22. PERFORMANCE ISSUES

This chapter describes a number of areas that often contribute to performance problems in Interlisp-D programs. Many performance problems can be improved by optimizing the use of storage, since allocating and reclaiming large amounts of storage is expensive. Another tactic that can sometimes yield performance improvements is to change the use of variable bindings on the stack to reduce variable lookup time. There are a number of tools that can be used to determine which parts of a computation cause performance bottlenecks.

Storage Allocation and Garbage Collection

As an Interlisp-D applications program runs, it creates data structures (allocated out of free storage space), manipulates them, and then discards them. If there were no way of reclaiming this space, over time the Interlisp-D memory (both the physical memory in the machine and the virtual memory stored on the disk) would fill up, and the computation would come to a halt. Actually, long before this could happen the system would probably become intolerably slow, due to "data fragmentation," which occurs when the data currently in use are spread over many virtual memory pages, so that most of the computer time must be spent swapping disk pages into physical memory. The problem of fragmentation will occur in any situation where the virtual memory is significantly larger than the real physical memory. To reduce swapping, it is desirable to keep the "working set" (the set of pages containing actively referenced data) as small as possible.

It is possible to write programs that don't generate much "garbage" data, or which recycle data, but such programs tend to be overly complicated and difficult to debug. Spending effort writing such programs defeats the whole point of using a system with automatic storage allocation. An important part of any Lisp implementation is the "garbage collector" which identifies discarded data and reclaims its space. There are several well-known approaches to garbage collection. One method is the traditional mark-and-sweep garbage collection algorithm, which identifies "garbage" data by marking all accessible data structures, and then sweeping through the data spaces to find all unmarked objects (i.e., not referenced by any other object). Although this method is guaranteed to reclaim all garbage, it takes time proportional to the number of allocated objects, which may be very large. (Some allocated objects will have been marked during the "mark" phase, and the remainder will be collected during the

"sweep" phase; so all will have to be touched in some way.) Also, the time that a mark-and-sweep garbage collection takes is independent of the amount of garbage collected; it is possible to sweep through the whole virtual memory, and only recover a small amount of garbage.

For interactive applications, it is not acceptable to have long interruptions in a computation for the purpose of garbage collection. Interlisp-D solves this problem by using a reference-counting garbage collector. With this scheme, there is a table containing counts of how many times each object is referenced. This table is incrementally updated as pointers are created and discarded, incurring a small overhead distributed over the computation as a whole. (Note: References from the stack are not counted, but are handled separately at "sweep" time; thus the vast majority of data manipulations do not cause updates to this table.) At opportune moments, the garbage collector scans this table, and reclaims all objects that are no longer accessible (have a reference count of zero). The pause while objects are reclaimed is only the time for scanning the reference count tables (small) plus time proportional to the amount of garbage that has to be collected (typically less than a second). "Opportune" times occur when a certain number of cells have been allocated or when the system has been waiting for the user to type something for long enough. The frequency of garbage collection is controlled by the functions and variables described below. For the best system performance, it is desirable to adjust these parameters for frequent, short garbage collections, which will not interrupt interactive applications for very long, and which will have the added benefit of reducing data fragmentation, keeping the working set small.

One problem with the Interlisp-D garbage collector is that not all garbage is guaranteed to be collected. Circular data structures, which point to themselves directly or indirectly, are never reclaimed, since their reference counts are always at least one. With time, this unreclaimable garbage may increase the working set to unacceptable levels. Some users have worked with the same Interlisp-D virtual memory for a very long time, but it is a good idea to occasionally save all of your functions in files, reinitialize Interlisp-D, and rebuild your system. Many users end their working day by issuing a command to rebuild their system and then leaving the machine to perform this task in their absence. If the system seems to be spending too much time swapping (an indication of fragmented working set), this procedure is definitely recommended.

Garbage collection in Interlisp-D is controlled by the following functions and variables:

`(RECLAIM)` [Function]

Initiates a garbage collection. Returns 0.

