THE NATURE OF THIS INTERLISP/LOOPS COURSE

Over the next five days, you will be learning some of the features of both INTERLISP and of the AI programming tool, LOOPS. The OSU AI Lab is a beta test site for the LOOPS language, and as such, we have received approval from the LOOPS development team at XEROX PARC to offer this course, and to reproduce some of the LOOPS course materials as offered by XEROX.

You will find that both INTERLISP and LOOPS are extremely flexible and will provide you with the capability build a tailor made programming environment to suit your particular needs. However it is also important to realize that in just five short days, it is impossible to cover all of the extensive facilities offered by INTERLISP and LOOPS. What we will do is to provide you with a starting point in your development of INTERLISP/LOOPS proficiency.

THE CONTENTS OF THIS NOTEBOOK

The OSU LOOPS class notebook contains the following materials.

1. three short papers concerning the INTERLISP environment and the LOOPS language [Notebook Section #II],
2. a set of exercises we will be using throughout the course [Notebook Sections #III - #XIII],
3. a set of descriptive materials about the TRUCKIN knowledge engineering game [Notebook Section #XIV],
4. samples of LOOPS gauges [Notebook Section #XV],
5. samples of LOOPS ClassBrowsers [Notebook Section #XVI],
6. samples of TRUCKIN game boards [Notebook Section #XVII],
7. LOOPS summary and manual [Notebook Section #XVIII], and
8. documentation for the SSI tool [Notebook Section #XIX].

COURSE TIMETABLE

Below is a tenative timetable for this course.

Tuesday, March 20

9:00 to 10:00 - meeting in CAE230
INTERLISP philosophy
INTERLISP specifics for getting started

10:00 to noon - introductory hands-on session for INTERLISP-D
the PA, the file system, DEdit, the INSPECTOR
[NOTEBOOK SECTION #III]

noon to 1:30 - lunch

1:30 to 2:00 - question and answer period in CAE 230

2:00 to 5:00 - continuation of morning exercises
[NOTEBOOK SECTION #III],
OSU turtle exercises
[NOTEBOOK SECTION #IV]

5:00 to 7:00 - dinner

7:30 to 11:00 - the machine room will be open for further work

Wednesday, March 21

9:00 to 10:00 - meeting in CAE230
Top Level LOOPS description,
object oriented programming

10:00 to noon - introductory hands-on session for LOOPS
[NOTEBOOK SECTION #V]

noon to 1:30 - lunch

1:30 to 2:30 - question and answer period in CAE 230,
active values

2:30 to 5:00 - continuation of morning exercises,
[NOTEBOOK SECTION #VI, #VII],
LOOPS turtle exercises
[NOTEBOOK SECTION #VIII]

5:00 to 7:00 - dinner

7:30 to 11:00 - the machine room will be open for further work

Thursday, March 22

9:00 to 10:00 - meeting in CAE230
topic - Managing the LOOPS environment, Browsers

10:00 to noon - hands-on session for LOOPS Browsers, building a Key Class Browser
[NOTEBOOK SECTION #IX]

noon to 1:30 - lunch

1:30 to 3:00 - meeting in CAE 230
chandra on OSU AI
introduction to the Rule Language

3:00 to 5:00 - knowledge engineering exercises
[NOTEBOOK SECTION #X]

5:00 to 7:00 - dinner

7:30 to 11:00 - the machine room will be open for further work

Friday, March 23

9:00 to noon - continuation of knowledge engineering exercises
[NOTEBOOK SECTION #XI]

noon to 1:30 - lunch

1:30 to 5:00 - meeting at Battelle

5:00 to 7:00 - dinner

7:30 to 11:00 - the machine room will be open for further work

Saturday, March 24

9:00 to 9:30 - meeting in CAE230
one tool built on top of LOOPS: the SSI, a system built on the SSI, MDX/MYCIN

9:30 to noon - SSI exercises
[NOTEBOOK SECTION #XII]

noon to 1:30 - lunch

11:00 to noon - meeting in CAE230
a tool for expressing
Diagnostic Systems - CSRL

2:00 to 5:00 - CSRL exercisesa
[NOTEBOOK SECTION #XIII]

5:30 to ??? - wrap up session (ie party) at jon's house
6000 Dublin Road
Dublin
(if lost call 889-2914)

A0096

Further steps in the flight from time-sharing

The Interlisp-D group.

Abstract

One of the goals of the Interlisp-D effort has been to provide Interlisp's programming support tools in a personal computing environment. This report outlines the current status of the Interlisp-D implementation, and describes some of the interactive programming tools that have recently been added to the system.

BACKGROUND

The Interlisp-D project was formed to develop a personal machine implementation of Interlisp for use as an environment for research in artificial intelligence and cognitive science [Burton *et al.*, 80b]. This note describes the principal developments since our last report almost a year ago [Burton *et al.*, 80a].

Principal characteristics of Interlisp-D

Interlisp-D is an implementation of the Interlisp programming environment [Teitelman & Masinter, 81] for the Dolphin and Dorado personal computers. Both the Dolphin and Dorado are microprogrammed personal computers, with 16-bit data paths and relatively large main memories (~1 megabyte) and virtual address spaces (4M-16M 16 bit words). Both machines have a medium sized local disk, Ethernet controller, a large raster scanned display and a standard Alto keyboard and "mouse" pointing device.

Both the internal structure of Interlisp-D and an account of its development are presented in [Burton *et al*, 80b]. Briefly, Interlisp-D uses a byte-coded instruction set, deep binding, CDR encoding (in a 32 bit CONS cell) and incremental, reference counted garbage collection. The use of deep binding, together with a complete implementation of spaghetti stacks, allows very rapid context switching for both system and user processes. Virtually all of the Interlisp-D system is written in Lisp. A relatively small amount of microcode implements the Interlisp-D instruction set and provides support for a small set of other performance critical operations. The at one time quite large Bcpl kernel has been all but completely absorbed into Lisp, for the reasons outlined in [Burton *et al*, 80b].

Interlisp-D is completely upward compatible with the widely used PDP-10 version. All the Interlisp system software documented in the Interlisp Reference Manual [Teitelman *et al.*, 78] runs under Interlisp-D, excepting only a few capabilities explicitly indicated in that manual as applicable only to Interlisp-10. The completeness of the implementation has been demonstrated by the fact that several very large, independently developed, application systems, such as the KLONE knowledge representation language [Brachman, 78], have been

---

A version of this paper is to appear in SIGART Newsletter, No. 77, August 1981.

brought up in Interlisp-D with little or no modification. Interlisp-D is in active use by researchers (other than its implementors) at both Xerox PARC and Stanford University and is now approaching the level of stability and reliability of Interlisp-10.

CURRENT PERFORMANCE

The performance engineering of a large Lisp system is distinctly non-trivial. We have invested considerable effort, including the development of several performance analysis tools, on the performance of Interlisp-D and, as a result, seen its performance improve by nearly a factor of five over the last year. Although relative performance estimates can be misleading, because of variation due to choice of benchmarks and compilation strategy, the overall performance of Interlisp-D on the Dolphin currently seems to be about twice that of Interlisp-10 on an otherwise unloaded PDP KA-10. Although this level of performance makes the Dolphin a comfortable personal working environment, we have identified a number of improvements which we anticipate will further improve execution speed by 20% to 100%.

MACHINE INDEPENDENCE

Another major thrust has been to reduce the dependencies on specific features of the present environment, so as to facilitate Interlisp-D's implementation on other hardware. Dependencies on the operating system have been removed by absorbing most of the higher (generally machine independent) facilities provided by the operating system into Lisp code. Gratuitous dependencies on attributes of the hardware, such as the 16-bit word size, have been removed and inherent ones isolated. In addition to an abstract desire for transportability, our sharing of code with other Interlisp implementation projects provides a on-going motivation for this effort.

EXTENDED FUNCTIONALITY

The principal innovations in Interlisp-D, with respect to previous implementations of Interlisp, involve the extensions required to allow the Interlisp user access to a personal machine computing environment.

Network facilities

While network access is a valuable facility in any computing environment, it is of particular importance to the user of a personal machine, as it is the means by which the shared resources of the community are accessed. Over the last year, Interlisp-D has incorporated both low level Ethernet access and a collection of various higher level protocols used to communicate with the printing and file servers in use at PARC. It is now straightforward to conduct *all* file operations directly with remote file servers. This both allows the sharing of common files (e.g., for multi-person projects, such as the construction of Interlisp-D itself), permits a user to move easily from one machine to another, and eliminates any constraints of local disk size. We have also begun to investigate the possibility of paging from a remote virtual memory elsewhere on the network. This would not only allow completely transparent relocation of a user's environment from one machine to another, but would open up a variety of interesting schemes for distributing a computation across a set of machines.

High level graphics facilities

Interlisp-D has always had a complete set of raster scan graphics operations (documented in [Burton, 80b]). More recent developments include a collection of higher level user graphics facilities, akin to those found in other personal computing environments. The most important of these is the Interlisp-D window package. This facility differs in spirit from most other window systems in that, rather than imposing an elaborate structure on programs that use it, it is a self consciously *minimal* collection of facilities which allow multiple programs to share the same display. Although some mechanism is necessary to adjudicate a harmonious sharing of the display, we feel that higher level display structuring conventions are still an open research question and therefore should not yet be incorporated into a mandatory system facility. The window package *does* provide both interactive and programatic constructs for creating, moving, reshaping, overlapping and destroying windows, in such a way that a program can be embedded in a window in a completely transparent (to that program) fashion. This allows existing programs to continue to be used without change, while providing a base for experimentation with more complex window semantics in the context of individual applications.

One such existing application is a display based, structural program editor. This editor, in contrast to the character orientation of most modern display based program editors, is the result of marrying display techniques (selection and command specification by pointing, incremental reprinting, *etc*) with the structure orientation of the existing Interlisp editor. Indeed, the two editors are interfaced so that the considerable symbolic editing power of the existing editor remains available under the display based one. Although our initial experience has been positive, the user interface is under continued revision as we gain further experience with this style of editing.

FUTURE PLANS

The area in which we anticipate most future development of Interlisp-D is the personal computing facilities, such as graphics and networking, and their integration into Interlisp's rich collection of programming support tools. While radical changes to the underlying language structures are made difficult by our desire to preserve exact Interlisp compatibility, we also expect some language extensions, including some form of object oriented procedure invocation.

One of the great strengths of Interlisp has been the many contributions made by its active, critical user community. We are hopeful that the recent commercial availability of Interlisp-D to other sites, and the consequent growth of its user community, will be a similar source of long term strength and, in the short term, significantly accelerate the pace with which Interlisp evolves away from its time-shared origins into a personal computing environment.

REFERENCES

Brachman, R. *et al.*
KLONE Reference Manual. BBN Report No. 3848, 1978.

Burton, R. *et al.*
Overview and status of DoradoLisp. *Proceedings of the 1980 Lisp Conference*, Stanford, 1980a.

Burton, R. *et al.*
Papers on Interlisp-D. Xerox PARC report, SSL-80-4, 1980b.

Teitelman, W. *et al.*
The Interlisp reference manual. Xerox PARC, 1978.

Teitelman, W. and Masinter, L.
The Interlisp programming environment. *IEEE Computer*, 14:4 April 1981, pp. 25-34.

The members of the Interlisp-D group are Beau Sheil, Bill van Melle, Alan Bell, Richard Burton, Ron Kaplan and Larry Masinter.

VI b.

Keep single-sided original for duplicating

Environments for exploratory programming

Beau Sheil

Cognitive and Instructional Sciences
Xerox Palo Alto Research Center, Palo Alto, California 94304 (USA)

[A version of this paper is to appear in Datamation, *February 1983]*

An oil company needs a system to monitor and control the increasingly complex and frequently changing equipment used to operate an oil well. A electronic circuit designer plans to augment a circuit layout program to incorporate a variety of vaguely stated "design rules". A newspaper wants a page layout system to assist editors in balancing the interlocking constraints that govern the placement of stories and advertisments. A government agency envisions a personal workstation that would provide a single, integrated interface to a variety of large, evolving data base systems.

Applications like these are forcing the commercial deployment of a radically new kind of programming system. First developed to support research in artificial intelligence and interactive graphics, these new tools and techniques are based on the notion of *exploratory programming*, the conscious intertwining of system design and implementation. Fueled by dramatic changes in the cost of computing, such *exploratory programming environments* have become, virtually overnight, a commercial reality. No fewer than four such systems were displayed at NCC '82 and their numbers are likely to increase rapidly as their power and range of application become more widely appreciated.

Exploration and implementation

Despite the diversity of subject matter, a common thread runs through our example applications. They are, of course, all large, complex programs whose implementations will require significant resources. Their more interesting similarity, however, is that it is extremely difficult to give complete specifications for any of them. The reasons range from sheer complexity (the circuit designer can't anticipate all the ways in which all the potential design rules might interact), through continually changing requirements (the equipment in the oil rig changes, as do the information bases that the government department is required to consult), to the subtle human factors issues which determine the effectiveness of an interactive graphics interface.

Whatever the cause, a large programming project with uncertain or changing specifications is a particularly deadly combination for conventional programming techniques. Virtually all of modern programming methodology is predicated on the assumption that a programming project is fundamentally a problem of *implementation*, rather than *design*. The design is supposed to be decided on *first*, based on specifications provided by the client; the implementation follows. This dichotomy is so important that it is standard practice to recognize that a client may have only a partial understanding of his needs, so that extensive consultations may be required to ensure a complete specification with which the client will remain happy. This dialogue ensures a fixed specification which will form a stable base for an implementation.

The vast bulk of existing programming practice and technology, such as structured design methodology, is designed to ensure that the implementation does, in fact, follow the specification in a controlled fashion, rather than wander off in some unpredictable direction. And for good reason. Modern programming methodology is a significant achievement that has played a major role in preventing the kind of implementation disasters that often befell large programming projects in the 1960s.

The implementation disasters of the 1960s, however, are slowly being succeeded by the design disasters of the 1980s. The projects described above simply will not yield to conventional methods. Any attempt to obtain an exact specification from the client is bound to fail because, as we have seen, the client *does not know and cannot anticipate* exactly what is required. Indeed, the most striking thing about these examples is that the clients' statements of their problems are really *aspirations*, rather than *specifications*. And since the client has no experience in which to ground these aspirations, it is only by exploring the properties of some putative solutions that the client will find out what is really needed. No amount of interrogation of the client or paper exercises will answer these questions; one just has to try some designs to see what works.

The consequences of approaching problems like these as routine implementation exercises are dramatic. First, the implementation team begins by pushing for an exact specification. How long the client resists this coercion depends on how well he really understands the limits of his own understanding of the problem. Sooner or later, however, with more or less ill-feeling, the client accepts a specification and the implementation team goes to work. The implementors take the specification, partition it, define a module structure that reflects this partitioning, freeze the interfaces between them, and repeat this process until the problem has been divided into a large number of small, easily understood and easily implementable pieces. *Control* over the implementation process is achieved by the imposition of *structure* which is then *enforced* by a variety of management practices and programming tools.

Since the specification, and thus the module structuring, is considered fixed, one of the most effective methods for enforcing it is the use of *redundant descriptions* and *consistency checking*. Thus the importance of techniques such as interface descriptions and static type checking, which require multiple statements of various aspects of the design in the program text in order to allow mechanical consistency checks to ensure that each piece of the system remains consistent with the rest. In a well executed conventional implementation project, a great deal of internal rigidity is built into the system in this way in the interests of ensuring its orderly development.

The problems emerge, usually at system acceptance time, when the client requests, not just superficial, but *radical* changes, either as a result of examining the system, or for some completely exogenous reason. From the point of view of conventional programming practice, this indicates a failure at specification time. The software engineer should have been more persistent in obtaining a fuller description of the problem, in involving all the effected parties, *etc.*. And this is often true. Many ordinary implementation exercises are brought to ruin by insufficient attention having been paid to getting the consequences of the specification fully agreed to. But that's not the problem here. The oil company didn't know about the new piece of equipment whose behavior is very different from the existing equipment on which the specification was based. No one knew that the layout editors would complain that it doesn't "feel right" now that they can no longer physically handle the copy (even in retrospect, it's unclear why they feel that way and what to do about it). *Etc., etc., etc.*. Neither would any amount of speculation by either client or software engineer have helped. Rather, it would just have prompted an already nervous client to demand whole dimensions of flexibility that would *not* in fact be needed, leaving the system just as unprepared for the ones that eventually turned out to matter.

Whatever the cause, the implementation team has to rework the system to satisfy a new, and significantly different, specification. That puts them in a situation that conventional programming methodology simply refuses to acknowledge (except as something to avoid). As a result, their programming tools and methods are suddenly of limited effectiveness, if not actually counterproductive. The redundant descriptions and imposed structure that were so effective in constraining the program to follow the old specification have lost none of their efficacy - they *still* constrain the program to follow the old specification. And they're difficult to change. The whole point of redundancy is to protect the design from a single (unintentional) change. But it's equally well protected against a single *intentional* change. Thus, all the changes have to be made all over the place. (And, since this should never happen, there's no methodology to guide or programming tools to assist this process.) Of course, if the change is small (as it "should" be), there is no particular problem. But if it is large, so that it cuts across the module structure, the implementation team finds that they literally have to fight their way out of their previous design.

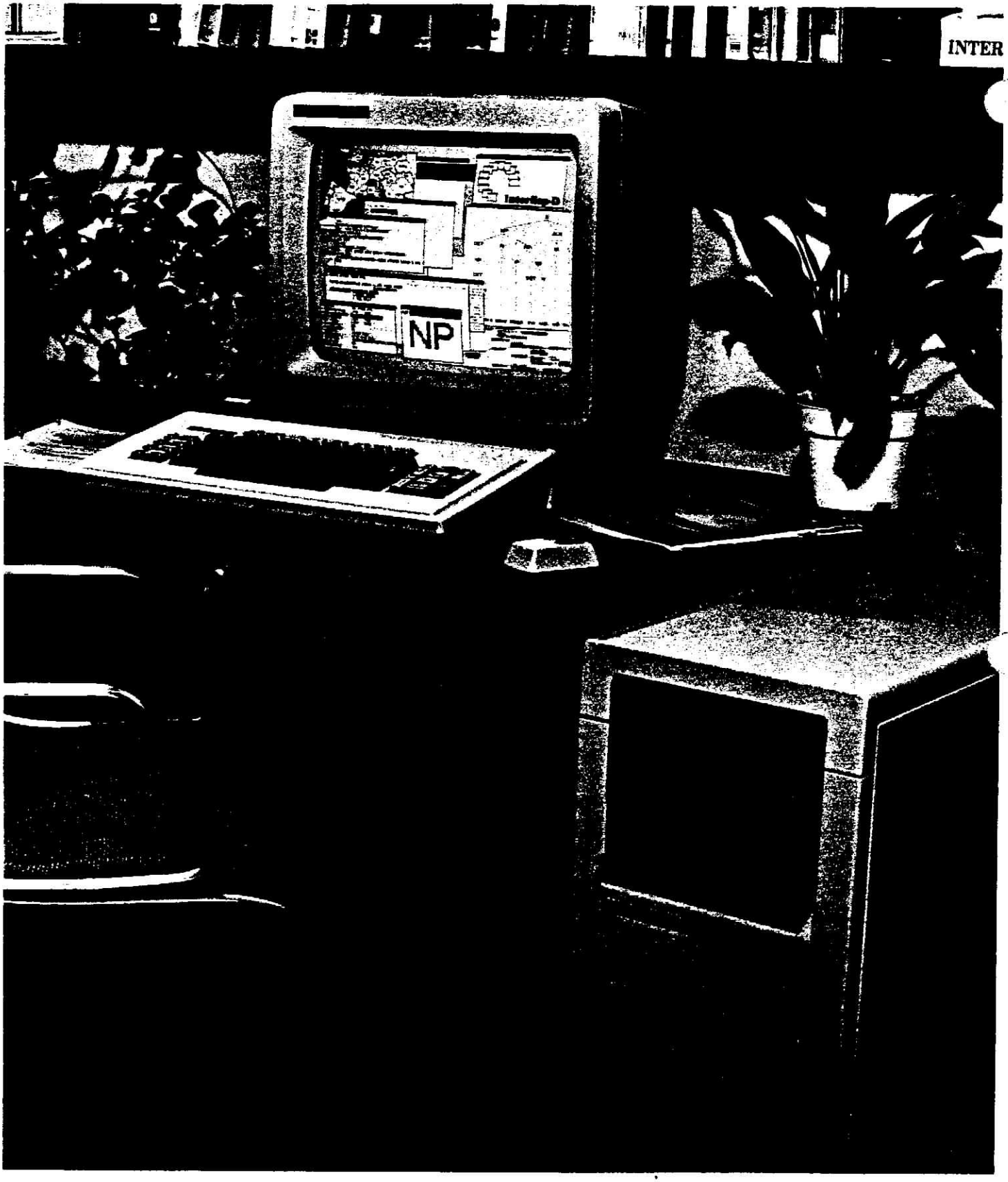
Still no major problem, if that's the end of the matter. But it rarely is. The new system will suggest yet another change. And so on. And on. After a few iterations of this, not only are the client and the implementation team not on speaking terms, but the repeated assaults on the module structure have likely left it looking like spaghetti. It still gets in the way (firewalls are just as impenetrable if laid out at random as they are when laid out straight), but has long ceased to be of any use to anyone except to remind them of the sorry history. Increasingly, it is actively subverted (enter LOOPHOLES, UNSPECs, *etc.*) by programmers whose patience is running thin. Even were the design suddenly to stabilize (unlikely in the present atmosphere), all the seeds have now been sown for an implementation disaster as well.

Programming as exploration

The alternative to this kind of predictable disaster is *not* to abandon structured design for programming projects which are, or which can be made, well-defined. That would be a tremendous step backwards. Instead, we should recognize that some applications are best thought of as *design problems*, rather than implementation projects. These problems require programming systems which allow the design to emerge from experimentation with the program, so that design and program develop together. Environments in which this is possible were first developed in artificial intelligence and computer graphics, two research areas which are particularly prone to specification instability.

At first sight, artificial intelligence might seem to be an unlikely source of programming methodology. However, constructing programs, in particular, programs which carry out some intelligent activity, is central to artificial intelligence. Since almost any intelligent activity is likely to be poorly understood (once something becomes well understood we usually no longer consider it "intelligent"), the artificial intelligence programmer invariably has to restructure his program many, many times before it becomes reasonably proficient. In addition, since intelligent activities are complex, the programs tend to be very large, yet they are invariably built by very small teams (often a single researcher). Consequently, they are usually at or beyond the manageable limits of complexity for their implementors. In response, a variety of programming environments based on the Lisp programming language have evolved to aid in the development of these large, rapidly changing systems.

The rapidly developing area of interactive graphics has encountered similar problems. Fueled by the rapid drop in the cost of computers capable of supporting interactive graphics, there has been an equally rapid development of applications which make heavy use of interactive graphics in their user interfaces. Not only was the design of such interfaces almost completely virgin territory as little as ten years ago, but even now, when there are a variety of known techniques (*e.g.*, menus, windows, *etc.*) for exploiting this power, it is still



Xerox 1108 Interlisp-D system

An exploratory programming system designed to be installed in the user's office. Processor, main memory (1.5 MBytes), rigid and flexible local disks, and Ethernet connection are all contained in the processor cabinet at lower right. The "mouse" pointing device, which moves a cursor image over the display according to sensed horizontal motion across the table, can be seen to the right of the keyboard, in front of the display.

Photo: Ken Beckman

very difficult to determine how easy it will be to use a proposed user interface and how well it will match the user's needs and expectations in particular situations. Consequently, complex interactive interfaces usually require extensive empirical testing to determine whether they are really effective and considerable redesign to make them so. While interface design has always required some amount of tuning, the vastly increased range of possibilities available in a full graphics system has made the design space unmanageably large to explore without extensive experimentation. In response, a variety of systems, of which Smalltalk is the most well known, have been developed to facilitate this experimentation by providing a wide range of built in graphical abstractions and methods of modifying and combining them together into new forms.

Exploratory programming systems

In contrast to conventional programming technology, which *restrains* the programmer in the interests of orderly development, exploratory programming systems must *amplify* the programmer in the interests of maximizing his effectiveness. Exploration in the realm of programming can require small numbers of programmers to make essentially arbitrary transformations to very large amounts of code. Such programmers need *programming power tools* of considerable capacity or they will simply be buried in detail. So, like an amplifier, their programming system must magnify their necessarily limited energy and minimize extraneous activities that would otherwise compete for their attention.

One source of such power is the use of interactive graphics. Exploratory programming systems have capitalized on recent developments in personal computing with extraordinary speed. Consider, for example, the Xerox 1108 Interlisp-D system shown on the facing page. The large format display and "mouse" pointing device allow very high bandwidth communication with the user. Exploratory programming environments have been quick to seize on the power of this combination to provide novel programming tools, as we shall see.

In addition to programming tools, these personal machine environments allow the standard features of a professional workstation, such as text editing, file management and electronic mail, to be provided *within the programming environment itself.* Not only are these facilities just as effective in enhancing the productivity of programmers as they are for other professionals, but their integration into the programming environment allows them to be used at any time during programming. Thus, a programmer who has encountered a bug can send a message reporting it while remaining within the debugger, perhaps including in the message some information, like a backtrace, obtained from the dynamic context.

Another apparent source of power is to build the important abstract operations and objects of some given application area directly into the exploratory environment. All programming systems do this to a certain extent; some have remarkably rich structures for certain domains, for example, the graphics abstractions embedded within Smalltalk. If the abstractions are well chosen, this approach can yield a powerful environment for exploration within the chosen area, because the programmer can operate entirely in substantively meaningful abstractions, taking advantage of the considerable amount of implementation and design effort that they represent.

The limitations of this approach, however, are clear. Substantive abstractions are necessarily effective only within a particular topic area. Even for a given area, there is generally more than one productive way to partition it. Embedding one set of abstractions into the programming system encourages developments that "fit" within that view of the world at the expense of others. Further, if one enlarges one's area of activity even slightly, a set of abstractions that was once very effective may become much less so. In that situation, unless there are effective mechanisms for reshaping the built in abstractions to suit the

changed domain, users are apt to persist with them, at the cost of distorting their programs. Embedded abstractions, useful though they are, by themselves enable only *exploration in the small*, confined within the safe borders where the abstractions are known to be effective. For *exploration in the large*, a more general source of programming power is needed.

