Error Management and Debugging in *Pan I*[1]

Michael L. Van De Vanter

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

Report No. UCB/CSD 89/554

A PIPER *Working Paper*

December 1989

Abstract

Pan is a complex, interactive system that is both under constant development and designed to be user-extensible. Under these circumstances it is important to have a clear policy for detecting and managing system errors (as opposed to errors in programs being edited), mechanisms to support the policy, and guidelines for their use.

This document describes our goals for error management in *Pan*, from the various perspectives of casual user, author of extension code, and author of *Pan* system code. It discusses briefly the current implementation (*Pan I* version 3.0), how we arrived at it, and some of its problems. Finally, it proposes a set of guidelines for future work on *Pan*.

[1]Sponsored by the Defense Advanced Research Projects Agency (DoD), monitored by Space and Naval Warfare Systems Command under Contract N00039-88-C-0292, and by a gift from Apple Computer Corp.

Contents

1 Introduction 1

Roles . . . 1

Overview . . . 2

Typographic Conventions . . . 2

2 Goals 2

2.1 The User's Perspective . . . 2

User Errors . . . 3

Internal Errors . . . 3

Managing User Errors . . . 3

Managing Internal Errors . . . 5

2.2 The Extender's Perspective . . . 5

2.3 The Implementor's Perspective . . . 6

3 Implementation Issues 6

3.1 Debugging vs. Production Versions . . . 6

3.2 Error Handling Primitives . . . 7

Editor-Error and Editor-Warn . . . 7

Definitional Facilities . . . 8

Contexts . . . 9

Implementation . . . 11

3.3 Resetting the Command Dispatcher . . . 11

Errors Under Program Control . . . 11

Breaks To Lisp . . . 11

Dealing With Unwinds . . . 12

Unwinds in Other Situations . . . 12

3.4 Detecting User Errors . . . 13

Prompting . . . 13

Protective Wrappers . . . 14

Functions for Testing . . . 14

Abstraction Layers and Responsibility . . . 14

3.5 Protecting *Pan* Internals . . . 16

Permanent Tests for Internal Errors . . . 16

Debugging Assertions in COMMON LISP . . . 17

Debugging Assertions in "C" . . . 18

3.6 Catching Lisp Errors Globally . . . 18

Trapping Lisp Errors . . . 18

COMMON LISP Error Detection 19
Foreign-Function Error Detection 19
3.7 Trapping Anticipated Lisp Errors 19
3.8 Tracing 20
Allegro Function Tracing 20
Allegro Function Advising 20
Custom Tracing in COMMON LISP 20
Custom Tracing in "C" 21
3.9 Statistics 22
Allegro Statistics 22
Custom Statistics in COMMON LISP 22

4 Guidelines for Extension Programming **23**
Debugging Version 23
Responding to User Errors 23
Prompt Arguments 24
Error Detection in Primitives 25
Protective Wrappers 26
More Help 27

5 Guidelines for Internal Programming **27**
5.1 User Errors 27
Implicit Tests 27
Explicit Tests 28
Internal Documentation 28
User-Lisp Errors 29
5.2 Protect Against Unwinds 30
5.3 Assertions 31
Assertions in COMMON LISP 31
Assertions in "C" 32
5.4 Tracing 33
COMMON LISP Function Tracing 33
Allegro Function Advising 33
Custom Tracing in COMMON LISP 33
Custom Tracing in "C" 34
5.5 Statistics 35
Custom Statistics in COMMON LISP 35

6 Acknowledgements **36**

7 References 36

1 Introduction

Policies and facilities for detecting, diagnosing, and recovering from internal errors are a crucial part of any complex software system. The need is especially great in *Pan* because it is interactive, under constant development as a research testbed, and designed to be user-extensible [3][2].

For the purpose of this document, an *error* is an internal state in *Pan* in which continued execution would lead to an untenable or unacceptable state.[3] An error under this sense is somewhat analogous to, but different from, a *Lisp error* as defined by the underlying language COMMON LISP (although the two definitions overlap) [8]. In this document, *error* will mean *Pan error* unless otherwise noted.

Roles

This discussion approaches error management from three points of view, corresponding to roles a person might fill when dealing with *Pan*.

- A *user* has work to do and just wants *Pan* to help.
- An *extender* writes additional, special-purpose commands for *Pan*, without necessarily being knowledgeable about or interested in the deeper aspects of *Pan*'s internal workings. It remains a goal of the project to support this kind of extension-level programming (although the *Pan extension language* has yet to be fully developed and documented).
- Finally, an *implementor* builds, debugs, and maintains the system at all levels.

Members of the project may fill more than one role, sometimes all three at once. It is important, however, to consider the needs of the three different roles separately. All three roles are important to the success of the project, but their needs and responsibilities vary.

[2]This is a working document that has evolved through several drafts as the issues came under discussion. Some of those issues remain partially open and are noted as such. Comments and additions should be kept for possible future revisions.

[3]This kind of error has nothing to do with mechanisms in *Pan* for editing and browsing ill-formed programs.

Overview

The paper begins by articulating goals for error management and debugging in *Pan*, viewed from the three perspectives. It continues with a discussion of implementation, historical, present, and proposed. Finally, it suggests a set of guidelines to be used in future *Pan* development, both at the extension and implementation level.

Typographic Conventions

Pan runs as a COMMON LISP image in which *Pan* objects are defined in and coexist with COMMON LISP. Parts of *Pan* depend on a substrate implemented in "C", imported into COMMON LISP by the Allegro "foreign-function interface."

This paper follows typographic conventions used in the *Pan* user manual [2]. COMMON LISP objects named in the text appear in typewriter font (as in `cons`), as do blocks of lisp code. *Pan objects* (the commands, functions, macros, options, and variables that constitute the extension language) appear in sans-serif (as in Editor-Error) when named in the text, but will appear in blocks of COMMON LISP code like any other code. Note, however, that names of *Pan* objects are by convention capitalized to distinguish them from *Pan* internal functions and COMMON LISP functions.

"C" code and objects also appear in typewriter font. "C" functions imported into COMMON LISP are known in COMMON LISP by different names; naming conventions make the distinction clear. For example the "C" function `swOpClearWindow` is imported into COMMON LISP as `sw-clear-window`.

Two special marginal notes appear throughout the discussion of implementation issues (see examples at right). These notes identify special problem areas (where projected solutions are not yet clear) and unfinished areas (where additional functionality is anticipated but not complete). *Problem* *Unfinished*

2 Goals

This section describes what we hope accomplish with error management in *Pan*. These goals vary, depending on which of three points of view we take.

2.1 The User's Perspective

A user would prefer not to bother with any of this. For this person, two kinds of errors can intrude while running *Pan*: user errors and internal errors.

User Errors

A *user error* occurs when *Pan* has been instructed to do something that is unsafe, incorrect, or doesn't make sense. Examples include attempting to move the cursor past the end of a buffer, to edit a file specified with a malformed or nonexistent directory path, and to paste from an empty clipboard.

It is tempting to assume that a conscientious user given adequate documentation will commit no user errors. In fact every user will commit many. Some user errors are inadvertent, some result from *trial and error learning* [5], and some take deliberate advantage of the error mechanisms ("move the cursor forward until it stops").

A special kind of user error is the *configuration error*. Users may write files containing *Pan* commands that, when loaded, set options, bind keyboard sequences to commands, create menus, and the like. The user can load configuration files explicitly (using Load-File), implicitly at start-up time by naming a file .panrc, or implicitly when visiting files in new buffers (using Auto-Load). In the COMMON LISP implementation of *Pan*, configuration errors can manifest themselves in different ways (definitional errors, macroexpansion errors, loading errors), but they are conceptually the same to a user.

