# History of Interlisp

Dr .Warren Teitelman
Google, Inc.

**1962**

I was first introduced to Lisp in 1962 as a first year graduate student at M.I.T. in a class taught by James Slagle. Having programmed in Fortran and assembly, I was impressed with Lisp's elegance. In particular, Lisp enabled expressing recursion in a manner that was so simple that many first time observers would ask the question, "Where does the program do the work?" (Answer – between the parentheses!) Lisp also provided the ability to manipulate programs, since Lisp programs were themselves data (S-expressions) the same as other list structures used to represent program data. This made Lisp an ideal language for writing programs that themselves constructed programs or proved things about programs. Since I was at M.I.T. to study Artificial Intelligence, program writing programs was something that interested me greatly.

However, Lisp was at that point in time just a language. Programming in Lisp consisted of submitting a job, usually as a deck of punched cards that was run in batch mode on a main frame. You would then pick up your output a few hours later, if lucky, otherwise the next day, and hope that it did not consist of a lengthy sequence of left parentheses or NILs, as would be the case if the program had certain kinds of bugs.

**1964**

The introduction of time-sharing at M.I.T. in 1964 dramatically changed the paradigm of software development. Instead of the developer doing their debugging offline, the user could now sit and interact directly with his program online. Originally developed as a way of making more efficient and economic use of a very expensive computer, time-sharing had the surprising side-effect of drastically reducing the amount of time it took to get a program working. Users experiencing this phenomenon reported that it was because they did not have to lose and then reestablish context so frequently, but could get very deep into their programs and the problems they presented, and stay there. The situation is analogous to trying to resolve an issue between two people via a conversation rather than sending letters back and forth. Regardless of how short the cycle of iteration is, e.g., if email is used instead of letters, if the process involves discovery and a lot of back-and-forth, it is much easier to do via a conversation. You can establish a context and stay focused until the problem is solved.


*Lisp50*, October 20, 2008, Nashville, Tennessee, USA.


**1965**

I personally experienced this phenomenon when I started working on my Ph.D. project in 1965. At first, I wanted to develop a general game playing program, one that could be given the rules for a new, simple game, and devise a strategy, possibly drawing on games it had previously mastered. (I was both ambitious and naïve!) I quickly realized that I was going to be spending a significant amount of effort changing my program as I evaluated its behavior and identified shortcomings. I would not be able to work out a design and them code and debug it.

This led me to the notion of building a system wherein the computer took an active role in helping me to make changes to a program:

> *The goal of artificial intelligence is to construct computer programs which exhibit the kinds of behavior we call 'intelligent' when we observe it in human beings. These programs are usually so complex that the programmer cannot accurately predict their behavior. He must run them to see whether any changes should be made. Developing these programs thus involves a lengthy trial and error process in which most of the programmer's effort is spent in making modifications. PILOT is a system designed specifically to facilitate making modifications in programs.*

The central innovation in PILOT was a concept I called Advising, wherein the user could treat a particular function or subroutine as a black box and without knowing what was inside the box, wrap "advice" (stealing the term from McCarthy's Advice Taker paper) around it that could operate before the function/subroutine ran, potentially changing its input parameters, after it ran, possibly changing its value, or detour around it entirely. However, what I really was envisioning was a programming environment:

> *This term is meant to suggest not only the usual specifics of programming system and language but also more elusive and subjective considerations such as ease and level of interaction, "forgivefullness" of errors, human engineering, and system "initiative." The programmer's environment influences, to a large extent determines, what sort of problems he can (and will want to) tackle, how far he can go, and how fast. If the environment is "cooperative" and "helpful", then the programmer can be more ambitious and productive. If not, he will spend most of his time and energy "fighting" the system, which at times seems bent on frustrating his best efforts.*

(I did not know at the time that this pursuit would occupy me for the next 20 years.)