Of course, the exact mechanisms which different exploratory systems propose as essential sources of programming power vary widely, and these differences are hotly debated within their respective communities. Nevertheless, despite strong surface differences, the systems share some unusual characteristics at both the language and environment level.

Languages

The key property of the programming languages used in exploratory programming systems is their emphasis on *minimizing and deferring the constraints* placed on the programmer, in the interests of minimizing and deferring the cost of making large scale program changes. Thus, not only are the conventional structuring mechanisms based on redundancy not used, but the languages make extensive use of *late binding*, *i.e.*, allowing the programmer to defer commitments as long as possible.

The clearest example is that exploratory environments invariably provide dynamic storage allocation with automatic reclamation (garbage collection). To do otherwise imposes an intolerable burden on the programmer to keep track of all the paths through his program that might access a particular piece of storage to ensure that none of them access or release it prematurely (and that someone does release it eventually!). This can only be done either by careful isolation of storage management or with considerable administrative effort. Both are incompatible with rapid, unplanned development, so neither is acceptable. Storage management must be provided by the environment itself.

Other examples of late binding include the *dynamic typing* of variables (associating data type information with a variable at run-time, rather than in the program text) and the *dynamic binding* of procedures. The freedom to defer deciding the *type* of a value until run-time is important because it allows the programmer to experiment with the type structure itself. Usually, the first few drafts of an exploratory program implement most data structures in general, inefficient structures such as linked lists, discriminated (when necessary) on the basis of their contents. As experience with the application evolves, the critical distinctions which determine the type structure are themselves determined by experimentation, and may be among the last, rather than the first, decisions to evolve. Dynamic typing makes it easy for the programmer to write code which keeps these decisions as tacit as possible.

The dynamic binding of procedures is more than a simple load-time linkage. It allows the programmer to change dynamically the subprocedures invoked by a given piece of code, simply by changing the run-time context. The simplest form of this is to allow procedures to be used as arguments or as the value of variables. More sophisticated mechanisms allow procedure values to be computed or even encapsulated inside the data values on which they are to operate. This packaging of data and procedures into a single object, known as "object oriented programming", is a very powerful technique. For example, it provides an elegant, modular solution to the problem of generic procedures (i.e., every data object can be thought of as providing its own definition for common actions, such as printing, which can be invoked in a standard way by other procedures). For these reasons, object oriented programming is a widely used technique in exploratory programming, and actually forms the basic programming construct of the Smalltalk language.

The dynamic binding of procedures can be taken one step further when procedures are represented as data structures which can be effectively manipulated by other programs.

While this is of course possible to a limited extent by reading and writing the text of program source files, it is of much greater significance in systems that define an explicit representation for programs as syntax trees or their equivalent. This, coupled with the interpreter or incremental compiler provided by most exploratory programming systems, is an extraordinarily powerful tool. Its most dramatic application is in programs that construct other programs that they later invoke. This technique is often used in artificial intelligence in situations where the range of possible behaviors is too large to encode efficiently as data structures but can easily be expressed as combinations of procedure fragments. An example might be a system which "understands" instructions given in natural language by analyzing each input as it is received, building a *program* which captures its "meaning", and then *evaluating* that program to achieve the requested effect.

Aside from such specialized applications, effective methods for mechanically manipulating procedures enable two other significant developments. The first is the technique of program development by writing interpreters for special purpose languages. Once again, this is a basic technique of artificial intelligence that has much wider applicability. The key idea is that one develops an application by designing a special language in which the application is relatively easy to state. Like any notation, such a language provides a concise representation which suppresses common or uninteresting features in favor of whatever the designer decides is more important. A simple example is the use of notations like context free grammars (BNF) to "meta-program" the parsers for programming languages. Similar techniques can be used to describe, among other things, user interfaces, transaction sequences, and data transformations. Application development in this framework is a dialectic process of designing the application language and developing an interpreter for it, since both the language and the interpreter will evolve during development. The simplest way of doing this is to evolve the application language out of the base provided by the development language. Simply by allowing the application language interpreter to call the development language interpreter, expressions from the development language can be used wherever the application language currently has insufficient power. As one's understanding of the problem develops, the application language becomes increasingly powerful and the need to escape into the development language becomes less important.

Programming tools

The second result of having procedures be easily manipulated by other procedures is that it becomes easy to write program manipulation subsystems. This in turn has two key consequences. First, the exploratory programming language itself can grow. The remarkable longevity of Lisp in the artificial intelligence community is in large part due to the language having been repeatedly extended to include modern programming language syntax and constructions. The vast majority of these extensions were accomplished by defining source to source transformations which converted the new constructions into more conventional Lisp expressions. The ease with which this can be done allows each user, and even each project, to extend the language to capture the idioms that are found to be locally useful.

Second, the accessibility of procedures to mechanical manipulation facilitates the development of programming support tools. All exploratory programming environments boast a dazzling profusion of programming tools. To some extent, this is a virtue of necessity, as the flexibility necessary for exploration has been gained at considerable sacrifice in the ability to impose structure. That loss of structure could easily result in a comensurate loss of control by the programmer. The programming tools of the exploratory programming environment enable the programmer to reimpose the control that would be provided by structure in conventional practice.

[Continued on page 10]

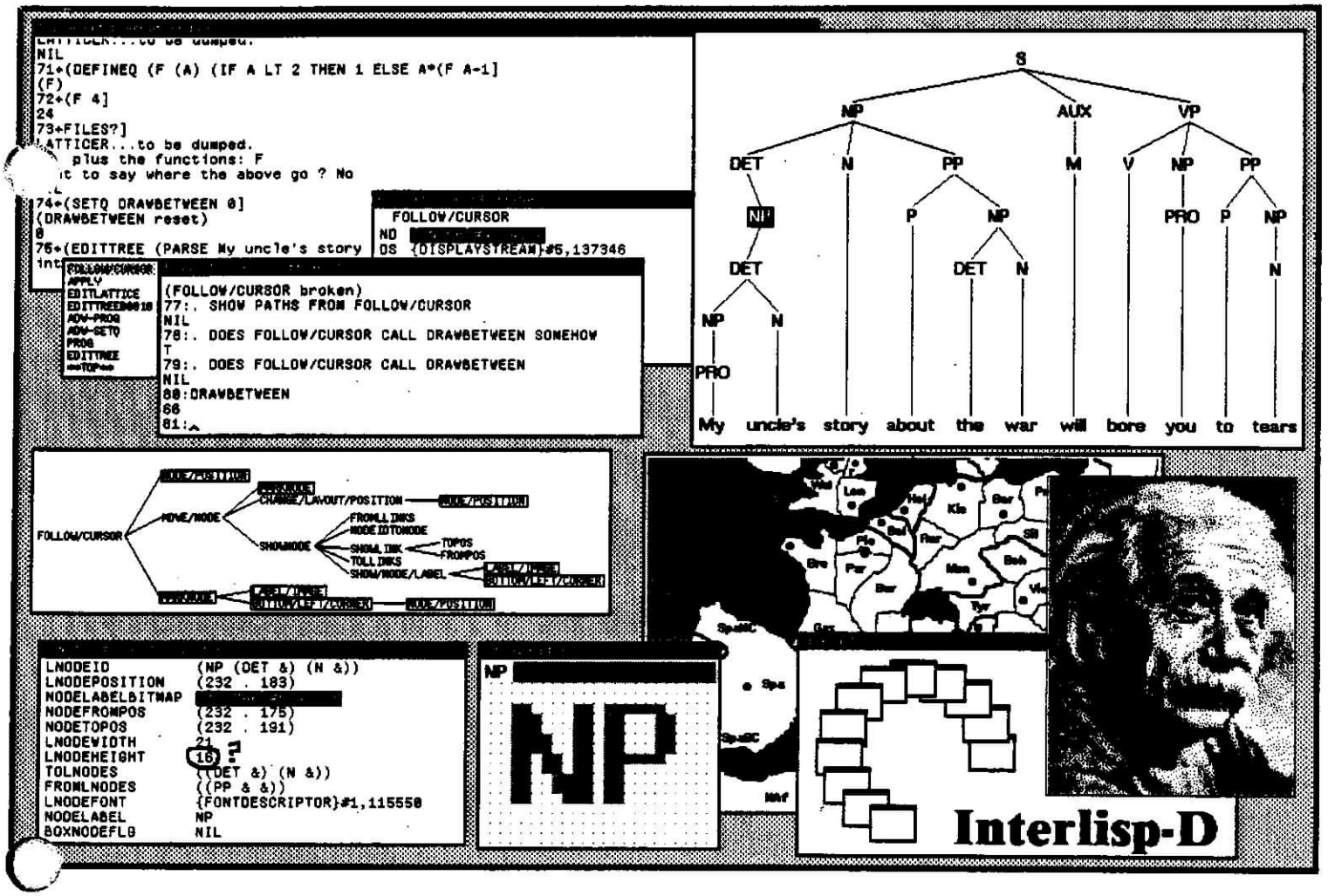
The two screen images at right show some of the exploratory programming tools provided in the Interlisp-D environment. The screen is divided into several rectangular areas or windows, each of which provides a view onto some data or process and which can be reshaped and repositioned at will by the user. When they overlap, the occluded portion of the lower window is automatically saved, so that it can be restored when the overlapping window is removed. Since the display is bitmapped, each window can contain an arbitrary mixture of text, lines, curves, and half-tone and solid area images.

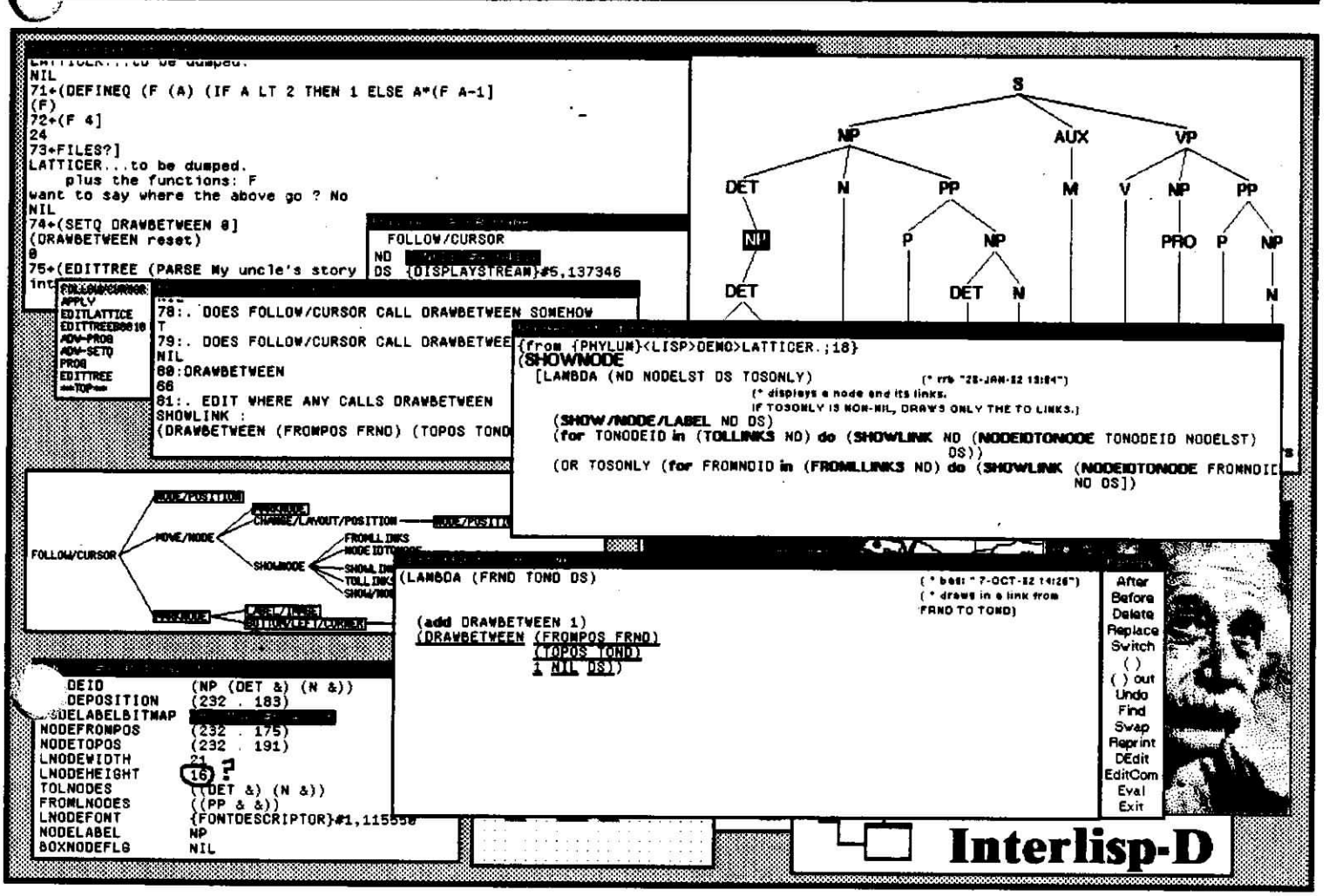
In the typescript window (upper left), the user has defined a program F (factorial) and has then immediately run it, giving an input of 4 and getting a result of 24. Next, he queries the state of his files, finding that one file has been changed (previously) and one function (F) has been defined but not associated with any file yet. The user sets the value of DRAWBETWEEN to 0 in command 74, and the system notes that this is a change and adds DRAWBETWEEN to the set of "changed objects" that might need to be saved.

Then, the user runs the program EDITTREE, giving it a parse tree for the sentence "My uncle's story about the war will bore you to tears". This opens up the big window on the right in which the sentence diagram is drawn. Using the mouse, the user starts to move the NP node on the left (which is inverted to show that it is being moved). While the move is taking place, the user interrupts the tree editor, which suspends the computation and causes three "break" windows to appear on top of the lower edge of the typescript. The smallest window shows the dynamic state of the computation, which has been broken inside a subprogram called FOLLOW/CURSOR. The "FOLLOW/CURSOR Frame" window to the right shows the value of the local variables bound by FOLLOW/CURSOR. One of them has been selected (and so appears inverted) and in response, its value has been shown in more detail in the window at the lower left of the screen. The user has marked one of the component values as suspicious by drawing on it using the mouse. In addition, he has asked to examine the contents of the BITMAP component, which has opened up a bitmap edit window to the right. This shows an enlarged copy of the actual NP image that is being moved by the tree editor.

Inside the largest break window, the user has asked some questions about FOLLOW/CURSOR, and queried the value of DRAWBETWEEN (now 66). The SHOW PATHS command brought up the horizontal tree diagram on the left, which shows which subprograms call each other, starting at FOLLOW/CURSOR. Each node in the call tree produced by the SHOW PATHS command is an active element which will respond to the user's selecting it with the mouse. In the second image, the user has selected the SHOWNODE subprogram, which has caused its code to be retrieved from the file (<LISP>DEMO>LATTICER) on the remote file server (PHYLUM) where it was stored and displayed in the "Browser printout window" which has been opened at middle right. User programs and extended Lisp forms (like **for** *and* **do***) are highlighted by system generated font changes. By selecting nodes in the SHOW PATHS window, the user could also have edited or obtained a summary description of any of the subprograms.*

Instead, the user has asked (in the break typescript window) to edit wherever anyone calls the DRAWBETWEEN program (which draws lines between two specified points). This request causes the system to consult its (dynamically maintained) database of information about user programs, wherein it finds that the subprogram SHOWLINK calls DRAWBETWEEN. It therefore loads the code for SHOWLINK into an edit window which appears under the "Browser print out window". The system then automatically finds and underlines the first (and only) call on DRAWBETWEEN. On the previous line, DRAWBETWEEN is used as a variable (the one the user set and interrogated earlier). The system, however, knows that this is not a subprogram call, so it has been skipped. If the user makes any change to SHOWLINK in the editor, not only will the change take effect immediately, but SHOWLINK will be marked as needing to be updated in its file and the information about it in the program database will be updated. This, in turn, will cause the SHOW PATHS window to be repainted, as its display may no longer be valid.


NIL
71+(DEFINEQ (F (A) (IF A LT 2 THEN 1 ELSE A*(F A-1]
(F)
72+(F 4]
24
73+FILES?]
LATTICER...to be dumped.
plus the functions: F
it to say where the above go ? No
74+(SETQ DRAWBETWEEN 0]
(DRAWBETWEEN reset)
0
76+(EDITTREE (PARSE My uncle's story
FOLLOW/CURSOR
ND
DS {DISPLAYSTREAM}#5,137346
APPLY
EDITLATTICE
EDITTREEB0010
ADV-PROG
ADV-SETQ
PROG
EDITTREE
==TOP==
(FOLLOW/CURSOR broken)
77:. SHOW PATHS FROM FOLLOW/CURSOR
NIL
78:. DOES FOLLOW/CURSOR CALL DRAWBETWEEN SOMEHOW
T
79:. DOES FOLLOW/CURSOR CALL DRAWBETWEEN
NIL
80:DRAWBETWEEN
66
81:
FOLLOW/CURSOR
MOVE/NODE
CHANGE/LAYOUT/POSITION
SHOWNODE
FROMLLINKS
NODEIDTONODE
SHOWLINK
TOPOS
FROMPOS
TOLLINKS
SHOW/NODE/LABEL
LNODEID (NP (DET &) (N &))
LNODEPOSITION (232 . 183)
NODELABELBITMAP
NODEFROMPOS (232 . 175)
NODETOPOS (232 . 191)
LNODEWIDTH 21
LNODEHEIGHT 16
TOLNODES ((DET &) (N &))
FROMLNODES ((PP & &))
LNODEFONT {FONTDESCRIPTOR}#1,115550
NODELABEL NP
BOXNODEFLG NIL
NP
S
NP
AUX
VP
DET
N
PP
M
V
NP
PP
NP
P
NP
PRO
P
NP
DET
N
DET
N
N
NP
N
PRO
My uncle's story about the war will bore you to tears
Interlisp-D


NIL
71+(DEFINEQ (F (A) (IF A LT 2 THEN 1 ELSE A*(F A-1]
(F)
72+(F 4]
24
73+FILES?]
LATTICER...to be dumped.
plus the functions: F
want to say where the above go ? No
NIL
74+(SETQ DRAWBETWEEN 0]
(DRAWBETWEEN reset)
0
75+(EDITTREE (PARSE My uncle's story
FOLLOW/CURSOR
ND
DS {DISPLAYSTREAM}#5,137346
APPLY
EDITLATTICE
EDITTREEB0010
ADV-PROG
ADV-SETQ
PROG
EDITTREE
==TOP==
78:. DOES FOLLOW/CURSOR CALL DRAWBETWEEN SOMEHOW
T
79:. DOES FOLLOW/CURSOR CALL DRAWBETWEE
NIL
80:DRAWBETWEEN
66
81:. EDIT WHERE ANY CALLS DRAWBETWEEN
SHOWLINK :
(DRAWBETWEEN (FROMPOS FRNO) (TOPOS TOND
{from {PHYLUM}<LISP>DEMO>LATTICER.;18}
(SHOWNODE
[LAMBDA (ND NODELST DS TOSONLY) (* rrb "28-JAN-82 13:04")
(* displays a node and its links.
IF TOSONLY IS NON-NIL, DRAWS ONLY THE TO LINKS.]
(SHOW/NODE/LABEL ND DS)
(for TONODEID in (TOLLINKS ND) do (SHOWLINK ND (NODEIDTONODE TONODEID NODELST)
DS))
(OR TOSONLY (for FROMNOID in (FROMLLINKS ND) do (SHOWLINK (NODEIDTONODE FROMNOID
ND DS])
FOLLOW/CURSOR
MOVE/NODE
CHANGE/LAYOUT/POSITION
SHOWNODE
(LAMBDA (FRNO TOND DS)
(* bas: " 7-OCT-82 14:26")
(* draws in a link from
FRNO TO TOND)
(add DRAWBETWEEN 1)
(DRAWBETWEEN (FROMPOS FRNO)
(TOPOS TOND)
1 NIL DS))
After
Before
Delete
Replace
Switch
( )
( ) out
Undo
Find
Swap
Reprint
DEdit
EditCom
Eval
Exit
LNODEID (NP (DET &) (N &))
LNODEPOSITION (232 . 183)
NODELABELBITMAP
NODEFROMPOS (232 . 175)
NODETOPOS (232 . 191)
LNODEWIDTH 21
LNODEHEIGHT 16
TOLNODES ((DET &) (N &))
FROMLNODES ((PP & &))
LNODEFONT {FONTDESCRIPTOR}#1,115550
NODELABEL NP
BOXNODEFLG NIL
S
NP
AUX
VP
DET
N
PP
M
V
NP
PP
NP
P
NP
PRO
P
NP
DET
N
N
Interlisp-D

Programming tools achieve their effectiveness in two quite different ways. Some tools are simply effective *viewers* into the user's program and its state. Such tools permit one to find information quickly, display it effectively, and modify it easily. A wide variety of tools of this form can be seen in the two Interlisp-D screen images on the previous page, including data value *inspectors* (which allow a user to look at and modify the internal structure of an object), *editors* for code and data objects, and a variety of *break and tracing* packages. Especially when coupled with a high bandwidth display, such viewers are very effective programming tools.

The other type of programming tool is *knowledge based*. Viewer based tools, such as a program text editor, can operate effectively with a very limited "understanding" of the material with which they deal. By contrast, knowledge based tools must know a significant amount about the *content* of a user's program and the *context* in which it operates. Even a very shallow analysis of a set of programs (*e.g.*, which programs call which other ones) can support a variety of effective programming tools. A *program browser* allows a programmer to track the various dependencies between different parts of a program by presenting easy to read summaries which can be further expanded interactively. Deeper analysis allows more sophisticated facilities. The Interlisp program analyser (Masterscope) has a sufficiently detailed knowledge of Lisp programs that it can provide a complete static analysis of an arbitrary Lisp program. A wide variety of tools have been constructed which use the database provided by this analysis to answer complex queries (which may require significant reasoning, such as computing the transitive closure of some property), to make systematic changes under program control (such as making some transformation wherever a specified set of properties hold), or to check for a variety of inconsistent usage errors.

Finally, integrated tools provide yet another level of power. The Interlisp system "notices" whenever a program fragment is changed (by the editor, or by redefinition). The program analyser is then informed that any existing analysis is invalid, so that incorrect answers are not given on the basis of old information. The same mechanism is used to notify the program management subsystem (and eventually the user, at session end) that the corresponding file needs to be updated. In addition, the system will remember the previous state of the program, so that at any subsequent time the programmer can undo the change and retreat (in which case, of course, all the dependent changes and notifications will also be undone). This level of cooperation between tools not only provides immense power to the programmer, but it relieves him of a level of detail that he would otherwise have to manage himself. The result is that more attention can be paid to exploring the design.

Contraction

A key, but often neglected, component of an exploratory programming system is a set of facilities for program *contraction*. The development of a true exploratory program is "design limited", so that is where the effort has to go. Consequently, the program is often both inefficient and inelegant when it first achieves functional acceptability. If the exploration is an end in itself, this might be of limited concern. However, it is more often the case that a program developed in an exploratory fashion must eventually be used in some real situation. Sometimes, the time required to reimplement (using the prototype program as a specification) is prohibitive. Other times, the choice of an exploratory system was made to allow for expected *future* upheaval, so it is essential to preserve design flexibility. In either event, it is necessary to be able to take the functionally adequate program and transform it into a program whose efficiency is comparable to the best program one could have written, in any language, had only one known what one was doing when one started.

The importance of being able to make this *post hoc* optimization cannot be overemphasized. Without it, one's exploratory programs will always be considered "toys"; the pressure to

abandon the exploratory environment and start implementing in a "real" one will be overwhelming; and, once that move is made (and it is *always* made too soon), exploration will come to an end. The requirement for efficient implementation places two burdens on an exploratory programming system. First, the architecture has to *permit* efficient implementations. Thus, for example, the obligatory automatic storage manager must either be so efficient that it imposes negligible overhead, or it must allow the user to circumvent it (*e.g.*, to allocate storage statically) when and where the design has stabilized enough to make this optimization possible.

Second, as the performance engineering of a large system is almost as difficult as its initial construction, the environment must provide *performance engineering tools*, just like it provides design tools. These include good instrumentation, a first class optimizing compiler, program manipulation tools (including, at the very least, full functionality compiler macros), and the ability to add declarative information where necessary to guide the program transformation. Note that, usually, performance engineering takes place not as a single "post functionality optimization phase", but as a continuous activity throughout the development, as different parts of the system reach design stability and are observed to be performance critical. This is the method of "progressive constraint", the incremental addition of constraints as and when they are discovered and found important, and is a key methodology for exploratory development.

Both of these concerns can be most clearly seen in the various Lisp based systems. While, like all exploratory environments, they are often used to write code very quickly without any concern for efficiency, they are also used to write artificial intelligence programs whose applications to real problems are very large computations. Thus, the ability to make these programs efficient has long been of concern, because without it they would never be run on any interesting problems. More recently, the architectures of the new, personal Lisp machines like the 1108 have enabled fast techniques for many of the operations that are relatively slow in a traditional implementation. Systems like Interlisp-D, which is implemented entirely in Lisp, including all of the performance critical system code such as the operating system, display software, device handlers, *etc.*, show the level of efficiency which is now possible within an exploratory language.

Prospects

The increasing importance of applications which are very poorly understood, both by their clients and by their would-be implementors, will make exploratory development a key technique for the 1980s. Radical changes in the cost of computing power have already made exploratory development systems cost effective vehicles for the *delivery* of application systems in many areas. As recently as five years ago, the tools and language features we have discussed required the computational power of a large mainframe (~$500K). Two years ago, equivalent facilities became available on a personal machine for ~$100K. A year later, ~$50K. Now, a full scale exploratory development system can be had for ~$25K. For many applications, the incremental cost has become so small over that required to support conventional technology that the benefits of exploratory development (and redevelopment!) are now decisive.