Internal Errors

In contrast, an *internal error* arises when *Pan*, in course of meeting a legitimate command from the user, arrives at a state where it cannot continue. From the user's perspective, *Pan* is simply broken, and it is somebody else's fault.

The boundary between the two classes of errors is blurry in practice, but it is important to respect this very important distinction in the mind of *Pan*'s users. *Pan*'s usability depends on handling user errors gracefully and constructively; it is also greatly enhanced by a scarcity of internal errors.

Managing User Errors

The guiding philosophy here is that the management of user errors is not only self-defense for system internals, but is also *Pan*'s primary help system.

> Think of the input not as an error, but as the user's first iteration toward the goal. [7]

The best response to user errors is good functional and user-interface design. There are two general approaches.

The best solution is a design that does not allow errors to occur in the first place. For example, *Pan*'s directory editor now makes it possible to select a file to visit by pointing, giving immediate feedback about which file was selected. With this interface, it is impossible to commit mis-specification errors for files (of which there are many kinds). As another example, *Pan*'s scrollbars make it possible to scroll to absolute (if imprecise) buffer locations by using the middle mouse button; it is impossible with this interface to specify a location that does not exist.

A closely related solution is to design the system so that it responds, not with a special kind of behavior, but in some simple and obvious way. For example, any attempt to scroll past the end of a buffer is clearly a user error, but the customary response is for the viewer to refuse (silently) to scroll further. Two aspects of the viewer display make this non-response adequate: the scroll bar bubble shows that the end of the buffer is visible, and the special display of empty lines past the buffer's end reinforces this fact.

However, a system as complex as *Pan* will always be vulnerable to user errors that demand extraordinary responses. These responses should follow the following guidelines, in decreasing order of importance.

- *Recover*. Retain the benefit of any work the user accomplished before issuing the offending command.
- *Reset*. Return quickly (and obviously) to a safe state, allowing the user to issue further commands with no unpleasant after-effects.
- *Educate*. Present the user with enough information to determine the nature of the error and, if possible, to avoid it in the future.
- *Be Nice*. Be as unobtrusive and polite as is permitted by the above goals.

In the special case of user configuration errors (see earlier discussion), the guidelines are essentially the same. There should be informative warnings, explaining the nature of the problem and what action was taken (ignored, default assumed, or whatever). Furthermore, other configuration statements in the file, both before and after, should be handled normally if they cause no errors.

Managing Internal Errors

For a user, encountering an internal error is like going to the dentist. It shouldn't happen at all, but when it does one hopes that the experience will be as productive and painless as possible. This suggests the following, again in decreasing order of importance.

- *Recover*. Retain the benefit of any work the user accomplished before issuing the ill-fated command.
- *Educate*. Make clear to the user that the error is internal; explain the cause so the user can avoid the error until it is corrected. Present enough information to help an implementor (not necessarily the user) diagnose and ultimately correct the cause.
- *Reset*. The user should be able to reset *Pan*, when desired, to a safe state, with as few unpleasant after-effects as possible.

All of this assumes, of course, that *Pan*'s documentation informs the user sufficiently about error handling and recovery.

2.2 The Extender's Perspective

Extension level commands are, from the user's perspective, just more *Pan*; they should respond to errors no differently than built-in commands. Thus, the extension language must assist and encourage the extender to follow the same guidelines and to write code that responds to errors as consistently as possible. This involves the following support.

- *Toolkit*. Commands and functions in the extension language should detect standard user errors, for example ensuring the validity of file specifications entered by the user. It should be possible for simple extension commands to rely solely on built-in mechanisms for error detection, notification, and recovery.
- *Responsibility*. The extender should be made aware of the division of responsibility for detecting user errors: which errors the extension language primitives will detect and which must be detected explicitly in extension code.
- *Protection*. Primitives of the extension language should help protect *Pan*'s internals against corruption through misuse by extension code.

When necessary for efficiency, this protection can be made optional, operating only during debugging.

2.3 The Implementor's Perspective

Goals for the implementor are somewhat the same as those for the extender, but with additional responsibility to make life as easy as possible for the extender. Additional goals include the following.

- *Consistency*. Internal error management should be standardized to the point of routine. This includes *Pan*'s behavior, as seen by users and extenders, as well as the code that will be read by other implementors.
- *Tools*. Whenever possible, standard tools should support internal error management. These are not necessarily the same tools as those used by extenders. For example they should include built-in tracing and error detection code.
- *User Interface*. Special user interface for implementors should enable dynamic control of *Pan*'s debugging and tracing facilities at a relatively fine granularity.

3 Implementation Issues

This section presents some of the technical issues involved with error management and debugging in *Pan*. It discusses some special problems along with existing solutions, unresolved difficulties (flagged by a special marginal note, see example at right), and unfinished work (also flagged by a special marginal note, see example at right). *Problem* *Unfinished*

This section does not discuss specific guidelines for programming in *Pan*; those appear in the following two sections.

3.1 Debugging vs. Production Versions

Configuration switches allow us to build versions of *Pan* that embody different choices in the tradeoff between safety and speed. We usually install a *production* version in the public binary file; for optimal performance, this version performs the minimum permissible degree of error checking (still a substantial amount). For development work we keep a *debugging* version in which switches are set the other way.

Configuration switches exist for the following choices (most of which are discussed in more detail throughout this section).

- Compile all COMMON LISP code with minimum regard for speed and maximum regard for internal error detection (vs. the opposite choices).
- Make load-time requests that improve error detection in the foreign-function interface at the expense of speed.
- Include with compiled COMMON LISP code all *Pan* **debug-assert** forms for internal error detection. When included, a separate switch for each module dynamically controls evaluation of the forms.
- Include with compiled "C" code all *Pan* tests for internal errors.
- Include with compiled COMMON LISP code all *Pan* **debug-trace** forms for custom tracing. When included, a separate switch for each defined trace controls trace output dynamically.
- Include with compiled "C" code *Pan* tracing statements for reporting on the foreign function interface. When included, a separate switch for each "C" module controls "C" tracing dynamically.

In the rest of this section, any error detection mentioned without explicit qualification is performed in every version of *Pan*, independent of these configuration switches.

3.2 Error Handling Primitives

This section introduces *Pan*'s two primitive functions for handling errors that can be detected under program control. Although this document emphasizes the management of errors detected at *run time*, while a user is running *Pan* to get work done, there are special problems associated with error management in other contexts. These primitives address some of these problems by combining a uniform client interface with context-sensitive behavior.

Editor-Error and Editor-Warn

Editor-Error and Editor-Warn, as their names suggest, are modeled on COMMON LISP **error** and **warn** respectively.[4] These two primitives are designed

[4]In some contexts Editor-Error and Editor-Warn behave identically to their COMMON LISP counterparts.

to be used uniformly for error reporting throughout *Pan*; their behavior changes dynamically to suit different contexts (more on contexts below).

In *Pan* terminology Editor-Error *signals* a fatal error, like its COMMON LISP counterpart. When the documentation for a *Pan* function or command mentions that "an error is signaled," it is understood that this happens with a call to Editor-Error. Also like COMMON LISP `error`, Editor-Error never returns to its caller.

Editor-Warn embodies the same error reporting strategy as Editor-Error in every context but allows execution to continue, in the manner of COMMON LISP `warn`.

Note that the primitive **Announce** is closely related to Editor-Error and Editor-Warn. Its behavior changes dynamically, with a reporting strategy that follows the other two in every context. **Announce**, however, is intended for informational messages, not error reporting, so it will not be mentioned further here.

Definitional Facilities

Elaborate mechanisms support the definition of *Pan objects*, the commands, functions, macros, options, and variables that constitute the extension language. Unfortunately, the flexibility of the COMMON LISP model for interpreting, compiling, and loading code, while generally advantageous, presents special problems for handling errors detected by these facilities.

