Predator: A Data Structure Precompiler

Marty Sirkin

1.0 General

1.1 Why have a data structure precompiler?

A great deal of the average programmer's time is spent writing, debugging and writing again many of the same routines to manipulate simple data structures and compositions of those basic data structures. Not only does this waste much time, but any programmer is capable of introducing errors into routines, even if similar routines have been written many times before.

Furthermore, programmers often write very complex programs because they know the internal structure of the data structures they are using. This tends to cause programming which is both far more complicated than need be, and usually specific to the one data structure being used. Hence if a programmer were to later decide that another data structure world be more appropriate, a large amount of code would need to be rewritten.

There are many examples in the literature of parameterized types and the like being used to solve the code reusability problem. Our solution differs from many of those because it changes the level of data structure programming to that of *simple tuples* (as in database programming). Data structures are considered to be nothing more than collections or sets containing data items. While there may be differences internally among the data structures (some may be more efficient at insertions, for example), this distinction is hidden from the programmer. Hence the level of programming is made far more simple. This simplicity of programming is important, because it will allow programmers to write algorithms and programs which are simpler, more easily debugged and more maintainable.

The data structure precompiler eliminates many of these problems by providing an environment in which the programmer has available the following:

- A simple method of describing data structures via *type expressions.*
- A small set of primitive functions (such as insert, delete, update...) for all supported data structures.
- A parser and code generator to provide working C code that implements the primitive functions for the data structures selected.
- A macro facility (similar to that in the C language) which allows programmers to specify their own functions and definitions.
- An include facility (not yet completed) which allows for include files similar to those in the C language so that precompiler definitions can be shared easily among programs.

Using this approach, a programmer can experiment with different data structures by simply changing the type expression, running the precompiler on the source file, and compiling the output C code from the precompiler. Since the actual code in the body of the program contains no knowledge of the data structure itself, no rewriting of the program need be done.

1.2 Efficiency of generated code

For a precompiler such as this to be successful, it is important that the code generated by it be comparable in efficiency to the code that could be generated by manual coding by the programmer. Assuming that it is, there is very little reason for programmers ever to go back to manually coding routines for basic data structures, and their compositions.

1.3 Testing the prototype

To understand how programmers really program (and to test the validity of our assumptions), the prototype will be constantly available for use. In addition, sample programming tasks will be available (simple ones) so that potential users can quickly try the precompiler out, We will ask programmers to program the task both manually in C, and with the precompiler.

The testing will hopefully provide us with two important pieces of information. It will give us user's input of ease of use, bugs, suggested improvements, etc. It will also allow us to compare the efficiency of the generated code to that which the programmer wrote to solve the same problem.

2.0 Using the precompiler

2.1 Packaging

The precompiler consists of the following files:

TABLE 1.

File name	Purpose
pred	The data structure precompiler
pred.pro	Optional list of alternate names for precompiler objects

2.2 Running the precompiler

The precompiler is a C language program which is run with one argument. That argument is the name of the source input file for the precompiler. All precompiler input files must end with the extension ".dac" (which stands for DAta structure C program). The extension does not need to be supplied when the program is executed. If no extension is supplied, the precompiler will add one.

For example, suppose that you wished to precompile a file whose name is: *cust.dac.* You would simply type: *pred cust* (Note that you could also type: *pred cust.dac* as well).

The precompiler will then generate all of the necessary C language files, including a make file. All that you need to do to then compile and link the program is to type:

```
make -f cust.mak
```

This then creates a program called *cust*, which can be run.

The precompiler does show status while it is running. It is a two-pass compiler, and prints status messages during each phase (as well as at the start and end of the program). Any errors that are encountered are noted, as well as any warnings (such as declaring a data structure, but not using it). Note that if there are any errors, the second pass (code generation) is not performed. If there are no errors, however, code will be generated (even if there are warnings).

2.3 Precompiler output

The precompiler generates the following output files:

TABLE 2. Precompiler output

Output file name	Purpose
<fname>.c	C file generated from original source.
<fname>_inc.h	Include file of actual data structures.
h_<fname>.c	C file of helper functions
h_<fname>.h	Include file of helper variables
<fname>.mak	Make file to compile/link program.

Note: <fname> is the name of the input source file (.dac file).

Note that the only file that the programmer should need to know about is the make file. All of the other files will be properly compiled and linked by the make file generated.

2.4 Format of the PRED.PRO file

The file **pred.pro** is an optional file that can be used to customize the precompiler. If it is to be used it must be located in the current working directory. The file contains sections of keywords which can be used to override the default names for the precompiler objects.

By using this file, for example, a programmer could change the name of the insert function from INS to INSERT (if that is what was desired). Or the name of the MACRO precompiler directive could be changed to TEXT_SUBS. The number of object names changed, and which ones are changed are up to the programmer. If, for any directive or function, the programmer wishes to use the default name, the string: <default> should be entered on that line.

