Massachusetts Institute of Technology
Artificial Intelligence Laboratory

Working Paper 268 February, 1985

# Exceptional Situations in Lisp

Kent M. Pitman

**Abstract**

Frequently, it is convenient to describe a program in terms of the **normal situations** in which it will be used, even if such a description does not describe the its complete behavior in all circumstances. This paper surveys the issues surrounding the description of program behavior in **exceptional situations.**

Keywords: Conditions, Exceptions, Handling, Lisp, Signalling.

# I. Introduction

During a computation, functions communicate with one another by passing arguments, by returning values, and by side-effect upon shared structures. Consider the following definitions:

| | |
|---|---|
| (PLUS $n_1$ $n_2$) | returns the sum of its two arguments, $n_1$ and $n_2$. |
| (TYI *instream*) | returns the next character on the input stream given by its argument, *instream*. This function performs an implicit side-effect upon *instream*, typically incrementing some internal scan pointer, in order to assure that subsequent calls attempts to read from the same stream do not try to re-read the same character. |

The descriptions of PLUS and TYI given above describe their behavior in what we shall call **normal situation**. Sometimes, however, an **exceptional situation** will arise which does not fit neatly into the normal description of a function. For example, PLUS might receive an argument which is not a number, or TYI might receive a stream which has no more available characters.

This paper will examine this phenomenon of exceptional situations and the programming techniques which can be employed to deal with such situations.

# II. The Nature of Exceptional Situations

It is important to recognize that this distinction between normal and exceptional situations is in some sense contrived. Making this distinction does not change the way programs behave; it simply changes the way we think about programs — hopefully for the better.

Suppose for a moment that when PLUS receives a non-numeric argument, it simply returns *false*.[1] We could simply define that in the normal situation PLUS returns the sum of its two arguments if they are numbers or *false* if they are not. This wouldn't change what PLUS did, only how we thought about it. In general, any program which could potentially be described in terms of normal and exceptional situations can be described with just a normal description by simply taking some union of the original normal and exceptional descriptions and declaring that to be normal.

Unfortunately, if we want to claim that it is normal to use PLUS with non-numeric arguments, then we have to put up with code such as:

```
(PLUS 'A 'B)
```

where a programmer calls PLUS explicitly with non-numeric arguments because he plans to take advantage of the fact that it will yield *false*. Perhaps the programmer doesn't know that the variable NIL holds the *false* object. Somehow we want to discourage the programmer from using PLUS in this fashion, and one way to do so is simply to describe such a use as "not normal."

### Simplifying Assumptions

Another reason for distinguishing between normal and exceptional situations shows itself in the use of **simplifying assumptions** that typically distinguish a prototype from a production quality program. A programmer may speed the development of a program by assuming the program will be needed only for some normal case. For example, the tokenizer for a parser might be initially coded to simply read characters from a stream until a token break, without handling the exceptional situation that occurs at the end of file when there is no next character. In so doing, the programmer pushes off worrying about a level of detail that would only be distracting during the prototyping phase. Later, after it is has been shown that the program works satisfactorily in normal situations, the programmer will typically "tighten up" the program to handle exceptional situations. This idea is explored in detail in [Rich 81] and [Lieberman 82].

[1] MULISP, a commerically available dialect of Lisp for microcomputers, does exactly this.

### Presentation

Another application for this same idea of simplifying assumptions may come after a program has been developed when it is being read by another programmer. The programmer may want to first understand what it does in the normal situation and then refine his understanding of it by reviewing its behavior in exceptional sitatutions. If the code for dealing with the normal situation is thickly interspersed with code attempting to recognize and deal with exceptional situations, the reader's ability to decipher the intent of the program may be greatly diminished.

This problem may be dealt with in at least two ways. One way would be to provide primitives within the language which encourage the separation of code to deal with normal and exceptional situations. Another way would be to provide editing technology which allowed the code for the exceptional situation to be hidden while the normal situation is examined on its own.