For example implementors use these facilities to create basic objects in *Pan*'s infrastructure. These definitions are compiled and loaded into *Pan* binaries, with the effect that basic objects are predefined and immediately available for use at run time. But a naive user might invoke the same definitional facilities in a configuration file that gets auto-loaded during a session with *Pan*. An error detected here is more appropriately handled like a run-time user error, with suitable warnings, defaults, recovery, and resetting of the command dispatcher.

Separate definitional mechanisms [4] [1] produce the *language descriptions* that drive the language-based components of *Pan*. Unlike *Pan* object creation, these mechanisms benefit from off-line preprocessing. Like *Pan* object creation they may be either preloaded into *Pan* binaries or loaded at run time, with the possibility of error in either context.

Pan's definitional mechanisms will eventually handle errors uniformly with calls to Editor-Error and Editor-Warn, relying on the behavior of the primitives to respond appropriately in the current context.

Contexts

Although most of the discussion in this paper emphasizes run-time issues, we have identified four different contexts whose requirements for error handling are distinct. Note that this discussion glosses over some subtle implementation issues in COMMON LISP (compiling vs. loading vs. macroexpansion vs. evaluation); these distinctions are important to the correct implementation of Pan's error handling primitives, but not to this discussion of requirements.

Run Time. This is the context most at issue in this paper; it motivated the earlier discussion of goals for error management. *Pan* is event-driven, so the run-time context can be viewed as a series of computations, each initiated by *Pan*'s *dispatcher* in response to an event (a user action).[5] A call to Editor-Error in this context terminates the processing initiated by the most recent event (see the next section on unwinding for more discussion of how this happens) and resets the dispatcher to await the next event. The error report in this context is typically a short message appearing in the annunciator of the active viewer, accompanied by a beep or canvas flash. A call to Editor-Warn reports the same way, but allows the computation to continue.

Some care is taken in *Pan* to ensure that any event-initiated computations can be terminated gracefully, but the actual thread of control is a bit more complex and its error recovery requirements less well-defined than the simple model presented above. Once the dispatcher has translated an event into a request for a specific command, it executes the following sequence of functions: *Problem*

1. Zero or more functions that have been registered with a special *before hook* for command dispatch.
2. An optional *before daemon* that may be defined for the specified command.
3. The specified *command*.
4. An optional *after daemon* that may be defined for the specified command.

[5]This model is confounded slightly by the existence of *timer* events. These do not originate directly from user events, but rather from elapsed time since the most recent user event. This discussion ignores the distinction becauses (a) *Pan*'s dispatcher manages these much the same way as user events, and (b) only very specific, low-level functions are performed in response to timer events.

5. Zero or more functions that have been registered with a special *after hook* for command dispatch.

The command dispatcher in the current implementation terminates the entire sequence in response to any call, anywhere in the sequence, to Editor-Error. It isn't clear, however, that this is the most desirable behavior.

Build Time. Some of *Pan*'s mechanisms (especially the definitional facilities) execute during the compilation, loading, and dumping that it takes to produce *Pan* binaries. In this context, the reporting and recovery behavior needed at run time is inappropriate, so Editor-Error and Editor-Warn behave essentially like `error` and `warn` respectively. A call to `error` reports the error message to standard output and drops into a Lisp break; `warn` reports the same way and continues.

Run-time Loading. There are normal situations in which *Pan* code is loaded into a running *Pan*. At present, the error handling context is simply inherited from the standard run-time context, but this is inadequate. The first error encountered in the file causes termination of the load, whereas it would be more convenient for the user/developer to have the definitional mechanisms attempt to continue after failing on a single form. Furthermore, the annunciator is inadequate for displaying the kinds of messages that can be generated. The error handling primitives should exhibit special behavior in this context, continuing loading as much as possible and recording error messages in a log of some sort. *Unfinished*

In contrast, run-time loading of language descriptions is now carried out in a special error handling context that is specialized for this purpose.

Batch Execution. *Pan* can execute certain functions in a batch mode, where ordinary run-time code executes without the window system and without user interaction. In this context, Editor-Error reports to standard output and terminates the COMMON LISP computation entirely via `excl:exit`. It also returns an error code to the shell so that encompassing makefiles can detect the failure.

Implementation

The primitive functions Editor-Error and Editor-Warn are implemented as indirect calls to functions bound to special variables.[6] Bound by default to `error` and `warn` respectively, these variables are dynamically rebound in contexts other than build time. This mechanism permits easy extension; new error handlers for new contexts can be bound dynamically as needed.

3.3 Resetting the Command Dispatcher

As described in the discussion of error contexts, *Pan* responds synchronously at run time to user *events* (keystrokes, mouse button clicks, menu selections, and the like) via an internal command dispatcher. In response to some events the dispatcher invokes a *Pan command*, waits for it to complete, and then awaits the next event. This discussion considers only errors that take place during command execution (see the earlier discussion of the run-time context for an explanation of those that don't).

When *Pan* code detects a user error at run time, it should terminate execution of the current command, report to the user, and reset the dispatcher to await the next event. Writing explicit control threads to support this behavior would be enormously complex. In *Pan* commands may call other commands, macros, and functions; errors can be detected at different levels of nested function calls.

Errors Under Program Control

Pan simplifies the thread of control by exploiting the Common Lisp `throw` mechanism. A run-time call to Editor-Error does not return. It prints a specified error message, beeps (or flashes the canvas), and unwinds the call stack with a distinguished `throw`. The dispatcher has the corresponding `catch`,[7] and resets appropriately.

Breaks To Lisp

Unfortunately not all Lisp errors can be detected and managed explicitly by *Pan* code (more about this later); every user will eventually fall into a

[6] Announce, mentioned earlier, is implemented the same way .

[7] The dispatcher actually uses the macro Execute-Protect, which contains the corresponding `catch`. Thus the same protection is available in contexts other than the dispatcher.

COMMON LISP "break loop." This allows an implementor to diagnose the problem by viewing the call stack on top of the dispatcher.

The user can reset the dispatcher from a break loop by typing `:resume`, presumably in the shell where the *Pan* process originated; this translates into a call to Editor-Error. In extreme failures, when *Pan* fails to reset, `:panic` typed to the break loop results in an attempt to save files from all modified buffers.

Dealing With Unwinds

Since error detection in *Pan* is distributed, programs must be ready for any *Pan* function to unwind instead of returning normally. In particular, no unpleasant side-effects (like open scratch files) should remain after an unwind.

Internally, COMMON LISP mechanisms such as dynamically scoped variables, `unwind-protect`, and `with-open-file` make it possible to deal effectively with unwinds. A number of *Pan* macros use these mechanisms to guard parts of *Pan*'s internal state against unwinds; these include at present:

Cursor-Motion-Protect
Save-Cursor-Excursion
With-Buffer
With-Buffer-Scope
With-Class-Scope
With-Help-Stream
With-Mouse-Icon
With-Text-Protection-Suspended
With-Variable-Binding

Ordinary extension code should rely on these macros and should never use the underlying mechanisms directly.

Unwinds in Other Situations

The user is often allowed to cancel a partially executed command, typically via a "Cancel" button on popup prompters. This is implemented with a call to Editor-Error, printing the message "Cancelled."

The command Abort is a shortcut for terminating commands legitimately under program control. It is useful when the thread of control would be greatly complicated by an explicit termination; it has much the same effect as Editor-Error, but with neither message nor beep.

3.4 Detecting User Errors

Graceful recovery from user errors requires that explicit tests be made whenever an error is possible and that Editor-Error be called when one is detected. Tests for user errors are distributed throughout the extension language. *Pan* functions often rely on other commands and functions to detect various errors.