The file consists of sections that each have a section title. The titles are:

- DIRECTIVES: - This section renames the precompiler directives. The directives are listed, one per line, directly below the DIRECTIVES: line. The directives, in order, are:

TABLE 3. Directives in the PRED.PRO file

Line	Default Name	Directive Use
1	SCHEMA	Directive to declare a schema
2	CURSDEF	Define array of cursors or dynamic cursors. Not currently used
3	CURSOR	Declares a cursor
4	MACRO	Declares a macro
5	LINK	Declares a link relationship
6	LCURSOR	Declares a link cursor. Usually not used by application programmers
7	COMPCURS	Composite cursor declaration

- FUNCTIONS: - This section renames the functions provided by the precompiler. They are listed, one per line, after the section header. The functions, in order, are:

TABLE 4. Functions in the PRED.PRO file

Line	Default Name	Function Use
1	INS	Insert an element into a container
2	ADV	Advance a cursor to the next valid element in a container
3	REV	Reverse a cursor to the previous valid element
4	DEL	Delete an element in a container
5	GETREC	Copies an element pointed to by a cursor to memory
6	UPD	Update a field of an element in a container
7	LAST_CURS_OP	Return status of last cursor operation on a cursor
8	RESET_CURSOR	Moves a cursor to start or end of a container
9	NEW_CURSOR	Allocate a dynamic cursor. Not currently working
10	FOREACH	Executes a code fragment for each element of a container found by a cursor
11	FIND	Moves a cursor to the first element that satisfies a predicate
12	SWAP	Swaps two elements in an unordered container
13	SIZE	Returns the size of a container
14	CURS_POS	Access cursor position. Declare a cursor position variable
15	MRU	Returns most recently used element in a container
16	LRU	Returns least recently used element in a container

- LAYERS - The current data structure layers which can be composed in type expressions are listed:

TABLE 5. Layers in the PRED.PRO file

Line	Default Name	Layer Use
1	ARRAY	Static ARRAY
2	MALLOC	Dynamically allocated elements
3	DELFLAG	Logically deletable items
4	DLIST	Doubly linked list
5	AVAIL	Avail list
6	SIZE	Size statistics kept
7	SEGMENT	Segmented elements
8	BINTREE	Ordered binary tree
9	PREDIND	Predicate-based indexing
10	ACTION	Function-based actions
11	USAGE	Most and least recently used elements in containers
12	LINKPOINT	Pointer-based link implementation.
13	LINKNEST	Nested loop link implementation.

2.5 Disclaimer

The precompiler is a work-in-progress. As a result not all of the functions and layers have been completed. The efficiency of the code generated is not yet as good as it will be, and, of course, there may be bugs. Please be patient with any lacks that there may be in the program - we're getting there.

Also, if you find any bugs, or have any comments or suggestions, please send them to:

`marty@cs.utexas.edu`

3.0 Basic Concepts

What follows is a description of the basic concepts of the data structure precompiler:

3.1 Elements

Elements are the basic building blocks from which data structures are built. In the precompiler they are nothing more than C structures. One can easily visualize them as tuples of data being stored in the data structure.

For example, a linked list of customer records would have the *element* type being the customer record (or struct).

Typedef statements are not currently supported for element parsing. So, a standard Predator element declaration would look like:

```
struct customer
{
  int age;
  char l_name[20], f_name[20];
};
```

In the above declaration, the name of the element would be "*customer*". In all future references to the element (such as in schema declarations), the name *customer* would have to be used to reference this element.

3.2 Schemas

A schema is a description of a data structure (as in database technology). The schema maps the abstract data structure that the programmer sees to a concrete data structure that the precompiler generates.

It is important to understand that schemas are not physical objects. Schemas are nothing more than templates that describe how Predator should create and generate a data structure. The data structure is not created until one (or more) *containers* are declared. In the Predator system one may have as many containers of a particular schema as desired.

The definition of a schema is done via a *type expression*. Each type expression defines how the precompiler should define the data structure. They are read left to right. Some examples of schemas follow:

```
SCHEMA CUST_ARRAY ON ELEMENT CUSTOMER = ARRAY[100];
```

This schema defines a definition of CUST_ARRAY as an array of 100 CUSTOMER elements.

```
SCHEMA CUST_ARR2 ON ELEMENT CUSTOMER = DELFLAG[ARRAY[100]];
```

This schema is very similar. However, each array element can now be deleted (it has a delete flag attached to it). Without the *delflag* modifier, no element in a data structure can be deleted.

```
SCHEMA CUST_LIST ON ELEMENT CUSTOMER = DLIST[MALLOC[]];
```

This final example shows a schema of a doubly linked list of customer records. Note that the *MALLOC* expression is needed. This tells the precompiler that the list is made up of dynamically allocated elements.

3.3 Layers

Layers are the building blocks that the precompiler uses in type expressions to build schemas. There are two types of layers: terminals and non-terminals. Terminal layers are "at the end" of a type expression. In other words, they are the bottom layer of the type expression. **No terminal may contain another layer as an argument.**

Non-terminals, on the other hand, *must* contain at least one other layer as an argument - they cannot be at the end of a type expression. All layers implement the same external interface (INSERT, DELETE...), so programmers should base the selection of layers on the program's needs, not based on distinct interfaces.

The table below lists the currently implemented layers:

TABLE 6. Implemented layers

Layer name	Layer description	Terminal	Layer definition
ARRAY	Array of n elements	Yes	ARRAY[#]
DELFLAG	Elements can be deleted	No	DELFLAG[type expression]

TABLE 6. Implemented layers

Layer name	Layer description	Terminal	Layer definition
MALLOC	Elements are dynamically allocated	Yes	MALLOC[]
DLIST	Doubly linked list. Ordered or unordered	No	DLIST[type expression <, fld1 <, fld2 ...>>]
AVAIL	Maintains available list of previously deleted nodes	No	AVAIL[type expression]
SIZE	Maintains size statistics for all containers of this schema	No	SIZE[type expression]
SEGMENT	Breaks element into multiple elements in multiple data structures	No	SEGMENT[te1, field1, te2,... fn, ten+1]
BINTREE	Ordered binary tree	No	BINTREE[type expression, fld1 <fld2 ...>]
PREDIND	Predicate-based index.	No	PREDIND[type expression, predicate]
ACTION	Performs a code section before or after a primitive function call on a container	No	ACTION[type expression, function, hint, <code>]
USAGE	Maintains LRU/MRU for a container	No	USAGE[type expression]