One consequence of this revolutionary change in the cost-effectiveness of exploratory systems is that our notion of "exploratory problem" is going to change. Exploratory programming was developed originally in contexts where change was *the* dominant factor. There is, however, clearly a *spectrum* of specification instability. Traditionally, the cost of exploratory programming systems, both in terms of the computing power required and the run-time inefficiencies incurred, confined their use to only the most volatile applications. Thus, the spectrum was arbitrarily dichotomized into "exploratory" (very few) and "standard"

(the vast majority). Unfortunately, the reality is that unexpected change is far more common in "standard" applications than we have been willing to admit. Conventional programming techniques strive to preserve a stability that is only too often a fiction. Since exploratory programming systems provide tools that are better adapted to this uncertainty, many applications, such as office information systems, which are now being treated as "standard" but which in fact seem to require moderate levels of ongoing experimentation, may turn out to be more effectively developed in an exploratory environment.

We can also expect to see a slow infusion of exploratory development techniques into conventional practice. Many of the programming tools of an exploratory programming system (in particular, the information gathering and viewing tools) do not depend on the more exploratory attributes of either language or environment and could thus be adapted to support programming in conventional languages like FORTRAN and COBOL. Along with these tools will come the seeds of the exploratory perspective on language and system design, which will gradually be incorporated into existing programming languages and systems, loosening some of the bonds with which these systems so needlessly restrict the programmer.

To those accustomed to the precise, structured methods of conventional system development, exploratory development techniques may seem messy, inelegant and unsatisfying. But it's a question of congruence: Precision and inflexibility may be just as disfunctional in novel, uncertain situations as procrastination and vacillation are in familiar, well-defined ones. Those who admire the massive, rigid bone structures of dinosaurs should remember that jellyfish still enjoy their very secure ecological niche.

Acknowledgement

Many of these ideas were first developed, and later much polished, in discussions with John Seely Brown and other colleagues in Cognitive and Instructional Sciences at Xerox PARC.

II c.

KNOWLEDGE PROGRAMMING IN LOOPS:
Report on an Experimental Course

Mark Stefik, Daniel G. Bobrow,
Sanjay Mittal, and Lynn Conway[1]

Knowledge Systems Area
Xerox Palo Alto Research Center
Palo Alto, CA 94304

Abstract

Early this year fifty people took an experimental course at Xerox PARC on knowledge programming in Loops. During the course, they extended and debugged small knowledge systems in a simulated economics domain called Truckin. Everyone learned how to use the Loops environment, formulated the knowledge for their own program, and represented it in Loops. At the end of the course a knowledge competition was run so that the strategies used in the different systems could be compared. The punchline to this story is that almost everyone learned enough about Loops to complete a small knowledge system in only three days. Although one must exercise caution in extrapolating from small experiments, the results suggest that there is substantial power in integrating multiple programming paradigms.

KNOWLEDGE PROGRAMMING is concerned with the techniques for representing knowledge in computer programs. It is important in many applications of AI, where the problems are messy. As in many situations in life, pat solutions and simple mathematical models just aren't good enough. Things break. Information is missing. Assumptions fail. Situations are complicated. To cope with messiness, AI researchers have found that large amounts of problem-specific knowledge are usually needed. This places a premium on the use of powerful techniques for representing and testing knowledge in computer programs.

Very few people have been trained to build knowledge systems. This is a critical bottleneck that limits the scope and impact of knowledge engineering. It limits the number of things that can be tried, the number of good ideas that are propagated, and the number of successful applications that influence the way that others perceive the field.

A few numbers may serve to put this in perspective. About one computer science researcher in ten does some work in AI, and perhaps a fifth of those work in knowledge engineering. In 1980, approximately 265 people graduated with Ph.D.'s in Computer Science, according to the "Snowbird Report" (Denning, et al., 1981). Fewer than a half dozen doctoral theses appear each year on some aspect of building knowledge systems. An estimate in a brochure by Teknowledge, Inc., indicates that there are only about sixty people in the world with high level expertise in the design and development of knowledge systems. Although precise figures for these populations are difficult to obtain, all the evidence suggests that the community is tiny, indeed.

---

[1] Now with the Defense Advanced Research Projects Agency (DARPA).

Thanks to Johan de Kleer, Richard Fikes and John McDermott for their reviews and comments on earlier drafts of this paper. We extend our special thanks to the course participants from Applied Expert Systems, Daisy Systems, ESL, Fairchild AI Lab, Lawrence-Livermore Laboratories, Schlumberger-Doll Research Laboratory, SRI International, Stanford University, Teknowledge, and Xerox Corporation. Their participation and feedback are vital to the ongoing experimental process for simplifying the techniques of knowledge programming. We enjoyed and will long remember their spirited involvement.

Training in knowledge engineering usually requires several years of study at one of a handful of universities. A group of us in the Knowledge Systems Area at Xerox PARC is trying to shorten this training time. Our goal is to increase the impact and scale of knowledge engineering by simplifying the methods of knowledge programming and making them more widely accessible. In doing this we have developed an experimental knowledge programming system called LOOPS (Bobrow & Stefik 1981; Stefik, et al., 1983a). Feedback about the adequacy of LOOPS is collected from beta-test sites which are using it to build knowledge systems. Feedback about the learnability of LOOPS is collected from participants in experimental courses.

Integration and Paradigms

An important principle of knowledge programming is that different paradigms are appropriate for different purposes. This contrasts with the use of a single programming paradigm for everything, be it logic programming as in Prolog (Clocksin & Mellish 1981), procedure-oriented programming as in Lisp (Winston & Horn 1978), object-oriented programming as in Smalltalk (Goldberg & Robson 1983), or whatever.

There are various metrics of cost for applying a programming paradigm across a spectrum of applications. Examples of metrics are the cost of learning, the cost of modifying, the cost of debugging, and the cost of running. These costs vary across paradigms and applications because different programming paradigms provide different ways of organizing information in programs. For a given metric and application, some programming paradigms can be more cost-effective than others. By allowing for choice and combination of paradigms, a knowledge programming system enables various costs to be lowered. For example, we attribute much of our success in the experimental courses to the low costs for learning and applying LOOPS. For each of the things that the course participants needed to represent in their knowledge systems, there was some paradigm in LOOPS in which the expression of the knowledge was concise and the learning cost was low. Although there is room for much more work on programming paradigms and their applications, the principle seems clear: it is expensive to use one simple programming paradigm for everything.

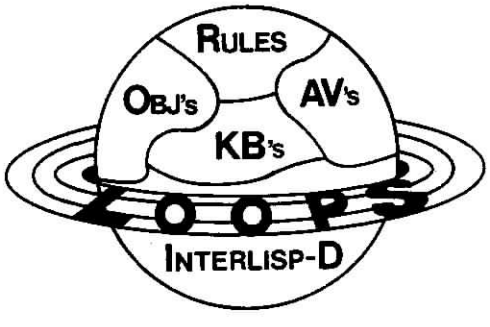


Figure 1.

The LOOPS Logo. Illustrating the different paradigms in the current version of LOOPS procedure-oriented, object-oriented, access-oriented, and rule-oriented. The ring is intended to suggest that LOOPS integrates the paradigms. They are not just complementary, but are designed to be used together in building knowledge systems.

As indicated in the LOOPS logo in Figure 1, LOOPS currently integrates four programming paradigms:

Procedure-oriented programming: In this paradigm, large procedures are built from small ones by the use of subroutines. Data and programs are kept separate. Most computer languages are like this. The procedure-oriented part of LOOPS is INTERLISP-D (Teitelman 1978, Xerox 1982). INTERLISP-D is shown at the base of the LOOPS logo to suggest that it provides the solid foundation on which the rest of LOOPS is built.

Object-oriented programming: In this paradigm, information is organized in terms of objects, which combine both instructions and data. Large objects are built up from smaller objects. Objects communicate with each other by sending messages. The conventions for communicating with an object by using messages constitute message protocols. Standardized protocols enable different classes of objects to respond to the same kinds of messages. Inheritance in a class lattice enables the specialization of objects.

Access-oriented programming: This paradigm is useful for programs that monitor other programs. Its basic mechanism is a structure called an active value, which has procedures that are invoked when variables are accessed. A useful way to think of active values is as probes that can be placed on the object variables of a LOOPS program. These probes can trigger additional computations when data are changed or read. For example, they can drive gauges that display the values of variables graphically.

Rule-oriented programming: This paradigm is specialized for representing the decision-making knowledge in a program. In LOOPS, rules are organized into rulesets which specify the rules, a control structure, and other descriptions of the rules. Two key features of the rule language are that it provides techniques for factoring control information from the rules, and also dependency-trail facilities, which provide mechanisms for "explanation" and belief revision.

These different organizational methods determine the way that information is factored and shared. Each paradigm provides a vocabulary and a set of composition methods for organizing information in a program.

Procedure composition: The composition methods of INTERLISP-D are forms of familiar control statements for iteration, recursion, and procedure call.

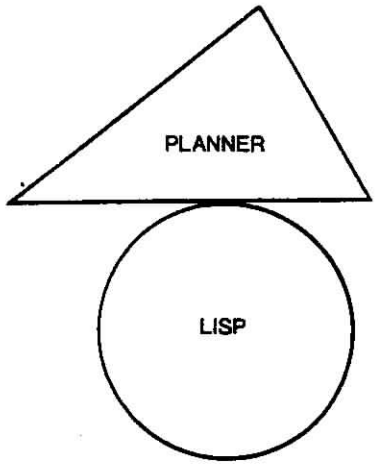


Figure 2.
Combining paradigms: The perch approach.

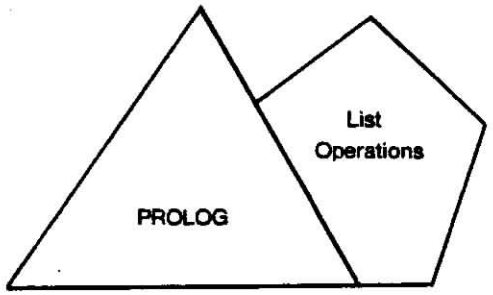


Figure 3.
Combining paradigms: The patch approach.

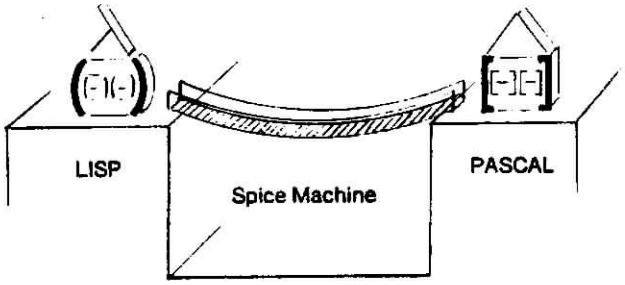


Figure 4.
Combining paradigms: The bridge approach.

Object composition: This paradigm provides several composition methods (shown in figure 6). The simplest is the specialization of methods and variables of a superclass. Special classes called Mixins are used to impart a specific set of behaviors to a number of subclasses. The term "mixin" is borrowed from Flavors -- (Weinreb & Moon 1981). Mixins exploit the multiple inheritance lattice by allowing inheritance to be factored. Composite objects extend the notion of objects to be recursive in structure so that multiple objects can be instantiated and linked together. Finally, perspectives in LOOPS are groupings of objects into a higher level object, such that each component is a view (or perspective) of the whole. Perspectives provide for the forwarding of messages to the appropriate view.

Access composition: Composition in this paradigm is done by nesting of active values. Analogous to the use of multiple probes in measuring a circuit, this composition assumes that the "probes" are for independent instruments and do not interfere with each other.

Rule composition: The LOOPS rule-oriented paradigm provides for the sharing of rules among rulesets. It makes use of the other paradigms for organizing the interactions between the rules. Thus rules can call rulesets directly (using the procedural orientation), or invoke rulesets by sending messages (using the object orientation), or invoke rulesets by changing data (using the access orientation).

Integration has two major themes in LOOPS: integration to allow the paradigms to be used together in building a knowledge system; and integration of a programming environment for creating and debugging knowledge systems.

Some examples illustrate the integration of paradigms in LOOPS: the "workspace" of a ruleset is an object, rules are objects, and so are rulesets. Methods in classes can be either Lisp functions or rulesets. The procedures in active values can be LISP functions, rulesets, or calls on methods. The ring in the LOOPS logo reflects the fact that LOOPS not only contains the different paradigms, but integrates them. The paradigms are designed not only to complement each other, but also to be used together in combination.

Some examples from other systems illustrate the non-integration of programming paradigms. For example, Figure 2 shows the connection between PLANNER and LISP. PLANNER was implemented in LISP, but a programmer could not easily intermix PLANNER and LISP procedures. A simple mistake by a "naive" programmer could easily crash the whole system. Figure 3 shows the connection of list operations to PROLOG, reflecting the fact that list operations were added late to PROLOG, after the initial design. Figure 4 illustrates another approach, illustrated perhaps by the Spice Machine. In this example LISP and PASCAL communicate over a narrow bridge, making mutual use awkward and costly.

The second theme of integration is the integration with the programming environment. For example, LOOPS extends to other paradigms many of the facilities of INTERLISP-D.

such as the display-oriented break package, editors, and inspectors. In LOOPS, this integration has led to the same synergy that is exploited in using multiple paradigms for application programs. For example, the notion of "breaking" on access to a function is extended to breaking on access to a variable by using active values to invoke the break package; the notion of tracing is extended to the notion of having gauges that can monitor the values of variables.

Getting Ready for the First Course

On January 6, we began to plan the first LOOPS course that would be offered on January 31 to our beta-test sites. We made a preliminary course outline, but we knew that we needed some way to draw the participants into programming in LOOPS. The idea of a video game was suggested, say rocket ships with LOOPS programs controlling the thrust and phasers. This idea was rejected as being both too frivolous, and computationally too expensive. Another suggestion was a game for placing tiles. We knew from Malone (1980) that there were principles for making games motivating. Our course participants would be computing and other professionals drawn from research organizations and AI start-up firms, who were interested in using LOOPS for building expert systems. We needed something that they would find useful and appealing.

As brainstorming continued, some pedagogical principles began to emerge. The game should draw on the real world knowledge of our students. Rocket ships and tiles were wrong, because people didn't have experience with such things from their everyday lives. A board game like Monopoly was considered, and then our first concept of *Truckin'* emerged. It would be a board game with road stops (see Fig. 5). The players would drive trucks around buying and selling commodities. Their job would be to plan a route and make a profit. There would be various hazards along the way, places where goods and profits could be lost. Players would need to buy gas occasionally.

By mimicking real life, *Truckin'* would provide the kinds of difficulties that knowledge engineers encounter in building expert systems. We could create a rich and animated simulation environment for the "independent truckers." The students would need to add knowledge to make their automated players more powerful. The simulation environment would draw on the student's real-world knowledge, and be rich enough to preclude a simple model. Much of the appeal of this was that the "common experience" character of *Truckin'* as a domain would enable us to side-step the usual knowledge acquisition bottleneck. The knowledge engineering experience would be accelerated by immediate feedback from the animated simulation. To help students get started, we would provide them with a simple expert system for playing *Truckin'*. We decided to teach students about knowledge programming in LOOPS by giving them a small knowledge system to extend.

At this point, we had less than a month to create the course materials, lectures, and *Truckin'*. Sleep would become a rare and precious commodity. The *Truckin'* data base began to take shape. The players would start at Union Hall, and would try to be parked at Alice's Restaurant at the end of the game. There would be various kinds of hazards along the road. The player with the most cash at Alice's at the end of the game would win.

LOOPS was able to accommodate changes as our ideas evolved. Initially, we thought of the hazards as being road stops. This was probably a carry over of our childhood experiences with board games. Then we added the idea of "bandits" that could move around just like the independent truckers. Bandits were represented as an inheritance combination of players and consumers. We used active values on variables of the road stops to update the display for commodity prices and inventories. This meant that we did not need to find every place in the program where these things could potentially be changed, in order to update the display. The features of LOOPS worked for us, providing convenient techniques for factoring the program. We became experienced consumers of our own knowledge programming system as we raced to get ready for the course.

The simulation was designed to cause goal conflicts. A truck going quickly over a rough road would probably have its fragile merchandise damaged. A truck going quickly past a weigh station would probably get an extra fine, unless he was lucky or the weigh station was busy. On the other hand, a truck going slowly past a bandit would probably get intercepted. There would be perishable goods and fragile goods. We considered explosive goods and other such things, but removed them when they failed to add anything new to the game. Our pedagogical style was to leave some things out in order to keep it simple. A player could take only three kinds of actions: buy, sell, and move.

To facilitate the "suspension of disbelief" in watching the animated simulation, artistic attention had to be given to the appearances of things. Icons for the various commodities, hazards, and trucks were created. We experimented with different configurations of the gameboard, moving away from the outside edge configuration of most gameboards in order to pack enough road stops on the display screen. Highways were drawn next to the road stops, with a gray background and little dashed white lines in the center. People looked at intermediate versions of the gameboard and told us that the abrupt motion of the trucks was startling. We modified the code to simulate braking so that trucks would slow down as they arrived at their destinations. The visual appearance of *Truckin'* became seductive. People were drawn into it.

Prior to this, we had used a simple gauge in our demo to illustrate the application of active values. It was a crude looking gauge and had little generality. We decided to extend the collection of gauges so that people could use them for debugging and for monitoring their independent truckers during the simulation. A family of gauges was designed (see Figure 7). For further ideas on style, we collected some professional catalogs of gauges, and sought advice from

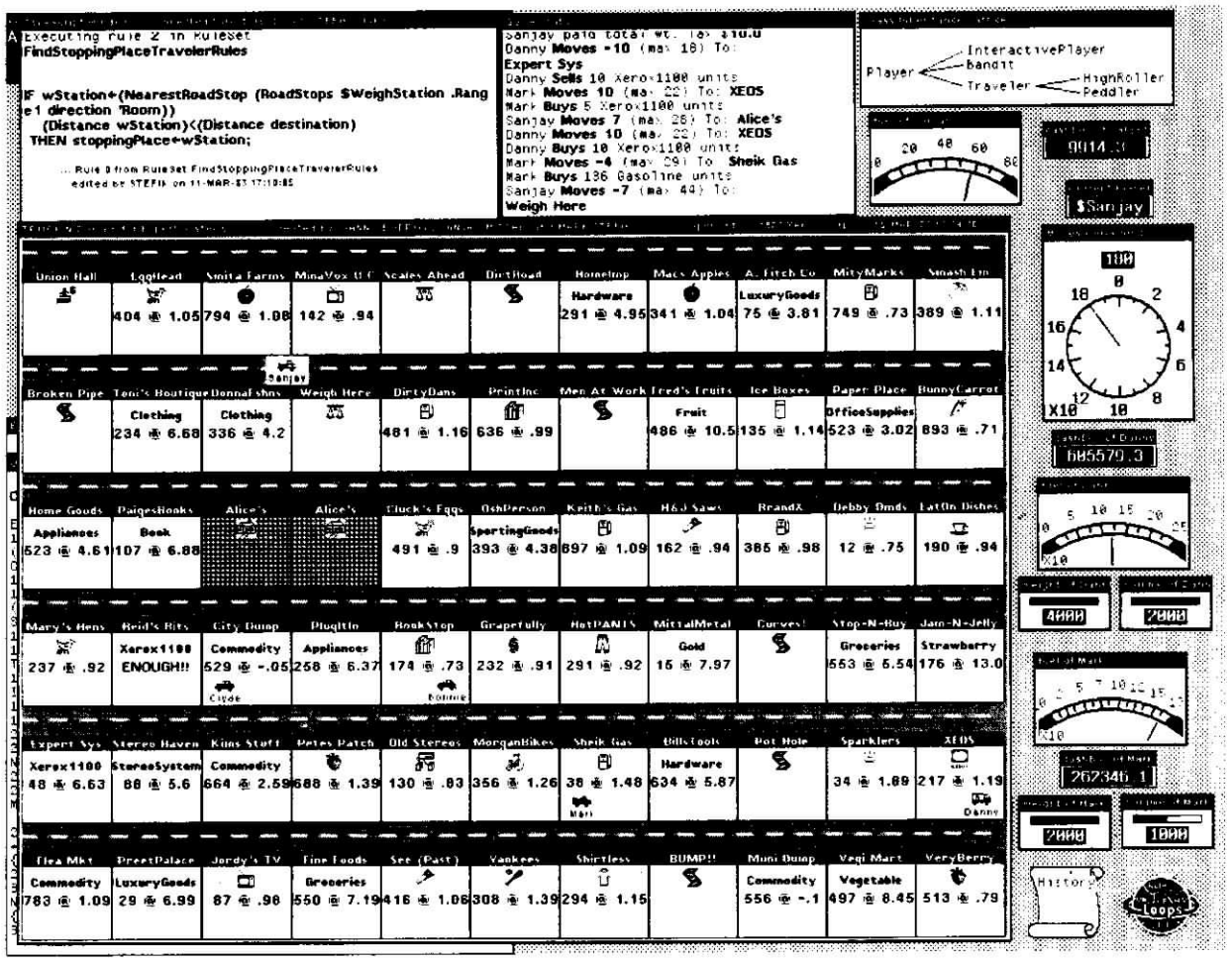


Figure 5. The Loops gameboard — for a game played by competing knowledge systems that emulate "independent truckers." The board's squares are road stops, connected by the highway drawn above. Roadstops can be producers, consumers, rough roads, weigh stations. Roadstops with icons are producers, where players can buy. Those with words (e.g., Clothing) are consumers, where players can sell. The trucks for the players are shown parked or moving along the highway (e.g., Sanjay). To the right, a panoply of gauges monitors the status of various players. In the upper left corner, a rule for one of the players is being traced.

Bruce Roberts on the Steamer project at BBN. The gauges went through several design reviews, to make the gauges simpler to use and modify. Because of the extensive use of multiple inheritance and the interactions on the display between the parts of the hybrid gauges, a number of design issues surfaced. During these reviews, we created names for certain categories of design errors that we encountered. For example, a *grainsize error* is a situation where the structural parts of an object (usually methods) are factored too coarsely for the fine control needed by its specializations. A *replication error* is a situation where almost the same structure is repeated in parallel classes, instead of factoring it in a way that would allow it to be shared. Such experiences gave us a deeper understanding of the programming issues that people would encounter in using the different paradigms.

About two weeks before the course would begin, we sent out notices to our beta-test sites inviting people to sign up for the course. We expected about a half dozen people. We advertised that our course would provide hands-on experience in extending a "mini-expert system." By word of mouth, the story spread. Over fifty people called us, requesting to get on the list. We split the list in half and scheduled the second course for the end of February. We didn't send out any more advertisements. We had gone public and now we had to make it work.

Suddenly it was the weekend before the course. We made some guesses about the appropriate distribution of prices and penalties. We created our first automated player, the Traveler, which would just travel along the board between Union Hall and Alice's Restaurant. As the Traveler cruised tirelessly around the game board, various bugs in the simulation surfaced. Meanwhile, we started work on a player to

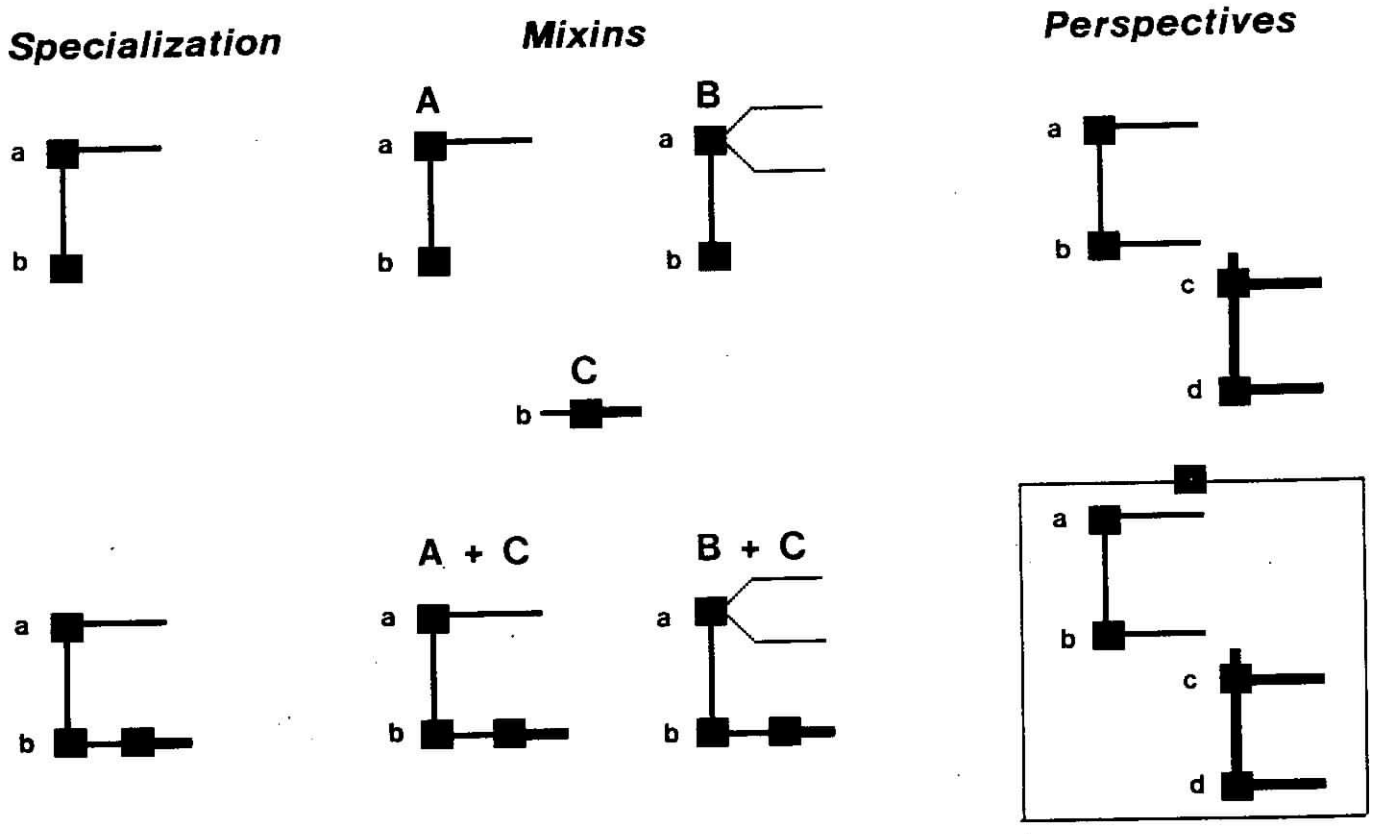


Figure 6. Object composition in Loops. The inheritance lattice enables many forms of structural sharing in Loops. The simplest form is *specialization*, that is, creating a subclass that overrides and augments variables and methods inherited from the superclass. When multiple superclasses are used, the resulting subclass mixes together the attributes from the superclasses. *Perspectives* provide a way of grouping objects to act as views of a higher level object. *Perspectives* automatically forward messages to the appropriate object.

specialize in luxury goods called HighRoller. We didn't have time to debug it very well before the course started. We reasoned that the bugs were acceptable, since they would provide things for the course participants to fix. We were right, but in hindsight, we had a lot of gall.