### Modularity

The idea that the language should encourage separation of normal and exceptional situations has appeal for another reason. It frequently happens that the same exceptional situation can occur in many different places in a program; and, in many cases, the way in which such situations should be dealt with does not vary with the location in the program.

For example, consider a program which is compiling a file from a remote machine. If the remote machine crashes, the compilation will probably want to simply fail no matter what routine was running. Languages should provide some way of centralizing the information about a program's behavior in such an exceptional situation so that it doesn't have to be needlessly repeated throughout the code for the program.

### Efficiency

In some cases, there may be efficiency reasons for considering some cases exceptional. For example, the compiled code for PLUS will be less efficient if it must contain explicit checks for its arguments being non-numbers. In MACLISP [Pitman 83], for example, the interpreted + function will "signal an error" if invoked with bad arguments, but a compiled calls to + will just "do the wrong thing" if that situation arises.[2] Because we distinguish normal and exceptional situations, we can conveniently describe the semantics of + as being compromised only in the exceptional situation; if we had not made this distinction, we would have to either make + do extra work in compiled code or describe the process of compiling + as "unreliable."

## III. Dealing With Exceptional Situations

When an exceptional situation arises in a program, there are a number of possible actions that can be taken.

### Ignoring Exceptional Situations

Our description of exceptional situations is general enough to include situations which are not normal but which the program does not recognize as such. The example of how compiled calls to + behave in MACLISP is an example of this behavior. Hence, one possible way in which programs can deal with exceptional situations is to fail to recognize them.

While such failure might not be something to encourage as a programming style, it is important to recognize that it does occur — frequently, in fact. It can happen any time an assumption of any kind is made by code but not mechanically verified (either at compile time or runtime).

---

[2]Specifically, it will ignore the type information of the arguments, interpreting them as if they were fixnums. The result may be mystifying to novice programmers.

### Modified Return Value Conventions

In some situations, the type of the return value is highly constrained. For example, since we know that many operations on numbers yield only other numbers, we could use non-numeric return values from such functions to convey other kinds of information. Our earlier suggestion that PLUS might return *false* when passed non-numeric arguments is an example of this technique.

A slightly more complicated variation of this technique arises with the MACLISP ERRSET primitive. ERRSET evaluates its argument, "trapping" any errors. The writers of ERRSET wanted to return the result of the computation if there was no error, but also to be able to detect whether there was an error or not. Since evaluation could return any kind of object, their problem was to devise a return value convention which expressed both of these pieces of information unambigously. Their solution was to return NIL if an error had occurred or a list the first element of which was the result of the evaluation if no error occurred.

### Multiple Return Values

The solution to this ERRSET "problem" on the LISP MACHINE [Weinreb 81] is somewhat more graceful because there it is possible for a function to simply return multiple values. LISP MACHINE LISP provides a special form called IGNORE-ERRORS which evaluates its argument returning the result as a "first return value" (or NIL if an error occurred). Since the case of NIL being returned in this position is ambiguous (the form might have returned NIL or an error might have occurred), it also returns a "second return value" which is NIL if no error occurred or non-NIL otherwise.

### Multiple Return Points

Especially in SCHEME [Steele 78], a popular way to deal with exceptional situations is simply to pass multiple continuations [Steele 76].

For example, we could imagine that our earlier case of TYI could be reformulated by making TYI take additional arguments of functions to be called in the case that either there is or is not a "next character." Consider the Scheme definition:

```
(DEFINE (TYI STREAM SUCCESS FAIL)
  (IF (STREAM-EMPTY? STREAM)
      (FAIL)
    (SUCCESS (STREAM-FIRST STREAM) (STREAM-REST STREAM))))
```

which would allow the caller to write:

```
(DEFINE (VIEW-STREAM STREAM)
  (TYI STREAM (LAMBDA (FIRST-CHAR REST-OF-STREAM)
                (TYO FIRST-CHAR) ;Type the first character
                (VIEW-STREAM REST-OF-STREAM))
              (LAMBDA () 'DONE)))
```

In this case, the normal and exceptional situations are factored out into separate functions so that they can be studied independently.