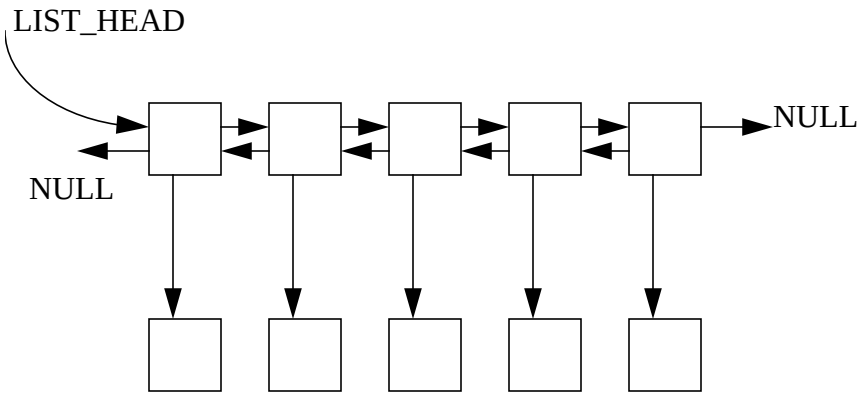
3.3.1 Layer descriptions

- ARRAY - This layer declares a static array of elements. In no event can the container hold more than the predeclared number of elements. Also, a simple array cannot reclaim data. So, if an element is deleted, it cannot be reused (see AVAIL for how to do this).
- MALLOC - This layer informs PREDATOR that it should dynamically create all new elements in the container. Thus at “new” or “insert” time, the element is “malloc”ed. Note that by itself the MALLOC layer does not connect elements in a data structure. It is simply a layer that determines how they are created. MALLOC is usually layered with other components to make a data structure.
- DELFLAG - This layer indicates that elements may be deleted. If this layer is not included the DEL function will still “work”, but will (at most) remove the element from the container, but not delete it.
- DLIST - This layer creates a double-linked list of elements. DLISTs may be ordered (see below).
- AVAIL - This layer creates an available list of previously deleted items. It also allows the insert function to try to get a new element from the avail list before it creates a new item. This can save time for dynamically allocated structures, and allows array structures to reuse data from earlier in the array.

- SIZE - This layer causes PREDATOR to maintain the size of the container. Without this layer defined the SIZE function will not work (in fact, it will not compile). Only include this layer for containers about which you need to know their size.
- BINTREE - This component is an ordered binary tree.
- PREDIND - This layer creates a predicate-based index. If a cursor has *exactly* the same predicate as a predindex layer, that cursor will use the predicate index for scans, foreaches... The predicates must match exactly, character by character. There are no restrictions as to what types of clauses the predicate contains.
- ACTION - This layer allows a user to have a section of code performed before or after a given function for a specified SCHEMA. For example, one might like to have a printf done to a special file after each insert is done to a container. Or a variable should be incremented each time a DEL is done on a cursor for a container.
- USAGE - This layer maintains the ordering in which the elements of a container are used (inserted, deleted or updated). The programmer may then use the functions MRU and LRU which return the CURSOR POS variable of the element of interest.
- SEGMENT - This is a very special layer. It partitions an element record into two or more sub-data structures. For example:

```
SEGMENT[DLIST[MALLOC[]], fld1, MALLOC[]];
```

The type expression above partitions each element into two parts. One record contains all of the fields up to and including “fld1”. The other part contains the rest of the fields of the element. The first partition records are connected into a DLIST, and each of the first partition records contains a pointer to the associated second partition record.



It is important to remember that all of the SEGMENT modifications are transparent to the programmer. He/she sees the original element as one record. PREDATOR is responsible for handling all of the segment references.

The reason for using SEGMENTS is efficiency. Consider the case of a sorting algorithm. The algorithm does its work by swapping data records. If the records are large (many fields) the SWAP operations

can be quite time consuming. The SWAP operation for SEGMENT merely swaps the first segment, and the *pointers* to the other segments. Thus if most of the data is not in the first segment, the SWAP is much faster. In fact, a PREDATOR version of QUICKSORT has been written using this idea.

A final feature in SEGMENT is that the first segment can be set up so that it contains *none* of the fields from the original element. This is done by leaving the first field argument blank int the type expression:

```
SEGMENT[ARRAY[100], , MALLOC[], fld1, ARRAY[100]];
```

This type expression has the first segment containing only the pointers to the other two segments. The first segment is stored in an ARRAY. The second segment is dynamically allocated and contains all data fields from the element up to and including fld1. The third segment is also stored in an array, and contains the rest of the fields.

3.3.2 Ordering in layers

Certain layers allow for elements to be ordered by the contents of one (or more fields). The ordering may be done on any field of the element, may be either ascending or descending, and may be done on string field (via a typecast). A maximum of five fields per layer is currently supported.

The format of an ordered layer is:

```
layer[sub_layer, [+-][(STRING)] fld1, [+-] [(STRING)] fld2...]
```

The syntax is as follows. After the subfield description the order fields are listed, separated by a comma. Each field can be preceded by a + or - (for ascending or descending). The default is ascending. If the field is a character string the typecast *(STRING)* is required.

The following are examples of an ordered layers:

```
dlist[malloc[], (STRING) l_name,  (STRING) f_name,  -age]
```

This layer is a doubly linked list of dynamically allocated customers. The list of customers is ordered by last name, then by first name (both ascending). Then (if there are any customers with the same first and last names) the list is ordered by *reverse* age.

```
dlist[avail[array[100]], - (STRING) mag_name,  date]
```

This layer is another list, this time based on a static array (using an avail list). The doubly linked list is ordered by magazine name, with the name in reverse order. Then the list is ordered by subscription date.