The Courses as Experiments

We have now run two intensive knowledge programming courses, and also repeated the second course to a small group using videotape. By the time of publication of this article, the course will have been run for over 100 people. The courses are organized to alternate lectures and hands-on exercises (see Table 1). So far, everyone taking the course has learned enough about the LOOPS knowledge programming system to do some practice exercises (such as creating a new kind of gauge) and to build an extended (smarter) *Truckin'* player.

The most important aspect of the courses for our purposes is the opportunity that they provide for refining both LOOPS and the course materials. For us, the courses are experiments, from which we are discovering how to make LOOPS and our teaching methods more effective. The basic structure of our experimental process is to run a course and take some measurements (for example, of the performance of students in terms of the problems that they complete, the questions that they ask, and the results of questionaires that they return). We then change some parameters and take the measurements again during the next course. By examining how the observations and measurements differ, we can form hypotheses to guide subsequent iterations of the course.

- We substantially increased the emphasis on tools and techniques for debugging, and formulated explicit heuristics for programming in LOOPS. We taught students to use tools for understanding the behavior of a system. The second course led students to use gauges for monitoring the values of variables, explanation facilities (Fig. 8) for understanding which rule made a particular decision, and breaking and tracing facilities for discovering why some rules do not fire.
- In some cases, we introduced intermediate problems in the exercises, having hypothesized that some of the

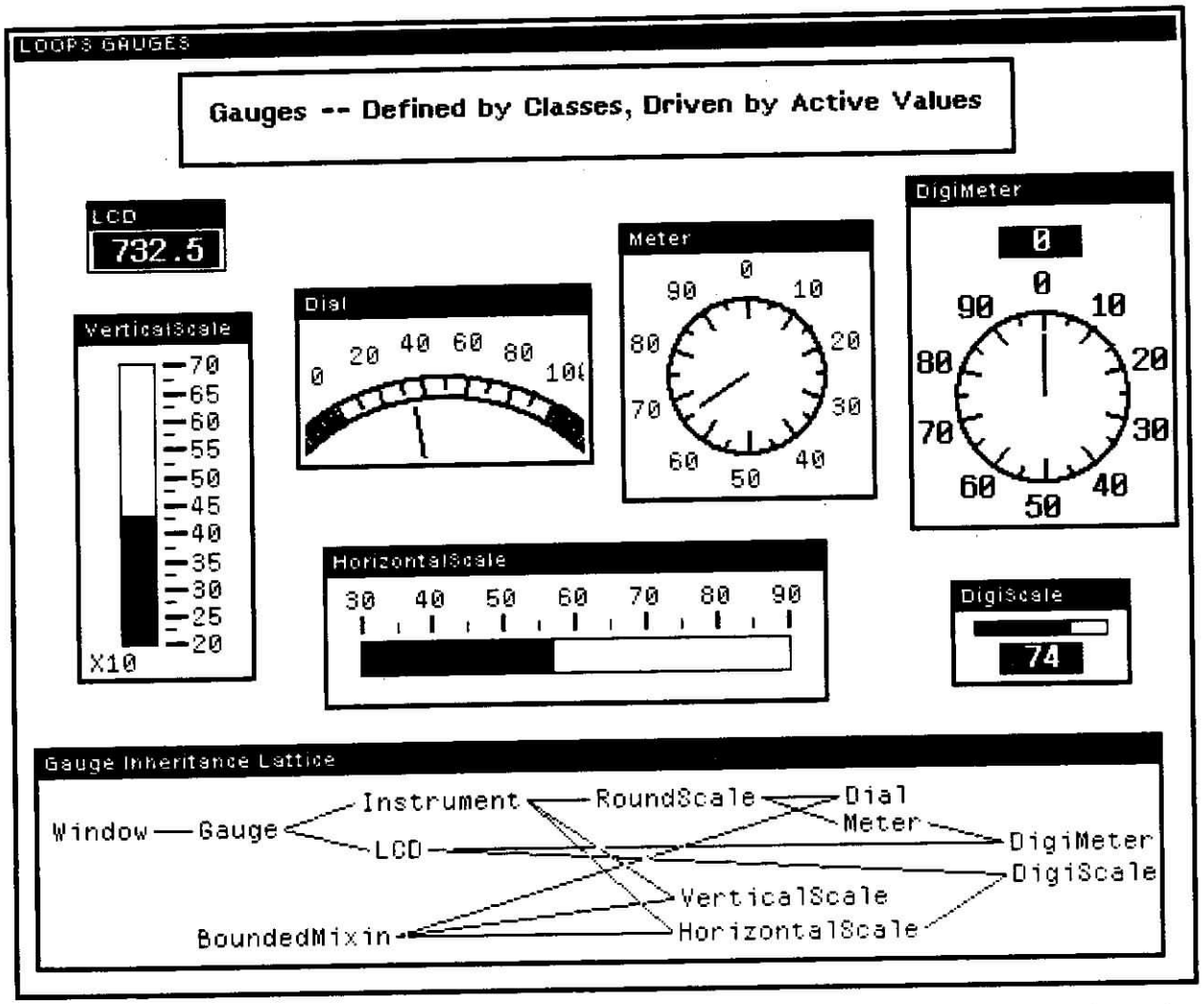


Figure 7. Loops gauges. Gauges are tools used to monitor the values of variables. They can be thought of as probes inserted onto the variables of an arbitrary Loops program. Gauges are defined in Loops as classes, and driven by active values the mechanism behind access-oriented programming in Loops. A browser at the bottom of the figure illustrates the relationships between the classes of gauges. From this figure, we can see that the DigiMeter is a combination of a Meter and an LCD.

steps between exercises were too difficult to take all at once.

- We fashioned a new starting player for the second course, called the Peddler, which did a better job than HighRoller in factoring the concerns of an independent trucker. We hypothesized that restructuring HighRoller was too difficult to do in a three day course.
- We adjusted the commodity prices and risks to provide a greater reward and selection pressure for more sophisticated and knowledgeable truckers.
- We improved the browsers, that is, our interactive graphics for "browsing" information in a knowledge base (see Figure 9). We believed that we could reduce much of the cognitive load for restructuring objects and accessing information if we provided more effective ways of making the right information visible.
- We fixed troublesome bugs in the rule compiler. During the first course, participants had to struggle with a compiler that did not reliably keep the generated LISP code in correspondence with the rules.

As a result of these changes, the participants in the second course were dramatically more successful than those in the first. In the first course, we had to slip the schedule for the knowledge competition by 90 minutes, in order to let people finish preparing their players. In the second course, people had players ready in about half of the allocated time, and spent the remaining time exploring other aspects of the system and tuning their strategies. Furthermore, the weakest player of the second course could easily dominate the best player from the first course.

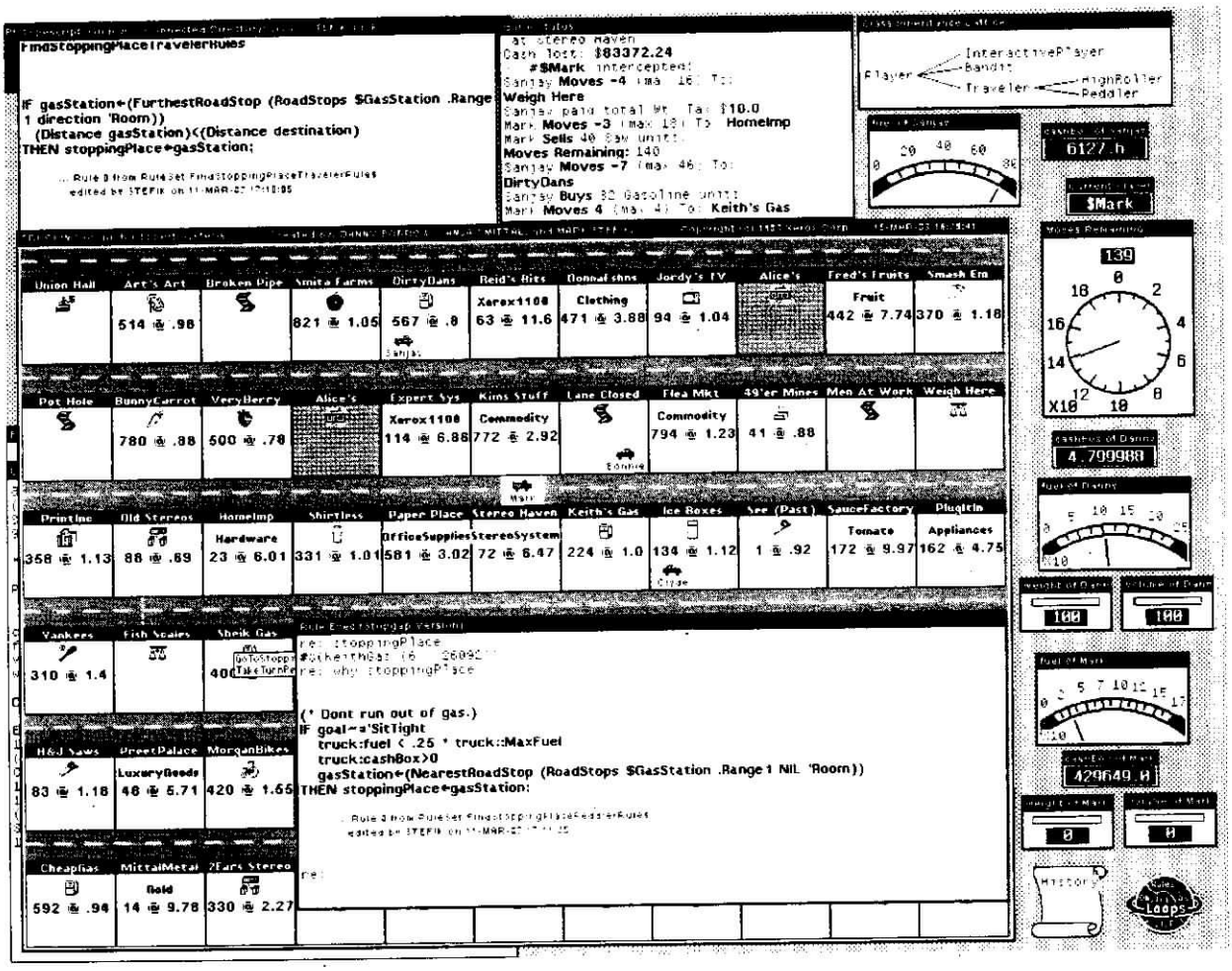


Figure 8. Seeing the knowledge behind a decision. In this figure the game is interrupted, causing the Rule Exec window to pop up over the gameboard. The user has asked why his truck picked a particular stopping place, and Loops has displayed the rule that made the decision.

People asked far fewer questions in the second course, and were able to complete many more of the exercises. In addition, the questionaires from the second course came back with radically different advice from those from the first course. The general response from the first course was "give us less on rules" and many people indicated substantial concern with many of the fundamental aspects of that paradigm. In the second course, the responses turned completely around. They said "give us more on rules and debugging."

We believe that in the first course the combination of a faulty rule compiler and lack of information on how to debug programs in this paradigm undermined confidence. During the second course, two members of a team were observed staring at a display. One of them said, "Why is it buying tomatoes?" and the other one elbowed him saying "Ask why! Ask why!" goading him into action at the LOOPS keyboard. They had learned their lessons well.

This process of simplifying methods and tuning the course in order to enhance learnability and propagatability reflects our interest in the engineering of knowledge (Conway 1981, Stefik & Conway 1982). In this case we are engineering languages and techniques for knowledge programming. The courses provide a source of feedback on the effects of changes to the course materials, paradigms and programming environments. In time, we would like to extend our work to provide a framework that would simplify the process of creating higher level organizations in expert systems (Stefik *et al.*, 1982).

The Knowledge Competition

A very enjoyable thing about the LOOPS course is the electric excitement that erupts during a knowledge competition. People seem to project themselves into the players that they have created. They have put their player through many simulations and many playing conditions. In a sense, they have taught it everything. But during the competition there is a moment of truth. The rules cannot be changed. Success

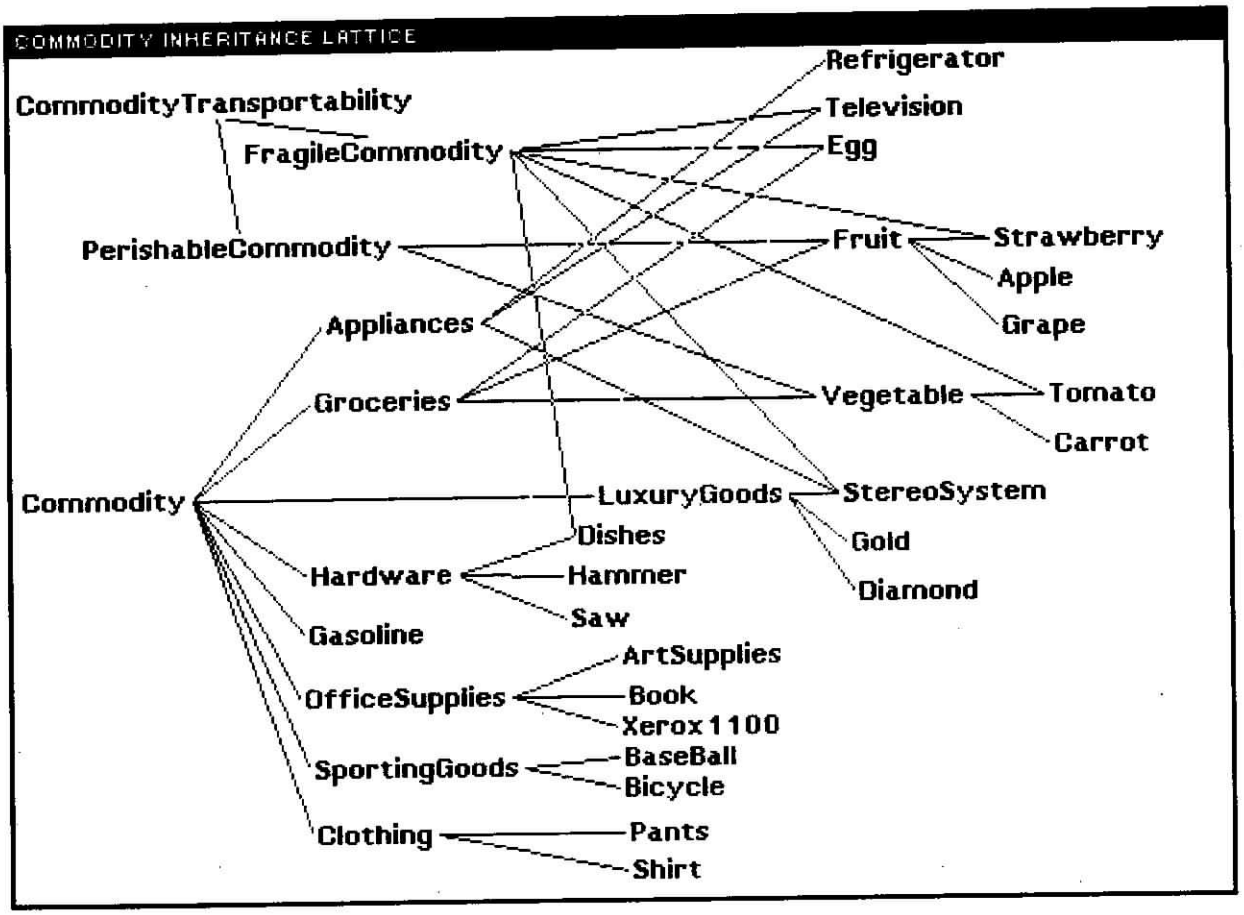


Figure 9. Class browser on commodities. Browsers are interactive programs used to browse through a knowledge base. The lines seen in a class browser indicate superclass relationships. For example, in this figure, a StereoSystem is a LuxuryGood, an Appliance, and a FragileCommodity. Browsers can be created to show other relationships too, and by selecting nodes in a browser, a user can access such further information.

in the short run is affected by chance, but on average, the most knowledgeable players will win.

The randomly generated game board comes up. As the simulation begins, there is a great deal of commentary and jibes as people compare their players. Who's ahead? Who just got robbed? In *Truckin'* the silliness of the ill-fated move is something that all the observers appreciate almost immediately. For example,

- A player may be racing to Alice's Restaurant. One move before the game ends it is unable to resist a business "opportunity" and doesn't make it to Alice's.
- A player may go to the closest place to sell some goods, even if it happens to be the City Dump, which unfortunately pays a "negative price."
- A player may become focused on a tight producer/consumer loop, making money faster than any other player on the board. If it is programmed to only buy fuel from stations along its route, but there is no gas station in the tight loop, the team will watch anxiously as the fuel gauge drops lower and lower.
- A player may try to park next to Alice's Restaurant near the end of the game, even if that happens to be the Union Hall, which confiscates all goods and cash.

In our experience so far, these oversights happen in the best of players. They provide a source of merriment during the competition, and an illustration of just how much knowledge is really needed to be powerful, even in an artificial environment.

The knowledge competition also serves as a source of examples and metaphors about the nature of knowledge. One example drawn from the first LOOPS course illustrates the interplay between knowledge and environment. For the first knowledge competition, two teams prepared players by simply fixing some of the bugs in the HighRoller. They had a private playoff just before the competition, and discovered that when both players were in the same game, the inventory of luxury goods on the game board became exhausted before the end of play. Neither player was able to cope with this situation. One of the heuristics that we now offer to teams

preparing for a knowledge competition is to test rules with many copies of the same player competing at once.

This interplay between knowledge and environment brings to mind the example of the ant on the beach (Simon 1981), in which the apparently complex movement of the ant is attributed to the complexity of the beach environment rather than the complexity of the ant. In *Truckin'*, the "ants" are mechanical and programmable. We have observed that even the complexity of the *Truckin'* environment creates a substantial selection pressure for resourceful and knowledgeable players. To win, the designers of the players must pit knowledge against complexity. Knowledge provides the adaptability needed for mastering the situations in the game.

The name "knowledge competition" was inspired by the observation that it is truly the knowledge of players that is competing, and the most adaptable player wins. Recently in connection with the interest in fifth generation computers, Feigenbaum and McCorduck (1983) have characterized knowledge as the new "wealth of nations." In the knowledge competition and *Truckin'*, the competitive advantages of knowledge in a player is concrete and observable in short experiments.

The success of the knowledge competition in motivating participation has led us to speculate on ways of alleviating the knowledge acquisition bottleneck. One idea is for a community of experts to interact through knowledge servers, which accept knowledge over a computer network and make themselves available for solving problems. Here again there would be a "competition" between different bodies of knowledge from the experts, competing to solve the problems that are posed.

Implications

Sometimes the effects of a technological change can be surprising and widespread. Although our research and experimentation with LOOPS has not run its full course, there have been a few expert systems started at our beta-test sites: three systems that perform parts of VLSI design, a program for playing Bridge, an investment advisor, a program for expressing specifications of parallel programs, a tester for LOOPS.

We sense that a technological change is emerging from such research on knowledge programming, a change in the infrastructure for building knowledge systems. The shift will have leveraging power in two ways: (1) the freeing of existing knowledge engineers from spending a year or two building the bottom of their knowledge representation systems, and (2) a measurable acceleration in the progress of the field if the simplified methods trigger an increase in numbers of practitioners from 100 to 1000 or more. Knowledge engineering can then begin to have a noticeable effect in many areas of our lives.

LOOPS COURSE OUTLINE

FIRST DAY:

9:00–9:15	Introduction.
9:15–10:15	Object-Oriented Programming: Classes - Objects - Variables - Methods - Inheritance - Documentation.
10:15-11:00	LOOPS Environment (Demonstration): Defining Methods - Editing - Printing - Inspecting Browsing - Gauges.
11:00-12:00	Exercise 1- Introductory hands-on session: Sending Messages - Browsing - Editing Inspecting.
12:00-1:00	Lunch.
1:00-2:00	Access-Oriented Programming: Active Values FirstFetch - NamedObjects - AtCreation - Nested Active Values - The LOOPS Break Package.
2:00–4:00	Exercise 2 - Gauges hands-on session: Specializing Classes - Instantiation - Using Gauges.
4:00–4:30	The *Truckin'* mini-Expert System.
4:30–5:00	Discussion.

SECOND DAY:

9:00–9:15	Introduction.
9:15-10:15	Rule-Oriented Programming: RuleSets Control Structures Recording Rule Invocations.
10:15-12:00	Exercise 3 - Rules hands-on session: Editing RuleSets - Debugging RuleSets.
12:00–1:00	Lunch.
1:00–2:00	Knowledge Representation Examples from *Truckin'*.
2:00–4:30	Exercise 4 - Knowledge programming hands-on session: Rule-Oriented Programming - multiple paradigm programming.
4:30–5:00	Discussion.

THIRD DAY:

9:00–9:15	Introduction.
9:15–11:00	Initial Development of your player: hands-on session.
11:00–12:00	Advanced LOOPS Features: Composite Objects - Perspectives vs. Mixins - Meta Classes - System Mixins - Knowledge Bases.
12:00–1:00	Lunch.
1:00–3:00	Final tuning of your player: hands-on Session.
3:00–4:00	The *Truckin'* Knowledge Competition.
4:00–4:30	LOOPS Environment: LOOPS Tester - Facilities for Bug Reporting.
4:30–5:00	Wrap-up: LOOPS support - User Packages - Future Directions.

Table 1.

References

Bobrow, D. G. & Stefik, M. (1981) *The* LOOPS *Manual.* Tech. Rep. KB-VLSI-81-13, Knowledge Systems Area, Xerox Palo Alto Research Center (PARC).

Clocksin, W.F. & Mellish, C.S. (1981) *Programming in Prolog.* Berlin: Springer-Verlag.

Conway, L. (1981) The MPC adventures: Experiences with the generation of VLSI design and implementation methodologies. *Proc. of the Second Caltech Conference on Very Large Scale Integration,* 5-28.

Denning, P. J., Feigenbaum, E., Gilmore, P., Hearn, A., Ritchie, R.W., & Traub, J. (1981) The Snowbird Report: A discipline in crisis. *Communications of the* ACM, 24:370-374.

Feigenbaum, E., & McCorduck, P. (1983) *The fifth generation: Artificial Intelligence and Japan's Challenge to the World.* Reading, MA: Addison-Wesley.

Goldberg, A., & Robson, D. (1983) *Smalltalk-80: The language and its implementation.* Reading, MA: Addison-Wesley.

Malone, T. W. (1980) *What makes things fun to learn? A study of intrinsically motivating computer games.* Technical Report CIS-7 (SSL-80-11), Xerox PARC.

Simon, H. A. (1981) *The sciences of the artificial.* Cambridge, MA: The MIT Press.

Stefik, M., Bobrow, D., & Mittal, S. (1983) *Knowledge programming in* LOOPS: *Highlights from an experimental course.* Video Report KSA-83-1, Xerox PARC.

Stefik, M., Bell, A. G., & Bobrow, D. G. (1983) *Rule-oriented programming in* LOOPS. Tech. Rep. KB-VLSI-82-22, Knowledge Systems Area, Xerox PARC.

Stefik, M., & Conway, L. (1982) The principled engineering of knowledge. *AI Magazine* 3(3):4-16.

Stefik, M., Aikins, J., Balzer, R., Benoit, J., Birnbaum, L., Hayes-Roth, F., & Sacerdoti, E. (1982) The organization of expert systems: A tutorial. *Artificial Intelligence* 18:135-173.

Teitelman, W. (1978) *Interlisp Reference Manual.* Technical Report, Xerox PARC.

Weinreb, D. & Moon, D. (1981) *Lisp Machine Manual.* Cambridge, MA: MIT Artificial Intelligence Laboratory.

Winston, P. & Horn, B. (1981) *Lisp.* Reading, MA: Addison-Wesley.

Xerox Corporation (1982) INTERLISP-D *users guide.* Pasadena, CA: Xerox Special Information Systems.

Note

LOOPS is available to selected Xerox customers designated as beta-test sites. The Knowledge Systems Area at Xerox PARC offers the intensive LOOPS course to selected applicants periodically for its research purposes.

III.

GETTING USED TO INTERLISP-D

WELCOME - to INTERLISP-D and to LOOPS!

INTERLISP-D presents an incredibly flexible and useful environment for the system builder and the casual user as well. As a lisp user coming from another dialect, you will find that certain features of INTERLISP-D take some "getting used to". But once mastered, you will find that INTERLISP-D is indeed a useful tool.

LOOPS is a powerful AI system building language incorporating object oriented and other paradigms of building AI systems that have proven to be extremely useful both for quick prototyping of systems as well as final system development.

As a caveat, let me emphasize WE ARE NOT COVERING ALL OF INTERLISP OR OF LOOPS in this class, rather only those parts which will give you some ability to get started, and to realize some of the capabilities of the system.

Before going further in this discussion, we must describe how to get INTERLISP-D up on the DLion volume which your team has been assigned.