In 1965, there were very few tools for developing Lisp programs, and those that were available, were very primitive. There was a utility called Prettyprint which printed out a nicely formatted representation of Lisp programs, using indentation to indicate depth of structure. A Trace facility was also available which modified specified functions to print on the terminal their input parameters on entry and their value on exit. You could think of this as a special case of Advising in that the advice was the same for all programs. There was also a Break package which enabled the user to cause program execution to halt at the entry point to specified functions. The user could then examine the value of the function's arguments (input parameters), even change them, then cause the function to run, and again gain control so as to examine the value that the function returned or side effects of the function's operation. The user could change input parameters and reexecute the function, or manually specify the desired value and have it be returned to the caller as though it had been the value produced by that function.

**1966**

When I received my Ph.D from M.I.T. in 1966, I took a position at Bolt, Beranek and Newman in Cambridge. At the time, BBN's computer was a DEC PDP-1, and Daniel Murphy had written a version of Lisp 1.5 for it. This Lisp was really just a toy – single user, slow, small address space, but I obtained copies of the Break and Prettyprint utilities (both being written in Lisp itself) from MIT, and read them in and thus began my pursuit of a Lisp programming environment.

**1967**

In 1967, BBN purchased an SDS 940 computer from Scientific Data Systems and began work building a time sharing system on it. The SDS 940 had a large address space and the ability to support a paging system. BBN was awarded an ARPA contract to provide a LISP system that could be distributed to other ARPA sites for doing A.I. research. Dan Murphy implemented Lisp 1.5 on the SDS 940, and we called this BBN Lisp. In BBN Lisp, I would over the next five years flesh out my vision of a programming environment for Lisp.

One of the interesting things about the SDS 940 was that it was a *hybrid* processor (SDS's term). In addition to performing the standard operations of a digital computer, the 940 also had some interesting *analog* properties: a portion of its memory could be used as a frame buffer to drive a display. (This may not seem like a big deal, but back in the 60s, every computer did not have an attached display!) I took advantage of this capability to write a circuit drawing program in which the structure of the circuit was maintained in and operated on by Lisp, which in turn would call primitives written in assembly language for performing display operations such as draw a line from A to B or display the following text at location (x,y). We were even able to hook up a stylus to use as input for pointing at various elements of the circuit on the display, and a driver for a Calcomp plotter to produce hardcopy of circuits being designed. This may have been the first example of a CAD program.

**1968**

Bob Kahn was at BBN during this period, and working on what would become the ARPA Net. I wrote a network simulation and display program for him in BBN LISP which enabled him to study various important aspects of networking, especially the ways in which networks become clogged, and to explore algorithms and heuristics for unclogging them. At the time we were looking at a network with perhaps a dozen nodes! We would start a packet from LA to Boston, and then pull Denver off line and see – on the display – whether the packet would automatically reroute to Salt Lake City, and go around Denver, etc.

In the area of programming tools, Peter Deutsch wrote a structure editor in Lisp for editing Lisp programs. Prior to this, Lisp source was prepared and edited offline in textual form and read into the Lisp system. Peter's editor enabled the user to edit Lisp programs without ever leaving Lisp. The editor provided operations for moving up, down, left or right in the list structure definition of a Lisp function, and to make insertions, deletions, or replacements, e.g. (-3 X) to insert X in front of the 3$^{rd}$ item in the current list, 2 to descend into the second item in the current list, 0 to ascend one level, etc. Other more sophisticated commands were soon added, such as a find command to search through all levels of the function being edited looking for a specified string or pattern, a mark command to mark, i.e. save, the current location, and a command to restore the context to one that had previously been marked, ability to define macros, etc.

The ability to edit a Lisp program *in situ* meant that a user could modify a running program and continue execution. For example, the user might be at a Break, evaluate the current function, identify a problem, edit the definition using the structure editor, and reevaluate the current, now modified function and go on.

