# CHAPTER 1

# INTRODUCTION

Interlisp is a *programming system*. A programming system consists of a programming *language*, a large number of predefined programs (or *functions*, to use the Lisp terminology) that can be used either as direct user commands or as subroutines in user programs, and an *environment* that supports the programmer by providing a variety of specialized programming tools. The language and predefined functions of Interlisp are rich, but similar to those of other modern programming languages. The Interlisp programming environment, on the other hand, is very distinctive. Its most salient characteristic is an integrated set of programming tools which know enough about Interlisp programming so that they can act as semi-autonomous, intelligent ''assistants'' to the programmer. In addition, the environment provides a completely self-contained world for creating, debugging and maintaining Interlisp programs.

This manual describes all three components of the Interlisp system. There are discussions about the content and structure of the language, about the pieces of the system that can be incorporated into user programs, and about the environment. The line between user code and the environment is thin and changing. Most users extend the environment with some special features of their own. Because Interlisp is so easily extended, the system has grown over time to incorporate many different ideas about effective and useful ways to program. This gradual accumulation over many years has resulted in a rich and diverse system. That is the reason this manual is so large.

Whereas the rest of this manual describes the individual pieces of the Interlisp system, this chapter attempts to describe the whole system language, environment, tools, and the otherwise unstated philosophies that tie it all together. It is intended to give a global view of Interlisp to readers approaching it for the first time.

## 1.1 INTERLISP AS A PROGRAMMING LANGUAGE

This manual does not contain an introduction to programming in Lisp. Sadly, primers and teaching materials for Lisp are few and quickly become dated. [Winston & Horn, 1981] discuss Lisp and its applications, but focus on MacLisp, with only a limited section on Interlisp in an appendix. [Siklossy, 1976] and [Weissman, 1967] are both sound, but a little dated. In this section, we simply highlight a few key points about Lisp on which much of the later material depends.

The Lisp family of languages (e.g., Interlisp, UCI Lisp [Meehan, 1979], FranzLisp [Foderaro, 1979], MacLisp [Moon, 1974], Lisp Machine Lisp [Weinreb & Moon, 1979], etc.) shares a common structure in which large programs (or functions) are built up by composing the results of smaller ones. Although Interlisp, like most modern Lisps, allows programming in almost any style one can imagine, the natural style of Lisp is functional and recursive, in that each function computes its result by selecting from or building upon the values given to it and then passing that result back to its caller (rather than by producing ''side-effects'' on external data structures, for example). A great many applications can be written in Lisp in this purely functional style, which is encouraged by the simplicity with which Lisp functions can be composed together.

Lisp is also a list-manipulation language. The essential primitive data objects of any Lisp are ''atoms'' (symbols or identifiers) and ''lists''(sequences of atoms or lists), rather than the ''characters'' or ''numbers'' of more conventional programming languages (although these are also present in all modern Lisps). Each Lisp dialect has a set of operations that act on atoms and lists, and these operations comprise the core of the language.

Invisible in the programs, but essential to the Lisp style of programming, is an automatic memory management system (an ''allocator'' and a ''garbage collector''). Allocation of new storage occurs automatically whenever a new data object is created. Conversely, that storage is automatically reclaimed for reuse when no other object makes reference to it. Automatic allocation and deallocation of memory is essential for rapid, large scale program development because it frees the programmer from the task of maintaining the details of memory administration, which change constantly during rapid program evolution.

A key property of Lisp is that it can represent Lisp function definitions as pieces of Lisp list data. Each subfunction ''call''(or *function application*) is written as a list in which the function is written first, followed by its arguments. Thus, `(PLUS 1 2)` is a list structure representation of the expression `1 + 2`. Each program can be written as a list of such function applications. This representation of program as data allows one to apply the same operations to programs that one uses to manipulate data, which makes it very straightforward to write Lisp programs which look at and change *other Lisp programs*. This, in turn, makes it easy to develop programming tools and translators, which was essential in enabling the development of the Interlisp environment.

One result of this ability to have one program examine another is that one can extend the Lisp programming language itself. If some desired programming idiom is not supported, it can be added simply by defining a function that translates the desired expression into simpler Lisp. Interlisp provides extensive facilities for users to make this type of language extension. In addition, the CLISP (Conversational LISP) package provides definitions for several commonly used programming constructs (`if ... then ... else`, `for` and `do` loops, etc.) that make many programs easier to express. Using this ability to extend itself, Interlisp has incorporated many of the constructs that have been developed in other modern programming languages.