Note that if you do not specify ordering fields in the SCHEMA definition that elements are added in chronological order.

The following layers currently support ordering:

- dlist
- bintree

3.3.3 Type Expression examples

The following are some examples of valid type expressions:

- ARRAY[n] - A static array of n elements.
- DLIST[MALLOC[]] - A doubly-linked list of dynamically allocated elements.
- DLIST[ARRAY[n], fld1] - An array of n items, organized into a list, ordered by field “fld1”.

- AVAIL[DELFLAG[[ARRAY[n]]] - An array with an avail list kept of deleted items.
- SIZE[DLIST[MALLOC[]] - A dynamic list which maintains the size of the container.
- DELFLAG[SIZE[SEGMENT[ARRAY[n], fld1, DLIST[MALLOC[], fld2], fld3, MALLOC[]]]] - This is a segmented data structure which contains items which can be deleted, and the size is maintained. Segment1 is an array of fields (up to fld1). Segment 2 is an ordered doubly-linked list (ordered by fld2) of fields up to fld3. The final segment is the rest of the fields with the records being dynamically allocated.

3.4 Containers

Containers are the collections of individual elements. Each container is a data structure. Containers are nothing more than instantiations of schemas. The container abstraction works because typical data structures that programmers use are nothing more than an implementation of the container abstraction.

So, for example, if you wanted to have a customer array called *my_custs* you would simply enter:

```
CUST_ARRAY my_custs;
```

If you wanted to have a customer list called *new_custs* the definition would be:

```
CUST_LIST new_custs;
```

Finally, a declaration of a customer list and an array of customer lists:

```
CUST_LIST a_list, list_arr[4];
```

Note that these are nothing more than C declarations. The precompiler creates a *typedef* for each schema. To create a container, you only have to declare the container.

NOTE: You may have as many containers of a specific schema as you would like. Each one is a separate data structure with its own memory. The only thing about the containers that is shared is the schema definition.

3.5 Cursors

Cursors are objects defined by the precompiler which allow the programmer to access elements in the data structure. Cursors are completely *opaque*, in that the programmer has no idea of how a specific cursor is implemented - nor does he/she need to.

Cursors are declared with the simple declaration:

```
CURSOR <cursor_name> ON <container_name> WHERE [<predicate>];
```

Where <cursor_name> is a unique cursor name, <container_name> is the name of a previously declared container, and predicate (which **must** be surrounded by parenthesis) is a valid “C” predicate that the cursor uses to determine which elements of a container to visit, and which to ignore. If the cursor is to traverse all elements in a container, the predicate can be removed. In this case the declaration would be:

```
CURSOR <cursor_name> ON <container_name>;
```

Each cursor is bound uniquely to one and only one container. A given container, however, may have many cursors declared on it. There is no cursor stability provided - it is up to the programmer to determine when cursors may become invalid due to insertions and deletions.

Each cursor has a search predicate. This predicate determines which elements of a container should be visited by a cursor. If, for example, you have a search predicate of TRUE (or 1), all elements of a container can be visited by a cursor. If, however, you were to specify *age > 30* as the predicate, only those elements

who have an age field whose value is > 30 would be visited by the cursor. This allows the programmer to raise the level of programming to a much higher level than before. If no predicate is specified, TRUE is assumed as the predicate. Predator contains a *query optimizer*, which is responsible for determining the best match of layer (within a container's type expression) to a cursor's search predicate. Predator selects the layer that will give the best performance for scanning the container for cursor matches.

The PREDIND layer is a special case. If a cursor has the *exact* same predicate as a predicate indexing layer in the type expression, Predator will use that layer for the scanning of the cursor. This is efficient when the number of elements in the container is >> the number of elements that satisfy the predicate.

In the next example, a cursor called *wealthy_young* is declared. It operates on the container *custs*, and will only stop on elements (customers) that are under 30 and earn more than $100,000 a year:

```
CURSOR wealthy_young ON custs WHERE (age < 30) && (salary >
100000);
```

Note that in the search predicate you do not need to reference the name of the container (or cursor) with the field names. If (in any Predator predicate) you simply specify a field name, Predator will try to match it to the proper container, and generate the associated code.

3.5.1 Cursor Positions

The precompiler allows the programmer to save, assign and compare cursor positions within a container via the *CURS_POS* construct (note that this name can be changed via the pred.pro file).

Cursor position variables are declared as follows:

```
CURS_POS var1, var2... ,varn;
```

Cursor position variables are used to hold the position of a cursor in a container. The position of a cursor is obtained with the *CURS_POS* field of the cursor. An example of saving a cursor position is:

```
var1 = <curs>.CURS_POS;
```

Cursors may be moved to new positions by assigning to their CURS_POS field:

```
<curs>.CURS_POS = var1;

<curs1>.CURS_POS = <curs2>.CURS_POS;
```

Finally, cursor positions may be tested. The following fragment finds an element, saves the position, and then prints a special message when that element is found in a FOREACH clause:

```
RESET_CURSOR(cursor, CONT_START);
FIND(cursor, cursor.age == 42);
var1 = cursor.CURS_POS;
FOREACH(cursor)
{
 if (var1 == cursor.CURS_POS)
    printf("At the element!\n");
}
```

3.5.2 CURSDEF

Ignore this section in the current version of Predator.

Each cursor that is to be used in a program should be declared with the CURSDEF directive. The syntax of the directive is:

```
CURSDEF c1, c2... cn;
```

This declaration must precede any CURSOR statements, and multiple CURSDEF statements are supported.