### Non-Local Transfer of Control

Sometimes, a convention may have been established between a group of programs so that when certain exceptional situations are detected, control is transferred to one of the callers in a non-local fashion. Such a transfer is generally accomplished by an escape procedure in SCHEME or the THROW primitive in Lisp.

Also in this category are the mechanisms some languages provide for aborting and restarting computations. These are typically just "syntactic sugar" for the same kind of non-local transfer of control already provided by THROW. For example, the ↑G function (which simulates the effect of the Maclisp abort character, Control-G) "throws" to a point which restarts the toplevel Read-Eval-Print loop.

## IV. Terminology

We have argued that it is useful to reason in the abstract about two separable kinds of behavior for programs, which we have called "normal" and "exceptional." This being the case, it seems appropriate that we evolve a special vocabulary, both in our natural language and in our programming languages, which acknowledges this distinction.

In modern Lisp dialects, we encourage the use of primitives such as DEFSTRUCT rather than CAR, CDR, and CONS because it allows us to get away from thinking about how objects are implemented and speak directly in terms of more abstract properties of objects. As we have seen, Lisp primitively provides many ways to deal with exceptional situations at the implementation level, yet many modern Lisp dialects do not standardize ways of imposing abstraction upon these situations so that it is clear which parts of a program are for dealing with the normal situations, which are for dealing with exceptional situations, and which are simply gluing the two parts together.

### Signalling

When a program detects an exceptional situation, it will typically want to announce the presence of the situation, allowing any special code which has been set up for dealing with that situation to be run. We will refer to this process of announcing an exceptional situation as **signalling** and to the piece of code which announces the situation as the **signaller.**

The class of exceptional situations which are recognized and signalled will be henceforth referred to as **conditions.**

We will think of signalling as a way for a program to request advice on which of several ways to proceed. These possible ways to proceed will be called **proceed options.**

### Handling

The pieces of code which are asked to select a proceed option will be called **handlers.** It a handler selects a proceed option, it will be said to have **handled** the situation. If it does not, it will be said to have **declined.** There is actually a third situation possible, which we shall call **bypassing,** where the handler performs some non-local transfer of control which implicitly eliminates the need to handle the error.

### Types of Situations

Some conditions are **non-fatal** in that the failure to handle them will not affect the correct behavior of the program if it simply continues on. Some common examples of places where non-fatal conditions might want to be signalled are "end of line" and "end of page" exceptions on output devices, the entry and exit of major and minor modes in an editor, the successful completion of subtasks in a multi-task project, or if-needed and if-used situations in a knowledge representation system. These non-fatal conditions are sometimes called "**hooks**" because there is a well-defined behavior that can occur if no advice is supplied, but advice can be given which will augment or override the default behavior.

Some situations may be considered **fatal** in the sense that code should not proceed beyond the point where they are signalled unless the condition is "corrected" by some form of external intervention. We will call such a fatal situation an **error.**

Error situations, sometimes called "**bugs**," may be divided into two classes: errors which are signalled, which we shall call **error conditions**; and errors that go unsignalled (usually only to cause confusion later in the program's execution).

The **COMMON LISP** manual [Steele 84] is careful to distinguish between two kinds of error situations by using the terminology "is an error" when an error might be signalled (but is not guaranteed to be) and "signals an error" to refer to situations where an error is guaranteed to be signalled. This terminology is compatible with the terms we have chosen in this discussion.

Many terms used in this paper occur also in documentation for the **LISP MACHINE** (*e.g.*, [Weinreb 83]). The terms used in this paper are intended to be compatible with their uses in those documents as well.