## 1.2 INTERLISP AS AN INTERACTIVE ENVIRONMENT

Interlisp programs should not be thought of as autonomous, external files of source code. All Interlisp programming takes place within the Interlisp environment, which is a completely self-sufficient environment for developing and using Interlisp programs. Not only does the environment contain the obvious programming facilities (e.g., program editors, compilers, debuggers, etc.), but it also contains a variety of tools which assist the user by ''keeping track'' of what happens, so the user doesn't have to. For example, the Interlisp file package notices when programs or data have been changed, so that the system will know what needs to be saved at the end of the session. The ''residential'' style, where one stays within the environment throughout the development, from initial program definition through final debugging, is essential for these tools to operate. Furthermore, this same environment is available to support the final production version, some parts providing run time support and other parts ignored until the need arises for further debugging or development.

For terminal interaction with the user, Interlisp provides a ''Read-Eval-Print'' loop. That is, whatever the user types in is `READ` by the system, executed (or ''`EVAL`''uated) and the result is `PRINT`-ed onto the terminal. (This interaction is also recorded by the programmer's assistant, described below, so the user

can ask to do an action again, or even to undo the effects of a previous action.) Although each interactive terminal listener (or ''executive'') defines a few specialized commands, most of the interaction will consist of simple evaluations of ordinary Lisp expressions. Thus, instead of specialized terminal commands for operations like manipulating the user's files, actions like this are carried out simply by typing the same expressions that one would use to accomplish them inside a Lisp program. This creates a very rich, simple and uniform set of interactive commands, since any Lisp expression can be typed at a command executive and evaluated immediately.

In normal use, one writes a program (or rather, ''defines a function'') simply by typing in an expression that invokes the ''function defining'' function (`DEFINEQ`), giving it the name of the function being defined and its new definition. The newly defined function can be executed immediately, simply by using it in a Lisp expression. Although most Interlisp code is normally run compiled (for reasons of efficiency), the initial versions of most programs, and all of the user's terminal interactions, will be run interpreted. Eventually, as a function gets larger or is used in many places, it becomes more effective to compile it. Usually, by that stage, the function has been stored on a file and the whole file (which may contain many functions) is compiled at once. `DEFINEQ`, the compiler (`COMPILE`), and the interpreter (`EVAL`), are all themselves Lisp functions that use the ability to treat other Lisp expressions and programs as data.

In addition to these basic programming tools, Interlisp also provides a wide variety of programming support mechanisms:

Structure editor
: Since Interlisp programs are represented as list structure, Interlisp provides an editor which allows one to change the list structure of a function's definition directly.

Pretty-printer
: The pretty printer is a function that prints Lisp function definitions so that their syntactic structure is displayed by the indentation and fonts used.

Break Package
: When errors occur, the break package is called, allowing the user to examine and modify the context at the point of the error. Often, this enables execution to continue without starting over from the beginning. Within a break, the full power of Interlisp is available to the user. Thus, the broken function can be edited, data structures can be inspected and changed, other computations carried out, and so on. All of this occurs in the context of the suspended computation, which will remain available to be resumed.

DWIM
: The ''Do What I Mean'' package automatically fixes the user's misspellings and errors in typing.

Programmer's Assistant
: Interlisp keeps track of the user's actions during a session and allows each one to be replayed, undone, or altered.

Masterscope
: Masterscope is a program analysis and management tool which can analyze users' functions and build (and automatically maintain) a data base of the results. This allows the user to ask questions like ''`WHO CALLS ARCTAN`'' or ''`WHO USES COEF1 FREELY`'' or to request systematic changes like ''`EDIT WHERE ANY` *(function)* `FETCHES ANY FIELD OF` *(the data structure)* `FOO`''.

Record/Datatype Package
: Interlisp allows a programmer to define new data structures. This enables one to separate the issues of data access from the details of how the data is actually stored.

File Package — Files in Interlisp are managed by the system, removing the problem of ensuring timely le updates from the user. The le package can be modi ed and extended to accomodate new types of data.

Performance Analysis — These tools allow statistics on program operation to be collected and analyzed.

These facilities are tightly integrated, so they know about and use each other, just as they can be used by user programs. For example, Masterscope uses the structural editor to make systematic changes. By combining the program analysis features of Masterscope with the features of the structural editor, large scale system changes can be made with a single command. For example, when the lowest-level interface of the Interlisp- D I/O system was changed to a new format, the entire edit was made by a single call to Masterscope of the form `EDIT WHERE ANY CALLS '(BIN BOUT )`. [Burton et al., 1980] This caused Masterscope to invoke the editor at each point in the system where any of the functions in the list `'(BIN BOUT )` were called. This ensured that no functions used in input or output were overlooked during the modi cation.

The new, personal machine implementations of Interlisp, such as Interlisp- D, also provide some new user facilities, and some new, interactive graphic interfaces to some of the older Interlisp programming tools:

Multiple Processes — The multiple and independent processes allowed in Interlisp- D simplify problems which require logically separate pieces of code to operate in parrallel.