**1969**

In 1969, BBN acquired a Digital Equipment PDP-10 to replace the SDS 940. The DEC PDP-10 had a 32 bit word, which meant that we had a 256K address space. At the time, that number was so big (the version of LISP 1.5 I used at M.I.T. in the early sixties had about 8K free space!), that we seriously considered not bothering to write a garbage collector. How could you ever run out of space with that much to start with??

Alice K. Hartley took over Dan Murphy's role in BBN Lisp. A number of new data types were added to augment lists and numbers: arrays, strings, large numbers, floating point numbers.

**1970**

As Lisp users began to write larger and larger programs, performance began to be an issue. A compiler had been available for Lisp programs since the early sixties. Compiling a Lisp function eliminated the overhead of interpreting conditional expressions, PROG, GO TO, and primitive arithmetic functions

such as IPLUS, IMINUS, etc. However, calling a function in Lisp, even from within a compiled Lisp function, was still a fairly heavy operation, because of the need to create a new frame on the stack, populate it with input values, and reverse the procedure when the function returned. We addressed this by providing various ways of avoiding a function call. For example, Lisp included a variety of searching functions such as Member, Assoc, etc., all of which used EQUAL for comparison. EQUAL could not be compiled open, but required a function call because of the possibility of having to recursively compare two expressions. We provided corresponding versions of such functions that used EQ instead of EQUAL, which was just a comparison of two pointers, which could therefore be compiled inline, thereby avoiding the function call. If the programmer knew that if a given item being searched for was either atomic, or else if a list, known to be the exact same list structure, they could use MEMB instead of MEMBER, thereby avoiding a function call. Similarly, we provided fast versions of RPLACA and RPLACD, the Lisp functions that physically alter list structures. FRPLACA and FRPLACD eschewed making any checks on their arguments but simply deposited the second parameter into CAR or CDR of their first parameter. This did have the unfortunate consequence of allowing a buggy program to actually clobber NIL, which did very bad things to LISP programs that took advantage of the fact that CAR and CDR of NIL were NIL. Once this happened, almost nothing worked correctly. As a result, we had to implement a check at the top level EVALQT for NIL being clobbered and to inform the user and then restore NIL.

To further reduce the number of function calls, Danny Bobrow came up with the idea of a Block Compiler. The Block Compiler would compile a collection of the programmer's functions into a single block. Calls from one function to another would not be visible on the stack and not require an external function call. We also improved the performance of a number of BBN Lisp tools such as Prettyprint, and the compiler itself, by block compiling them.

In order to better profile where a program was expending resources – compute time, free storage, large numbers, or any other measurable quantity – I wrote a Breakdown package that operated by using the same paradigm as that employed by Break and Advise packages. It wrapped user-specified functions in a call to a function that would compute the value of the quantity being measured, call the specified function, and then compute the value again, and save the aggregate count/value. The user could see a roll up of the resources expended by various components of his program and thereby focus his performance tuning in the appropriate areas.

Another significant extension to the Lisp environment came in 1970 when Danny Bobrow and Alice Hartley designed and implemented the "spaghetti stack". This functionality introduced a new data type, the stack pointer, and enabled running programs to search the current execution stack, e.g., find the second occurrence up the stack of the function FOO, and return the name of the function that called FOO, to alter the normal flow of control, e.g., return from a specified stack pointer a specified value (very useful when debugging programs in order to manually by pass a known problem), and to evaluate an expression or variable in a specified context, e.g. what is the value of x as of six function calls back up the stack. Spaghetti stack functionality was similar to the notion of exceptions, catch, and throw in Java. While the full generality of the spaghetti stack was rarely used, RETFROM – return a specified value, i.e. unwind the stack to the indicated stack pointer, RETEVAL – evaluate an expression in the specified context and return it, STKEVAL – evaluate an expression in the specified context but don't unwind the stack, EVALV – evaluate a variable in the specified context, STKNTH, and STKPOS all saw heavy use, especially in implementing various commands in the Break package, and by DWIM.