`(RECLAIMMIN` *N*`)` [Function]

Sets the frequency of garbage collection. Interlisp keeps track of the number of cells of any type that have been allocated; when it reaches a given number, a

garbage collection occurs. If *N* is non-NIL, this number is set to *N*. Returns the current setting of the number.

RECLAIMWAIT [Variable]

Interlisp-D will invoke a RECLAIM if the system is idle and waiting for your input for RECLAIMWAIT seconds (currently set for 4 seconds).

(GCGAG *MESSAGE*) [Function]

Sets the behavior that occurs while a garbage collection is taking place. If *MESSAGE* is non-NIL, the cursor is complemented during a RECLAIM; if *MESSAGE*=NIL, nothing happens. The value of GCGAG is its previous setting.

(GCTRP) [Function]

Returns the number of cells until the next garbage collection, according to the RECLAIMMIN number.

The amount of storage allocated to different data types, how much of that storage is in use, and the amount of data fragmentation can be determined using the following function:

(STORAGE *TYPES PAGETHRESHOLD*) [Function]

STORAGE prints out a summary, for each data type, of the amount of space allocated to the data type, and how much of that space is currently in use. If *TYPES* is non-NIL, STORAGE only lists statistics for the specified types. *TYPES* can be a litatom or a list of types. If *PAGETHRESHOLD* is non-NIL, then STORAGE only lists statistics for types that have at least *PAGETHRESHOLD* pages allocated to them.

STORAGE prints out a table with the column headings **Type**, **Assigned**, **Free Items**, **In use**, and **Total alloc**. **Type** is the name of the data type. **Assigned** is how much of your virtual memory is set aside for items of this type. Currently, memory is allocated in quanta of two pages (1024 bytes). The numbers under **Assigned** show the number of pages and the total number of items that fit on those pages. **Free Items** shows how many items are available to be allocated (using the **create** construct, Chapter 8); these constitute the "free list" for that data type. **In use** shows how many items of this type are currently in use, i.e., have pointers to them and hence have not been garbage collected. If this number is higher than your program seems to warrant, you may want to look for storage leaks. The sum of **Free Items** and **In use** is always the same as the total **Assigned** items. **Total alloc** is the total number of items of this type that have ever been allocated (see BOXCOUNT, in the Performance Measuring section below).

Note: The information about the number of items of type LISTP is only approximate, because list cells are allocated in a special way that precludes easy computation of the number of items per page.

Note: When a data type is redeclared, the data type name is reassigned. Pages which were assigned to instances of the old data type are labeled ****DEALLOC****.

At the end of the table printout, STORAGE prints a "Data Spaces Summary" listing the number of pages allocated to the major data areas in the virtual address space: the space for fixed-length items (including datatypes), the space for variable-length items, and the space for litatoms. Variable-length data types such as arrays have fixed-length "headers," which is why they also appear in the printout of fixed-length data types. Thus, the line printed for the BITMAP data type says how many bitmaps have been allocated, but the "assigned pages" column counts only the headers, not the space used by the variable-length part of the bitmap. This summary also lists "Remaining Pages" in relation to the largest possible virtual memory, not the size of the virtual memory backing file in use. This file may fill up, causing a **STORAGE FULL** error, long before the "Remaining Pages" numbers reach zero.

STORAGE also prints out information about the sizes of the entries on the variable-length data free list. The block sizes are broken down by the value of the variable STORAGE.ARRAYSIZES, initially (4 16 64 256 1024 4096 16384 NIL), which yields a printout of the form:

```
variable-datum free list:
le 4       26 items;    104 cells.
le 16      72 items;    783 cells.
le 64      36 items;    964 cells.
le 256     28 items;   3155 cells.
le 1024     3 items;   1175 cells.
```

```
le 4096     5 items;   8303 cells.
le 16384    3 items;  17067 cells.
others      1 items;  17559 cells.
```

This information can be useful in determining if the variable-length data space is fragmented. If most of the free space is composed of small items, then the allocator may not be able to find room for large items, and will extend the variable datum space. If this is extended too much, this could cause an `ARRAYS FULL` error, even if there is a lot of space left in little chunks.