3.5.3 Arrays of cursors

Ignore this section in the current version of Predator.

The precompiler supports arrays of cursors. This can be useful when you wish to have multiple cursors declared on a container, or to have an array of cursors declared on an array of containers (in other words, cursor[n] is declared on container[n]). Some examples will illustrate these features:

```
CURSDEF curs[2];

CURSOR curs[0] ON cont WHERE TRUE;

CURSOR curs[1] ON cont WHERE TRUE;
```

This example declares two cursors, which are in an array, which access the same container, and can be used in some algorithm to manipulate the container. Note that a reference such as: ADV(curs[var]) (where var is an integer variable) is both possible and quite powerful.

```
CURSDEF curs[2];

CURSOR curs[] ON cont WHERE TRUE;
```

This example does exactly the same as the previous one. Note the open array syntax, which the precompiler interprets to mean: “All of the elements in the array.”

```
CURSDEF curs[5];

CURSOR curs[] ON cont[] WHERE TRUE;
```

This final example allocates an array of cursors to an array of containers. So, curs[0] is declared to span cont[0], curs[1] on cont[1] and so on.

3.6 Implicit predicates

Predicates are used in many different ways in PREDATOR. Predicates are used in the where clauses of cursor declarations. They are used in the PREDIND layer. They are also used to specify the code to execute in a FOREACH statement. Finally, they are used in FIND statements as the secondary search clause.

When predicates are specified in PREDATOR, the cursor part of the field declaration is implicit. For example:

```
FIND(curs3, age > 30);
```

means that PREDATOR will search (using the curs3 cursor) for records where the age field (of curs3) is greater than 30. The implied cursor in a cursor declaration is the cursor being declared. In the FIND and FOREACH statements it is the specified cursor. And the PREDIND layer uses *any* cursor as the implied match. Examples of this are:

- CURSOR curs ON cont WHERE (age == 30 && strcmp(l_name, “Miller”) < 0);
- SCHEMA cont ON ELEMENT elem = PREDIND[DLIST[MALLOC[]], age == 30];
- FIND(curs2, ss# < 123456789);

- FOREACH(curs1)
  {
  printf(“Name: %s, %s\n”, l_name, f_name);
  };

If you wish to use a field of another cursor inside of a predicate, simply use a construction like the one below:

```
FIND(curs, age <= curs2.max_age);
```

This example sets the curs cursor on the first element which has an age field that is less than or equal to the max_age field of the element that curs2 (a different cursor) points to.

3.7 Links

It is often necessary to interconnect elements in multiple containers.Predator uses the link construct for this purpose. Links are declared as follows:

```
LINK <link_name> ON <many/one> <cont_name> TO <many/one> <cont_-
name> USING <implementation> WHERE <pred>;
```

Links in Predator are declarative. They are placed at the top of the program (before the application code), but after the container declarations. As with other Predator constructs, links are automatically maintained. When a program inserts, deletes or updates an element in a container that is linked, Predator automatically generates the proper code to maintain the link.

A single link in Predator is able to interrelate elements based on predicates that can be as complicated as desired. Predicates can use both conjunctions and disjunctions. Elements in predicates use the container name followed by a period, followed by the field name to reference a field. For example, if we wished to link employees with departments where the department number of the employee is the same as the department number of the department we would have a link declaration of:

```
LINK a_link ON MANY emps TO ONE dept USING pointer WHERE emps.-
dept_num == dept.dept_num;
```

Another example might be to link employees to departments where the employee name is the same as the manager name in a department record. In other words, this is the “manages” link that links the manager record with his/her department (note that the comparison of strings must be valid C syntax):

```
LINK manages ON ONE emps TO ONE dept USING nestedloop WHERE strc-
mp(emps.name, dept.manager) == 0;
```

For those familiar with databases, links are nothing more than simple database joins.

Note: Currently Predator does not support links that use only one container (i.e. a self- link). Nor is the <many/one> flag yet implemented.

Programmers may choose the method that Predator uses to generate links. The *USING* clause of a link statement specifies how Predator will implement that link. The following implementations of links are currently supported:

- POINTER - Predator uses pointers to implement links. The extra work of maintaining the link is located in insert/delete/update operations.
- NESTEDLOOP - Predator uses a nested loop algorithm (i.e. cross-product) to determine the linked elements. The overhead of this method is located in the FOREACH/RESET/ADV operations.

3.8 Composite cursors

Links were described in the previous section. The composite cursor (or COMPCURS) is the method by which programs use the information maintained in the link relationship.

Composite cursors allow the programmer to range a set of cursors over a set of containers, and to only set the cursor's values to proper tuples that satisfy the composite cursor's predicate. Composite cursors may retrieve tuples of cursors over as many containers as desired. The syntax of the composite cursor statement is as follows:

```
COMPCURS <cc_name> USING <curs1> <cont1> <curs2> <cont2> [...]
WHERE <pred>;
```

Each cursor declared in the statement must be paired with a container name, and the cursor MUST NOT have been previously declared - it must be a "new" cursor. You must have a cursor/container pair for each container referenced in the predicate.

As with links, composite cursors are declarative - they are placed above the application code and below the link and container declarations.

Composite cursors are referenced via a FOREACH statement. The syntax of the foreach statement is similar to that for container cursors:

```
FOREACH(<comp_cursor>)
{
  <code>
}
```

With the above statement, the <code> section will be executed once for each combination of elements that satisfy the composite cursor predicate. In each iteration of the loop, each container cursor will be set to the proper element, and the <code> section may reference fields in that container as normal cursor references.