DWIM, the most well known, and in some cases, reviled, feature of BBN LISP was introduced in 1970. DWIM stands for Do-What-I-Mean and embodies my view that people time is more valuable/expensive than computer time. When I first started programming in FORTRAN in 1960, I was appalled at receiving the error message, "on line 70, DIMENSION is misspelled". If the Fortran Compiler knew this to be the case, why didn't it accept this and go on an compile my program? It's almost like the computer was the parent and I, the programmer, was the child, and the computer was sending the programmer to his room in punishment for making a mistake. To me, that was wrong.

So one night when using a model 33 teletype whose keys were sticking, causing doubling of characters and consequent undefined function or unbound atom errors to occur, I had the epiphany that any competent Lisp programmer watching over my shoulder, even without knowing the semantics of my program or what I was trying to accomplish, would nevertheless have understood what I was typing despite the typos, so why not have the computer recognize my intent and correct my mistakes?

In order to accomplish this, the BBN-LISP interpreter was modified so that rather than throw an error when an undefined function or unbound atom was encountered, instead it would call the function FAULTEVAL (or in some cases, FAULTAPPLY). FAULTEVAL would be initially defined to throw an error, but it could be redefined by the user. When the user turned DWIM on, FAULTEVAL was redefined to instead call a program that used various heuristics to identify and attempt to correct the error. Spelling correction was the most common scenario. An algorithm was implemented that took advantage of the most common types of errors made by a touch typist, e.g., doubled characters, transpositions, case error, etc. A spelling list appropriate for the context of the error was searched, and a metric computed for each item on the list that measured the difference between that item and the unknown word. If the match was sufficiently close, e.g., the only difference being a doubled character or a transposition, the correction was performed without the user having to approve. Otherwise, the user was offered the closest match and asked to approve the correction. If the user approved or the correction was automatically done, a message was printed on the terminal and computation would continue as though the error had not occurred. If the user was not at the terminal, after an appropriate interval, DWIM would default to Yes or No depending on how close the match was. It was not uncommon for a user to perform some editing, then start a computation, go get some coffee, and come back to find the computation complete with several corrections having been made.

DWIM was also programmed to handle the case where the user typed a number instead of '(' or ')' because of failure to hit the shift key, e.g. 8COND instead of COND. This kind of error was particularly nasty to fix, because not only did it cause a misspelled function or variable, but totally altered the structure of the expression being evaluated. For the user to manually fix such an error using the structure editor required not only removing the 8 or 9, but rearranging the list structure. Having be able to DWIM handle such errors was quite helpful.

Spelling correction was also used in contexts besides evaluating Lisp expressions. For example, there was a spelling list of edit commands that was used to correct a mistyped editor command. When loading a file where the file name was not found, a spelling list of previously encountered file names would be used.

I later used DWIM to extend the syntax of Lisp by taking advantage of the fact that an unrecognized expression would cause a call to FAULTEVAL, where such an expression could be translated to an equivalent Lisp expression. For example, iterative statements were implemented by translating them into the equivalent PROG, MAPC, MAPCAR, et al, when FAULTEVAL was called because FOR or WHILE, etc., were not the name of defined Lisp functions. Similarly, the expression (X + Y) would be translated to (PLUS X Y) the first time it was evaluated because X was not a defined function.

Another innovation introduced to BBN LISP in 1970 was the History package. The idea was rather than simply performing the operations requested by the user, call functions, edit expressions, perform break commands, etc., and discarding that information, to have an agent that would record what the user entered so that the user could examine the history, and replay portions of it, possibly with substitutions. (The history feature of the UNIX C-shell introduced in the late 70's was in fact patterned after the Interlisp history package.) The history also contained any messages displayed to the user during the execution of the corresponding event, e.g., any DWIM corrections, or messages about global variables being reset or functions being redefined, etc.