LogingIn on the DLion

Here are the instructions for starting INTERLISP-D on your machine. First, you need to know one term: Volume. For the DLions, "volume" designates one of three possible user areas on the DLion hard disk. These volumes are (currently) completely independent. So that, for example, one team may use Volume 1 on a certain machine in the day time, while another team uses Volume 2 on the same machine in the evenings, with no interaction of things stored on the two Volumes.

To log in on the Machine/Volume to which you are assigned, first find the machine you have been assigned then do the following:

- (you'll see the machine with a box containing some instructions bouncing from place to place on the screen - we call this (not too originally) the bouncing box)

- depress the left mouse button (this brings the LISP Install tool)

- get the arrow into the Install tool window (you do this by moving the mouse around - ie the mouse controls the position of the arrow)

- put the arrow at the end of the line that reads: "Volume: Lisp"

- button the left mouse button (this will bring a triangular character after "Lisp")

- Now input the number of the LISP volume you have been assigned.

- Now left button the "Start Lisp Volume" item that is a little farther down the window.

- Now you'll see a confirmation icon. Button the left button again. (Now your INTERLISP volume will load. What this really means is that you are restarting an existing INTERLISP virtual memory.)

- When INTERLISP comes up, the first thing it will do is ask you for your machines "PUP Number". You'll see what it thinks the PUP number is right after. In your case, just give INTERLISP a carriage return. (The PUP Number is the number by which other machines can "talk" to your machine over the ETHERNET.)

This completes the loadup procedure that you'll be using. Of course there are other ways of starting up LISP, but during the course, you'll only be returning to your volume as per the above procedure.

The INTERLISP Philosophy

INTERLISP has grown up over many years and has had the benefit of many gifted people. The INTERLISP philosophy is (in a nutshell) to provide a system that "helps" the user and at the same time can be tailored to make a programming environment that suits individual preferences. The custom tailoring of INTERLISP is accomplished through the setting of MANY individual variables. These environment setting variables all have "reasonable" default values that are intended to give the casual user an environment that she will find useful.

There are two distinct classes of users that INTERLISP is easy for. First there are the users that have used it over some period of time. This group has become familiar with most of the "environment setting" variables that are included in INTERLISP, and have developed an environment that is "comfortable" for them.

The second class is made up of casual users that really don't want to bother with figuring out all the variables to get "just the right" environment for them.

That leaves most of you - users who are not yet familiar with all the environment variables, but still (will) feel irrated that they cannot set the action of INTERLISP just the way they want. The best advise is - be patient. Soon you will become familiar enough to begin building your own INTERLISP environment. In the mean time, you'll still find INTERLISP to be pretty "neat".

The Programmer's Assistant

The first thing that you'll see once you enter the PUP number, is a number followed by ‹leftArrow› [sorry, i don't have a ‹leftArrow› character]. This means that the INTERLISP top level is waiting for you to enter something.

The INTERLISP Top Level

The INTERLISP top level (the Programmer's Assistant - PA) is much more flexible than top levels in other dialects. As a design goal, the top level of INTERLISP was made to allow the user both different forms of specifying input to INTERLISP and many ways of manipulating what instructions she has already issued.

For purposes here we will say that the top level of INTERLISP comes in three flavors: the atom eval flavor, the expression eval flavor, and the expression apply flavor.

The atom eval flavor you are used to. Typing an atom name followed by ‹cr› causes the atom to be evaled and the result printed on the screen.

The expression eval form you are almost used to. Typing a non-atomic s-exp causes the expression to be evaled assuming the CAR of the list is a function. The only difference is that when

you close the final paren, INTERLISP takes the s-exp automatically - that is you don't type a ‹cr›.

The expression apply flavor will be unfamiliar. Typing the name of a function, immediately followed by a ‹leftParen›, a list of arguments, then a ‹rightParen› causes INTERLISP to APPLY the function to the arguments WITHOUT EVALing the arguments.

EX: LIST(a b c)

This facility is overall handy. Just be careful how you use it. For example, do

EX: SET(a (x y z))
SET(b a)
b‹cr›

The TTYIN Editor

At the top level, you are actually in an editor called TTYIN. In TTYIN you have available editing operations on the current line of code you are constructing.

By depressing the ‹leftArrow› key you obtain the deleteCharacter function.

In addition to this mundane capability, the mouse is also active for TTYIN. To insert a character, you simply mouse the left button, move the cursor to the desired location, let up on the mouse button, then make the insertion you want. Note this just amounts to moving the cursor location via the mouse.

To make deletions in the current line, you can mouse the right button to move the cursor to the place for the deletion (note that the area from the first mouse position to the last one becomes inverted), then let up on the mouse. Everything in the inverted part of the line will be removed.

EX: do SET(a (x y z)
[note exp is not completed]
now change the a to b

```
now change the (x y ..)
to (xy ..)

complete the line

practise this a little to get a feel for TTYIN
```

There are other capabilities provided by TTYIN. You might want to check them out in the manual.

REDO, FIX, UNDO

The PA has the notion of "history lists". These lists store what you have done in previous instructions. This enables the PA to give you the facility to act on selected (by you) previous instructions. Three of the most important of the actions you can request of the PA are REDO, FIX, and UNDO.

The action of REDO is pretty easy to understand.

```
EX:     41<leftArrow> SET(a (x y z))
           [the 41 is just for ex]
           [notice the PA told you that a is reset]
        42<leftArrow> SET(a (u v w))
        43<leftArrow> REDO 41 <cr>
        44<leftArrow> a <cr>
```

FIX is also easy to understand. With FIX, the command referred to will be brought back for you to be altered as desired using TTYIN.

```
EX:     FIX 41
        [now fix line 41
           to do SET(b (x y z))]
```

UNDO is another very useful PA instruction. The name is self explanatory.

```
EX:     41<leftArrow> SET(a (x y z))
        42<leftArrow> SET(a (u y))
        43<leftArrow> SET(y q)
```

```
44<leftArrow> UNDO 42 <cr>
45<leftArrow> a <cr>
```

In the examples above, we have used UNDO, FIX, and REDO by referencing commands by their line number. You can also reference commands symbolically.

```
EX:     SET(a x)
        SET(a (u i))
        (LIST 'x a))
        UNDO SET <cr>
        a <cr>
```

The PA includes many additional facilities for referencing groups of commands, for changing one item is a command, and so on. See the manual for further details.

The HISTORY List

In addition to the facilities directly offered by the PA, there is an additional overlay available in our LOOPS load up: the active history icon. This provides a direct history list which is an active window (active window are windows in which the mouse has some defined functionality).

```
EX:     put the arrow into the history window icon
        mouse the right button
        select EXPAND
```

Now you have an active history list which you can manipulate.

```
EX:     SET(a (t r w))
        SET(a (i o u))

        now to "refresh" the memory list
          point the cursor at the title
             bar of the history window
          mouse the right button
          select UpDate
```

In the active history list you can select the operations we

have talked about by mouse actions. Mousing the left button over an item in the active history list causes that item to be REDOne. Mousing the middle button over an item causes a menu to pop up that will let you choose what to do to the selected item.

EX: get used to the active history window when you feel comfortable with it, then select the title item SHRINK, and the active window will disappear, and the history icon will come back

DEdit

In our work we will make use chiefly of two of the available editors: TTYIN (which you already know about) and DEdit. In this section, we'll explore DEdit.

A Stack Based Screen Editor

DEdit is a screen based LISP-structure editor. After you get familiar with it, you'll find DEdit to be very useful for operating on LISP structures. DEdit knows about the structure of LISP expression. The starting point for using DEdit is to select some s-exp (done by pointing the arrow at the object and buttoning the left button). If the object you want to select is an atom, then pointing to any place in the atom will do. If the s-exp is a list, then you select it by buttoning its enclosing paren (either the one at the start or the one at the end).

When you select an s-exp using DEdit, what you are really doing is pushing that s-exp onto a stack. DEdit will indicate to you what s-exp is on top of the stack by placing a heavy underline under that s-exp. The second item on the stack will be shown to you by a dotted underline.

The basic logic of using DEdit is to think of yourself as using a post-fix notation for the operations you want to carry out. Eg, when you want to replace one s-exp with another, then first select the s-exp you want to replace. Next select the s-exp you want to insert. Then select REPLACE from the EditOps. The EditOps live in a small window just to the right of the main DEdit window.

So the overall operation could be described as

select Target - select Source - select Operation

DF and DV

The function to invoke from top level to edit a function is DF, to edit a variable use DV.

```
EX:     SET(a (x y z))
        DV(a)
         [this will throw you to DEdit
          change the value of a to be (x z)
          exit DEdit by selecting the EditOps EXIT
                with the left mouse button]
```

The Edit Buffer

So far you know how to use DEdit to change the existing structure of an s-exp. Suppose you want to add something to the existing structure. While in DEdit if you start entering information from the keyboard, then an "Edit Buffer" will wake up. This Edit Buffer is another example of TTYIN, so you know how to handle it.

```
EX:     there is a function called
        factorial that is defined
        in your loadup. PrettyPrint
        the current definition.
        Notice the error in it.
        Now do
          DF(factorial)
                          1
        and fix the error
```

Defining Functions

So far you know how to edit functions, but not how to define them. In this section we'll cover that.

---

1 HINT: INTERLISP is case sensitive.

Functional Forms in INTERLISP

In ELISP (for example) we have available different functions to define functions of different sorts; eg EXPRs and FEXPRs. In INTERLISP, we use the same function, but put different forms inside the function to obtain different effects.

4 Kinds Of Functions

For purposes here, we will think of there being four different kinds of INTERLISP functions.

1. LAMBDA-SPREAD Functions

In defining these functions, we specify an argument list; the arguments for a LAMBDA-SPREAD function are evaluated. (LAMBDA-SPREADs are like EXPRs in ELISP.)

```
EX:     pp(exampleLambdaSpread)
        (exampleLambdaSpread 'a 'b 'c 'd)
```

2. LAMBDA-NOSPREAD Functions

LAMBDA-NOSPREAD functions are specified by having there be an atomic argument for the function. However many arguments the function is called with are evaluated. And the arguments are accessed within the function by using the auxiliary function ARG. Inside a LAMBDA-NOSPREAD with an argument N, the first argument can be accessed by (ARG N 1). The binding of N itself is the number of arguments that have been specified. (LAMBDA-NOSPREADs are like LEXPRs in ELISP.)

```
EX:     pp(exampleLambdaNoSpread)
        (exampleLambdaNoSpread 'a 'b 'c 'd)
          [see the manual to find more
           about the useful output function
           PRINTOUT]
```

3. NLAMBDA-SPREAD Functions

NLAMBDA-SPREAD functions behave just like LAMBDA-SPREADs do, EXCEPT the arguments are not evaluated. (There is no functional ELISP equivalent.)

EX: pp(exampleNLambdaSpread)
(exampleNLambdaSpread a b c d)

4. NLAMBDA-NOSPREAD Functions

NLAMBDA-NOSPREAD functions do NOT behave like LAMBDA-NOSPREADs. The form is the same: these functions take a single argument. But here the single argument is bound to a list of the unevaluated arguments that the function is called with. (NLAMBDA-NOSPREADs are like FEXPRs from ELISP.)

EX: pp(exampleNLambdaNoSpread)
(exampleNLambdaNoSpread a b c d)

DEFINEQ

At this point, you know the various functional forms for INTERLISP, but now how to define a function. The easiest way is to use DEFINEQ. DEFINEQ itself is an NLAMBDA function that can be used to define any number of functions in one fell swoop. The way to use DEFINEQ is

```
(DEFINEQ (⟨nameOfFunction1⟩
           (⟨functionForm1⟩ args1 body1))
         (⟨nameOfFunction2⟩
           (⟨functionForm2⟩ args2 body2))
         ...)
```

where the functionForms are LAMBDA or NLAMBDA.

The ⟨functionForm⟩ parts default to LAMBDA if not supplied.

EX: define a function with no arguments,
and no body called sillyIncrement
then DF sillyIncrement
and make it to be a LAMBDA-SPREAD
function of 1 argument, whose action
is to just increment the argument.

EX: make a function called reasonableDefine

```
that takes one argument, a litAtom,
makes a function named ‹litAtom›, puts
you into DEdit to create the definition
of ‹litAtom›, and prints the message
        ‹litAtom› defined
on exit.
```

The File Package

Near the beginning of this set of notes, the INTERLISP philosophy was stated as (in part) making things easy for the user. One of the best examples of this can be seen in the file package.

The Idea of the File Package

The File Package is a system that attempts to free the user from most concerns about where his various functions and values are stored. In most other dialects of LISP, the user must keep track of his functions and variables himself.[2] In addition, in most other dialects, the general mode of "editing/testing" can be thought of as a loop first going to the source file, then loading the (changed) source, then testing. In INTERLISP, there is no real idea of a text file for the source that is to be accessed by the user. Rather, you think of your functions as running AND being edited directly in INTERLISP, then let the file system worry about where they should be stored.

Telling The File Package Where To Store Things

5. FILES?

One way to tell the file system where to store things is by using the function FILES?(). Suppose you have created a function (as you just did above) and now want to let the file system know where to store it. Simply say FILES?() and INTERLISP will let you declare where the new function is to be stored.

---

[2] ELISP provides some overlays that mitigate this.

EX: do FILES?()
and inform the file system
that you want to save reasonableDefine
in a file called LOOPSCLASSUTIL

6. xxxCOMS

EX: pp(LOOPSCLASSUTILCOMS)
and note the contents of that
variable

The file package "works" by keeping a variable named xxxCOMS (if the file name is xxx) and storing in that variable all the information about what is supposed to be in file xxx, and actions to take on storing or loading file xxx. FNS and VARS are just two file package commands; there are MANY others to allow you to specify things like special actions on loading or saving the file, initializing values for variables, and so on. See the manual for details.

7. Saving Your Work

You'll want to save your work from time to time in permanent form. There are two directories you can use right now: the hard disk {DSK} and the floppy disk {FLOPPY}.[3] You can connect to whichever one you want by doing, for example

CONN {FLOPPY}

Once you are connected to your directory of choice AND you have informed the file system of what you want in a given file (note: you only have to inform it once. Ie, if you have a new function FOO, and do FILES?(), then the file system will let you tell it where to put FOO. If you now do FILES?() a second time, then you will not be asked about FOO again.) then you can use MAKEFILE(<fileName>) to store away to the current directory your file.

---

[3] NOTE: the permanent disk is not truly permanent now - someone may come along and get rid of whatever you want to save. It is recommended to do your "permanent" saves to {FLOPPY}

```
EX:     insert your new floppy into the machine.
        FLOPPY.FORMAT(class)
          this will format the floppy.
          Do this ONCE. This function
          destroys anything currently
          on the floppy.
        CONN {FLOPPY}
        MAKEFILE(LOOPSCLASSUTIL)
          now you've saved the file
          LOOPSCLASSUTIL
        dir()
          this will show you the
          contents of the currently
          connected directory.
```

To reload the file, use the function LOAD, eg LOAD(LOOPSCLASSUTIL)

Once you load a file at the beginning of a session, the file system will keep track of any changes that are made to that file. Then you can simply use the function CLEANUP() at the end of the session. CLEANUP forces the rewriting of any files whose contents have changed. Note that in part the action of CLEANUP is controlled by the setting of CLEANUPOTIONS. You might check this out if you're going to be a serious user.

IV

INTERLISP TURTLE EXERCISES

In this exercise, you will bring together some of the things you have learned about INTERLISP-D, as well as see some of the the capabilities of the DLion display (but not very much).

We have at our disposal a wonderfully dutiful critter, a TURTLE named ART. And we have a number of instructions we can issue to ART. Among them

- CENTER - brings up ART in the middle of a special TURTLE cage (read, WINDOW)
- TCLEAR - (alas) sacks ART (ie clears the TURTLE cage)
- FORWARD(<distanceToTravel>) - tells ART to move forward by a specified amount, leaving a trail of wasted linebackers in his wake.
- JUMP(<distanceToJump>) - instructs ART to jump by the specified amount like H___ to avoid the rush
- POINT(<absoluteDirectionToPoint>) - tells ART to turn to a specified direction.
- TURN(<relativeTurningAngle>) - tells ART to turn a specified number of degrees from his current direction.
- DrawTurtle, EraseTurtle, ComputeTurtleArrayIndex - these are the base functions that the others rely on.

EX:
```
CENTER()
CLEAR()
CENTER()
FORWARD(50)
TURN(90)
JUMP(80)
```

ART is really an instance of a record called TURTLE. You can look directly at the record for ART by using an INTERLISP display facility call the INSPECTOR.

EX:
```
(INSPECT ART)
  first you get a menu.
 select AsRecord with the left button.
```

Now you'll have another menu to
specify what kind of record ART is.
Button TURTLE.

You then get a small little square next
to your cursor.
At this point INTERLISP is waiting for
you to place the "window" for the
inspector of ART.
(i think it may be an NFL inspector)
Anyway, you should mouse
the left button where ever you
want this inspect window to come up.

The inspector is handy, because it permits you to look at various INTERLISP constructs in a structured way, and also allows you to set parameters in the object you display.

EX: Change the CURRENTX field of ART.
mouse CURRENTX with the left button.
mouse CURRENTX with the middle button,
then select SET. This will get you
a mouse window in which you can set
a new CURRENTX.

Close the mouse window and the inspect
window by mousing the windows with the
right button and selecting CLOSE.

DrawTurtle()
note that ART hopes over to the
new CURRENTX.

EX: Get used to the basic
functions that ART has.

Now we'll make ART a little smarter.

EX: construct a function
called BACK that will
make ART go backwards
by a specified amount.
(after all, there may be some
big suckers on the other side

of that line!)[1]

Now lets give ART some other interesting patterns to run.

EX: make a function called SQUARE that takes one argument and makes ART draw a square in the TURTLE CAGE with the length on one side being the argument we provide.

Finally, lets give ART some company.

EX:
```
(SETQ BROCK (CREATE TURTLE))
  this creates another TURTLE
  named BROCK

(SETQ turtleWindow
      (fetch WINDOW of ART))
  this gives you the actual
  TURTLE CAGE that ART is using

use the inspector to set the
   window that BROCK uses
   to be the same as ART's
   window
```

What we want to do now is to make our turtle functions be applicable to two turtles.

EX: modify all the turtle functions to take as the first argument the name of the turtle to which we apply the function. Instead of just altering the functions we already have, make new functions and perform the alterations on the new functions. Call the new functions the same name

---

[1] Make life easy for yourself. Think about the relation of BACK to the existing function FORWARD.

but with the prefix osu. Use the function MOVD to initially create the new function, then edit the new one.

Eg,

```
MOVD(FORWARD osuFORWARD T)
```

The T is a copy flag, see the manual for the meaning.

```
DF(osuFORWARD)
```

Now we have a set of turtle functions that can be apply to arbitrary turtles.

EX: make a function osuSquareDance that will cause ART to draw a square around the center of the screen, while his partner BROCK does the same thing, just 180 degrees out of phase with ART[2]

EX: save all of your TURTLE functions

---

2 The width of your turtle cage is 550, the height is 630.

V

XIV. 49.

GETTING USED TO LOOPS

Yesterday, you learned the basics of using INTERLISP-D. Today, we'll begin to learn LOOPS. LOOPS is a language designed to offer a number of the different paradigms for system construction which have been found useful for building AI systems.

Introduction

Programming Paradigms In LOOPS

LOOPS offers (currently)[1] four different programming paradigms:

1. procedure oriented programming - the "normal" way of building programs out of procedures which can "call" each other,

2. object oriented programming - a way of conceptualizing a program to be number of interacting "objects" which get work done by passing (locally understood) messages to each other,

3. data oriented programming - the data driven style of programming, and

4. rule oriented programming - the view of a program as a knowledge base collection of rule groups which are interpreted by some external interpreter.

At its current stage of development, one approach for understanding LOOPS is to view systems that are constructed in LOOPS as being object oriented, with the other modes of expression being submodes. This will become more clear a little later.

---

[1] One possibility for the future that is being considered by the LOOPS development team is a constraint style of program construction.

Display Oriented Tools

One of the most important aspects of LOOPS from a system building perspective is the wealth of display oriented tools that INTERLISP/LOOPS offers. These include

- DEdit facility,
- the Inspector,
- inbuilt "browsers" for displaying LOOPS the relations between LOOPS objects, and most importantly
- the ability to customize your own "browsers".

As we'll see, the browser notion is a powerful system building tool.

Brief Introduction To The ClassBrowser

In todays class, we will be mainly learning about LOOPS classes, but in order to manipulate the classes we will be using, you need some familiarity with the LOOPS ClassBrowser. In a later class, we'll see how browsers work and will build a browser. But for now, we'll just view the browser as a tool.

Bringing Up A Browser on a LOOPS Class

LOOPS classes are LOOPS objects which are related to each other by their location in a class hierarchy.[2] The ClassBrowser is a tool to bring up a graphical lattice structure showing the class hierarchy. A ClassBrowser is brought up by specifying the "StartingList" of the part of the class lattice that we want to see. Suppose we have a LOOPS class called osuCSProfessor, and we want to view osuCSProfessor. We do this by sending a New message to the LOOPS class ClassBrowser. (Hold on, you'll learn about messages and message passing below.)

---

[2] We'll see what this means later today.

EX: (<leftArrow>New $ClassBrowser Show '(osuCSProfessor))

Once we have a LOOPS class displayed in the browser there are many actions we can take to let us either display what is in the class or to change what is there, both using the ClassBrowser. In general, the left mouse operations in a ClassBrowser are display operations while the middle mouse operations are edit operations. When you hold the arrow on an class in the class browser, and select a mouse button, a menu will appear that will offer options. If a menu item has a star at its end, then there is a submenu that may be popped by selecting that class with the middle mouse button.

EX: left mouse the class osuCSProfessor
select PP* with the middle mouse button
select PP with the left mouse button

In the exercise above, you can see that the browser will give you access to the "innards" of an class, but the power of a browser goes much beyond that.

EX: close the current ClassBrowser by mousing the right button anywhere inside its window and selecting CLOSE

(<leftArrow>New $ClassBrowser Show '(Person))

Now we are looking at a part of the LOOPS class hierarchy that is currently loaded. Note that you can understand by simply looking at this graphic that, for example, osuCSProfessor is somehow related to Person. You don't now really know all the "methods" (read for now functions) that any of the classes knows about, but you can see some kind of relation between the classes.

The above illustrates the real power of the browser idea from a system building viewpoint. When you set out to create a system in LOOPS, a good starting point is to consider all the classes you are going to be concerned with, and structure the inter-relations between these classes using a browser. After these top level inter-relations are expressed, then the time comes for filling in the details, ie the "innards" of the classes.

Now that you have a little idea of what the browser is used for, we'll go on to describe the nature of LOOPS classes.

LOOPS Classes

LOOPS classes from a conceptual view may be considered as active agents which have both local storage and which understand messages. They correspond to the construct of "data abstraction" from the programming language community [3], but you should always view them as active agents.

They also should be thought of as a the holder of "typical instance" type information, similar to a frame-like idea.

LOOPS classes may be instantiated, a process that yields an "instance" of the class (not too surprisingly).

Storage

The local storage of a LOOPS class comes in two flavors:

- CVs (Class Variables) and
- IVs (Instance Variables).

CVs are LOOPS class variables which are declared and exist in the class and which may be accessed by all instances of the class.

IVs are LOOPS class variables which are declared in LOOPS classes, but which can exist in either the class or any instance of the class.

---

[3] minus the idea of import and export variables and procedures, but plus the idea of instantiation and inheritance

1. Basic Operations On IVs

As with any kind of storage facility, IVs have fetch and store operations. The fetch operation for IVs is

(GetValue ⟨loopsName|variable⟩ ⟨variableHoldingCVName⟩)

Here ⟨loopsName⟩ is the LOOPS name (read POINTER) for a LOOPS class. For example, there is a LOOPS class we call osuCSProfessor. LOOPS maintains a pointer to the internal structure for that class. To get that pointer we do

($ osuCSProfessor)
or just
$osuCSProfessor[4]

So in the above to get the IV named 'salary for the LOOPS class osuCSProfessor, we would do

EX: (GetValue $osuCSProfessor 'salary)

Alternatively, we could have set some variable to be that pointer then used the variable in the GetValue form.

EX: (SETQ currentLOOPSclass $osuCSProfessor)
(GetValue currentLOOPSclass 'salary)

Note that we are fetching the value of an IV from a class.

---

[4] $ is both a function and a read macro.

EX: mouse the left button on osuCSProfessor
and select PP

We can also set the value for the salary IV in osuCSProfessor.

```
EX:     (PutValue $osuCSProfessor 'salary 30000)
            [the state ran out of money!]
        (GetValue $osuCSProfessor 'salary)
```

So far we've looked at fetching or setting IVs in classes. Now let's consider instances. Instances can be either "named" or "unnamed", depending on somethings you'll understand a little later. Our first exposure to instances here will be of the named variety. You have an instance in your loadUp of $osuCSProfessor named $jones. Right now the local state of $jones for the variable salary is not set. But you will still get a value for salary if you fetch it from $jones. Can you guess what it'll be and why?

```
EX:     (GetValue $jones 'salary)
```

The salary IV that is set in the class that $jones is an instance of (ie $osuCSProfessor) is returned unless there is a local setting in $jones. The value in the IV at the class level then acts as a default value for all instances in the class.

On the other hand, setting the value in $jones will not effect any other members of the class.

```
EX:     (PutValue $jones 'salary 41000)
        (GetValue $osuCSProfessor 'salary)
        (GetValue $smith 'salary)
           [$smith is another instance of
            the same class. He has no
            local state for salary.]
```

2. Basic Operations On CVs

CVs have the same two kinds of operations, but the action is a little different. Remember, CVs do not exist in instances at all, only in classes. Suppose we have a declared CV in $osuCSProfessors called maxSalary.

EX: look at the PP for
osuCSProfessor
again to locate
the CV maxSalary.
Note its value.

```
(GetClassValue $osuCSProfessor 'maxSalary)
(GetClassValue $jones 'maxSalary)

(PutClassValue $jones 'maxSalary 50000)

get a fresh PP of $osuCSProfessor

(GetClassValue $smith 'maxSalary)
```

CVs (which only exist in at the class level) ARE SETTABLE from the instance level. And once reset, the effect is felt in all instances of the class.