A few examples illustrate the use of composite cursors:

```
COMPCURS ex1 USING e_curs emps d_curs dept WHERE e_curs.age < 30 &&
a_link;
```

This example retrieves cursor pairs (one to emps and one to dept) where the employee's age is less than thirty, and the department numbers match (from the link relationship defined in the section above). To use the composite cursor, you might want to print out each employee's name and department number, as well as the department name:

```
FOREACH(ex1)
{
  printf("Name: %s  Dept num: %d  Dept name: %s\n", e_curs.name,
         d_curs.dept_num, d_curs.dept_name);
}
```

Another (more complicated example) interrelates employees, departments, and orders placed by employees to show all of the orders by members of a specified department:

```
LINK link1 ON ONE emps TO MANY orders WHERE emps.emp_num == order.-
emp_num;

COMPCURS ex2 USING e_curs emps d_curs dept o_curs orders WHERE
 a_link && link1 &&e_curs.dept_num == dyn_var;
```

There are two important points to notice in the example above. The department number comparison (on the department container) is compared against a variable (dyn_var). Thus this composite cursor can be

used to query against many different departments. Second, each tuple found in a foreach statement will be a triple of the employee, the department and an order (all from the same department). They can be printed or summed as desired.

If, in the previous example, we had wished to restrict to a particular order type, we could have added another term to the predicate to restrict the orders to those desired.

Note: Currently you may not place a link predicate directly in a composite cursor predicate. You must first declare the link, and then use the link name in the composite cursor.

Note: You may have as many link predicate elements and container cursor elements in the predicate as you wish, but they MUST be conjunctive (at the top level). For example, the following is not legal:

```
COMPCURS ex3 USING ec emps dc dept WHERE link1 || link2;
```

whereas the following is:

```
COMPCURS ex4 USING ec emps dc dept WHERE link1 && ((dc.dept_num ==
42) || (dc.dept_num == 49));
```

3.9 Precompiler directives

All of the data declarations for the precompiler are done via *precompiler directives.* These directives are placed in the source of your .DAC file, before and outside of the scope of the actual code of your program. The directives are read and translated on the first pass of the precompiler, and are removed from the output .C file generated by the precompiler.

Precompiler directives are case-sensitive and are free-form (they are not tied to a specific column in the source file).

3.10 Primitive Functions

The primitive functions supplied by the precompiler allow the programmer to manipulate containers, elements and cursors in an easy, simplified way which elevates the level of programming to that used in database programming. The algorithms are simple and clean, and can be debugged quickly and easily.

The functions supplied by the precompiler are:

- INS - Add an element to a container.
- RESET_CURSOR - This moves a cursor to the start or end of the container it traverses. The second argument to RESET_CURSOR is one of: **CONT_START** or **CONT_END**. Note that FOREACH always does a RESET_CURSOR.
- ADV - Moves the cursor to the next element in the container that satisfies the search predicate. If no more elements are found, the cursor is placed at the end of the container.
- REV - Moves the cursor to the previous item that satisfies the search predicate. If no more elements are found, the cursor is placed at the start of the container.
- DEL - Deletes the element pointed to by a cursor.
- GETREC - Copies the element pointed to by a cursor to a memory area specified in the call.

- UPD - Updates a field of the element pointed to by a cursor to a value specified in the call. If the field to be updated is a string the special typecast (STRING) must be placed before the value. If the field to updated is an array, the special typecast (ARRAY) must be placed in front of the value. See Figure 4.2, “Explanation of sample program,” on page 20 for more information.
- LAST_CURS_OP - This returns the status of the last operation performed on the specified cursor. The only functions which change the status code returned by LAST_CURS_OP are: ADV, REV, FIND. Possible values are:

TABLE 7. Status values

Value	Meaning
GOOD	Previous function completed without errors
NOT_GOOD	Previous function did not complete
EOR	Previous operation did not find an element - End of Records

- FOREACH - Performs a code fragment for each element in a container. Note that since you specify a cursor for this function, the elements visited are only those visited by the cursor. This function is very useful in performing an operation on each element of a container.
- FIND - This function finds the next element in a container based on a predicate specified in the call. Since this call uses a cursor, it will only search the elements found by that cursor. In other words, both predicates must be satisfied for a record to be found. Note: This call does NOT reset the cursor before the find operation. That is left to the programmer.
- SWAP - This function swaps two records in a container. It is passed two cursors which are positioned over two different elements. The data values in the elements are then swapped.
- NEW_CUROSR - This function declares a dynamic cursor. Not yet fully implemented.
- SIZE - This function returns the size of the container argument. It will only work, however, if the container has a schema for which the SIZE layer is defined.

As with the precompiler directives, the function names can be changed to any other name the programmer desires via the PRED.PRO file.

TABLE 8. Functions provided with the precompiler

Function	Format
INS	INS(<container_name>,<element>,<cursor>);
ADV	ADV(<curs_name>);
REV	REV(<curs_name>);
DEL	DEL(<curs_name>);
GETREC	GETREC(<curs_name>, <addr>);
UPD	UPD(<curs_name>, <field>, <value>);

TABLE 8. Functions provided with the precompiler

Function	Format
LAST_CURS_OP	LAST_CURS_OP(<curs_name>)
RESET_CURSOR	RESET_CURSOR(<curs_name>, <which end>)
NEW_CURSOR	NEW_CURSOR(<curs_name>, <cont_-name>, <predicate>)
FOREACH	FOREACH(<curs_name> {<oeprations>}
FIND	FIND(<curs_name>, predicate);
SWAP	SWAP(<curs1_name>, <curs2_name>);
SIZE	SIZE(<container_name>);
MRU	MRU(<container_name>);
LRU	LRU(<container_name>);

3.11 Simplified cursor operations

3.11.1 Cursor advance/reverse operations

The precompiler is able to understand the following two constructs:

- <cursor>++
- <cursor>--

These operations are exactly the same as the functions ADV and REV. This is nothing more than a simplified way of specifying them, so that the code can look more “C-like”.