(STORAGE.LEFT) [Function]

Provides a programmatic way of determining how much storage is left in the major data areas in the virtual address space. Returns a list of the form (*MDSFREE MDSFRAC 8MBFRAC ATOMFREE ATOMFRAC*), where the elements are interpreted as follows:

MDSFREE The number of free pages left in the main data space (which includes both fixed-length and variable-length data types).

MDSFRAC The fraction of the total possible main data space that is free.

8MBFRAC The fraction of the total main data space that is free, relative to eight megabytes.

This number is useful when using Interlisp-D on some early computers where the hardware limits the address space to eight megabytes. The function 32MBADDRESSABLE returns non-NIL if the currently running Interlisp-D system can use the full 32 megabyte address space.

ATOMFREE The number of free pages left in the litatom space.

ATOMFRAC The fraction of the total litatom space that is free.

Note: Another important space resource is the amount of the virtual memory backing file in use (see VMEMSIZE, Chapter 12). The system will crash if the virtual memory file is full, even if the address space is not exhausted.

Variable Bindings

Different implementations of lisp use different methods of accessing free variables. The binding of variables occurs when a function or a PROG is entered. For example, if the function FOO has the definition (LAMBDA (A B) *BODY*), the variables A and B are bound so that any reference to A or B from *BODY* or any function called from *BODY* will refer to the arguments to the function FOO and not to the value of A or B from a higher level function. All variable names (litatoms) have a top level value cell which is used if the variable has not been bound in any function. In discussions of variable access, it is useful to distinquish between three types of variable access: local, special and global. Local variable access is the use of a variable that is bound within the function from which it is used. Special variable access is the use of a variable that is bound by another function. Global variable access is the use of a variable that has not been bound in any function. We will often refer to a variable all of whose accesses are local as a "local variable." Similarly, a variable all of whose accesses are global we call a "global variable."

In a "deep" bound system, a variable is bound by saving on the stack the variable's name together with a value cell which contains that variable's new value. When a variable is accessed, its value is found by searching the stack for the most recent binding (occurrence) and retrieving the value stored there. If the variable is not found on the stack, the variable's top level value cell is used.

In a "shallow" bound system, a variable is bound by saving on the stack the variable name and the variable's old value and putting the new value in the variable's top level value cell. When a variable is accessed, its value is always found in its top level value cell.

The deep binding scheme has one disadvantage: the amount of cpu time required to fetch the value of a variable depends on the stack distance between its use and its binding. The compiler can determine local variable accesses and compiles them as fetches directly from the stack. Thus this computation cost only arises in the use of variable not bound in the local frame ("free" variables). The process of finding the value of a free variable is called free variable lookup.

In a shallow bound system, the amount of cpu time required to fetch the value of a variable is constant regardless of whether the variable is local, special or global. The disadvantages of this scheme are that the actual binding of a variable takes longer (thus slowing down function call), the cells that contain the current in use values are spread throughout the space of all litatom value cells (thus increasing the working set size of functions) and context switching between processes requires unwinding and rewinding the stack (thus effectively prohibiting the use of context switching for many applications).

Interlisp-D uses deep binding, because of the working set considerations and the speed of context switching. The free variable lookup routine is microcoded, thus greatly reducing the search time. In benchmarks, the largest percentage of free variable lookup time was 20 percent of the total ellapsed time; the normal time was between 5 and 10 percent.

One consequence of Interlisp-D's deep binding scheme is that users may significantly improve performance by declaring global variables in certain situations. If a variable is declared global, the compiler will compile an access to that variable as a retrieval of its top level value, completely bypassing a stack search. This should be done only for variables that are never bound in functions, such as global databases and flags.