3. Inheriting CVs and IVs

CVs and IVs may be inherited from higher levels in the class structure (as shown in the browser). The easiest way to find out where in the hierarchy various CVs and IVs live is to use the left browser menu item WHEREIS.

EX: select WHEREIS on
osuCSProfessor
then the subitem
CV, and find out
where all the CVs
live

do the same for the
IVs of osuCSProfessor

The action of CVs as holders of data for all instances (or for

the class itself) and of IVs as actual storage locations for instances and default value in classes remains unchanged.

```
EX:     (GetClassValue $jones 'maximumAge)
        (PutClassValue $jones 'maximumAge 99)

        (GetClassValue $professor 'maximumAge)
        (GetClassValue $MrT 'maximumAge)
          [$MrT is an instance of
                $citProfessor]
```

To generalize: the setting of the CV takes place in the class in which it resides.

There is one important thing to point out regarding how CV are to be used when they are inherited. Note from the about that maximumAge is a CV that is declared in $professor.

EX: what do you suppose will happen if you do

```
(PutClassValue $citProfessor 'maximumAge 22)
```

try it and see

CVs that are inherited to INSTANCES of an class through the class hierarchy can be set in classes in which they do not live! Try to figure out why this is a reasonable design feature for a language like LOOPS.

There are other ways of fetching and storing the values of CVs and IVs. For example, you can fetch the local state only of an IV.[5] See the manual for details.

Further, we have talked about the "values" of CVs and IVs, but in reality, both CVs and IVs have property lists. Look back at

---

[5] If it is not defined locally, LOOPS will return a ?. You can change this to something else if you don't like it.

the PP of some class. Note the form for an IV or CV will look like

EX: (<nameOfVar> <valForVar> doc <someDocumentation>)

The "doc" is just one example of a property. You can define your own properties for any CV or IV. Again, see the manual for details.

4. Adding And Deleting IVs and CVs

The easiest way to add either a new IV or a new CV is to use the middle mouse menu item Add*.

Similarly, you can use the middle mouse selection Delete to get rid of some unwanted IV or CV.

Messages

Now that we have some idea of IVs and CVs, we'll move right along and talk about LOOPS "methods". Methods are the means by which a LOOPS class knows how to respond to a message that is sent to it. The way to send a message to a LOOPS class is by the form

(<leftArrow> <loopsClass> <methodName> <arg1> <arg2> ...)

EX: (<leftArrow> $jones setSalary 35000)

look at setSalary with PPMethod

The first argument in ANY method is self. This self refers to the current class to which you have sent the message. The binding of self to the class to which the message is sent will be important a little later.

Look back at the PPMethod for setSalary and see what else should have been potentially set by the method.

EX:
```
(GetClassValue $professor 'maxSalary)
(<leftArrow> $jones setSalary 39000)
(GetClassValue $professor 'maxSalary)
```

Methods that are to be understood by instances reside in LOOPS in

- either the class that the instances are instantiated from or
- some class that is up the class hierarchy from the instantiating class (in LOOPS parlance - on the supers list of the class).

THIS IS A VERY IMPORTANT NOTION. YOU'LL DO YOURSELF A BIG FAVOR IF YOU REREAD IT SEVERAL TIMES AND MAKE SURE YOU UNDERSTAND WHAT IS BEING SAID.

To create a LOOPS method, we can use the Add* middle mouse menu selection, then the DefineMethod sub selection.

EX:
```
create a new method to reside in
$professor whose action will
be to set the age IV and up date
the maxAge CV. Call this new method
setAge
```

On what LOOPS entities will setAge operate on?

EX:
```
(<leftArrow> $jones setAge 39)
(<leftArrow> $MrT setAge 65)
```

The method you have created will be available in $professor AND in all instances of LOOPS classes under $professor in the class hierarchy. The method is inherited. This is one reason why the

self argument is so important. self sets the context for evaluation, even for a method which is not locally present.

One thing should be bothering you at this point:

> if methods that reside in LOOPS classes are understood by the (possibly indirect) instances of those classes, then how does one write a method that will be understood by the LOOPS class itself?

One example of a message that you need to send to a class rather than to an instance, is the message that requests a new instance to be built.

```
EX:     (SETQ moorthy
              (<leftArrow> $osuCSProfessor New))
        (<leftArrow> moorthy SetName 'moorthy)

         [the important part is
            the sending of the New
            message]
```

Where does this new method reside? Not in $osuCSProfessor, nor on any class on its supers list. The answer involves a different kind of LOOPS class called a meta-class. The base distinction between a class and a meta-class is that instantiations of any meta-class are themselves classes. In a sense then, meta-classes are "higher up" in an analogous way that normal classes are "higher up" than instances.

Although this may be a little confusing right now, the operational way to understand this is that if you want some class itself to understand some method, then the cleansest way to proceed is to put that method into the meta class of the one you are dealing with.

The way that you can alter the meta-class of some class is

1. be SURE that you have the meta-class defined FIRST

2. select Edit* with the left mouse button

3. change the entry for MetaClass that is near the top of the definition of the class

The Notion Of SPECIALIZATION In LOOPS

In the above, we saw that both variable (in either the IV flavor or the CV flavor) and methods are inherited in LOOPS. But this important feature of the language is lost unless we also have in hand the concept of Specialization.

In LOOPS, a new class may be "tacked on" to some existing class in the class hierarchy (as shown in the browser) by

EX: middle mouse $osuProfessor
select Add*
select Specialize
give the name
osuHistoryProfessor

do a PP on $osuHistoryProfessor

select WHEREIS on
$osuHistoryProfessor
and see what CVs are
known

Note that $osuHistoryProfessor was created, and in fact it was empty. BUT CVs were known to it. That was through the inheritance mechanisms of LOOPS. The important idea is that in LOOPS, classes can be Specialized from existing classes, and only NEW IVs, CVs, or Methods that do not exist in the classes on the supers list of the new class need concern us.

The importance of specialization cannot be overempahsized. We'll be doing an exercise later in the course on making a browser in which we will use the notion of specialization quite heavily.

VI

copy on darker

stored on [indigo]<Loops>loopscourse>ExerciseGettingFamiliar.bravo

Last Edited: MJS November 10, 1983 4:07 PM

Getting Familiar with Loops

Prepared by the LOOPS DesignTeam

Danny Bobrow, Sanjay Mittal, and Mark Stefik

Purpose of these Exercises

These exercises provide an introduction to the Loops programming environment. They introduce the Loops *Browser* and *Inspector*, which are important tools for defining and editing classes. Notation for accessing variables, and invoking methods is explained. The exercises also provide examples of the use of tools for understanding and debugging programs -- such as gauges and the object break facility.

Notation

In the dialogs below, regular font is used to show what the system types, and bold font is used to show what the user types.

Preliminaries

Go to your workstation for the course and LOGIN according to the instructions in the handout titled "How to Start Loops".

Part A. Browsing through the class lattice.

(1) Display the class browser starting at the class *Commodity* as follows:

(←New $ClassBrowser Show '(CommodityTransportability Commodity))

Position the browser by moving the cursor (the cursor moves by moving the mouse) and clicking the left mouse key when you have placed the browser window in the desired position on the screen. A recommended position is the lower right hand side of the screen.

(2) Scrolling and shaping the Browser.

If the lattice is two big to fit in the window, *scroll bars* can be used to shift the part of the lattice that appears.

To shift the lattice horizontally, slide the cursor through the Browser slowly past the bottom of the window. A horizontal scroll bar will appear and the cursor will turn into a double-headed arrow. Holding down the left mouse button in the scroll bar causes the lattice to shift to the left. Holding down the right mouse button causes the lattice to shift to the right. The middle button is used to indicate a proportional space in the lattice for positioning it in the window. For example, holding and releasing the button in the middle of the scroll bar causes the middle of the lattice to be displayed in the window.

On a Dandelion with a two-button mouse, the CENTER key on the keyboard is used in place of the middle mouse button.

(EX) Use the scroll bar in the Commodity class lattice to familiarize yourself with the control of scrolling.

To shift the lattice verically, move the cursor into the Browser and slide it slowly past the left edge of the window. A vertical bar scroll bar will appear.

(EX) The vertical scrolling commands are analogous. Discover them by trying the different mouse buttons.

(EX) Reshape the browser using the right mouse button.

(3) Printing and editing selected objects. Here's a summary of using the mouse to interact with a browser. Selection takes place on the *upward* transition of the mouse button.

At the Top Level of a Browser

Left button Brings up a menu of *Printing* options

Middle button Brings up a menu of *Editing* options

Right button Brings up a menu of *Window* options

To get a description of an option, keep a button depressed over the menu option for a few seconds and watch the black prompt window.

In a Browser Menu

Left button *Default* option

Middle button *SubOptions* -- spawns a menu of suboptions.

Menu items followed by an asterisk (e.g., Doc*) have suboptions.

(EX) Move the cursor in the Browser to Apple, and click with the left mouse button. A menu of print options will appear next to the cursor. Select the Print* option with the left key.

(EX) Try out *PrintSummary* and *PP* in the *Print** sub-menu, and *Doc** in the *Method* sub-menu.

(EX) Use the browser to *Inspect* a class. (Hint: *Inspect* is a suboption of *Edit**).

(EX) The *Doc** option makes it convenient to browse documentation in the classes.

Experiment with the Doc* option. On the class *Apple*, what is the documentation of the *SetPrice* method? What is the documentation on the *qty*, *pr*, and *price* instance variables? What is the difference between *pr* and *price*, according to the documentation? What is the default documentation that you get when you select *Doc** with the left button, without looking at suboptions?

Part B. Seeing where variables and methods are inherited from.

In Loops, variables and methods are inherited through the class lattice. The PrintSummary option and PP! options show things that are inherited from above. PP! (pronounced "pretty print bang") prints out all of the inherited information.

(EX) Select *PP!* (suboption of Print*) on *Groceries* in the browser.

The method names are a clue to where they are inherited from. Each method appears in the listing as

(selector methodName doc (some comment.)*)

The names of the methods are *ClassName.Selector* by convention in Loops. Several of the methods of *Grocery* are inherited.

The *WhereIs* option (suboption of Print*) is a good way to find out where a particular instance variable, class variable, or method is inherited from. *WhereIs* brings up another menu, offering IVs, CVs, and Methods. Selecting any of these (by moving the cursor and clicking with the left mouse key) will offer a menu of the names of IVs, CVs, etc. The browser will flash the class from which the variable or method is inherited.

(EX) Select the *WhereIs* option on Strawberry. Use it to find out where several of the instance variables and methods come from.

The menu of IVs, CVs, or Methods will still be around after the first selection - just select another item to see where it is. To make this menu disappear, move the cursor outside this menu and click the left button.

If nothing flashes in the browser, look in the black prompt window at the top of the screen for a message. The node from which the selected IV, CV or Method is inherited may not be visible in the Browser.

(EX) Try to see where all the different parts of Strawberry class are inherited from.

Part C. Creating Instances.

(1) Create an instance of Apple

(← $Apple New 'MyApple)

This creates an Apple and gives it the Loops name *MyApple*. Instances with names can be accessed by preceding the name with a dollar sign.

(2) Print its structure

(← $MyApple PP)

(EX) Create instances of other classes. You can give them names as in C(1) above or save a pointer to the instance in a Lisp variable as follows:

(SETQ app1 (← $Apple New))

In this example, the instance is not given a name but it can be referenced by using the Lisp variable *app1*.

(EX) Print out the second apple.

(← app1 PP)

Some classes cannot be instantiated. For example, the class *LuxuryGoods* is an abstract class which does not describe a particular kind of thing that can be instantiated.

(EX) See what happen when you try to instantiate *LuxuryGoods*. Print out *LuxuryGoods* to see how the "abstract" class is specified. (Hint: look at the meta class.)

Part D. Setting values of variables.

(1) To retrieve the value of an instance variable in Loops, the function *GetValue* can be used, or its shorthand *@*. For example,

(GetValue $MyApple 'price)
(@ $MyApple price)
(@ app1 price)

Use this to get the value of some instance variables of *$MyApple* or *app1*.

(2) To set a variable in Loops, the function *PutValue* is called as follows:

(PutValue $MyApple 'price 60)

A shorthand for putting values is:

(←@ $MyApple price 60)

(EX) Set the instance variable *qty* of *$MyApple* to 30. Use the GetValue @-shorthand to verify that the value was changed. Also try sending a PP message to *$MyApple*. Looking at the printout, how can you tell which values are set locally, and which are inherited?

Part E. Invoking a method on an instance.

(1) In Loops, the syntax for sending messages is

(← object Selector arg1 arg2 arg3 ...)

The leading left arrow means send a message. :

(EX) For example, send a Display message to *$MyApple* to display its icon as follows:

(← $MyApple Display)

(EX) Send a SetPrice message to *MyApple* to set its price

(← $MyApple SetPrice 30)

Use the browser to see the name of the function that implements the *SetPrice* method.

Part F. Inspecting an object

(1) To inspect an object's structure, send it an *Inspect* message

(← $MyApple Inspect)

The same message for inspecting that works above for instances, also works for classes. Classes can also be inspected through the class browser.

(2) Changing the value of an instance variable
Select the IV name by moving the cursor over its name in the inspect window and clicking the left mouse key. Next, press the middle key over the same selection to bring up a command menu. Selecting *PutValue* from this menu will allow you to enter a new value for the IV in the prompt window.

(EX) Using the inspector, change the variables *pr* and *qty* of *$MyApple.*

(3) Inspecting value structure and properties.

Selecting the IV value (using the right column, not the IV name!) allows operations on the values and properties of the IV. Select a value with the left key and bring up a command menu with the middle key.

(EX) Try *Inspect* and *Properties* on some selected values of *$MyApple.*

(4) Commands in the title bar of the Inspect window.
Other operations are possible on the inspected object by pointing the cursor in the dark title region of the inspect window and holding the middle key. Move the cursor over the desired operation with the middle key pressed. To see a description of an operation, hold it over the option for the section and look in the prompt window. Releasing the key executes the selected command.

(EX) Get a description of IVs, and LocalValues.

(EX) Try *Redisplay, IVs, Class, LocalValues.* Try going back and forth between the LocalValues and AllValues.

(EX) Using the @ notation, change the *pr* again. Notice (sigh), that the inspector does not update its display automatically. To make sure that the displayed values are current, use the *Redisplay* option in the menu from the title bar.

Part G. Monitoring changes via active values.

(1) Inspector windows are only "refreshed" with new values when they are first drawn, or when a *Redisplay* option is selected. This means that they can appear on the screen with outdated values. A way of maintaining up to date data on the screen is to use gauges. Unlike inspector windows, gauges are automatically updated when Loops variables change. The following code attaches a *.VerticalScale* (a kind of gauge) to the *price* instance variable of *MyApple*.

(←New $VerticalScale Attach $MyApple 'price)

Suggestion: Before you type the closing right parens, move the cursor out of the typescript window. If you can't find the gauge, it may be under the typescript window. Move the typescript window to get it.

(2) Change the value of *price* as in part D and see the gauge change.

(EX) Create a Browser for the class *Gauge*. Using the Browser, use the *ClassDoc* option (suboption of *Print**) to get information about the different gauge classes.

(EX) Attach a different gauge such as *Dial*, *Meter*, or *DigiMeter* instead of *VerticalScale* to the IV *qty* of *MyApple*. Test the gauge by changing the value using parts D.

(3) Gauges are attached through *active values*. These active values appear in the value of Loops variables, and have a structure that can be examined.

(EX) Inspect the value of *price* using the inspect facility described in part E. See that the form of the value has changed. Inspect it down to the actual value. What is the *PutFn* and *GetFn* for the active value that drives the *VerticalScale*?

Part H. Debugging via active values.

(1) A perplexing problem in many programming systems is finding out what part of a large program smashed the value of a particular variable. In Loops, active values enable the system to enter a Break whenever the value of particular variable is changed. For example,

←(BreakIt $MyApple 'price)

will cause a break when the *price* variable of *MyApple* is changed.

(EX) Put a variable break on *MyApple* as above. Change the value of the variable. Reacquaint yourself with the Break package commands.

(EX) Change the variable price using the *SetPrice* message, and verify that it enters the break again. What are the arguments to *Apple.SetPrice*?

(2) To remove a break, use the *UnBreakIt* function as follows:

(UnBreakIt $MyApple 'price)

(EX) Change the value of *price* again, and verify that you no longer enger a break.

(EX) Put a variable break on *qty* verify that it breaks when you change the value.

(EX) Inspect the *qty* of MyApple now that it is broken. Examine the active value that causes the

Break. Again inspect down to the actual values.

Part I. Editing an object.

Any object can be edited using the Lisp editor by sending the object Edit message. For editing a class, if you have the class in a browser, you can use one of the Edit commands in the middle menu (see Step A for using the browser). E.g.

(← $Apple Edit) will allow you to edit class *Apple*
(← $MyApple Edit) will allow the same for instance *MyApple.*

(EX) Edit *MyApple* to change value of *pr.*

(EX) Add a new instance variable *growIn* to *$MyApple* with value *Washington.*

(EX) Add a description for a new instance variable *color* to the class *Apple*, and give it the default value *Red.* Now use the @-shorthand to get the value of *color* for the instance *$MyApple.*

HAPPY LISPing, LOOPing, RULing and TRUCKINg!!

If you need any help contact one of the course instructors.

VII

Copy darker and @ 98%

Filed on {INDIGO}<Loops>LoopsCourse>ExerciseGauges.bravo
Edited: MJS November 10, 1983 4:58 PM

Class Specialization and Mixins

(The "gauges" exercise.)

by the Loops Design Team

Daniel Bobrow, Sanjay Mittal, and Mark Stefik

Purpose of these Exercises

These exercises provide some practice in object-oriented programming, in specializing and combining classes. The examples are drawn from the "gauges" that are part of Loops. Gauges are objects used to create dynamic graphical images of values of Loops variables. Every one should be able to complete parts 1 through 4. Part 5 is for those who finish quickly and want to try something harder, more on their own.

Notation

In the dialogs below, regular font is used to show what the system types, and bold font is used to show what the user types.

Preliminaries

Go to your workstation for the course and LOGIN according to the instructions in the handout titled "How to Start Loops".

Part 1. Making A SubClass (a blinking LCD)

Step 1. The idea of this exercise is to make a new gauge which blinks each time it receives the message *Set*. It should blink once if the value has gone down, four times if the value has gone up, and not at all if the value is unchanged. It should display its current value as is done by an LCD. Since it's behavior is identical to LCD except for its response to the message *Set*, a good first step is to make a new subclass of LCD. Put up a browser on the set of classes for gauges by typing:

(Browse GAUGESCLASSES)

A small square will appear to allow you to place the browser down where you want it. Hold down the left mouse button until the outline of the browser is positioned properly, and then release the button. It will probably be convenient to place the browser in the lower left area of the screen.

Step 2. Use the Browser to make a subclass of *LCD*. This can be done by using the *Specialize* suboption of *Add**. After you select *Specialize* in the menu of suboptions, you will be prompted in the prompt window (the small black window in the upper left hand corner) for the name of the new class. It will look like:

Class Name: **BlinkLCD**

Step 3. Document the new class. This can be done by editing the *BlinkLCD* class definition. Using the browser, select *Edit** with the left button. This will invoke the Lisp editor on the class definition.

Adding documentation requires adding both a *doc* property name, and its value -- a comment. Use the gauge browser to see the documentation on *BoundedMixin* for an example. This example illustrates an *editing idiom* that is useful in cases where two items need to be inserted, Select the atom *Edited* in the *MetaClass* part of the *BlinkLCD* source. Then type a list like the following into the edit buffer:

(doc (* BlinkLCD is a blinks when it changes value.))

Then click *Before* in the edit commands. This inserts the list containing the property name and the value into the right part of the definition. Finally, use the *(out)* command to remove the parentheses around the list. Exit DEDIT when you are finished..

Step 4. Study the definition of LCD. Decide which methods and variables that you think will need to be specialized.

Hint. It should be enough to specialize just the *Set* message. Consider using ←*Super* together with the message *Blink* inherited from Window. The *Blink* can be invoked with **(← self Blink 3)** to make the window blink 3 times).

Step 5. The next step is to define a *Set* method for *BlinkLCD*. This code should check the current value of the IV *reading*, do its blinking, and then send *reading* to the *Set* method of the Super class. One way to define a method is to use the *DM* suboption of *Add**. This will place you into DEDIT with a functional template for the method. Edit the template to look something like:

```
(LAMBDA   (self reading)
            (* Comment describing what the Set method does.)
          [COND
           ((IGREATERP (@ reading) reading))
             (* Blink once if the value has gone down)
             (← self Blink 1))
           (T (* otherwise ...)  ]

              (←Super self Set reading]
```

After leaving the editor on the function, go back and look at the class definition of **BlinkLCD** again, and see how the documentation that you entered was automatically included in the class.

Step 6. Create an instance of **BlinkLCD** and display it on the screen. (*Hint:* The new gauge will be created next to the cursor. Move the cursor so that the gauge will not be buried beneath the typescript window.)

```
(SETQ wink (← ($ BlinkLCD) New))
(← wink Update)        this is necessary to make the LCD appear
```

Step 7. Now try out your **BlinkLCD** by sending *Set* messages to it, to see whether it performs as you expected.

```
(← wink Set 10)
(← wink Set 55)
(← wink Set 35)
```

(← wink Set 35)

Part 2. Defining and Using A Mixin Class

The idea of blinking is a "modification" that could be made to any of the gauges in Loops. The previous exercise did this by specializing a particular gauge. In this exercise, we define a *Mixin* class that can be combined with any of the gauges to convert it into a blinking gauge.

Step1. The first step is to define the Mixin class, *BlinkMixin.* We will want this class to have *Object* as its only superclass. Since *Object* does not appear in any browser on the screen at this point, we can not use the *Specialize* option as before. Instead, we use a procedural way of defining the class as follows:

DC (BlinkMixin)

The super class of **BlinkMixin** will be defaulted to *Object.* Document this mixin as in Part 1..

Look at the browser. **BlinkMixin** will not appear there. To make it appear in the browser, add it via the *AddRoot* option in the title region menu. Remember to look up in the prompt window in the upper left.

Step 2. Now define a *Set* message using DM as in part 1. The same code should work for *Set.* You can copy the code from BlinkLCD by typing:

(← ($ BlinkLCD) CopyMethod 'Set 'BlinkMixin 'Set)

Edit the method to change the comment, and switch the number of blinks done on lowering and raising. To edit the method use the EM suboption in the browser.

Question: What does *←Super* in this method do?

Step 3. The next step is to use the mixin in the definition of a new class, *BlinkMeter. BlinkMeter* should take the blinking behavior that we defined previously for *LCD,* and combine it with the behavior for *Meter.* Define a **BlinkMeter** as follows:

DC(BlinkMeter (BlinkMixin Meter))

Use the editor to document this class as usual.

Step 4. Now make an instance of an **BlinkMeter** and test it as before.

```
(SETQ bm1 (← ($ BlinkMeter) New))
(← bm1 Update) this is necessary to make the gauge appear on the screen
(← bm1 Set 13)
...
```

Part 3. Using gauges to monitor internal state of an object

Now let us look at the internal working of *bm1* by using another gauge to monitor instance variables of *bm1*. Create a vertical scale gauge and attach it to the *reading* of bm1.

(← $VerticalScale New 'VS1)

(← $VS1 Attach bm1 'reading)

Note that when the *reading* of bm1 gets above 100, the vertical scale gauge *pins* at the top, and a ?? is shown in the upper left corner. Change the scale of *VS1* to encompass a larger range:

(← $VS1 SetScale 0 1000)

Now create an instance of *LCD* and attach it also to the IV *reading*. Attach another instance of *LCD* to *displayVal* in *bm1*. Watch what happens to each of these *LCD's* as you send *VS1 Set* messages. Click the middle button over one of the *LCD's*, and try *Attached?* and *Detach*.

Part 4. Defining A DigiDial

The *DigiMeter* in the Gauges file is a combination of a *Meter* and an *LCD*. In this exercise, we will define a *DigiDial* which will be a combination of a Dial and an *LCD*.

Step1. Study the definition of a *DigiMeter*. Look at at the definitions of its methods: *ComputeScale*, *Set*, *SetParameters*, *ShowSetting*, and *Update*. These methods combine the behaviors of a *Meter* and an *LCD* so that a *DigiMeter* exhibits the "sum" of the behaviors.

Hint: The invocation of DoMethod in *DigiMeter.SetParameters* may be an example of bad programming style. Could this be replaced by a ←Super ? If yes, what would be the advantage?

Step2. Define a class for *DigiDial* with supers as follows:

DC (DigiDial (Dial LCD))

Document this class as in Part 1.

Step 3. Now define methods for *DigiDial* to combine the behaviors of a Dial and an LCD analogous to those in DigiMeter. Document these methods as usual.

Step 4. Instantiate and test the *DigiDial*, and show it off to us!!!

Part 5.Making Yet Another Subclass (a counter example)

Step 1. The idea of this exercise is to make ***CounterLCD***, a new "counting gauge" which counts the number of times it receives the message *Increment*. Define this as a new subclass of *LCD*. Define the *Increment* message in terms of the Set message. Create an instance *c1* and test it.

Step 2. Attach this gauge to bm1. Since in this case we want to have the message *Increment* sent instead of *Set*, we must type:

(← c1 Attach bm1 'reading 'Increment)

This will cause c1 to be called with the message *Increment* every time bm1 is Set. Try this out.

Step 3. If you have time, define and create a *Reset* message, which resets a counter to zero. Try making a *Counter* mixin.

VIII

Copy @ 98%
on darker

Filed on {INDIGO}<Loops>LoopsCourse>ExerciseObjectTurtle.bravo
Edited: MJS August 3, 1983 7:21 PM

Using Procedures and Objects

Purpose of these Exercises

This exercise illustrates some techniques for combining *procedure-oriented programming* with *object-oriented programming.* We will again use the Turtle Graphics domain for this exercise. Instead of using records to represent turtles, we will create a *Turtle* class and instantiate it. The exercise will give you an opportunity to compare the characteristics of the *Turtle* programs implemented as records and as objects. It will also provide an example of exploiting *specialization* in the object representation, to create a modified turtle program.