3.11.2 Direct cursor manipulation

The precompiler is able to understand and generate code for direct manipulation of the fields that are contained in the element that a cursor points to. This allows for far more readable code to be written.

There are two types of direct cursor manipulation: read and write.

3.11.3 Field accessing

The precompiler supports direct manipulation of fields for a cursor as if the cursor were the element itself. For example:

```
if (cust_cursor.age == 15)
   printf(“Age is 15!!!);
```

This example looks at the age of the element that the cursor points to. If it is 15, it performs the printf statement. Anyplace you can think of that you would want to read a field of an element you may use this construct.

3.11.4 Update operations

This same idea can be used to simplify the UPD function. The precompiler understands assignments to cursor fields, and generates the correct code. For example:

```
cust_cursor.age = 25;
```

This line updates the age field to 25. It is exactly the same as the following UPD function call:

```
UPD(cust_cursor, age, 25);
```

In fact, the cursor form of the UPD function generates exactly the same code as the UPD instruction. It is up to the programmer to choose the form he wants.

Note that the string and array casts that are available for UPD are also available for the cursor form of the command. Example:

```
cust_cursor.last_name = (STRING) “Joe”;
```

or....

```
cust_cursor.arr = (ARRAY) my_array;
```

3.12 Macros

The precompiler provides a generalized macro facility which is very similar to that in the C language. Macros are declared as follows:

```
MACRO <macro_name>(arg_list) <file_name>;
```

<macro_name> is the name of the macro to look for in the source text.

(arg_list) is a list of arguments. If the macro has no arguments, the empty parenthesis still must be included.

<file_name> is the name of the file in the current directory which contains the text to be substituted for the macro.

What happens when a macro is found in the source text is this: The macro file is opened and read. Any variable substitutions are done. The new code fragment is the evaluated and code is generated for it.

There are several important points to be made about the macro facility:

- Macros may call other macros, up to 15 levels deep. This is much more powerful than what C allows.
- Macros (in the macro files) can be as long as desired. This is also more powerful than C.
- The macro code that is read in is evaluated, so that if it contains other function calls (or macros) those calls are evaluated as well. Thus, the macro substitution is not just static text, as it is in C.
- The macro code that is read in is not error checked (currently). This is done during the first pass of the precompiler, and the macro expansion is done during the second pass.

4.0 Important notes

This section contains descriptions of several features/bugs of the current version of the compiler. It is important to understand each of these points, as they will arise when using the compiler.

4.1 The parser is picky

The current compiler has a custom parser which is not as robust as the one next version compiler, which is based on LEX and YACC. There are several places in which the compiler is not as robust as it should be. In these places, it is important to stick to the rules stated below to avoid problems:

1. Function calls should always have the left parenthesis *immediately* following the function name. Ex: ADV(cursor); as opposed to ADV (curs);

2. Arguments within functions are best placed by putting the first argument right after the open parenthesis, and by placing commas right after arguments, a space following the comma., and the close parenthesis right after the last argument.
   Ex: INS(cont, elem, curs); as opposed to INS(cont , elem , curs);

4.2 The precompiler does not understand forward declarations

Therefore, make sure that all Predator primitives are declared before they are used. For example, place all element declarations before the schema declarations in which they are used, and all schema declarations before the container declarations, etc.

4.3 The compiler generates all code to keep all invariants

When a programmer writes the declarations at the top of a Predator program, he is declaring invariants over the data structures. For example, suppose a container is an ordered binary tree of malloced elements. The ordering is an invariant over the container. When a programmer writes a data structure independent algorithm using Predator statements (such as INS, DEL, UPD) the code to maintain the ordering is automatically generated by Predator. An example illustrates the point:

```
SCHEMA my_sch ON ELEMENT customer = dlist[malloc[], (STRING)
                                          l_name, (STRING) f_name];

my_sch a_cont;
CURSOR curs ON a_cont;


void main(void)
{
 ...
RESET_CURSOR(curs, CONT_START);
curs++; curs++;
curs.l_name = (STRING) “Smith”;
```

The example above has a cursor that points to the third element of a_cont. When the update statement is found, Predator will update the value, and will generate code that will repair the ordered double-linked list if needed. The programmer only need concern himself about updating the value. All other constraints (including linking of multiple containers) is automatically handled by Predator.

4.4 Predator has no cursor stability

This means that Predator does not handle the invalidation of cursors when the underlying element under the cursor is deleted. For example:

```
CURSOR curs1 ON a_cont;
CURSOR curs2 ON a_cont;

void main(void)
{
  ...
  RESET_CURSOR(curs1, CONT_START);
  curs1++; curs1++;
```

```
curs2.CURS_POS = curs1.CURS_POS;
DEL(curs1);
```

The cursor *curs2* is now invalid. It is up to the programmer to keep track of this, and not to reference curs2 until it is restored to a valid value.

Cursors which point to elements which are updated (and possibly moved), as well as cursors that are part of a composite cursor are still valid when the underlying element is updated. However, since the element may have moved within the container, there is no guarantee that continuing the scan of the cursor (with an ADV or a FOREACH) will reach all of the elements that should be reached.