Global variable declarations should be done using the `GLOBALVARS` file package command (Chapter 17). Its form is `(GLOBALVARS` VAR_1 **...** VAR_N`)`.

Another way of improving performance is to declare variables as local within a function. Normally, all variables bound within a function have their names put on the stack, and these names are scanned during free variable lookup. If a variable is declared to be local within a function, its name is not put on the stack, so it is not scanned during free variable lookup, which may increase the speed of lookups. The compiler can also make some other optimizations if a variable is known to be local to a function.

A variable may be declared as local within a function by including the form `(DECLARE (LOCALVARS` VAR_1 **...** VAR_N`))` following the argument list in the definition of the function. Local variable declarations only effect the compilation of a function. Interpreted functions put all of their variable names on the stack, regardless of any declarations.

Performance Measuring

This section describes functions that gather and display statistics about a computation, such as as the elapsed time, and the number of data objects of different types allocated. TIMEALL and TIME gather statistics on the evaluation of a specified form. BREAKDON gathers statistics on individual functions called during a computation. These functions can be used to determine which parts of a computation are consuming the most resources (time, storage, etc.), and could most profitably be improved.

(TIMEALL *TIMEFORM NUMBEROFTIMES TIMEWHAT INTERPFLG* —) [NLambda Function]

Evaluates the form *TIMEFORM* and prints statistics on time spent in various categories (elapsed, keyboard wait, swapping time, gc) and data type allocation.

For more accurate measurement on small computations, *NUMBEROFTIMES* may be specified (its default is 1) to cause *TIMEFORM* to be executed *NUMBEROFTIMES* times. To improve the accuracy of timing open-coded operations in this case, TIMEALL compiles a form to execute *TIMEFORM* *NUMBEROFTIMES* times (unless *INTERPFLG* is non-NIL), and then times the execution of the compiled form.

Note: If TIMEALL is called with *NUMBEROFTIMES*>1, the dummy form is compiled with compiler optimizations on. This means that it is not meaningful to use TIMEALL with very simple forms that are optimized out by the compiler. For example, (TIMEALL '(IPLUS 2 3) 1000) will time a compiled function which simply returns the number 5, since (IPLUS 2 3) is optimized to the integer 5.

TIMEWHAT restricts the statistics to specific categories. It can be an atom or list of datatypes to monitor, and/or the atom TIME to monitor time spent. Note that ordinarily, TIMEALL monitors all time and datatype usage, so this argument is rarely needed.

TIMEALL returns the value of the last evaluation of *TIMEFORM*.

(TIME *TIMEX TIMEN TIMETYP*) [NLambda Function]

TIME evaluates the form *TIMEX*, and prints out the number of CONS cells allocated and computation time. Garbage collection time is subtracted out. This function has been largely replaced by TIMEALL.

If *TIMEN* is greater than 1, *TIMEX* is executed *TIMEN* times, and TIME prints out (number of conses)/*TIMEN*, and (computation time)/*TIMEN*. If *TIMEN*=NIL, it defaults to 1. This is useful for more accurate measurement on small computations.

If *TIMETYP* is 0, TIME measures and prints total *real* time as well as computation time. If *TIMETYP* = 3, TIME measures and prints garbage collection time as well as computation time. If *TIMETYP*=T, TIME measures and prints the number of pagefaults.

TIME returns the value of the last evaluation of *TIMEX*.

(BOXCOUNT *TYPE N*) [Function]

Returns the number of data objects of type *TYPE* allocated since this Interlisp system was created. *TYPE* can be any data type name (see TYPENAME, Chapter 8). If *TYPE* is NIL, it defaults to FIXP. If *N* is non-NIL, the corresponding counter is reset to *N*.

(CONSCOUNT *N*) [Function]

Returns the number of CONS cells allocated since this Interlisp system was created. If *N* is non-NIL, resets the counter to *N*. Equivalent to (BOXCOUNT 'LISTP *N*).

(PAGEFAULTS) [Function]

Returns the number of page faults since this Interlisp system was created.