Notation

In the dialogs below, regular font is used to show what the system types, and bold font is used to show what the user types.

Preliminaries

Go to your workstation for the course and LOGIN according to the instructions in the handout titled "How to Start Loops".

Part I. *Using Programs as Data*

This part of the exercise uses a recursive lisp program to help in translating the record-based Turtle Graphics program into an object-based program.

Step 1. *Create a Turtle class.*

First define the *Turtle* class using the function DC as shown below.

(DC 'Turtle).

Having created the *Turtle* class, you next need to add to it some class and instance variables corresponding to the record variables in the original Turtle Graphics program.

Create a browser for Turtle, and then use the editor to add the following instance variables to *Turtle*:

CURRENTX CURRENTY HEADING and *ERASED?*

EX: poke around in $LatticeBrowser and $ClassBrowser to see how the CVs for left and middle button items look

LocalCommands

The CV LocalCommands tells the browser processing mechanism whether the method associated with a menu item is understood by the browser itself or by the object you have buttoned. If a method name is contained in the CV LocalCommands, then when you button a menu item, the message associated with that menu item will be sent to the browser itself with two arguments

1. obj - a binding to the object that you have buttoned in the browser (ie binding to the POINTER to the LOOPS object) and
2. objName - a binding to the name of the object you have buttoned.

In addition self is bound to the instance of the browser you are working in.

If on the other hand the menu item you chose is NOT contained in LocalCommands, then the browser mechanism will send the associated message to the LOOPS object you have buttoned, with self bound to the LOOPS pointer for that object.

EX: poke around in the LocalCommands of $ClassBrowser

From the above you can see that if a browser item is a local command, then the method that responds must be of two arguments: obj and objName.

EX: look at the local command GetSubs of $ClassBrowser

1. When To Make Something Local

Although not something that LOOPS imposes on you, I find the following programming disipline useful when constructing browsers.

> In addition to any other considerations, make any menu item which interacts with the user be a local browser command. Make methods which run in the objects themselves do no interactive operations.

If you follow this disipline, then you will factor cleanly any LOOPS system you build into "program control component" and "user interactive component". I find such a factoring useful especially for debugging purposes.[2]

GetSubs

The GetSubs method of a browser does just what the name implies, it finds the "sub objects" of the current object. Note "sub objects" can be defined however you want to define it.

- ClassBrowser - look down the class hierarchy
- SupersBrowser - look up the class hierarchy
- MetaBrowser - look along the meta link
- etc.

---

[2] There are special considerations when you want to interact with LOOPS under a mouse process. See the INTERLISP manual, or talk to me about this if you want to someday build "interactive" operations under from a browser.

The important point is that $LatticeBrowser has methods which send a GetSubs message to self. The methods in $LatticeBrowser do a lot of the work of any browser. But they use a method that we may easily specialize for our own purposes.

EX:
```
do WHEREIS
on $ClassBrowser
and find how many of its
methods are up higher
in LatticeBrowser

try to find an example
of GetSubs being used
in some method in
$LatticeBrowser
```

Building A KeyClassBrowser

Suppose we have a very large set of LOOPS classes that is hard to view all at one time in a browser.

EX: (<leftArrow>New $ClassBrowser Show 'Object)

Now that you believe that you really do sometime have TOO many LOOPS classes for getting a gestalt view, lets decide how to build a browser to help us.

Here is one possibility to accomplish what we want.

- a browser that will only show those LOOPS classes which we have designated as "Key Classes"
- but will allow us to "expand" the view to show a normal class browser starting from any key class we choose in our KeyClassBrowser

Designation of KeyClass

The first thing we have to do is decide how we are going to mark, or keep track of key classes.

```
EX:     add a new CV to
        $Object called
        KeyClasses
```

Now make three new methods for $Class:

```
EX:     MakeKeyClass
          makes self be a
          key class by
          using PutClassValue

        UnMakeKeyClass
          removes self
          from the list
          of KeyClasses

        KeyClass?
          a predicate function
          to test if self is
          a key class
```

Now lets make some of the objects in the class hierarchy to be on the list of KeyClasses using the functions we just made.

```
EX:     pp(KeyClasses)
          this is a list i have
          provided. You are to
          make each Class on the
                                  3
          list into a key class
```

Now lets make a specialization of ClassBrowser.

---

3 Remember that sending a message to Object for example doesn't make sense. You can however send a message to the pointer $Object.

```
EX:     specialize $ClassBrowser
          to a new Class called
          $KeyClassBrowser
```

All that we have to do now to get our new browser to make an instance and display in the way we want is to specialize the method GetSubs from $ClassBrowser.

```
EX:     copy the method
        GetSubs from $ClassBrowser
        to $KeyClassBrowser

        look at the method and decide
        what we have to do next
```

GetSubs in $ClassBrowser sends a message off to obj to fetch the subs of obj.[4] Lets simply change the name of the method that goes to obj, then go back and create that method.

```
EX:     do EM on GetSubs
        in $KeyClassBrowser

        change SubClasses
        to SubKeyClasses
```

Now we have to make sure that obj will understand the method SubKeyClasses. What we want is a method that will

- fetch its subs in the class hierarchy (it can do this by issuing a SubClasses message to self)
- collect the subs that are KeyClasses

---

[4] I think they should have called it "OBI-ONE".

- for those subs that are not key classes, continue searching down the class hierarchy looking for key classes, stopping when one is found

Be sure you understand what i am saying in this logic. It may be helpful to draw pictures of the class hierarchy.

EX: DefMethod a new
method in $Class
called SubKeyClasses
to accomplish the above

test your method to
make sure it works

If you get just plain stuck in this, then look at the function jonsClass.SubKeyClasses in your loadUP.

CONGRATULATIONS!

EX: (<leftArrow>New $KeyClassBrowser Show 'Object)

Now most of the battle is done. But there are still a few things that are needed in our KeyClass browser.

EX: make any instances of
$KeyClassBrowser come up
with "Key Class Browser"
in the title bar instead
of "Class Browser"

EX: now add new
functionality to
the $KeyClassBrowser
by adding left button
functions:
MakeRegularClassBrowser
MakeSupersBrowser

MakeMetaBrowser

this involves adding
things to LeftButtonItems
and creating new methods
for $KeyClassBrowser

Question: WHY was building this new browser such a "relatively" easy operation using LOOPS?

Advanced Features For $KeyClassBrowser

In the regular notion of browsers, there is no real idea that browsers can "cooperate" with each other to present a consistent view of the LOOPS world to the user. In this section, i'll suggest ways you might further develop your $KeyClassBrowser so that you use it not only to get a gestalt view of all of the LOOPS objects, but also to "manage" the view of LOOPS objects (as shown in OTHER browsers) that is presented.

THIS SECTION IS OPTIONAL!

One basic problem with browsers is that you have to constantly run about closing old browsers. The reason is that if you have say two browsers up, and you alter one of them is some way (i mean alter the class hierarchy (eg)) in one of them, then you could be in trouble if you try to use the other one. Suppose you destroy a class in one, then try to specialize it in the other.

Solution: make your KeyClassBrowser a sort of index you use into the others. Only pull up a class browser by using the KeyClassBrowser AND keep track of all the browsers "spawned" from the key class browser. Any time you generate a NEW ClassBrowser, close the OLD one.

Further modification: Instead of just killing it, close it, cash its pointer on a stack, and build some functionality that will allow you to "pop back up" to it.

Another mild problem is doing WHEREIS using a ClassBrowser, and the object that has the method you want to find is not IN the current browser. Of course, the browser will let you know in the PromptWindow where the object is, but sometime you want more.

Solution: specialize WHEREIS so that if the object you want to highlight is not in the current browser, that a SupersBrowser will pop up AND blink the appropriate object for you.

This could go on and on. There are MANY useful things that could be done to extend the notion of cooperative browsers. As you think more about this, even if you don't implement your idea, please tell me about it.

X

Darken @ 98%

Filed on {INDIGO}<Loops>LoopsCourse>ExerciseRules.bravo
Edited: MJS November 10, 1983 5:00 PM

Editing and Debugging RuleSets

by the Loops Design Team

Daniel Bobrow, Sanjay Mittal, and Mark Stefik

Purpose of these Exercises

This set of exercises explores the programming and debugging facilities for using rules in Loops. In this session we will extend the behavior of a rule-driven "player" of the *Truckin* game. The techniques introduced in this session will provide experience in representing and debugging knowledge representation that will be needed in the later sessions.

Notation

In the dialogs below, regular font is used to show what the system types, and bold font is used to show what the user types.

Preliminaries

Go to your workstation for the course and LOGIN according to the instructions in the handout titled "How to Start Loops".

Part I. Basic Debugging Tools

This exercise is to use the auditing, tracing, and breaking facilities of the rule language to understand the rules behind the behavior of an automated player of *Truckin* called *Traveler.* A *Traveler* is a class for players that commute between *UnionHall* and *AlicesRestaurant*, stopping for gas and *WeighStations* along the way.

There is a listing of *Traveler* in your course notebook. You may want to refer to it as we continue.

Step 1. Getting Started

In the discussion which follows, we will use the term *gameMaster* to refer to that which runs the game. Depending on which version of *Truckin'* is being used, the gameMaster may be one or more programs, databases, and processes, or even more than one computer. Some variations on this will be discussed later in this class. For the purposes of this discussion, it is convenient to refer to it all as simply the gameMaster.

The first step is to set up a new *Truckin* simulation. Do this as follows:

(← $Truckin New)
(← PI BeginGame)

PI is an abbreviation for *PlayerInterface.* It's short, since you may type it a lot.

The gameMaster will then ask you in the prompt window to choose a kind of player. Use the mouse to select *Traveler* in the menu next to the prompt window, and type the name "Trav1" for the name of the driver. The gameMaster will then ask you for the kind of truck. Select *FordTruck* (or any other brand that you prefer). Then the gameMaster will ask you about gauges on *Trav1* and it is recommended that you choose DEFAULT and that you place the fuel gauge to the right of the game board.

When the gameMaster asks you for the second player, select "No" with the cursor.

The gameMaster will put up an inspector window of game parameters for you to set. These parameters control such things as the length of the game, the number of bandits, and so forth. You can set the values of the parameters by selecting a parameter and using the *PutValue* option. To see a description of a particular parameter, select the value of a parameter and select the *Properties* option. It is recommended that you set the value of the parameter gameDuration to 60 for this exercise.

When you are finished, click DONE in the menu next to the inspector. The gameMaster will then put up a bar chart for you player's cash.

Then the simulation will begin. *Trav1* will begin commuting back and forth on the board. If you depress the CTRL key, the *Traveler* will pause in its travels. The *fuel* gauge should go down at each move, and the *cashbox* bar chart will change whenever the *Traveler* spends or receives money. Every few turns, the bandits (*Bonnie* and *Clyde*) will make their moves.

The typescript window will type a rule before each move of *Trav1*. This is because the *Traveler* was compiled in the system with rule tracing turned on (compiler options T).

Step 2. Using the Rule Executive

Find the menu labeled *Interrupt* at the top center of the display. This menu should have an entry for the player *Trav1*. Clicking *Trav1* in this menu will invoke the Rule Exec window. Do this now. (It may take a few seconds for the request to be noticed -- it waits until the current player finishes its move.) The player interrupt will disappear.

The Rule Exec will prompt you with "re:". ***Notice that time keeps marching on, even though Trav1 is suspended.*** You can resume the simulation by typing "OK" to the rule exec as follows:

re: **OK**

Do this, and click *Trav1* again, to get the feel of using the Rule Exec to interrupt execution. Inside the rule exec, you can type expressions in the rule language. For example:

re: **destination**

will cause the current destination of your *Travl* to be typed. You can look at the values of other variables (such as *truck* or *stoppingPlace*) as well. You can also use compound terms as in

re: **truck:fuel**

The number you get back should correspond to the value shown by *Travl's* fuel gauge. You can also look at class variables, as in

re: **truck::MaxFuel**

You can make adjustments to your player, as in

re: **truck:cashBox←20000**

This should cause the *cashBox* bar chart to get updated immediately. (Of course, this would be cheating if done in a RuleSet, since your player should only acquire *"cash"* by selling merchandise.) Lastly, you can send messages, as in

re: **(← self Show)**

Note: Be sure to use rounded parentheses in these expressions, not brackets. In the Rule langauge, brackets are used to control precedence of operators, and parentheses are used for function calls and messages.

or, if you are not at *UnionHall*,

re: **(← truck:location:prev PP)**

This illustrates that *self* is set to the work space, and also shows off the *Show* method that *Traveler* inherits from *Player.*

Step 3. Changing Compilation Options

One way of changing compilation options is to send a *TurnOn* message to a player class. For example,

re: **(← $Traveler TurnOn 'A)**

To edit a RuleSet, enter the rule exec and create a browser for players as follows:

re: **(Browse $Player)**

Using the browser, select *Traveler* (with the middle button). Using the left mouse button, select EM* (for EditMethod), and you will be shown a list of the Methods for *Traveler*: *BuyGas*, *FindStoppingPlace*, *GoToStoppingPlace*, and *TakeTurn*. Select the *FindStoppingPlace* method. The TTYIN edit window will come up automatically -- ready for you to examine or make changes to the RuleSet.

WARNING!!! Do not use "Reprint" on a RuleSet. This will do terrible things to its readability. The rule editor that your are using is our 'StopGap" version. In the next version of the Rule Language, we will use a structural editor and a more Lisp-like notation.

Verify that the RuleSet has auditing turned on. If not, change the compiler options declaration to read as follows:

Compiler Options: A ;

When you have finished editing the RuleSet, type ↑X to exit. A pop-up menu will then appear by the cursor. The *Help* option in the menu can be used to see a description of the other options.

Most of these options are intended for use in debugging the Rule Compiler. You may find it interesting to use them to examine the LISP code currently generated by the rule compiler under different compiler options.

To compile the rules and quit, select OK.

Step 4. Asking *Why*

Leave the rule exec and let the simulation run for a while. (The audit trail is created incrementally, as the program runs, so you need to wait for it to complete another turn.) Then interrupt *Travl* and use the audit trail to answer "why" questions as follows:

re: **why stoppingPlace**

Using the rule exec in combination with auditing is a handy way of discovering which rules were responsible for particular decisions. (Perhaps this should be called *how* instead of *why*.) Try asking *why* for other variables such as destination. You can ask *why* for compound variables as in

re: **why truck:fuel**

in this case you should get back the message "Rule not known." because the *fuel* variable is not set by a RuleSet compiled with auditing turned on.

If "Why" is typed without arguments, the Rule Exec uses the previously entered expression. For example

re: **stoppingPlace**
(AlicesRestaurant 123.45)
re: **why**
--- that is, why stoppingPlace

IF (Distance destination)<=.Range1 · (RoomToParkP destination)
THEN stoppingPlace←destination;
Rule 4 from FindStoppingPlaceTravelerRules

Step 5. Suspending and Waking *Truckin'*

Another menu at the top of the screen is labeled *GameControl.* This menu has options for suspending, killing, and waking the current simulation. The gameMaster consists of several processes for the different players, the clock, and scheduling. If you enter a Lisp break, or are editing, you may want to stop the frenzy of activity on the screen so that you can work. This menu is for that purpose. Practice suspending and waking the gameMaster. (But don't *kill* the game yet!)

Step 6. Breaking on Rule Invocation

In this section we will see how to step through the execution of a RuleSet. Using the Browser and rule editor, change the compiler options for the *FindStoppingPlace* method of *Traveler* to read as follows:

Compiler Options: BT;

This indicates that the rule should "break" to the rule exec whenever a rule is tested or executed. Exit the rule editor and rule executive and let the simulation run.

As the simulation continues, you will see a rule print out in the Typescript window, and then the rule executive will pop up. By typing "OK" to the rule exec as in

re: **OK**

You may find it useful to suspend the game while you are here.

You can step through the execution of the rules. Note that the break occurs *before* executing the left or right hand sides of the rules. Step through the execution of *FindStoppingPlace* a few times to see how it works. You may want to use this feature for some subtle case of debugging particular RuleSets.

When you are tired of typing ok, try

re: **(← $FindStoppingPlaceTravelerRules Off 'BT)**

and let the *Traveler* run. You may have to type OK a few more times, until the Lisp interpreter let's go of the function with the "break code" in it.

Step 7. Debugging with Gauges

In this section we will see how to create extra gauges to help with debugging. We will begin by putting a gauge on *stoppingPlace.* Enter the User Exec by depressing CTRL-LeftShift.

The User Exec is an alternative to the Rule Exec. The User Exec expects Lisp expressions and provides the Interlisp-D environment (e.g., the history list). The Rule Exec expects rule language expressions and provides rule facilities (e.g., why questions). The User Exec can also be entered by typing UE as a command to the Rule Exec.

To create a gauge on *stoppingPlace,* type:

```
← (←New $LCD Attach $Travl 'stoppingPlace)
← OK
```

For obscure and temporary reasons, the ←New syntax doesn't currently work in the StopGap Rule language.

The *Truckin' Manual* describes more automatic ways of installing gauges on the instance variables of a player.

Step 8. Listing RuleSets

To get a hardcopy listing of the RuleSets associated with a class, use the function *ListRuleSets.* The listings will appear on the local printer. Get one of the course instructors to show you where it is. To make a listing, type the following to the user exec.

```
← (ListRuleSets 'Traveler)
```

Part II. Creating A New Player

The purpose of this exercise is to practice making a new kind of *Player* called a *BigMac,* which is a revised version of the *Traveler.* A *BigMac* is a class of player that commutes between two of the *AlicesRestaurants* in the simulation. *A BigMac* (a hungry driver of a Mac truck) will presumably eat a lot, visiting *UnionHall* only when it runs out of money and gets towed there. A *BigMac* always drives a *MacTruck.*

Step 1. Setting Up

You may want to restart the game before continuing with this exercise. Kill the game using the GameControl menu. Using the mouse, close a few of the windows at the top of the screen until you find the Loops Logo (Saturn). Depress the left mouse button, and a menu should pop up. Select the *SetUpScreen* option to restore the screen to its original state. You may want to create a new browser for players as:

```
← (Browse $Player)
```

To create a new player that is a specialization of *Traveler* use the *Specialize* option in the Browser. When you are prompted for a class name in the prompt window, type:

Class Name: **BigMac**

The Browser will now indicate a new class for *BigMac.*

You may need to shape the browser window to see *BigMac.*

Step 2. Replacing a Method

To insure that *BigMac* always drives a *MacTruck*, we need to replace its *SelectTruck* method. Initially, the method for *SelectTruck* is inherited from *Player*, and prompts for a truck.

To replace this method use the EM! option in the Browser.

Either approach will put you into a Lisp editor on the Lisp function that implements the method. Edit the function so that it just returns *'MacTruck*.

Step 3. Adding an Instance Variable to the WorkSpace

To change the behavior of *BigMac*, it will first be necessary to add an instance variable to record the *nextDestination*. Edit *BigMac* with a Lisp Editor of your choice. (Hint: You may access it through the *Edit* option of the Player Browser.) When you have added the instance variable, that portion of *BigMac's* definition should look as follows:

(InstanceVariables (nextDestination NIL doc (* Next destination. A different AlicesRestaurant.)))

You may want to add some documentation to *BigMac* itself. If you do, the relevant portion of *BigMac's* definition should look approximately like this:

(MetaClass PlayerMeta
doc (* A Player that commutes between AlicesRestaurants, eating hamburgers.)
Edited: (edited: *MyName* "21-Feb-83: 15:31"))

BigMac inherits other instance variables from *Traveler*, but they don't show in the source because they are not introduced at this level of the inheritance lattice. To see them, you can prettyPrint a summary of it through the browser (using the *PrintSummary* option).

Step 4. Specializing the Rules of *Traveler*

In this section, you should go back and edit the rules for the *TakeTurn* method and modify them for a *BigMac* player. In debugging your rules, the techniques introduced in Part 1 for auditing and breaking RuleSets and adding gauges, will be of use.

Hints:

1. The course handout "*Truckin'* Query Functions" describes a set of functions for accessing information in the world of the *Truckin'* simulation. These functions will be discussed later in the course, but you may find it helpful to browse this document when you are trying to understand the *Traveler* rules.

2. You will probably want to replace the *TakeTurn* method of *Traveler* with one specialized for *BigMac*. You can use the EM! option in a player browser to do this.

3. *BigMac* should initialize his *destination* and *nextDestination* on the first call. In the method for *TakeTurn*, the following rule may be a useful substitution for some of the existing rules:

```
(* On first call, initialize destination and nextDestination.)
IF ~destination
THEN alices←(RoadStops 'AlicesRestaurant)
     destination←(CAR alices) direction←(DirectionOf destination)
     nextdestination←(CADR alices);
```

This rule assumes that *alices* is defined as a temporary variable of the rule set.

4. In addition, a new rule like the following may be appropriate in the method for *GoToStoppingPlace*:

```
(* Switch destination and nextDestination when you arrive.)
IF truck:location=destination
THEN temp←destination
     destination←nextDestination
     nextDestination←temp
     direction←(DirectionOf destination);
```

5. If you have trouble with the behavior of *BigMac*, use the auditing, breaking, and gauging facilities you have learned about to understand the behavior.

Step 4. Saving the Rules on a File

To save your RuleSet on a file, type the following to a User Exec or at Top-level lisp.

(FILES?)

Lisp will ask you whether to save various instances and functions. Type BIGMAC for all of the things that you want to save. Type] (a right square bracket) for all of the things that you don't want to save, such as *Bonnie, Clyde,* and other things not related to your file. You can type LINEFEED (LF key) to mean *same as previous.*

Then make a file containing your *BigMac* player as follows:

MAKEFILE(BIGMAC)

This file can later be retrieved by typing

LOAD(BIGMAC).

[Optional] Part III. How Auditing Works

This section is intended for those who finish their *BigMac* player early, and would like to learn more about how the auditing works in the rule language. This section is a tour of the auditing facility.

To see how an audit trail works interrupt your player with ↑F. Use the left mouse button to

select the top item of the "trace back" menu to the left of the rule exec window. This will create an inspector for the workSpace.

If there is no trace back window, type **(← self Inspect)** instead.

Select the value of the destination variable with the left mouse button. (It will turn black). Depress the middle mouse button and a menu should pop up. Select the *Properties* option. This will spread out the properties of destination. The interesting part of this is the *reason* property of destination. The value of the *reason* property should be an instance of a *StandardAuditRecord.* If you inspect that record, you can "inspect" all the way to the *rule object* which prints out when you type the *why* question.

The Lisp code generated for the RuleSet must not only save values, but must also create the audit records and link them to the *reason* properties when it executes. To see this auditing code, you may want to invoke the rule editor on the *FindStoppingPlace* method of *Traveler,* exit the editor with ↑X, and select the EF menu option to examine the Lisp code. You should be able to find the code that makes the audit trail. Say DE to the Lisp TTY editor if you want DEDIT. After looking at the code, exit the editors.

The next step is to look at the Audit Class declaration for the RuleSet. Select the *EditAllDecls* option in the Rule Compiler menu. This will put you in the editor again, except that several additional "default" declarations will now be made explicit. In particular, you can now see the declaration for the audit class. Exit the rule editor.

To see where the meta-assignment statement for saving the rule in the audit record came from, edit the class *StandardAuditRecord.*

A *StandardAuditRecord* saves only a pointer to a rule. The specification of what to save in an audit record is made by meta-assignment statements - either in the class for the audit record, or in the RuleSet. The class for the audit record must have instance variables for all of the values to be saved. This facility can be used for experimenting with belief revision systems. See the Rules Manual for details. This material is beyond the scope of the 3-day Loops course.

Filed on {INDIGO}<Loops>LoopsCourse>ExerciseCompetition.bravo
Edited: MJS August 3, 1983 7:37 PM

Knowledge Programming

by the Loops Design Team

Daniel Bobrow, Sanjay Mittal, and Mark Stefik

Purpose of these Exercises

These exercises provide experience in building a small knowledge system in Loops: a *Truckin'* player. This *Truckin'* player will be your "entry" in the knowledge competition at the end of the course. The exercises will help you to prepare your player, and tell you how to enter the knowledge competition, and what to expect.

Notation

In the dialogs below, regular font is used to show what the system types, and bold font is used to show what the user types.

Preliminaries

Go to your workstation for the course and LOGIN according to the instructions in the handout titled "How to Start Loops".

Part I. Specializing an Existing Player

This exercise is to augment the knowledge of a modest knowledge system incrementally. For this purpose, we have provided a sample *Truckin'* player as follows:

Peddler (50 rules): - a player created for the second Loops course with a structure that makes it convenient to create *evaluation functions* that prioritize player decisions.

Find the listing of *Peddler* in your course notebook. You may want to refer to it as we continue.

Step 1. Creating A Player

Create a player that is a specialization of *Peddler.* As in the *BigMac* exercises, either a browser to create your specialized player.

You will need to name your player. Here are some names that have been used before:

HappyHauler	PeterLorry
Routier	Maverick
SafeSam	
SpendThrift	
Toyota	
TravelingSalesman	

When you pick a name, register it with us and we will be alert for name conflicts before the knowledge competition.

Step 2. Hints About *Peddler*

Peddler is a reasonably good player, but its knowledge base lacks some fine points. Here are some known weaknesses which you may choose to remedy. (You may discover other weak spots.)

1. *Peddler* uses "rating RuleSets" to pick *Producers*, *Consumers*, and stopping places near *AlicesRestaurant*. But it shows little flexibility in its selection of which *AlicesRestaurant* to go to at the end, or which *GasStations* to visit. You may want to extend the rating idea to these other decisions.

2. *Peddler* looks for low prices when it picks a *Producer*, rather than computing its probable profit (by looking ahead for a possible *Consumer*). Late in the game, it may even buy goods for which there are no attractive *Consumers*.

3. *Peddler's* gasoline logic is poorly organized. When *Peddler* is low on gas, it can completely forget about other things, like stopping at *WeighStations* or getting to *AlicesRestaurant*. If *Peddler* gets in a tight loop between a *Producer* and a *Consumer*, and there is no gas station in-between, it will go back and forth content on making a tidy profit but forgetting to go outside that range for gas.