A great number of commands at the extension level need no explicit tests at all. This is an important aspect of the extension language, and it should be maintained carefully. This section describes some of the error detection mechanisms currently in place.

Prompting

Many tests for user errors are designed to ensure the well-formedness of user data, typically supplied in response to prompters. *Pan*'s command definition mechanism allows arguments to be declared specially, with the result that the user will be prompted automatically for the desired data whenever the command is invoked by the dispatcher. The prompting mechanism guarantees well-formedness of any value it returns, unwinding with a call to Editor-Error when it cannot return such a value.

Well-formedness is defined by an optionally specified "type" for each argument (the default type is :string). Types supported at present include

:command	Specification of a *Pan* command
:form	A Lisp form
:integer	An integer
:key-sequence	Specification of a key-stroke sequence
:pathname	A file specification
:string	Any text string[8]
:symbol	A Lisp symbol
:yes-or-no	Boolean answer to a question

Additional prompt types can (and should) be added whenever we observe that one or more commands routinely apply additional predicates to prompt argument values; additional prompt types anticipated at present include `:buffer`, and `:option-variable`. *Unfinished*

[8] In the current implementation, tabs may be entered, but no other non-graphic characters.

A number of additional options allow the prompting for each argument to be customized as needed, for example with messages and displayed default values.

When the implementation of a *Pan* command is too complex to use this kind of declarative argument prompting, the prompting function for each type (for example Prompt-For-String and Prompt-For-Pathname) may be called explicitly. In every case the well-formedness of the return value is guaranteed. Each prompting function has (or should have) documentation that explains the criteria in effect for the type.

Protective Wrappers

Pan Macros can be written to provide safe environments for various kinds of operations, further reducing the need for extension code to protect explicitly against user errors.

The single example at present is Cursor-Motion-Protect, which ensures the consistency of cursor-related data structures. Extension level code may, within a call to Cursor-Motion-Protect, ignore buffer boundaries while moving about. Any attempt to move illegally aborts with a low-level call to Editor-Error; during the recovery Cursor-Motion-Protect (using `unwind-protect`) restores crucial data structures.

Functions for Testing

Even when operations are known to be safe (for example Delete-Region signals an error if the buffer is protected), the semantics of a particular command may require an explicit test. The extension language includes many variables and predicates that make this kind of information visible, for example Text-Protected, Buffer-Empty?, Selection-Valid?, and EOB? (is the cursor at end of buffer?).

Abstraction Layers and Responsibility

A difficulty with all these mechanisms is keeping them in balance, maintaining a clear division of responsibility for user error testing. We would like the abstraction layers of the system to be arranged (and documented) so that that during any command execution *Problem*

- a test for every potential user error is made, but
- no test is performed redundantly.

Although existing code is improving as it gets rewritten, the problem is confounded by the many abstraction layers in some cases, the lack of internal documentation in other cases, and the variety of user contexts that determine which correctness criteria are in effect. A short example drawn from the current implementation (version 3.0) suggests the complexity.

> The top level command for visiting a file, Visit-Named-File, declares a prompt argument of type `:pathname`. When the user invokes Visit-Name-File, Prompt-For-Pathname is called implicitly to collect an argument from the user. Prompt-For-Pathname ensures that the argument is a well-formed file specification (returned as a COMMON LISP pathname), and has a directory component that exists. However, Visit-Named-File must additionally test to ensure that the pathname (if it specifies an existing file at all) does not specify a directory, since a directory is a legitimate file in some contexts[9] but cannot be visited like other files.[10]
>
> Visit-Named-File calls Visit-File, which first ensures that no buffer already contains the specified file. Visiting the same file in two *Pan* buffers is considered to be an error.[11] Visit-File does not presume that the file exists, unless the user has requested that it be visited "read-only" in which case it signals an error should the file not exist. It does assume implicitly that the file (if it exists) is not a directory.
>
> In contrast, Insert-File uses the same prompt argument mechanism as Visit-Named-File, but insists that the specified file exist and not be a directory, signaling an error otherwise.
>
> Both Visit-File and Insert-File may call Create-Region-From-File, which assumes that the specified file exists. Create-Region-From-File in turn checks that the file is readable before calling `internal-create-region-from-file`, which would fail if the file were not readable. `internal-make-region-from-file` in turn checks that the file has nonzero length before calling one of the file reading functions written in "C", none of which are intended to work on zero length files.

[9] That this ambiguity is an artifact of UNIX file system semantics makes it no less a problem.

[10] Directories actually can be visited with *Pan*'s *directory editor*; Visit-File invokes the directory editor when the specified directory turns out to be a directory.

[11] This error is not always detectable, again because of UNIX file system semantics.

The situation is actually considerably more complex, given the number of different interactions possible with the file system, and the ongoing evolution of the command set. Existing abstraction layers do not satisfy the criteria precisely. They usually err on the side of conservatism at the cost of redundant tests.

3.5 Protecting *Pan* Internals

The importance of handling user errors gracefully and informatively justifies the use of considerable resources, both in implementation effort and in run-time overhead. A suggested above, we would prefer to detect *any* potential user error, describe the problem precisely, and reset reliably.

The same isn't true for potential misuse of the extension language by extension-level code. We owe the extender (and implementor for that matter) a certain amount of safety and resilience, but not at too great an expense. Furthermore, it is impossible to protect against *any* potential error, and it would be a waste of time to try. This section discusses how we can protect against these kinds of errors.

Permanent Tests for Internal Errors

When the user invokes one of *Pan*'s commands, a certain amount of error detection comes free. Commands (when executed by the dispatcher) use the prompt module to guarantee well-formed arguments, and an error is signaled when the user fails to supply them. It isn't always clear, however, how much argument checking to do in contexts where only programming errors could be at fault, and how to respond when an error is detected.[12] *Problem*

Sometimes the overhead of a permanent, explicit test is justified. This might be the case when

- a particular function appears especially vulnerable to misuse,
- the potential for disaster is great, or
- the likely symptoms of an error would be misleading.

In some cases, a call to Editor-Error might be a sufficient response to an internal error detected this way. This only makes sense, however, when the

[12]This applies also to commands when called internally as functions, since automatic type checking is only applied to arguments supplied by prompting the user.

problem can be diagnosed and explained adequately in the error message, since the stack will not be available for examination afterward.

We have at present no conventional mechanism for permanent tests (as opposed to debugging assertions, see below) in situations where a call to Editor-Error is not appropriate. Some error tests in the current implementation, which may fall in this category, now call Editor-Error. It is an open question whether this category is necessary at all. If so, it should be carefully articulated as a guideline; otherwise Editor-Error and debugging assertions will presumably suffice. *Unfinished*

Debugging Assertions in COMMON LISP

The most general mechanism for protecting *Pan*'s internals against misuse is *Pan*'s `debug-assert` and its supporting machinery. This macro generalizes COMMON LISP `assert` with a single additional argument (a control variable), placed before the standard arguments. The operation of `debug-assert` may be controlled in two ways.

- A *Pan* configuration switch[13] determines whether `debug-assert` code will be included in compiled *Pan* code.
- A control variable in each *Pan* module dynamically controls whether assertions in the module (when they have been included at all) are evaluated.[14]

Debugging assertions are typically included in debugging versions of *Pan*, but not included in production versions. The default setting for all control variables is `t`, so that when assertions are included, all debugging assertions are evaluated. There should be, but isn't yet, a user interface[15] for assertion control variables; it should be possible to inspect their values (a list in the help buffer would suffice) and adjust them individually. *Unfinished*

Debugging assertions also document important assumptions and invariants; these are sometimes more concisely expressed as Lisp predictates than as prose. The use of `debug-assert` is is not as prevalent as it should be, and there are probably still tests with calls to Editor-Error that should be replaced by uses of `debug-assert`. *Unfinished*

[13]Set *make* variable LISP_DEBUG to `t` in src/lib/make/def.allegroforms.make.