## V. Signalling

We now turn to the question of what kind of information needs to accompany a signal. It may help here to appeal to a fairly concrete example.

Let us imagine a situation in the control program for a robot butler which is about to put food on the table where the robot notices that the eggs it is about to serve to serve are green. We might imagine that the signaller's code would look something like:

```
To Serve a Tray-Of-Food:
  For each Food in the Tray-Of-Food,
    If the Food's Color is not the Food's Expected-Color,
       Signal a Bad-Food-Color condition
         specifying the Food, its Color, and its Expected-Color.
  Carry the Tray-Of-Food to the Dining-Room.
  Place the Tray-Of-Food on the Table.
```

Unfortunately, the "code" does not provide enough information for a potential handler to advise it about how to proceed beyond the point where the condition is signalled. Since the handler may not have access to the data structures necessary for for correcting the problem and since the signaller has not provided "code" to implement any corrections, multiple correction strategies will not in general be possible.

For this reason, when a condition is signalled, we should remember that two kinds of information may need to be provided: a description of the condition and a description of what, if anything, the signaller is prepared to do in order to proceed from the condition.

In the case that proceeding is possible, the above code not suffice. What would be needed might look more like:

```
To Serve a Tray-Of-Food:
  For each Food in the Tray-Of-Food,
    If the Food's Color is not the Food's Expected-Color,
       Signal a Bad-Food-Color condition
         specifying the Food, its Color, and its Expected-Color
         and heeding advice on which of the following ways to proceed:
          To Add-Food-Coloring of a given Color:
             Put Food-Coloring of the given Color to the Food.
          To Serve-Food-Anyway:
             Continue;
          To Throw-Food-Away:
             Remove the Food from the Tray-Of-Food.
  Carry the Tray-Of-Food to the Dining-Room.
  Place the Tray-Of-Food on the Table.
```

## VI. Handling

When a condition is signalled, handlers may exist which could potentially handle the error. We will defer for the moment the question of how such handlers are located and for now concern ourselves with what those handlers will want to do once they are located.

Informally, we may think of a handler as analogous to a piece of sage advice. But like any advice, there may be times when it is appropriate and times when it is not. Some of those times may be partly determined by issues of scope which will be discussed later; others will be determined by inspecting the description of the condition and the options the signaller has suggested are available.

Some appropriate means will have to be provided for inspecting the condition. In MACLISP, the means provided[3] is a function to examine the condition state at the current (or a given) stack frame. This made it quite difficult to test condition handlers out of context. Experience with LISP MACHINE LISP suggests that an object-based approach (where the handler receives an argument representing the condition's description and is provided with a means of inspecting that object) is more flexible.

After inspecting the description, the handler might want to do any of a number of things. For example, to continue our earlier robot scenario:

- It might suggest that it is acceptable to serve the food anyway.
- It might suggest that the food not be served.
- It might suggest that the color of the food be changed.
- It might suggest replacing the food with another kind of food.
- It might suggest replacing the food with "better" food of the same kind.

These items involve giving advice about how to proceed. In order to give such advice, the handler would have to have verified that the particular proceed option it was suggesting was acceptable to the signaller.

Regardless of the signaller's situation, the handler may always decline to handle the condition.

- It might say it has no suggestion to about how to proceed.

Also regardless of the situation of the signaller, the handler may always bypass the situation entirely. For example:

- It might blow up the building.

In this case, the situation is resolved by obliterating some outer context in which the situation arose in the first place. This is closely related to what happens in a non-local transfer of control.

### Provisional Handling

Some advice may want to be given provisionally. For example, the handler might want to say: "If you can find nothing better to do, proceed in the following way..."

On the LISP MACHINE, the `CONDITION-BIND-DEFAULT` primitive addresses this issue, but forces the decision about whether advice is to be provisional or not to be decided at coding time (when the user must select between `CONDITION-BIND` and `CONDITION-BIND-DEFAULT`) rather than allowing the user to defer the decision until run time.