As with DWIM, the History package grew out of my "laziness" and desire to offload manual tasks to the computer. Frequently during the course of an online session, I found myself wanting to redo a particular operation, or perform the same operation with different parameters. I could see this operation on my terminal just a few lines back. Why couldn't I just tell the computer, "Do that again"?

Perhaps the most important piece of information stored in each history event was the information required to UNDO that operation. This was especially valuable in the context of editing. UNDO is functionality that every user now expects in an editor, but it was first introduced in BBN-LISP in 1970. The UNDO functionality provided in BBN-LISP still surpasses that available in today's editors in that the user could UNDO operations out of order. For example, after performing a series of four or five editing operations, the user could realize that the information deleted in the first operation is needed, and would be able to UNDO just that operation by explicitly referring to that operation using the history package, without affecting the intervening operations.

In addition to being able to UNDO edit operations, the user could also UNDO operations that were typed in at the top level or in a Break. This was most frequently to undo assignments. It could also be used to undo an entire edit session, rather than undoing one command at a time, sort of a revert operation for S-expressions. The user could also arrange to have functions that they defined to be undoable by storing information on the history list.

**1971**

In 1971, CLISP, one of the less successful features, although loved by some, was added to BBN-LISP. CLISP (for Conversational LISP) was my attempt to make the Lisp syntax more palatable by supporting infix notation and fewer parentheses. The user could write (IF X IS LESS THAN Y AND Z IS NULL THEN X + Y ELSE Z * 2) which would translate to the Lisp expression (COND ((AND (LESSP X Y) (NULL Z)) (T (TIMES Z 2))). As with iterative statements, expressions in CLISP were translated when first encountered via the DWIM/FAULTEVAL technique. Alternatively, the user could invoke a DWIMIFY function which would, without actually evaluating any expression, sweep through a program and perform all of the corrections or translations that would have been performed if the program had been run. This was especially useful if the user wanted to compile a function without having run it. A CLISPIFY function was provided to convert LISP conventional Lisp programs into CLISP.

A better received enhancement was the File Package, added in 1971. For those familiar with UNIX, this was essentially a "make" for Lisp. The user could specify the set of functions, global variables, property lists, et al, to be contained in a specified file, and then "make" that file. When the file was loaded in a subsequent session, this information would be recorded and available. Whenever a component known to be in a specified file was modified, the system would know that the corresponding file needed to be rewritten. A cleanup function was provided that would write out all files that contained components that had been changed. The user would be informed about any items created or modified during the course of his session that did *not* appear in any of the user's files, and therefore might be lost if the user abandoned his session without saving them somewhere. The only thing missing from the File Package that would be provided in UNIX Make was the notion of dependencies.

**1972**

In 1972, Danny Bobrow and I left BBN and went to the newly formed Xerox Palo Alto Research Center – PARC. BBN continued to provide the low level support for the Lisp system, i.e., compiler, garbage collector, and all of the SUBRs, whereas the center of activity for the various packages and utilities moved with me to PARC. Both sites continued to be supported by ARPA,

and to indicate this partnership and shared responsibility, we renamed BBN-LISP to be Interlisp.

Around the ARPA net, Interlisp continued to use the DEC PDP-10 as its principal platform, although ports to various other machines were performed, especially the DEC VAX. However, PARC, because of political reasons, could not purchase a Digital Equipment computer. Xerox, having just purchased Scientific Data Systems, was concerned about how Digital Equipment Corporation, with whom they were now competing, would be able to do with such an event. Given the hardware expertise at PARC, the simplest solution was to build our own computer, and so we built MAXC (for Multiple Access Xerox Computer), which emulated a PDP-10 instruction set and could run the code we got from BBN.