BREAKDOWN

TIMEALL collects statistics for whole computations. BREAKDOWN is available to analyze the breakdown of computation time (or any other measureable quantity) function by function.

(BREAKDOWN FN_1 ... FN_N) [NLambda NoSpread Function]

The user calls BREAKDOWN giving it a list of function names (unevaluated). These functions are modified so that they keep track of various statistics.

To remove functions from those being monitored, simply UNBREAK (Chapter 15) the functions, thereby restoring them to their original state. To add functions, call BREAKDOWN on the new functions. This will not reset the counters for any functions not on the new list. However (BREAKDOWN) will zero the counters of all functions being monitored.

The procedure used for measuring is such that if one function calls other and both are "broken down", then the time (or whatever quantity is being measured) spent in the inner function is *not* charged to the outer function as well.

BREAKDOWN will *not* give accurate results if a function being measured is not returned from normally, e.g., a lower RETFROM (or ERROR) bypasses it. In this case, all of the time (or whatever quantity is being measured) between the time that function is entered and the time the next function being measured is entered will be charged to the first function.

(BRKDWNRESULTS *RETURNVALUESFLG*) [Function]

BRKDWNRESULTS prints the analysis of the statistics requested as well as the number of calls to each function. If *RETURNVALUESFLG* is non-NIL, BRKDWNRESULTS will not to print the results, but instead return them in the form of a list of elements of the form (*FNNAME #CALLS VALUE*).

Example:

```
← (BREAKDOWN SUPERPRINT SUBPRINT COMMENT1)
(SUPERPRINT SUBPRINT COMMENT1)
←(PRETTYDEF '(SUPERPRINT) 'FOO)
FOO.;3
←(BRKDWNRESULTS)
FUNCTIONS   TIME    #CALLS  PER CALL   %
SUPERPRINT  8.261    365    0.023     20
SUBPRINT   31.910    141    0.226     76
COMMENT1    1.612      8    0.201      4
TOTAL      41.783    514    0.081
NIL
←(BRKDWNRESULTS T)
((SUPERPRINT 365 8261) (SUBPRINT 141 31910) (COMMENT1 8
1612))
```

BREAKDOWN can be used to measure other statistics, by setting the following variables:

BRKDWNTYPE [Variable]

To use BREAKDOWN to measure other statistics, before calling BREAKDOWN, set the variable BRKDWNTYPE to the quantity of interest, e.g., TIME, CONSES, etc, or a list of such quantities. Whenever BREAKDOWN is called with BRKDWNTYPE not NIL, BREAKDOWN performs the necessary changes to its internal state to conform to the new analysis. In particular, if this is the first time an analysis is being run with a particular statistic, a measuring function will be defined, and the compiler will be called to compile it. The functions being broken down will be redefined to call this measuring function. When BREAKDOWN is through initializing, it sets BRKDWNTYPE back to NIL. Subsequent calls to BREAKDOWN will measure the new statistic until BRKDWNTYPE is again set and a new BREAKDOWN performed.

BRKDWNTYPES [Variable]

The list BRKDWNTYPES contains the information used to analyze new statistics. Each entry on BRKDWNTYPES should be of the form (*TYPE FORM FUNCTION*), where *TYPE* is a statistic name (as would appear in BRKDWNTYPE), *FORM* computes the statistic, and *FUNCTION* (optional) converts the value of form to some more interesting quantity. For example, (TIME (CLOCK 2) (LAMBDA (X) (FQUOTIENT X 1000))) measures computation time and reports the result in seconds instead of milliseconds. BRKDWNTYPES currently contains entries for TIME, CONSES, PAGEFAULTS, BOXES, and FBOXES.