Windows — The ability to have multiple, independent windows on the display allows many di erent processes or activities to be active on the screen at once.

Inspector — The inspector is a display tool for examining complex data structures encountered during debugging.

The gure found at the beginning of this chapter shows a standard user display within Interlisp- D. One window displays a list of messages available for browsing, using an experimental mail reading system. This operates in parallel with the user's other activities, continually monitoring the remote mail server and watching for any new messages. The ''DEdit'' window is editing an Interlisp function. The ''Chat'' window o ers a direct connection to a remote machine (this one is a remote le server). There are two nested break windows showing the environment of an interrupted evaluation. And in the lower right, there is a Masterscope display showing all the possible execution paths to some function.

Some of the newer implementations of Interlisp have embedded within them an entire operating system written in Interlisp. For the most part, that is of no concern to the user (although it is nice to know that one *can* write programs of this complexity and performance within Interlisp!). However, some of the facilities provided by this low level code allow the use of Interlisp for applications that would previously have been forced into a relatively impoverished system programming environment. In particular, Interlisp- D provides complete facilities for experimenting with distributed machines and services on a local area network, plus access to all the services that such networks provide (e.g., mail, printing, ling, etc.).

## 1.3 INTERLISP PHILOSOPHY

The extensive environmental support that the Interlisp system provides has developed over the years in order to support a particular style of programming called ''exploratory programming'' [Sheil, 1983].

# INTRODUCTION

For many complex programming problems, the task of program creation is *not* simply one of writing a program to fulfill pre-identified specifications. Instead, it is a matter of exploring the problem (trying out various solutions expressed as partial programs) until one finds a good solution (or sometimes, any solution at all!). Such programs are by their very nature evolutionary; they are transformed over time from one realization into another in response to a growing understanding of the problem. This point of view has lead to an emphasis on having the tools available to analyze, alter, and test programs easily. One important aspect of this is that the tools be designed to work together in an integrated fashion, so that knowledge about the user's programs, once gained, is available throughout the environment.

The development of programming tools to support exploratory programming is itself an exploration. Noone knows all the tools that will eventually be found useful, and not all programmers want all of the tools to behave the same way. In response to this diversity, Interlisp has been shaped, by its implementors and by its users, to be easily extensible in several different ways. First, there are many places in the system where its behavior can be adjusted by the user. One way that this can be done is by changing the value of various ''flags'' or variables whose values are examined by system code to enable or suppress certain behavior. The other is where the user can provide functions or other behavioral specifications of what is to happen in certain contexts. For example, the format used for each type of list structure when it is printed by the pretty-printer is determined by specifications that are found on the list `PRETTYPRINTMACROS`. Thus, this format can be changed for a given type simply by putting a printing specification for it on that list.

Another way in which users can effect Interlisp's behavior is by redefining or changing system functions. The ''Advise'' capability, for instance, permits the user to modify the operation of virtually any function in the system by wrapping user code ''around'' the selected function. (This same philosophy extends to the break package and tracing, so almost any function in the system can be broken or traced.) Experimentation is thus encouraged and actively facilitated, which allows the user to find useful pieces of the Interlisp system which can be configured to assist with application development. This is even easier in systems like Interlisp-D, where the entire system is implemented in Interlisp, since there are extremely few places where the system's behavior depends on anything outside of Interlisp (such as a low level system implementation language).

While these techniques provide a fair amount of tailorability, the price paid is that Interlisp presents an overall appearance of complexity. There are many flags, parameters and controls that affect the behavior one sees. Because of this complexity, Interlisp tends to be more comfortable for experts, rather than casual users. Beginning users of Interlisp should depend on the default settings of parameters until they learn what dimensions of flexibility are available. At that point, they can begin to ''tune'' the system to their preferences.

The various implementations of Interlisp share not only this general philosophy, but a philosophy about each other also. Interlisp is available in highly compatible versions across several machines. The community of Interlisp implementors is committed to maintain this level of compatibility. One testimony to this is the existence of pieces of very old code in modern versions of Interlisp that have been inherited from the original BBN-Lisp system nearly 15 years ago. Many of the function definitions in the core of the system have not changed since 1977, over many different versions of Interlisp.

Appropriately enough, even Interlisp's underlying philosophy was itself discovered during Interlisp's development, rather than laid out beforehand. The Interlisp environment and its interactive style were first analyzed in Sandewall's excellent paper [Sandewall, 1978]. The notion of ''exploratory programming'' and the genesis of the Interlisp programming tools in terms of the characteristic demands of this style of programming was developed in [Sheil, 1983]. The evolution and structure of the Interlisp programming environment are discussed in greater depth in [Teitelman & Masinter, 1981].