**1974**

In 1974, on a visit to Stanford, I met Larry Masinter, who showed me a number of impressive extensions to Interlisp that he had prototyped. These included a much more sophisticated version of Interlisp's iterative statement, as well as what he called a Record package, which enabled a user to label various components of a list structure and refer to them by name, thereby eliminating the CADADRs and CDADDRs that made Lisp programs so difficult to use. The Record package also had the advantage that the user could change a record definition, and his program would automatically adopt the new structure. For example, if PERSON were defined as (RECORD PERSON (FIRSTNAME LASTNAME TITLE)), the expression (X:TITLE) would translate to (CADDR X). If the user later changed the definition of PERSON to (RECORD PERSON (FIRSTNAME INITIAL LASTNAME TITLE)), all expressions involving TITLE would automatically be retranslated to use CADDDR.

I was fortunate to convince Larry to come to Xerox PARC as an intern, and later pleased to act as his de facto thesis adviser as he pursued and received his Ph.D. at Stanford. For his Ph.D. work, Larry developed another widely used feature in Interlisp: Masterscope.

Masterscope would analyze a large program and build a data base of relationships between the various components that could then be queried using a natural language front end. For example, WHO CALLS FOO AND USES MUMBLE, EDIT WHERE X IS USED FREELY AND Y IS BOUND, etc. As LISP programs became larger and more complex, and were being built by teams of programmers, rather than a single programmer, functionality such as that provided by Masterscope was invaluable in understanding, maintaining, and extending programs.

**1975**

By 1975, Interlisp had become so rich in functionality that it was clear that word of mouth was no longer sufficient and in depth documentation was needed, especially since there was a large and growing community of users at the various ARPA sites that had little or no direct contact with the developers of Interlisp at PARC and BBN. I therefore began work on the first Interlisp manual, which turned out to be a year long project. When completed, the manual was over 500 pages and heavily indexed. It was written using PUB, a text formatting program developed at Stanford by Dan Swinehart and Larry Tesler. (This was back in the days when the only WYSIWYG editor was PARC's Bravo which ran only on the Alto.) I had the idea of using the fact that the manual was machine readable, and heavily indexed, to use it to provide online help and documentation. The user could type in something like TELL ME ABOUT FILE PACKAGE and see on his terminal/screen the relevant text. In a break, the user could simply type '?' and see an explanation of the input parameters for the current function.

**1976**

In 1976, Dan Ingalls gave a presentation at one of our weekly Dealer meetings at PARC in which he demonstrated the first window system. Written in and for Smalltalk, the user interface and paradigm it provided for enabling the user to manage and work with multiple contexts was so compelling that I immediately began to consider how we might provide such a mechanism for Interlisp. At the time, although Peter Deutsch had developed a byte-coded instruction set for Interlisp for the D-machines (Dandelion, Dolphin, and Daybreak), implementing Interlisp on these machines was not yet viable as they were somewhat underpowered for Lisp development. Bob Sproull came up with the idea for what would turn out to be the first client-server window system: use the Alto as the window server and Interlisp running on MAXC as the client and develop a protocol for having Interlisp tell the Alto what to display, and for the Alto to tell Interlisp about mouse clicks. Bob developed the ADIS (for Alto Display) package and I wrote DLISP in Interlisp. DLISP included a window manager and windowing system that enabled overlapping windows, cut and paste, etc. J Moore implemented a text package that would support display and editing of text in windows. I demonstrated this functionality at IJCAI in 1977, and presented a paper, a Display Oriented Programmer's Assistant.

**1979**

In 1979, PARC began the design of the Dorado, which is the first real personal computer. A project to specify the requirements for what we called an Experimental Programming Environment (meaning an environment which supported experimental programming) was started. We drew on the experiences of the three programming communities at PARC: Smalltalk, Interlisp, and Mesa. This led to the Cedar project, which I joined in 1980. The availability of the Dorado also made possible building a Lisp with a native display capability, which led to the Interlisp-D project.

**1993**

In 1993, the ACM Software Systems award was given to the Interlisp team: "For their pioneering work in programming environments that integrated source-language debuggers, fully compatible integrated interpreter/compiler, automatic change management, structure-based editing, logging facilities, interactive graphics, and analysis/profiling tools in the Interlisp system."