In any system where defaults may be asserted, there is the question of how to resolve a situation where two defaults have been asserted.

For example, on the LISP MACHINE, when a signal goes unhandled via the normal mechanism (set up by `CONDITION-BIND`), default handlers are searched from the inside out (along the dynamic call

[3]Actually, MACLISP has only an error system, not a condition system, but we will assume that the same principles apply.

chain) until a handler is found. It is a subject of debate whether this is the correct order, or whether the order should be reversed.[4] Few systems have ever reached the necessary complexity to actually provide such conflicting defaults. In time, as systems grow larger and need this kind of fine tuning, better data will be available about what programmers really need and the answers to this question will perhaps become more clear.

### Classifying Conditions

When is condition handler is trying to determine whether its advice would be applicable in a given situation, it could be considerably helped by the availability of a class hierarchy for condition types. For example, a piece of advice appropriate for "file errors" could be detected as appropriate for "file not found errors" using such a mechanism.

The implementors of the LISP MACHINE condition system insist that this is a situation where the ability for a condition type to have multiple superclasses has proven nearly essential. They argue that some errors do not fit neatly into a strict hierarchy; for example, if an "end of file" condition is signalled from within the `READ` function, is this a "read error" or a "file error"? If it is possible for a condition type to have multiple superclasses, it may be specifically type "file read error" which could inherit from both of the more general types "read error" and "file error." This feature allows handlers a great deal of flexibility in deciding which conditions they are willing to advise.

### Classifying Proceed Options

Some advice may be more specific than others. For example, suppose that the handler for the our `Bad-Food-Color` condition wants to suggest that the food color be changed and the signaller is prepared to `Add-Food-Coloring`. There might want to be some notion of a class hierarchy for proceed options such that `Add-Food-Coloring` is recognized as an appropriate substitute for `Change-Color`.[5]

## VII. Connecting a Signaller with a Handler

When a condition is signalled, there are some questions about how an appropriate handler should be located.

### Scope of Handlers

The first decision to be made is whether handlers for conditions are to be "globally assigned" or "locally bound."

The trend in programming today is strongly in the direction of eliminating any notion of "global" assignment in favor of primitives that bind things. Hence, dialects are more likely to provide primitives with names like `CONDITION-BIND` which provide handlers for conditions within a certain scope[6] than primitives with names like `CONDITION-SET` which side-effect some "global" default handler.

On the LISP MACHINE, for example, the `CONDITION-BIND` special form makes a set of handlers available within the dynamic context of the execution of its body. It also accomplishes the additional service of partitioning the applicability of handlers according to condition type. For example, the form

```
(CONDITION-BIND ((type1 handler1)
                 (type2 handler2)
                 ...)
  form1 form2 ...)
```

will attempt to use a given $handler_i$ for signals generated within its body only if those signals are of the associated $type_i$ (or some subclass of that type).

[4]For example, it is the author's belief that `CONDITION-BIND-DEFAULT` should instead search outside in.

[5]The author is not aware of condition systems for any existing languages that support the notion of a class hierarchy for proceed options.

[6]The scope could in principle be either dynamic or lexical, though in practice dynamic scope or some very close variant always seems to be preferred in existing systems.

The alternative might have been for the user to write:

```
(CONDITION-BIND (#'(LAMBDA (CONDITION)
                     (COND ((TYPEP CONDITION 'type1)
                            (FUNCALL handler1 CONDITION))
                           (T (DECLINE CONDITION))))
                #'(LAMBDA (CONDITION)
                     (COND ((TYPEP CONDITION 'type2)
                            (FUNCALL handler2 CONDITION))
                           (T (DECLINE CONDITION))))
                ...)
  form1 form2 ...)
```

### Default Handling

One exception to this might be that defining a new condition type for use by others could also involve the creation of a global **default handler** for the condition. In object systems, this could be implemented as a method for the objects of the given condition type which is sent only if no handler was found. This idea of a default handler could be used to implement the distinction between error and non-error conditions in some systems by making the default handler for a non-error condition simply return (telling the signaller to proceed in whatever way it feels best), and making the default handler for an error enter an interactive debugger.

