don" },
   { 10040, 53, 96.3,        "Accounting",  "Akerson, Mary" },
   { 10060, 65, 98.8,         "Nutrition", "Zacks, William" },
   { 10050, 23, 96.1, "Computer Sciences",   "Thomas, Jeff" },
   { 10080, 31, 98.7,     "Culinary Arts",   "Geraci, Bart" },
   { -1 }
};

// Main.

main()
{
  int i;
  E   *e;

  open( k );
  init_curs( c );
  init_curs( cs );

  for (i=0, e=rawdata; e->empno != -1; i++, e++) {
    insert( c, *e );
  }

  // You may pass c as a actual to formal c
  // and assign c to temporary cursor variable v:

  printf( "f( c ):\n" );
  f( c ); // Legal.
  v = c;  // Legal.

  // You may not pass cs as an actual to formal c,
  // nor assign cs to temporary cursor variable v:

#if 0
  printf( "\n" );
  printf( "f( cs ):\n" );

  f( cs ); // Not legal.
  v = cs;  // Not legal.
#endif

  // You may pass c and cs as actuals to generic formal gx,
  // and assign c and cs to generic temporary cursor variable gv:
```

```
  printf( "\n" );
  printf( "gf( c ):\n" );

  gf( (GC) &c );  // Legal.

  printf( "\n" );
  printf( "gf( cs ):\n" );

  gf( (GC) &cs ); // Legal.

  gv = (GC) &c;   // Legal.
  gv = (GC) &cs;  // Legal.

  close( k );
  exit( 0 );
}
```

Table of Contents