Example:

```
←(SETQ BRKDWNTYPE '(TIME CONSES))
(TIME CONSES)
←(BREAKDOWN MATCH CONSTRUCT)
(MATCH CONSTRUCT)
←(FLIP '(A B C D E F G H C Z) '(.. $1 .. #2 ..) '(.. #3
..))
(A B D E F G H Z)
←(BRKDWNRESULTS)
FUNCTIONS  TIME    #CALLS  PER CALL   %
MATCH      0.036    1       0.036    54
CONSTRUCT  0.031    1       0.031    46
TOTAL      0.067    2       0.033
FUNCTIONS  CONSES  #CALLS  PER CALL   %
MATCH      32       1      32.000    40
CONSTRUCT  49       1      49.000    60
TOTAL      81       2      40.500
NIL
```

Occasionally, a function being analyzed is sufficiently fast that the overhead involved in measuring it obscures the actual time spent in the function. If you were using TIME, you would specify a value for *TIMEN* greater than 1 to give greater accuracy. A similar option is available for BREAKDOWN. You can specify that a function(s) be executed a multiple number of times for each measurement, and the average value reported, by including a number in the list of functions given to BREAKDOWN. For example, BREAKDOWN(EDITCOM EDIT4F 10 EDIT4E EQP) means normal breakdown for EDITCOM and EDIT4F but executes (the body of) EDIT4E and EQP 10 times each time they are called. Of course, the functions so measured must not cause any harmful side effects, since they are executed more than once for each call. The printout from BRKDWNRESULTS will look the same as though each function were run only once, except that the measurement will be more accurate.

Another way of obtaining more accurate measurement is to expand the call to the measuring function in-line. If the value of BRKDWNCOMPFLG is non-NIL (initially NIL), then whenever a function is broken-down, it will be redefined to

call the measuring function, and then recompiled. The measuring function is expanded in-line via an appropriate macro. In addition, whenever BRKDWNTYPE is reset, the compiler is called for *all* functions for which BRKDWNCOMPFLG was set at the time they were originally broken-down, i.e. the setting of the flag at the time a function is broken-down determines whether the call to the measuring code is compiled in-line.

GAINSPACE

If you have large programs and databases, you may sometimes find yourself in a situation where you need to obtain more space, and are willing to pay the price of eliminating some or all of the context information that the various user-assistance facilities such as the programmer's assistant, file package, CLISP, etc., have accumulated during the course of his session. The function GAINSPACE provides an easy way to selectively throw away accumulated data:

(GAINSPACE) [Function]

Prints a list of deletable objects, allowing you to specify at each point what should be discarded and what should be retained. For example:

```
←(GAINSPACE)
purge history lists ? Yes
purge everything, or just the properties, e.g., SIDE,
LISPXPRINT, etc. ?
just the properties
discard definitions on property lists ? Yes
discard old values of variables ? Yes
erase properties ? No
erase CLISP translations? Yes
```

GAINSPACE is driven by the list GAINSPACEFORMS. Each element on GAINSPACEFORMS is of the form (*PRECHECK MESSAGE FORM KEYLST*). If *PRECHECK*, when evaluated, returns NIL, GAINSPACE skips to the next entry. For example, you will not be asked whether or not to purge the history list if it is not enabled. Otherwise, ASKUSER (Chapter 26) is called with the indicated *MESSAGE* and the (optional) *KEYLST*. If you respond **No**, i.e., ASKUSER returns N, GAINSPACE skips to the next entry. Otherwise, *FORM* is evaluated with the variable RESPONSE bound to the value of ASKUSER. In the above example, the *FORM* for the "**purge history lists**" question calls ASKUSER to ask "**purge everything, ...**" only if you had responded **Yes**. If you had responded with **Everything**, the second question would not have been asked.

The "**erase properties**" question is driven by a list SMASHPROPSMENU. Each element on this list is of the form (*MESSAGE . PROPS*). You are prompted with *MESSAGE*

(by ASKUSER), and if your response is **Yes**, *PROPS* is added to the list SMASHPROPS. The "**discard definitions on property lists**" and "**discard old values of variables**" questions also add to SMASHPROPS. You will not be prompted for any entry on SMASHPROPSMENU for which all of the corresponding properties are already on SMASHPROPS. SMASHPROPS is initially set to the value of SMASHPROPSLST. This permits you to specify in advance those properties which you always want discarded, and not be asked about them subsequently. After finishing all the entries on GAINSPACEFORMS, GAINSPACE checks to see if the value of SMASHPROPS is non-NIL, and if so, does a MAPATOMS, i.e., looks at every atom in the system, and erases the indicated properties.