[14]For example, `*region-debug*` controls assertion evaluation in the "region" module.

[15]User interfaces for debugging mechanisms are loaded dynamically when needed from *Pan* library module "pan-debug."

Debugging Assertions in "C"

An entirely separate mechanism supports debugging assertions in *Pan*'s "C" modules. These are coded as explicit tests for erroneous conditions and calls to `printError`[16] when one is detected. `printError` is a simple wrapper for `fprintf`, directing the message to the stream `stderr`.

Most debugging assertions are simple range checks on arguments, since *Pan*'s "C" modules retain very little state between calls.

The operation of these debugging assertions is controlled by conditional compilation. A configuration switch[17] determines whether assertion checking code will be included in compiled "C" modules. There is no mechanism for controlling assertion checking dynamically.

3.6 Catching Lisp Errors Globally

A *Lisp error* originates from the underlying COMMON LISP, when an error is "signaled" in the nomenclature of the language definition [8]. These are often the result of bugs in *Pan* code, but there are also certain user errors that can only be detected by the Lisp errors they produce, for example attempting to execute a Lisp form with mismatched parentheses.

Trapping Lisp Errors

An early FRANZ LISP implementation of *Pan* was able to bind custom error handling functions to two different error traps. These error handlers had access to information about the nature of the error, from which they could diagnose the problem and choose how to respond [6]. One possible response was just a call to Editor-Error, and tests for some user errors presumably relied on this mechanism.

The Allegro implementation of COMMON LISP is far less flexible.[18] The only tool for trapping Lisp errors is `excl:errorset`. Unfortunately, this trap catches every kind of Lisp error and provides no information in the program about the nature of the error.[19] This is much too coarse-grained to serve as a global error handler. *Problem*

Thus, a Lisp error in the current implementation suspends *Pan* and drops the user into a break loop. This should not be allowed to happen

---

[16] `printError` is linked from the *Pan* "C" library `cdebug`.

[17] Include string "`-DDEBUG`" in the definition of *make* variable `OPTS` in src/pan/Makefile.

[18] We believe that future releases of Allegro COMMON LISP will be better in this respect.

[19] An optional argument requests that informative messages be printed to the console.

in any situation where a Lisp error is anticipated or expected; anticipated errors should be detected by local uses of `excl:errorset` (see below).

When an unanticipated Lisp error does occur, the user can usually reset the *Pan* dispatcher from a break loop by typing `:resume`, which translates into a call to Editor-Error.

Common Lisp Error Detection

The Allegro implementation of Common Lisp supports a certain amount of internal rund-time checking[20] in compiled code. A *Pan* configuration switch[21] requests (using Common Lisp `proclaim`) maximal error detection at run time.

Foreign-Function Error Detection

Allegro's "foreign-function interface" supports type checking on calls to "C" functions.[22] A *Pan* configuration switch[23] requests run-time argument type checking on foreign function calls; this must be specified at the time "C" object modules are loaded into Common Lisp, and it cannot be controlled dynamically.

3.7 Trapping Anticipated Lisp Errors

Allegro's `excl:errorset` is the only way for *Pan* to trap Lisp errors, but it provides no diagnostics beyond success or failure. Thus, it is only effective in narrow contexts where diagnosis is relatively simple, usually the execution of a single Common Lisp function. In ordinary cases, a call to Editor-Error is the appropriate response to an error detected this way.

When the diagnosis isn't obvious from the static context of the call, `excl:errorset` can be instructed (by an optional argument) to print diagnostic messages on the console. We would rather trap the the error and announce the message using *Pan* facilities, but we cannot do so straightforwardly in the current Allegro implementation. During run-time file loading, for example, *Pan* responds to any Lisp error by announcing "Error during loading," and the user must examine the console for details.

We might experiment with redirection of the stream to which this kind *Unfinished*

[20] It isn't clear how much.

[21] Set *make* variable `OPT_COMPILE` to `nil` in src/lib/make/def.allegroforms.make.

[22] Unfortunately, Allegro neglects to type check return values.

[23] Set *make* variable `OPT_LOAD` to `nil` in src/lib/make/def.allegroforms.make

of information is written by the COMMON LISP implementation. It may be possible to capture it and place it in a special log maintained inside *Pan*, a log whose visibility would be more easily managed than the console.

3.8 Tracing

Tracing is a valuable aid to debugging. *Pan* supports several kinds of tracing, but they have not been well integrated and they lack a uniform user interface.

Allegro Function Tracing

The Allegro implementation of COMMON LISP provides a mechanism for selectively tracing function calls and returns. We haven't built a convenient user interface for this mechanism, but one could be added.[24] *Unfinished*

The default behavior of Allegro function tracing is somewhat limited by the fact that many internal objects in *Pan*, used as argument and return values, are not intelligibly printed by the default scheme. This can be improved somewhat by adding custom printers to *Pan*'s internal data structures,[25] but we have done so for only a few data structures. *Unfinished*

Allegro Function Advising

The Allegro implementation of COMMON LISP provides a mechanism for attaching before- and after-daemons to selected functions. We have used this mechanism very little and not at all in support of error management and debugging. For debugging, this could represent a useful compromise between simple function tracing (above), since it offers more flexibility, and custom tracing (below), since they can be added dynamically to running code. We haven't built a convenient user interface for this mechanism, but one could be added. *Unfinished*

Custom Tracing in COMMON LISP

Sometimes, built-in COMMON LISP function tracing is inadequate; it may print too much, too little, or at an inappropriate level of abstraction. *Pan*'s "debug-trace" module is a customized tracing facility that allows special tracing code to be added where appropriate.

Like debugging assertions, custom tracing can be controlled in two ways.

---

[24] Any interface like this would reside in the "pan-debug" library module.

[25] The COMMON LISP **defstruct** takes a **:print-function** argument.

- A *Pan* configuration switch[26] determines whether custom tracing code will be included in compiled *Pan* code.

- A name for each programmer-defined "trace" allows the programmer to control dynamically whether the specified trace (when it has been included) is performed.

Pan's "debug-trace" module is at present very little used. *Unfinished*

Custom Tracing in "C"

An entirely separate mechanism supports tracing in *Pan*'s "C" modules. These are coded as explicit calls to `printTrace` (for strings) and `putTrace` (for characters).[27] These are simple wrappers for `fprintf` and `putc` respectively, directing output to the stream `stderr`.

Nearly every function exported from *Pan*'s "C" modules[28] contains tracing statements. Most trace statements print the name of the "C" function on entry and as many of the arguments as can be printed meaningfully. When there is a return value, a separate trace statement prints this too. In the few cases where a "C" module retains interesting state, trace statements may print additional information, for example whether a call to `sw-cursor-set` "hits" the cached location of the displayed cursor.

Custom tracing in "C" modules is controlled in two ways.

- A *Pan* configuration switch[29] determines whether tracing code will be included in compiled "C" modules.

- A static variable in each "C" module dynamically controls tracing in the module (when it has been included). A special function in each "C" module allows this variable to be set.[30]

The "pan-debug" module in *Pan*'s run-time library contains a user interface to "C" tracing. When loaded, "pan-debug" adds special menus that allow tracing to be set, module-by-module or all at once.

It isn't possible at present to redirect output from "C" module tracing, *Unfinished*

---

[26]Set *make* variable `LISP_DEBUG` to `t` in src/lib/make/def.allegroforms.make.

[27]`printTrace` and `putTrace` are linked from *Pan* "C" library `cdebug`.

[28]There are currently around 100 exported "C" functions.