### Synchronous and Asynchronous Conditions

The "error systems" provided in existing Lisps are primarily **synchronous**. That is, the signalling of a condition from code causes that code to block pending advice from handlers, which are then run synchronously in the same process.

Any discussion of **asynchronous conditions** would require new terminology to define what it means to **interrupt** a running process, and is beyond the scope of this paper.

## VIII. Summary

The descriptions of programs may usefully be divided into two parts, which describe their behavior in **normal situations** and in **exceptional situations.**

This distinction is important when developing programs because it allows programmers to make **simplifying assumptions** about the nature programs. It can also have important consequences on the **presentation** of a program's code, by visually separating the code that handles normal and exceptional situations. Aspects of programs' **modularity** and **efficiency** can also be influenced also be affected by the decision to make this distinction.

Most languages already provide adequate control mechanism for dealing with exceptional situations, including **ignoring** such situations, varying **return value conventions**, varying the **number of return values**, and allowing programs to exit via **non-local transfer of control** such as THROW.

Unfortunately, although these mechanisms are powerful enough to implement appropriate kinds of control structures, they are not sufficiently abstract to encourage as a language for talking about exceptional situations. New language primitives must be evolved which interface to these mechanisms in an abstract way.

In this paper, we have surveyed various aspects of existing condition systems for Lisp and have established terminology which may be used in describing their behavior in an abstract way.

## References


[Conway 75] R.W. Conway and D. Gries, *An Introduction to Programming: A Structured Approach Using PL/1 and PL/C-7*, Winthrop Publishers, Inc., Cambridge, MA, 1975.

[Lieberman 82] H. Lieberman, "Seeing What Your Programs Are Doing," Memo 656, MIT Artificial Intelligence Laboratory, Cambridge, MA 02139, February 1982.

[Liskov 79] B. Liskov, et al, *CLU Reference Manual*, Technical Report 225, MIT Laboratory for Computer Science, Cambridge, MA 02139, October 1979.

[Pitman 83] K. Pitman, *The Revised Maclisp Manual* (Saturday Evening Edition), Technical Report 295, MIT Laboratory for Computer Science, Cambridge, MA 02139, May 1983.

[Rich 81] C. Rich and R.C. Waters, "The Disciplined Use of Simplifying Assumptions," Working Paper 220, MIT Artificial Intelligence Laboratory, Cambridge, MA 02139, December 1981.

[Steele 76] G.L. Steele, Jr., and G.J. Sussman, "LAMBDA, The Ultimate Imperative," Memo 353, MIT Artificial Intelligence Laboratory, Cambridge, MA 02139, March 1976.

[Steele 78] G.L. Steele, Jr., "The Revised Report on SCHEME: A Dialect of LISP," Memo 452, MIT Artificial Intelligence Laboratory, Cambridge, MA 02139, January 1978.

[Steele 84] G.L. Steele, Jr., *Common LISP: The Language*, Digital Press, Burlington, MA, 1984.

[Weinreb 81] D.L. Weinreb and D.A. Moon, *Lisp Machine Manual* (Fourth Edition), MIT Artificial Intelligence Laboratory, July 1981.

[Weinreb 83] D.L. Weinreb, *Signalling and Handling Conditions*, Document #990097, Symbolics, Inc., Cambridge, MA 02139, 1983.


## Acknowledgments


Many of the ideas in this paper were inspired by the "New Error System" for the Symbolics Lisp Machine systems [Weinreb 83]. Other ideas descend from discussions with Eugene Ciccarelli, David Moon, and Daniel Weinreb.