## 1.4 HOW TO USE THIS MANUAL

This document is a reference manual, not a primer. We have tried to provide a manual that is complete, and that allows Interlisp users to find particular items as easily as possible. Sometimes, these goals have been achieved at the expense of simplicity. For example, many functions have a number of arguments that are rarely used. In the interest of providing a complete reference, these arguments are fully explained, even though they would normally be defaulted. There is a lot of information in this manual that is only of interest to experts.

Users should not try to read straight through this manual, like a novel. In general, the chapters are organized with overview explanations and the most useful functions at the beginning of the chapter, and implementation details towards the end. If you are interested in becoming acquainted with Interlisp using this manual, the best way would be to skim through the whole book, reading the beginning of each chapter.

A few notes about the notational conventions used in this manual:

Lisp object notation: All Interlisp objects in this manual are printed in the same font: Functions (AND, PLUS, DEFINEQ, LOAD); Variables (MAX.INTEGER, FILELST, DFNFLG); and arbitrary Interlisp expressions: (PLUS 2 3), (PROG ((A 1)) ), etc.

Case is significant: An important piece of information, often missed by newcomers to Interlisp, is that *upper and lower case is significant.* The variable FOO is not the same as the variable foo, which is not the same as the variable Foo. By convention, most Interlisp system functions and variables are all-uppercase, but users are free to use upper and lower case for their own functions and variables as they wish.[1]

This manual contains a large number of descriptions of functions, variables, commands, etc, which are printed in the following standard format:

(FOO BAR BAZ _ ) [Function]

This is a description for the function named FOO. FOO has two arguments, BAR and BAZ. Some system functions have extra optional arguments that are not documented and should not be used. These extra arguments are indicated by ‘‘ ’’.

The descriptor [Function] indicates that this is a function, rather than a [Variable], [Prog. Asst. Command], etc.. For function definitions only, this can also indicate the function ‘‘type’’: [NLambda Function], [NoSpread Function], or [NLambda NoSpread Function], which describes whether the function takes a fixed or variable number of arguments, and whether the arguments are evaluated or not.

[1] One exception to the case-significance rule is provided by the Interlisp CLISP facility, which allows iterative statement operators and record operations to be typed in either all-uppercase or all-lowercase letters: (for X from 1 to 5 ) is the same as (FOR X FROM 1 TO 5 ). The few situations where this is the case are explicitly mentioned in the manual. Generally, one should assume that case is significant.

## 1.5 REFERENCES

[Burton, et al., 1980] Burton, R. R., L. M. Masinter, A. Bell, D. G. Bobrow, W. S. Haugeland, R.M. Kaplan and B.A. Sheil, ''Interlisp-D: Overview and Status'' in [Sheil & Masinter, 1983].

[Foderaro, 1979] Foderaro, John K., *The FRANZ LISP Manual* University of California, Bekeley, California (1979).

[Meehan, 1979] Meehan, J. R., *The New UCI Lisp Manual* Lawrence Erlbaum Associates, Hillsdale, New Jersey (1979).

[Moon, 1974] Moon, David, *MACLISP Reference Manual* Version 0, Laboratory for Computer Science, MIT, Cambridge, Massachusetts, (1974)

[Sandewall, 1978] Sandewall, Erik, ''Programming in the Interactive Environmnet: The LISP Experience'' *ACM Computing Surveys*, vol 10, no 1, pp 35-72, (March 1978).

[Sheil, 1983] Sheil, B.A., ''Environments for Exploratory Programming'' *Datamation*, (February, 1983) also in [Sheil & Masinter, 1983].

[Sheil & Masinter, 1983] Sheil, B.A. and L. M. Masinter, ''Papers on Interlisp-D'', Xerox PARC Technical Report CIS-5 (Revised), (January, 1983).

[Siklossy, 1976] Siklossy, L., *Let's Talk Lisp* Prentice-Hall, Englewood Cliffs, New Jersey (1976).

[Teitelman & Masinter, 1981] Teitelman, W. and L. M. Masinter, ''The Interlisp Programming Environment'' *Computer*, vol 14, no 4, pp 25-34, (April 1981) also in [Sheil & Masinter, 1983].

[Weinreb & Moon, 1979] Weinreb, D. and D. Moon, *Lisp Machine Manual* Artificial Intelligence Laboratory, MIT, Cambridge, Massachusetts, (January 1979).

[Weissman, 1967] Weissman, C., *LISP 1.5 Primer* Dickenson Publishing Company, Belmont, California (1967).

[Winston & Horn, 1981] Winston, P. H., and B.K.P. Horn, *LISP* Addison-Wesley, Reading, Massachusetts (1981).

**References**