[29]Include string "`-DTRACE`" in the definition of *make* variable `OPTS` in src/pan/Makefile.

[30]For example, tracing in the "C" frame module is controlled by the "C" function `swFrameSetTrace`, exported into COMMON LISP as `sw-set-frame-trace`.

but it should be. In particular, it should be integrated with both kinds of tracing in COMMON LISP so that all tracing output appears in the same stream.

3.9 Statistics

Another valuable aid to debugging, somewhat related to tracing, is the general ability to gather statistics about the frequency with which various internal events occur.

Allegro Statistics

Allegro COMMON LISP provides a profiling mechanism that can count and and report the number of times specified functions are called. This can be useful for discovering which parts of the system are being most heavily, but is relatively inflexible. See the Allegro User Guide for details on how to use it.

Custom Statistics in COMMON LISP

Pan's "statistics" module allows more detailed statistics gathering code to be added where appropriate. The facility supports simple counters, ratios, percentages, sums, differences, and histograms.

Like debugging assertions and custom tracing, statistics gathering can be controlled in two ways.

- A *Pan* configuration switch[31] determines whether custom statistics gathering code will be included in compiled *Pan* code.
- A name for each programmer-defined "statistic" allows the programmer to control dynamically whether the specified statistic (when it has been included) is gathered.

The *Pan* option Print-Stats-On-Termination, when set to t, requests that all defined statistics be printed by *Pan* when the user exits.

Pan's "statistics" module is at present used very little outside of the semantic analysis (Colander) modules. *Unfinished*

[31] Set *make* variable LISP_DEBUG to t in src/lib/make/def.allegroforms.make.

4 Guidelines for Extension Programming

This section describes *Pan* tools and conventions for error management in *extension level* code. A premise of *Pan*'s extension language is that relatively little explicit attention to error management should be necessary if you follow a few basic guidelines.

This discussion presumes knowledge of basic definitional mechanisms in *Pan*'s extension language, including

Define-Char-Class
Define-Command
Define-Constant
Define-Flag-Variable
Define-Function
Define-Hook
Define-Hook-Function
Define-Macro
Define-Option-Variable
Define-Variable

Debugging Version

Use a *debugging* version of *Pan* for developing and testing new commands. This provides much better internal error checking, and it enables *Pan*'s custom tracing and statistics gathering facilities.

Responding to User Errors

You must write your code to detect any situation where a course of action requested by the user is nonsensical or would be dangerous to the internal workings of *Pan*. When this happens, your code must enforce *Pan*'s policy for responding to user errors. This policy is to

1. *Abort* the currently executing command.
2. *Restore* buffer contents to a safe state so that the user loses no work.
3. *Beep* or flash to alert the user.
4. *Announce* the nature of the error in terms the user can understand.
5. *Reset* the command dispatcher to await the next command.

When writing extension code, you are obliged only to detect and diagnose user errors; a call to Editor-Error does the rest. For example, *Pan*'s text cursor must not move past the end of a text buffer (EOB).

```
(Define-Command Next-Character()
  "Move cursor forward."
  (when (EOB?)
    (Editor-Error "Cursor at EOB"))
  ... )
```

A call to Editor-Error does not return; instead, it aborts the command that is currently executing and unwinds all procedure calls on the stack back to the dispatcher. The dispatcher resets, and awaits the next user action. The first argument to Editor-Error is a COMMON LISP format string. Editor-Error applies `format` to the string and any remaining arguments, and *announces* the result in the panel of the active viewer, along with a beep.

Designing error messages is an art; it demands careful thought and good judgement. Messages must be terse, so they will fit into a *Pan* viewer's one-line *annunciator*, but they must help the user identify precisely the offending action. They must point out that a mistake has been made, but they must not appear surly or insulting. They must accomplish all this using terminology that the user will understand, not the terminology of the implementation. They should the object of continuing refinement, since almost nobody gets them right *a priori*.

In most cases you can avoid the problem altogether by exploiting error management facilities already present in *Pan*'s extension language. Some of the guidelines below suggest how.

Prompt Arguments

A *Pan* command, when invoked, may ask the user to specify one or more arguments. Error detection in the presence of user input is crucial but tiresome. *Pan*'s "prompt" module can do most of it for you, and you can specify most of it declaratively.

For example, the command Insert-File asks (declaratively, in the lambda list) the user to enter text that specifies a file, and then, if valid, converts the textual specification into a COMMON LISP *pathname* value.

```
(Define-Command Insert-File
           ((file :prompt :pathname "Insert file:"))
  ... )
```

The keyword `:pathname` is a "type" specification to the prompt module. It requests that user input be subject to well-formedness criteria appropriate to the type, and that an appropriate COMMON LISP value be returned. If the user input is unsuitable, the prompt module calls Editor-Error with a diagnosis of the problem. Thus, the prompt module *guarantees* that the argument `file` in Insert-File is a well-formed pathname.

The prompt module supports the following types; more will be added.

:command	Specification of a *Pan* command
:form	A Lisp form
:integer	An integer
:key-sequence	Specification of a key-stroke sequence
:pathname	A file specification
:string	Any text string[32]
:symbol	A Lisp symbol
:yes-or-no	Boolean answer to a question

Note, however, that this particular kind of type-checking (supported by the "prompt" module) is not in effect when you call Insert-File as a *function* from other code. In this case your code must guarantee the validity of any argument you supply; commands like Insert-File are counting on it.

Error Detection in Primitives

Many functions in *Pan*'s extension language protect against fundamental user errors. If you are willing to have your extension command terminated abruptly when such an error is detected (as usual, via calls to Editor-Error), then you can just use these functions assuming the best.

For example, any function that changes the contents of a buffer's text representation must ultimately use one (or both) of Insert-Region or Delete-Region. These two call Editor-Error immediately when a buffer's text has been "protected." No other functions need to make this test.

Having a command terminated abruptly is a bad idea if it would leave behind unpleasant side-effects. This generally doesn't happen in extension code, as long as you use protective wrappers in the right places (see below) and you don't open files explicitly. If you can't make your command inherently immune to unwinds, then you must make explicit tests (checking the local value of option variable Text-Protected in the buffer for example).

[32] In the current implementation, tabs may be entered, but no other non-graphic characters.

Protective Wrappers

Certain *Pan* operations are inherently complex and full of delicate side-effects. An unwind from the middle of a cursor movement, for example, could leave any number of internal data structures (and the screen display) in disastrous disagreement. Several *Pan* macros guard specifically against this kind of damage; only a few are important at the extension level.

Whenever you intend to move the cursor about, do so within the scope of Cursor-Motion-Protect. This guarantees the integrity of cursor, text, and screen data, even in the presence of calls to Editor-Error (or other unwinds).

```
(Define-Command Next-Character()
  "Move cursor forward."
  (Cursor-Motion-Protect
    <go ahead and try it>
  ... ))
```

When you expect a command to run long enough that the user might notice the delay, changing the appearance of the mouse cursor is a helpful courtesy. However, you must eventually change the cursor back to what it was before, even if a call to Editor-Error unwinds. The macro With-Mouse-Icon changes the cursor and guarantees that it will be restored in any case.

```
(Define-Command Grind()
  (With-Mouse-Icon :think "starting to grind ..." "done"
    ... ))
```

Text protection (controlled by the option variable Text-Protected) disallows any changes to the contents of a buffer's text representation. When you want to modify a buffer in spite of possible text protection, do so within a call to macro With-Text-Protection-Suspended. This guarantees that buffer protection will be restored to its former state (on or off), even if a call to Editor-Error unwinds.

```
(Define-Command Flush-Buffer()
  (With-Text-Protection-Suspended
    (Delete-Region (Make-Buffer-Region))))
```

Macro With-Variable-Binding operates analogously for general *Pan* variables. Use it to change the value of a variable for the duration of an operation; the prior value is guaranteed to be restored upon completion.

```
(Define-Command Poke-Around()
  (With-Variable-Binding (Load-Verbose nil)
    ... ))
```

More Help

These guidelines are necessarily vague in places and are subject to change. Here are some other ways to use these facilities productively.