You can change or add new entries to GAINSPACEFORMS or SMASHPROPSMENU, so that GAINSPACE can also be used to purge structures that your programs have accumulated.

Using Data Types Instead of Records

If a program uses large numbers of large data structures, there are several advantages to representing them as user data types rather than as list structures. The primary advantage is increased speed: accessing and setting the fields of a data type can be significantly faster than walking through a list with repeated CARs and CDRs. Also,

compiled code for referencing data types is usually smaller. Finally, by reducing the number of objects created (one object against many list cells), this can reduce the expense of garbage collection.

User data types are declared by using the DATATYPE record type (Chapter 8). If a list structure has been defined using the RECORD record type (Chapter 8), and all accessing operations are written using the record package's **fetch**, **replace**, and **create** operations, changing from RECORDs to DATATYPEs only requires editing the record declaration (using EDITREC, Chapter 8) to replace declaration type RECORD by DATATYPE, and recompiling.

Note: There are some minor disadvantages with allocating new data types: First, there is an upper limit on the number of data types which can exist. Also, space for data types is allocated a page at a time, so each data type has at least one page assigned to it, which may be wasteful of space if there are only a few examples of a given data type. These problems should not effect most applications programs.

Using Incomplete File Names

Currently, Interlisp allows you to specify an open file by giving the file name. If the file name is incomplete (it doesn't have the device/host, directory, name, extension, and version number all supplied), the system converts it to a complete file name, by supplying defaults and searching through directories (which may be on remote file servers), and then searches the open streams for one corresponding to that file name. This file name-completion process happens whenever any I/O function is given an incomplete file name, which can cause a serious performance problem if I/O operations are done repeatedly. In general, it is much faster to convert an incomplete file name to a stream once, and use the stream from then on. For example, suppose a file is opened with `(SETQ STRM (OPENSTREAM 'MYNAME 'INPUT))`. After doing this, `(READC 'MYNAME)` and `(READC STRM)` would both work, but `(READC 'MYNAME)` would take longer (sometimes orders of magnitude longer). This could seriously effect the performance if a program which is doing many I/O operations.

At some point in the future, when multiple streams are supported to a single file, the feature of mapping file names to streams will be removed. This is yet another reason why programs should use streams as handles to open files, instead of file names.

For more information on efficiency considerations when using files, see Chapter 24.

Using "Fast" and "Destructive" Functions

Among the functions used for manipulating objects of various data types, there are a number of functions which have "fast" and "destructive" versions. You should be aware of what these functions do, and when they should be used.

"Fast" functions: By convention, a function named by prefixing an existing function name with `F` indicates that the new function is a "fast" version of the old. These usually have the same definitions as the slower versions, but they compile open and run without any "safety" error checks. For example, `FNTH` runs faster than `NTH`, however, it does not make as many checks (for lists ending with anything but `NIL`, etc). If these functions are given arguments that are not in the form that they expect, their behavior is unpredictable; they may run forever, or cause a system error. In general, you should only use "fast" functions in code that has already been completely debugged, to speed it up.

"Destructive" functions: By convention, a function named by prefixing an existing function with `D` indicates the new function is a "destructive" version of the old one, which does not make any new structure but cannibalizes its argument(s). For example, `REMOVE` returns a copy of a list with a particular element removed, but `DREMOVE` actually changes the list structure of the list. (Unfortunately, not all destructive functions follow this naming convention: the destructive version of `APPEND` is `NCONC`.) You should be careful when using destructive functions that they do not inadvertantly change data structures.