5.0 Sample program

5.1 The program

The following is the file **greet.mac** that is called from the main program:

printf(“Hello there, %s\n”, field);

The following is the main program:

- #include <stdio.h>

```
struct customer
{
  char l_name[20];
  char f_name [20];
  short age;
}

SCHEMA cust_array ON ELEMENT customer = delflag[array[100]];
cust_array new_custs;

CURSDEF cust_cursor;
CURSOR cust_cursor ON new_custs WHERE age > 30;

MACRO greet(field) greet.mac;

struct customer test_cust;

void main()
{
  short loop;

  strcpy(test_cust.l_name, “Smith”);
  strcpy(test_cust.f_name, “Joe”);
  for (loop = 15; loop < 60; loop = loop + 5)
  {
     test_cust.age = loop;
```

```
    INS(new_custs, test_cust, cust_cursor);
  }

  printf("Doing a Foreach:\n");
  FOREACH(cust_cursor)
  {
   greet(l_name);
   DEL(cust_curs);
  }
 }
```

5.2 Explanation of sample program

The following is an explanation of the program listing above, in the order of the program:

1. The element *customer* is declared. Each customer has three fields, a first name, a last name and an age.
2. The *cust_array* schema is declared. Each container of type cust_array is an array of 100 customers, each of which can be deleted.
3. An actual container of customers, *new_custs* is declared.
4. A cursor is declared on container *new_custs* with a search predicate.
5. A macro is defined. *Greet* is a macro that takes ne argument. The code to replace greet with is located in the file *greet.mac* The code for the file is listed above. Each time that "greet" is found in the text of the program it will be replaced by the text in the specified file. And, any occurrences of the formal variable "field" in the file will be replaced by the actual variable specified in the macro call.
6. A sample customer is declared. Notice that it is declared as a normal structure.
7. The name fields of the sample customer are filled in.
8. A loop adds elements (all with the same name) with different ages from 15-55. The *insert* function returns the position of the new record in the cursor *cust_cursor.*
9. Then for each element that cursor *cust_cursor* finds, the *greet* macro is called to print out the last name. Then the element is deleted.

6.0 Examples of all directives, functions

The following is a source program, that doesn't do anything all that interesting, but does illustrate examples of each of the directives and functions available with the precompiler.

```
#include <stdio.h>

struct customer        /* Defines an element */
{
 char l_name[20];     /* Which has a last name, first name, age */
 char f_name [20];
 short age;
}
```

```
                /* Declares that cust arrays are arrays   */
                /*  of 100 customer elements, which can   */
                /*  be deleted.                           */

SCHEMA cust_array ON ELEMENT customer = delflag[usage[size[array[100]]]];

                /* Declare a container of schema cust_array.*/
cust_array new_custs;

                /* Declare two cursors.                  */
                /* Both cursors range over all elements */
CURSOR cust_cursor ON new_custs WHERE 1;
CURSOR cust2 ON new_custs;

                /* Declare a macro with two fields.      */
MACRO greet(f1, f2) gr.mac;

                 /* Create one customer element to be used.*/
struct customer test_cust;

                /* Declare a cursor position variable. */
CURS_POS position;

void main()       /* Main procedure.                     */

                /* Set up an element, and insert it.     */
 strcpy(test_cust.l_name, “Smith”);
 strcpy(test_cust.f_name, “Jones”);
 test_cust.age = 55;
 INS(new_custs, test_cust, cust_cursor);

                /* Change the age and insert another. */
 test_cust.age = 33;
 INS(new_custs, test_cust, cust_cursor);

                /* Reset the cursor to the start of   */
                /*  the container.                    */
 RESET_CURSOR(cust_cursor, CONT_START);

                /* Find the first customer whose age  */
                /*  is less that 42.                  */
 FIND(cust_cursor, age < 42);

                /* Save the position of this element. */

                /* Print the results of the find.     */
 printf(“Results of the find are: %d\n”,
       LAST_CURS_OP(cust_cursor));

                /* Update the name of the element that*/
                /*  the cursor points to.  Do it both */
                /*  with the function, and explicitly.*/
                /* Also, update the age.              */
 UPD(cust_cursor, f_name, (STRING) “Fred”);
```

```
 cust_cursor.l_name = (STRING) “Jones”;
 cust_cursor.age = 30;

                /* Move the cursor forward and back  */
                /* Note: This sequence causes the cursor to be */
                /*  EOR, so an update on the cursor would fail. */
 ADV(cust_cursor);
 REV(cust_cursor);
 ADV(cust_cursor);
 cust_cursor++;
 REV(cust_cursor);
 cust_cursor--;


                /* Get last record into memory.          */
                /* Print results.                        */
 RESET_CURSOR(cust_cursor, CONT_END);
 GETREC(cust_cursor, test_cursor);
 printf(“Name: %s, %s   Age: %d\n”, test_cursor.l_name,
       test_cursor.f_name, test_cursor.age);

                /* Swap the first and last records.      */
 RESET_CURSOR(cust2, CONT_START);
 SWAP(cust_cursor, cust2);

                /* For each element in the container, greet */
                /*  tham (and print a message to the saved  */
                /*  element) and then delete the element.   */
 FOREACH(cust_cursor)
 {
   greet(l_name, f_name);
   if (position == cust_cursor.CURS_POS)
     printf(“You are special!!\n”);
   DEL(cust_cursor);
 }

                /* Print the size of the container. */
 printf(“Size of container is: %d\n”, SIZE(new_custs));

                /* Find the least recently used element. */
                /* Set cursor to MRU element.          */
                /* Print results.                      */
                /* Move cursor to saved position, and print */
 position = LRU(new_custs);
 curs.CURS_POS = MRU(new_custs);
 printf(“Name of MRU: %s, %s\n”, curs.l_name, curs.f_name);
 curs.CURS_POS = position;
 printf(“Name of LRU: %s, %s\n”, curs.l_name, curs.f_name);
}
```