- Review existing code in the user level editing code of *Pan*. Files named with the suffix `-cmds` usually contain extension level code.
- Use *Pan*'s internal documentation features, available through the help system.
- Show your code to one of *Pan*'s implementors and ask for suggestions on error management.

5 Guidelines for Internal Programming

Guidelines for internal programming begin with those for extension level programming, described in the previous section. Those guidelines are also in effect for internal programming and will not be repeated here. This section concentrates on internal issues that are intentionally hidden from the extension programmer.

5.1 User Errors

The best method for handling user errors is to design the system and user-interface carefully enough that they cannot occur; this approach was mentioned, along with some examples, earlier in the paper. The remainder of this section describes how to handle user errors that do not submit to this optimal solution.

It is a *Pan* policy that all user errors be detected and handled appropriately. Editor-Error implements most of the mechanisms for response; the difficult part is detection and diagnosis.

Distinguishing between user errors and internal errors can be difficult at times; furthermore, the criteria sometimes change as the system evolves.

Implicit Tests

Use special prompt arguments for commands when possible. This greatly simplifies validation of user input.

Avoid an explicit test when you can rely on low-level error detection mechanisms (for example, built-in guards against illegal cursor movement and text protection). Just assume the best (but be aware of potential stack unwinding). Your code will be more readable, and redundant tests will be minimized.

When you call a higher-level function, consult the documentation (and code) to determine what user errors, if any, are detected. Add explicit tests only for user errors not detected by functions you call. On the other hand, if a function tests for an error that cannot possibly occur in your context, reconsider your choice of function. Perhaps there is a lower level function that will not perform the test, but which relies on its clients to prevent the error. If there is no such lower level function, consider restructuring that part of the system so there is one. Examples of these layers may be found among file-handling commands and functions.

Explicit Tests

Simplify control structures by exploiting the fact that a call to Editor-Error does not return. Good style has the error tests, when possible, inserted *into* the thread of control, rather than being part of it. In practice this means that calls to Editor-Error should appear most often in `when` and `unless` forms and seldom in `if`, `cond`, and `case` forms.

```
(Define-Function Whiz-Bang (object direction)
  (when (bogusp object)
    (Editor-Error "Can't whiz a ~S" object))
  (unless direction
    (Editor-Error "Can't whiz without direction"))
  ...)
```

Internal Documentation

Pan's elaborate internal help system makes information available at run time; it can be configured to present just information most useful to users or all information that might be relevant to implementors too.

Among other features, each *Pan* object may have a documentation string associated with it. This is analogous to the documentation strings for COMMON LISP objects, but with a powerful user interface for browsing the information.

Use this feature for every *Pan* object you create. For functions and commands, add to the ordinary specification (arguments, return values, side effects) additional information about boundary conditions. Under which conditions will the function "signal an error" (understood to mean a call to Editor-Error)? What preconditions are simply assumed?

User-Lisp Errors

Certain kinds of user errors can be detected only by evaluating a COMMON LISP form and awaiting a Lisp error. Allegro offers only one mechanism for trapping Lisp errors, `excl:errorset`.

Unfortunately, `excl:errorset` reports only success or failure (and a return value if any); in case of failure no diagnostic information is available. Use `excl:errorset` only in narrow contexts where a specific kind of user error can be anticipated, and where diagnosis is guaranteed by context. A typical context for `excl:errorset` is execution of a single COMMON LISP form that involves the file system or user input (see examples below).

When `excl:errorset`'s optional second argument is `t`, it prints to the console a description of any error trapped (information not available to the caller). Use this option only when the information would add to the diagnosis implied by the context. For example, it wouldn't help in a context where a failure always implies a missing file, but it would help when loading Lisp code, since the additional information can help the user locate the problem.

Allegro's `excl:errorset` returns multiple values: a boolean indicating success or failure and the resulting value (if successful). How to handle these depends on the circumstances. Sometimes you just want to protect against irrelevant break loops, as when deleting a scratch file, and you care about neither value.

```
(when (probe-file (scratch-file))
  (excl:errorset (delete-file (scratch-file))))
```

Other times, you may want to treat the return value separately from success or failure, as when loading a file of Lisp code; this example takes three separate actions, corresponding to three possible outcomes of the call to `load`: it returns `t`, it returns `nil`, or it signals an error.

```
(unless (excl:errorset
          (if (load file :if-does-not-exist nil)
              (Announce "~A loaded" filename)
              (Editor-Error "~A not found" filename))
          t)
  (Editor-Error "Error during load of ~A" filename))
```

And sometimes, you want to use both values together, as when expanding a UNIX file specification; in this example the `multiple-value-bind` form returns either the result, when it succeeds, or `nil` if it signals an error.

```
(cond
  ((multiple-value-bind (success? result)
     (excl:errorset (excl::tilde-expand-unix-namestring name))
     (and success? result))
  (t (Editor-Error  "Unknown user in ~S." name))))
```

Ideally Lisp would pass control to a *Pan* error handler in case of any Lisp error not already wrapped by `excl:errorset`. Unfortunately, Allegro does not make this possible (and `excl:errorset` provides too little information to be useful). So, in case of Lisp errors not wrapped by `excl:errorset`, *Pan* gets suspended and enters a break loop. If the guidelines here are followed, however, this should never happen in response to a user error, only in response to internal *Pan* bugs.

5.2 Protect Against Unwinds

Unwinds can strike at any time. A call to a *Pan* function might, instead of returning, unwind the stack directly back to the dispatcher with a `throw`. Take great care that your code leaves behind no unpleasant side-effects when this happens; there are several basic techniques for doing so.

- Use dynamic binding for special variables that establish important context (instead of `setf`).
- Use COMMON LISP `unwind-protect`.
- Use `with-open-file` instead of `open` to read files.

Avoid indiscriminate use of these techniques. Instead incorporate them in *Pan* primitives whose main job is to control context and side-effects.

For example, the special variables `*buffer*` and `*class*` establish the context for many of *Pan*'s functions. When setting bindings in new classes you can use the macro With-Class-Scope to change this context temporarily.

```
(Define-Macro With-Class-Scope (class &body body)
  `(let ((*class* ,class))
     ,@body))
```

If you must temporarily modify a non-special variable, protect it with `unwind-protect`. For example, With-Variable-Binding works this way for *Pan* variables. A simpler example would be the preservation of an explicit global stack.

```
(push-my-stack value)
(unwind-protect
  (progn
    ...)
  (pop-my-stack))
```

5.3 Assertions

Assertions differ from tests for user errors in two ways. First, they guard against failures that the user should not be able to provoke. Second, they must make available much more information, since diagnosis must be manual.

Assertions in COMMON LISP

Whenever an internal *Pan* function appears particularly vulnerable to failure through misuse (especially when such a failure would be difficult to diagnose) add a *Pan* debugging assertion.

```
(debug-assert *module-debug*                ;control var.
              (zerop foo)                   ;test form
              ()                            ;list of places
              "Value of foo is ~D, not 0"   ;message
              foo)                          ;message args
```

The first argument is a variable that controls assertion checking dynamically. There should be one control variable per module; for example the control variable in the "region" module is `*region-debug*`. There should be, but isn't yet, a convenient user interface for examining and setting the values of these variables. All control variables have default value `t` at present, so all assertion checking is routinely performed in debugging versions of *Pan*. *Unfinished*

`debug-assert` passes its remaining arguments to COMMON LISP `assert`. The assertion in the example ensures that variable `foo` has value 0; when the assertion fails (namely, when the value of `foo` is *not* 0), `assert` prints the specified message using `format` and enters a break loop.

Note that the call to debug-assert is removed from compiled code in production versions of *Pan*. To activate assertion checking in a production version, reload the module to run interpretively and push the symbol :debug onto the COMMON LISP list *features*.

Assertions in "C"

Debugging assertions in *Pan*'s "C" modules are handled in much the same spirit, but with a different implementation.

- Control conditional compilation with the "C" DEBUG switch, set in the appropriate makefiles.
- Write explicit error tests. Announce failures with a call to printError, a wrapper to fprintf (with output to stderr).
- Return a special error value to the Lisp code that called the function.

```
int
swPanelSetFlag(dsply, flag, value)
     int dsply;
     int flag;
     int value;
{
  ...
#ifdef DEBUG
  if (flag < 0 || flag >= NFLAGS) {
    printError("swPanelSetFlag: flag%d out of range",
               flag);
    return(FALSE);
  }
  ...
#endif DEBUG
  ...
}
```

Add assertions to every "C" function that is exported into Lisp. Assume that arguments from Lisp are correct in the ordinary case, but use assertions to check arguments for rationality whenever there is enough local state in "C" to do so.

There is no mechanism to control assertion checking in "C" dynamically. One could be added, but it does not seem necessary.

5.4 Tracing

Common Lisp Function Tracing

When it will suffice, Common Lisp function tracing is the most convenient form of tracing available. Control function tracing dynamically, on a function-by-function basis, using `trace` and `untrace`

```
(trace Insert-Region Delete-Region)
```

See documentation on the Allegro implementation of Common Lisp for more details on use of `trace` and `untrace`.

Common Lisp function tracing is confounded by the presence of *Pan*'s complex, large (and sometimes circular) objects being passed as arguments and return values; default methods for printing these objects are at best hard to read, and can sometimes cause Lisp errors themselves. Correct this shortcoming by supplying useful printing functions for important data structures.

```
(defstruct (tnode ...
             (:print-function print-tnode))
  parent
  children
  ...
  )

(defun print-tnode (tnode stream level)
  (format stream "..." ...))
```

Allegro Function Advising

When simple tracing is inadequate, but you want to operate dynamically in an existing image without adding custom tracing code (see below), you can use the Allegro advise mechanism. This allows you to dynamically attach procedures to be run as before- or after-daemons on any function you specify. See the Allegro Users Manual for more details.

Custom Tracing in Common Lisp

Add custom tracing code in situations where Common Lisp function tracing does not suffice:

- when argument values do not print intelligibly in the default manner;

- when higher-level information would be much more useful than argument listings; and
- when a particular kind of trace must be implemented in several locations, not tied to a specific function.

Pan's "debug-trace" module supports a general form of tracing. To use it, begin by creating a new "trace" by name.

```
(def-debug-trace watch-widgets
  :documentation "Trace operations on widgets")
```

Insert trace statements at the appropriate places, as many as needed for the particular trace.

```
  ...
  (debug-trace watch-widgets
               "Widget ~A gets created."
       ...)
  ...
  (debug-trace watch-widgets
               "Widget ~A enlarged by ~D."
               ...)
  ...
```

You can control the trace dynamically with calls to `debug-trace-on` and `debug-trace-off`.

```
(debug-trace-on watch-widgets)
```

See the "debug-trace" module in src/lib/allegro/debug-trace.cl for more details.

Custom Tracing in "C"

Tracing in *Pan*'s "C" modules is handled specially for several reasons.

- Although it is possible to use COMMON LISP tracing on functions bound to "C" functions, argument passing across the foreign-function interface is sufficiently problematic that it is helpful to know what arguments a "C" function actually receives.
- Some arguments to C functions (e.g. textnode pointers) do not submit gracefully to printing.

- Additional custom tracing, beyond reporting argument variables, can be most helpful.

Every "C" function should include at least simple argument tracing.

```
int
swPanelSetFlag(dsply, flag, value)
     int dsply;
     int flag;
     int value;
{
  ...
#ifdef TRACE
  if (panelTrace) printTrace("swPanelSetFlag%d,%d,%d\n)",
(                              dsply, flag, value);
#endif TRACE
  ...
}
```

When tracing code reports arguments (the most common usage at present) position the trace statement before any debugging assertions. When a "C" function returns an important (and printable) value, add an additional trace just before the return that reports this value. Add additional traces for monitoring important retained state in "C" modules, for example whether a call that sets the screen cursor location "hits" the cached value.

Every "C" module ("panel" for example) includes a static variable (in this case `panelTrace`) that controls tracing dynamically. Default value for these values is `FALSE`, but they may be adjusted with calls to "C" functions (for example `swPanelSetTrace`) imported into Lisp (where it is known as `sw-set-panel-trace`). *Pan* library module "pan-debug" will, when loaded, add a menu for changing "C" tracing variables.

5.5 Statistics

In some situations you may want to observe aggregate rather than specific behavior in your code. In these cases statistics are more suitable than tracing.

Custom Statistics in COMMON LISP

Pan's "statistics" module supports a general form of statistics gathering. Create a new "statistic" by name before starting what you want to observe.

The statistic may be a simple "counter," a "ratio" or "percentage" (based on counters), a "sum" or "difference" (of two counters), or a "histogram."

```
(def-statistics-counter calls-to-window-update
  "Number of calls to window update")

(def-statistics-counter update-cache-miss
  "Number of cache misses in window update")

(def-statistics-ratio update-cache-miss calls-to-window-update
  "Ratio of window update cache misses to calls.")
```

Then, at the appropriate places, increment counters. The default increment is 1, but you can specify another increment with an optional argument.

```
...
(statistic calls-to-window-update)
...
```

Finally, when you wish to see a summary of current values for statistics, call `print-statistics`. You may also set *Pan* option Print-Stats-On-Termination to `t`, which requests a summary when you exit *Pan*.

6 Acknowledgements

Pan, including all policies and mechanisms described here, was designed and implemented by Robert A. Ballance, Jacob Butcher, and Michael L. Van De Vanter with guidance from Susan L. Graham. Christina Black made many helpful suggestions concerning this document and the issues it raises.

7 References

1. BALLANCE, R. A. Syntactic and Semantic Checking in Language-Based Editing Systems. PhD Dissertation, Computer Science Division, EECS, University of California, Berkeley, 89/548, December 1989.
2. BALLANCE, R. A. AND VAN DE VANTER, M. L. Pan I: An Introduction for Users. Computer Science Division, EECS, University of California, Berkeley, 88/410, September 1987.

3. Ballance, R. A., Van De Vanter, M. L. and Graham, S. L. The Architecture of Pan I. Computer Science Division, EECS, University of California, Berkeley, 88/409, August 1987.

4. Butcher, J. LADLE. Computer Science Division, EECS, University of California, Berkeley, 89/519, November 1989, Master's Thesis.

5. Hiltz, S. R. and Kerr, E. B. Learning Modes and Subsequent Use of Computer Mediated Communication Systems. *Proceedings SIGCHI Conference on Human Factors in Computing Systems*, Boston, MA (April 1986).

6. Lane, D. J. Porting Pan I to Allegro Common Lisp. Computer Science Division, EECS, University of California, Berkeley, 88/453, September 1988.

7. Lewis, C. and Norman, D. A. *Designing for Error*. In *User Centered System Design: New Perspectives on Human-Computer Interaction*, D. A. Norman and S. W. Draper, Eds. Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, 411–432.

8. Steele, Jr., G. L. *Common Lisp: The Language*. Digital Press, 1984.