4. *Peddler* avoids *PerishableCommodities* and *FragileCommodities*, because it lacks knowledge for dealing with them. It needs to know about *RoughRoads*, and that perished commodities can spoil, and that damaged merchandise must be taken to the dump to avoid fines. There is a lot of money to be made on such merchandise.

5. *Peddler's* rating schemes using a numeric method for combining factors. You may choose to use a symbolic approach to combine the factors more rationally.

Step 3. Strategy for Preparing Your Knowledge System

Here are some suggestions for building and debugging your player:

Keep it simple!

You don't have a great deal of time (6-8 hours) to build your system, and you will be learning Loops (and perhaps coping with bugs) at the same time. In the previous Loops

courses, the most successful entrees were created by people who fixed bugs and made small improvements to the players rather than starting a complete re-design.

Ask Why!

During the first Loops course, people sometimes used only conventional programming techniques and struggled to find bugs that the auditing facilities could easily have pinpointed for them. Part of what makes rules special are the special facilities for auditing rules. Use this! It will help you pinpoint shortcomings in your rules! Also use the gauges and breaking and tracing facilities. These tools for understanding your player really help.

Save your player frequently!

Computer systems are subject to occasional crashes. Think conservatively. Save your files every 15 - 30 minutes, and make listings every hour or two. Be sure to type **(FILES?)** before doing your MAKEFILE. (Or use the lisp function CLEANUP).

Try a crowded board!

Some important phenomena only show up during a crowded game, as will be the case during the knowledge competition. By analogy with ecological systems, the stiffest competition for a player is often itself. Identical players, like members of the same species, compete for the same ecological niche. Be sure to exercise your player on such a board before the competition. A good way to test your player is to compete it against several copies of itself and several *Peddlers*.

Try different trucks!

Trucks have different storage capacities, speeds, ranges, etc. The performance of your rules is likely to be different for the different classes of trucks. Use a browser to determine the characteristics of the different trucks and try your player with different trucks so that you can understand how your rules interact with the truck characteristics.

Lisp is legitimate!

If you find that the rule language is too confining for you, you can always express some of the knowledge in a Lisp function, called directly or installed as a method on your player or some other object. This is completely "legitimate" in *Truckin'*, and may be appropriate since the *StopGap* version of the Rule Language has a somewhat awkward interface. You may also find it useful to define some new classes of Loops objects.

Step 4. Technical Note -- Creating, Copying, and Installing RuleSets

In working on your player, you may want to define some new RuleSets as methods. To define a RuleSet as a method, use the DefRSM suboption under DM* in the browser.

You may want to create a new version of one of the Rating RuleSets. One way to do this is to use the CopyRules method of RuleSets. For example, to create a rating RuleSet of GasStations for a player named *RoadRunner*, you may proceed as follows:

← (← $PeddlerRateConsumers CopyRules RoadRunnerRateGasStations)

We recommend that you use your player name as a prefix on your RuleSetNames, to avoid name conflicts during the knowledge competition.

Step 5. Save Your Player Periodically

After naming your player, inform the file system by typing:

← FILES?)

to the Lisp Executive. When it prompts you for a file for the class of your player, use the player name for the file name. When you have finished with this, create a file for your player:

←MAKEFILE(*MyPlayerName*)

All computers and systems are subject to failure. Be conservative. Save your player every half hour or so, to avoid losing your work.

Part II. Entering the Knowledge Competition

The knowledge competition is the usually exciting finale to our course. Here are the things you need to do to participate.

9:15 am ---

1. *Register and file your player.* Your player should be ready, loaded in your machine, and backed-up in the file system. You should see the course instructors to register your player name, and the name of a driver for it. (This is to insure that the names are unique and short enough for the competition.) Make sure that your player loads correctly from a file. You should also make sure that you have saved everything your player needs (using *FILES?*), and that all rule tracing and breaking has been turned off.

9:30 am

2. *Competition debriefing.* Before starting the competition, we will spend a few minutes talking about the idea of a knowledge competition, and will ask each group to spend 1-2 minutes talking about their player. One person in your group should be ready to tell us the main ideas you are trying.

10:00 am

4. *Starting the competition.* We will then bring the distributed game master on line and tell you the network address of the *PostMaster.*

Note: The game clock will start so that the competition starts *automatically* at 10:15. If you fail to complete the following instructions on time, you can still enter your player, but the competition will have already started and you will be behind.

5. *Go to your workstation.* Type:

(SlaveTruckin)

This will start the "slave" to the game master, and will also run the competition with the "simulation display" turned off. The game master will then ask you the following questions:

Name of your machine: ***JonesMachine***
What is the address of the PostMaster: ***12365***

For the machine name, you should append "Machine" to your last name. For example, if your name is "Jones", then you should enter "JonesMachine" as above. The address of the Postmaster will be the number that we gave you in step 3.

You will then be prompted to start your players in the usual way. Use your registered *driver* name. The promptwindow will begin to print messages about the competition starting soon.

6. It's a lot more fun to watch the game with everyone else, so you can hear the cheers and groans and gossip as everybody watches and comments on the performance and luck of the road!

Good Luck!

SYSTEMS BUILT ON TOP OF LOOPS

Thus far in the class, you have been doing exercises to build your INTERLISP/LOOPS expertise. Today, we'll be looking at several systems which have been built on top of LOOPS.

The first system we'll look at is the SIS tool, a tool constructed to allow some of the generic ideas of browsers to be applied and extended. Then we'll go on to look briefly at a medical diagnosis system called MDX/MYCIN, a system built following the OSU-AI paradigm of diagnostic expert system design, and which is implemented on top of both LOOPS and the SIS. Finally, we'll end the course by looking at some of the capabilities of CSRL, a language which has been developed to allow the expression of diagnostic systems in a straightforward way.

THE SIS TOOL

The Structured Instance System is a LOOPS tool which has been designed to extend the notions of "environment control" to the realm of named instances. One of the important design goals of the system defined ClassBrowser is that the browser should be an aid to the LOOPS user for organizing and dealing with the world of LOOPS classes she is developing. The SIS applies that same top level idea to LOOPS named instances.

The SIS has its root in the distinguished LOOPS class called StructuredInstanceObject. At the class level it includes a tool called the StructuredInstanceObjectClassBrowser, a specialization of the normal ClassBrowser. Going to the instance level, the SIS includes a browsing tool called the InstanceBrowser that allows manipulation of the relations between the instances of a subclass of StructuredInstanceObject. Finally, the SIS includes another graphical tool for showing the current setting for a user chosen IV of a collection of instances, the ValueLattice.

StructuredInstanceObjects

The SIS includes acts on any LOOPS class which has StructuredInstanceObject on its supers list. There are two properties that such LOOPS classes have that are important to understand.

- First, any instantiation of such a class is a named LOOPS object that can be viewed (along any number of user defined relations) in relation to other members of the same class. These relationships that may exist between the instances of the StructuredInstanceObject are called "InstanceLinks".

- Second, there is a special kind of IV that can be defined for any SturcturedInstanceObject: the "UserIV". The notion here is the SIS system has been constructed to allow knowledge engineers to build systems for potential end users. At the end user level, many IVs are of no interest at all - they may exist in a system only in terms of "making the system work", but the end user may not need to know of their existence. To hide such IVs from the end user, the SIS distinguishes a special sort of IV (the UserIV) for express interactions that may be necessary with the end user.

The StructuredInstanceObjectClassBrowser

The starting point for jumping into a number of SIS defined objects is the StructuredInstanceObjectClassBrowser.

EX: do
    ssi]

The function ssi brings up the a normal class browser starting from the root SturcutredInstanceMeta, and a StructuredInstanceObjectClassBrowser starting from the root StructuredInstanceObject.

Within the SIS system, there are many new menu items that you find defined at each of the three level of browser type tools. Whenever you see a menu item called "StardardFunctions*" you can expect to find a submenu of more normally defined browser operations.

EX: mouse the object $organization
with the left button, then
mouse the StandardFunctions*
with the middle button and
look at the items in the submenu

do the same sort of operation
with the middle mouse menu item
StandardFunctions*

In addition to the standard functions, there are a number of new operations that you will find too. The middle mouse sub menu for example, includes extensive operations for storing and retrieving objects from secondary storage.

EX: conn (FLOPPY)
dir()

now middle mouse $organization
and select Save* with the middle button
the select the item SaveClassAndInstances

now do dir() again

now middle mouse $jonsTopLevel,
select LoadSubObject* with the
middle button, and then select
LoadSubObject

this will bring a menu from which
you could select a file to load that
would restore any object inferior to
jonsTopLevel that has been stored

mouse any button outside the selection
menu to tell the system you don't want to
load any file right now

On the left selection menu for the StructuredInstanceObjectClassBrowser, you'll find a item BrowseInstances that will bring up an InstanceBrowser on the current instances of the object buttoned.

Before going on to look at the IB, you may want to play a little bit with some of the other menu items for the StructuredInstanceObjectClassBrowser, but do NOT do CompileMethods (from the middle button menu).

The InstanceBrowser

The heart of the SIS facilities is the InstanceBrowser.

EX: select the left button item
BrowseInstances on $organization

when it asks for your starting list
enter]

when it asks for the relation you want
to show select (partOfSuper partOfSub)

place the IB in the normal way

You now have up an InstanceBrowser for the instances of $organization as viewed along a certain relation. A design goal for the SIS was to allow the user many facilities for graphically editing the relation between his instances. These graphical facilities are located on the middle menu items for the IB.

EX: mouse $xerox with the middle button
select MoveInstance

(at this point the IB goes into
"gather mode" to gather up
all the supers you want for
the new location for $xerox.)

follow the instructions you will see
in the PromptWindow to move $xerox
to have a super of $battelle and
no subs

select MoveInstance again to
put $xerox back where it was
to start with.

EX: experiment with the other
middle button items in the IB

The IB can be brought up for any define relation for a StructuredInstanceObject.

EX: go back to the class level
and bring up an IB for
the instances of $organization
along the relation
(typeOfSuper typeOfSub)

place the new IB next to the
old one

In addition to the facilities that you have seen so far, the SIS system was a vehicle for some initial experimentation into the notion of "co-operating browsers". At one level, there is cooperation between IBs.

EX: select $xsis with
with the left button
in the IB showing the
relation (typeOfSuper typeOfSub)

select TotallyKillInstance
from the menu

note what happens to the
other IB you have up

EX: select $battelle
with the left button
in the IB showing
(partOfSuper partOfSub)

select SpawnNewBrowser

note that the old one
(the one from which you
spawned) disappears

The Value Lattice

In addition to being able to graphically edit the relations that exist between instances, the SIS also provides a way to access (in this case either display or change) the values of a selected UserIV in all the instances. Suppose for example that you want to have a UserIV for $organization call projects, and that you'd like to store in that IV the names of ongoing project at each level in an organization.

EX: select the title menu item
SetIVofInterest in the IB
showing (partOfSuper partOfSub)
for $organization

select the IV projects

select the title menu item
MakeValueLattice in the same IB

select the ValueOnly mode of
presentation

place the ValueLattice

The structure of the ValueLattice is exactly the same as the IB from which it was invoked, but the print characters for each node show the current setting for the IVofInterest that is currently set.

The action of the mouse is very simple for the ValueLattice.

- the left button is just a display device that will highlight a node you select, and highlight the corresponding node in the parent IB. This is to help not get lost in the ValueLattice.

EX: select any node in the
ValueLattice with the
left mouse button, and
note the action in the
parent IB.

- The middle mouse button is for resetting the IVofInterest for a selected node.

EX: select the node for
$columbus-ai in the
ValueLattice with the
middle button

when asked, type in
(DARPA)

note the effect on the
ValueLattice

close the parent IB
and see what happens
(what would you
WANT to happen?)

Discussion

The SIS is a system that is designed to extend programming environment offered by LOOPS to the realm of named, structured instances of LOOPS objects. In so doing, the idea of cooperating browsers was introduced. As was hinted at in yesterday's exercises, this notion can be very powerful if viewed in a light of trying to help the user manage his thinking. If you have suggestions, comments, or criticisms of the few parts of the SIS that you've seen, don't hesitate to make them known.

MDX/MYCIN

MDX/MYCIN is a medical diagnosis system operating in a subdomain of the MYCIN system: bacterial meningitis. At the end of the notebook, you'll find a paper describing the system and the motivation behind construcing it. For purposes of this exercise, what we will do is

- bring the system up and run it on a typical case
- imagine that we are participating in a knowledge engineering seesion whose purpose is to debug a portion of MDX/MYCIN

EX: button mmConcept
with the middle mouse
button and select
BringUpCommandMenu

The command menu is a permanent menu (as opposed to a pop-up menu such as you have seen before) that allows the user to just ask for a desired function.

EX: button Diagnose
in the command menu
that you have up for
mmConcept

select the case
mycinCase232
for running

The system will now bring up an IB, and proceed to diagnosis the case following an MDX approach. Note how clear cut it is to follow the action of what the system is doing at any one time.

When the job is done, the IB will show in inverted video the MDX/MYCIN specialists which have been established.

MDX/MYCIN Debugging Session

At this point we can imagine that you are a knowledge engineer. At your side is your resident medical expert. You have just run case 232 and your expert is requesting to see the establishing numbers for each specialist.

EX: set the IVofInterest
for the IB that is
shown to MostRecentResult

bring up a ValueLattice

Now the medical expert points out that the establishing value say for diploPneumonia is too low. Instead of 1, it should have been 2.

Now you as the knowledge engineer must (with the help of the medical expert) figure out what part of the domain knowledge is incorrect.

The first thing to do is to look at the individual knowledge groups inside the specialist diploPneumonia.

EX: left mouse $diploPneumonia
middle mouse TruthTableFunctions*
select BrowseTruthTables

Now you are looking at the individual TruthTables (ie knowledge groups) that reside in the diploPneumonia specialist.

EX: set the IVofInterest
for the IB you have up
on the knowledge groups
to MostRecentResult

bring up a ValueLattice
for the knowledge group
IB

Note we are now playing the same game that we did before for the specialist level except one level down; ie, at the knowledge group level.

Now your resident medical expert says (from looking at the establishing results for the individual knowledge groups) that the problem is in the diploPneumoia.headInjury knowledge group.

At this point you have pinpointed a potential problem and you can now revise the domain knowledge for that ONE knowledge group that you have found to be in error.

EX: left mouse the
diploPneumonia specialist

select TruthTableFunctions*
with the middle mouse button

select EditTruthTable

select diploPneumonia.headInjury

Now you will find the truth table you have fingered as the culprit in a DEdit window, ready for you to alter.

Imagine you have done the necessary modifications, and exit DEdit.

Now you have made the necessary change to your domain knowledge. Consider what portions of the system must now be re-tested in order to verify that the change is acting as you want it to.

Comments

There are several points to be made about the expert system you have just seen. First, the system is easily extensible. (Why?)

Second, the system is easy to debug because knowledge is factored in a relatively clean way on two levels.

Third, the principle of NAMING the knowledge groups and the specialists themselves allows easy of use by the medical expert.

USING CSRL IN INTERLISP-D

CSRL is a language for implementing diagnostic expert systems. This chapter emphasizes the details of loading and interacting with CSRL on a Xerox 1108 (hereafter called a Dandelion), rather than describing the language and motivating it. It assumes that the you, the reader, have some familiarity with using a Dandelion and the LOOPS language.

Conventions in this document: Since the printer does not have a true backarrow character, a "←" is used instead in this document. Examples showing user interaction indicate what the user enters by underlining it.

1. Loading CSRL

Before you can load CSRL, your Dandelion must be in Interlisp running LOOPS. A fresh version of LOOPS is recommended. CSRL with the Auto-Mech expert system takes up about 700 pages.

To load CSRL, obtain the relevant floppy from Tom Bylander (his office is Caldwell 408), insert the floppy in into the Dandelion, and type in:

←LOAD({FLOPPY}LOADCSRL)

Before any files are loaded, you will be asked 2 questions. The first question finds out if you want to load the source code for CSRL, and the second asks you if you want to load the Auto-Mech expert system which is written in CSRL. If you are doing this for the first time, answer the first question "n" and the second question "y". The following sections are written in the context that you have loaded the Auto-Mech system.

2. The CSRL Browser

The CSRL browser allows you examine, modify, and run a CSRL expert system.

Exercise 1: Creating a CSRL Browser

To get a CSRL browser for Auto-Mech, enter:

```
←(←New $CSRLBrowser Show '(Auto-Mech Specialist))
```

> The cursor will prompt you (by changing to a box shape) to place the browser on the display. You will probably need to Recompute the browser (using the title menu) in order to display the whole lattice. Warning: Do not select the ShowValues item in the title menu until you have run a case.

The lattice that is displayed shows you the "specialist" structure of the expert system. FuelSystem, for example, is a "subspecialist" of Auto-Mech and a "superspecialist" of Vacuum, Delivery, and other specialists. Each specialist of Auto-Mech is associated with a hypothesis about the state of an automobile engine, e.g., FuelSystem is associated with the hypothesis that something is wrong with the fuel system (the subsystem that delivers a mixture of fuel and air to the cylinders of the engine). The hypotheses of FuelSystem's subspecialists are (as you might expect) sub-hypotheses of FuelSystem's hypothesis.

The remainder of this section briefly describes the commands available to you on the browser. Following sections describe more details about using the browser and creating your own expert system.

2.1. Left Button Commands

Print	Prints a specialist or some part of it on the PPDefault window. Selecting this item brings up the following menu.	
	Specialist	Prints the whole specialist.
	Declarations	Prints the declarations of the specialist. These indicate its super- and subspecialists.
	Knowledge Group	Prints a knowledge group of the specialist. Another submenu is displayed for selecting which knowledge group to print. Knowledge groups correspond to

major decisions to be made by the specialist.

Message — Prints a procedure that responds to a particular CSRL message (not the same as a LOOPS message). Another submenu is displayed for selecting the procedure to print. The Specialist class contains the default procedures for messages.

Doc — Retrieve documentation on the specialist or some part of it. It has the same submenu structure as the Print command. This command also lets you retrieve documentation for parts which are inherited by the specialist. Currently all the specialists in Auto-Mech are subclasses of the Specialist class, which contains default information for all specialists.

WhereIs — Find out where a part of the specialist is inherited from.

Unread — Unread the specialist into the current display stream.

Diagnose — Do diagnosis starting with this specialist. Submenus for selecting what case to diagnose, and what message to send are displayed. This command is covered in more detail in the next section.

2.2. Middle Button Commands

Add — Add a new item to the specialist. Brings up the following submenu.

Specialist — Lets you edit the specialist. This is no different from using the Edit command to edit the specialist.

Declarations — Lets you edit the declarations. This is no different from using the Edit command to edit the declarations.

Knowledge Group — Add a knowledge group to the specialist. You are prompted to

type in the name of the knowledge group in the prompt window and then you are sent to the editor. If you type in the name of a previously defined knowledge group, that is what you will edit.

Message — Add a message to the specialist. A submenu is displayed for selected what message you want to add (what messages can be sent is predefined), and then you are sent to the editor. If you select a previously defined message, that is what you will edit.

BoxNode — Boxes the node. This is useful (in fact necessary) to use the Copy command.

Copy — Allows you to copy a knowledge group or message from the specialist that was selected to the specialist which is currently boxed. Submenus let you select the knowledge group or message of your pleasure.

Delete — Allows you to delete a knowledge group, message, or the specialist itself if it is a tip specialist, i.e., a specialist with no subspecialists. Submenus let you select the knowledge group or message. An additional submenu of one item is displayed to ok the deletion. Clicking the mouse outside the menu cancels the deletion. Warning: Undoing a knowledge group or message deletion is not possible. To undo a specialist deletion you need to add the specialist back as a subspecialist of the appropriate specialist, and edit the declarations of the specialist.

Edit — Allows you to edit the specialist or some previously defined part. Its submenu is the same as the Add command. The difference is that for the Knowledge Group and Message subcommands, you select what you want to edit from another submenu.

Rename — Allows you to rename a knowledge group or the specialist itself.

2.3. Title Menu Commands

ShowValues — Brings up a sub-browser which shows the confidence values for the specialist in the current case. See next section for more details.

Recompute, AddRoot, DeleteRoot, SaveInIT — Same as for class browsers. Recompute is not automatically called when a specialist's declarations are changed, so you will need to Recompute the browser "manually".

2.4. Shift Commands

If the left shift key is depressed when you click the **left** mouse button, a summary of the specialist is printed in the PPDefault window. This will be referred to as the Print Summary command.

If the left shift key is depressed when you click the **middle** mouse button, you will be sent to the editor to edit the specialist. You can use the Edit command to do the same thing.

Exercise 2: Operating the Browser

If you haven't brought up the browser for Auto-Mech yet, it is suggested that you do so now.
Try using the Print, Print Summary, and Doc commands on the specialists.
Use the Edit command to bring up a specialist or a part of one in the editor. If you make no changes before you exit the editor, no processing will be done. If you make changes, and exit the editor, then if an error is discovered, you will go back to the editor; otherwise the changes that you have made will take effect (and are undoable). Selecting the Stop item from the Exit submenu (use the middle button to display the submenu) will let you exit the editor without making any changes. Use the Print command to confirm this.
Use the Copy command to copy the summary knowledge group of Specialist to Choke. You will need to use the BoxNode command on Choke first. Confirm the copy with the Print command. Use the Delete command to remove the summary knowledge group from Choke.
Rename the Choke specialist to UsedToBeChoke. Rename it back to Choke.

3. Running a Case

Exercise 3: Running Auto-Mech

Left button the Auto-Mech specialist in the browser, select the Diagnose command, "new case?" and "Establish-refine". Auto-Mech will now present questions for you to answer in the TTY window. Also, some information about what the specialists are doing is displayed. The specialist that is currently executing is boxed in the browser. For anyone who doesn't want to think of answers to the questions, use the following:

```
Do you have problems starting your car? n

Does the car stall? n

Does the car run rough? y

Does the problem occur while idling? n

Does the problem occur on loading? y

Does the problem occur while the engine is both
        hot and cold? y

Have you eliminated ignition as a possible cause
        of the problem? y

Is any fuel delivered to the carburetor? y

Have you been getting bad gas mileage? n

Are there any cracked, punctured or loose vacuum hoses? u

Can you hear hissing while the engine is running? n

Are the vacuum hoses old? y

Can you see cracks in the carburetor gasket? y

Has FuelSystem completed diagnosis? n

Is the air filter old? n

Has FuelSystem completed diagnosis? n

Have you tried a higher grade of gas? y
```

3.1. What Happens When a Case is Run

What you just did was to send an Establish-refine message to the Auto-Mech specialist in the context of a new case. Upon receiving this message, the Auto-Mech specialist sent itself an Establish message, some questions were asked, and then the specialist sent itself a Refine message, which then called Auto-Mech's subspecialist, FuelSystem, with Establish and Refine messages. As this "establish-refine" process was applied further down the hierarchy, some of the specialists were sent Establish messages, but not Refine messages. This happened either because the specialist had no subspecialists or because the specialist did not have a high enough confidence value.

In CSRL, a seven-point confidence value scale is used. For convenience, we use the integers from -3 to +3, which can be loosely interpreted as:

3	Hypothesis is confirmed.
2	Hypothesis is very likely.
1	Hypothesis is mildly likely.
0	Evidence for the hypothesis is inconclusive.
-1	Hypothesis is mildly unlikely.
-2	Hypothesis is very unlikely.
-3	Hypothesis is disconfirmed.

If the confidence value is 2 or 3, the specialist is said to be established. For -2 or -3, the specialist is said to be rejected. Otherwise the specialist is suspended.

A sub-browser is available to display the confidence values in a graphical manner by selecting the ShowValues item from the title menu.

Exercise 4: Creating a Confidence Value Browser

> Select the ShowValues items in the Auto-Mech browser. You will be prompted to display the browser window on the screen. Try out the menus in this browser. Note: The confidence value browser won't work unless the top node, Auto-Mech, has a confidence value in the current case.

Since you can use the main browser to send any message to any

specialist, you are able to "explore" any diagnosis that was not originally done.

Exercise 5: Diagnosing From Inside the Hierarchy

> Left button one the specialists that has a confidence value but was not refined, and select Diagnose, "current case?", and Refine from the menus. After this processing is finished, go to the confidence value browser, and select Recompute from the title menu.

3.2. The Current Case

CSRL remembers what the current case is by setting the variable currentCase. Cases correspond to instances of the class CSRLCase, which has methods for implementing a simple question-asking facility. CSRLCase also keeps track of old cases. Presently there is no facility (but one is planned) for renaming or saving cases.

3.3. Tracing CSRL

The trace information that the diagnose provides you is done using the functions TraceCSRL and UntraceCSRL. Both of them are NLambda-NoSpread functions, and take arguments corresponding to the following forms:

Specialist	Trace all specialists (TraceCSRL Specialist)
‹specialist›	Trace the specialist (TraceCSRL Mixture)
Message	Trace all messages (TraceCSRL Message)
(Message to ‹specialist›)	Trace messages to this specialist (TraceCSRL (Message to ValveOpen))
(Message from ‹specialist›)	Trace messages sent from this specialist (TraceCSRL (Message from FuelSystem))
Rule	Trace all rules. This will only trace the rules of specialists that are currently traced. (TraceCSRL Rule)

(Rule of <specialist>)
Trace rules in the specialist. This will only trace the rules if the specialist is traced.
(TraceCSRL (Rule of Carburetor))

(Rule of <name> kg)
Trace rules of knowledge groups with this name
(TraceCSRL (Rule of summary kg))

For example, the present tracing level was done by (TraceCSRL Specialist Message). UntraceCSRL removes a previous TraceCSRL.

Exercise 6: Tracing Rules in Selected Specialists

Trace the rules of each of FuelSystem's immediate subspecialists, and then diagnose a new case.

4. Making Your Own Expert System

This section gives a brief incomplete account of how to build a simple expert system in CSRL. In particular, it contains no information on how to change the default Refine procedure. It also depends on your ability to figure out how the Auto-Mech system. The section also includes exercises on building part of a expert system for diagnosing problems in a house (or apartment or mansion if you like).

4.1. Making the First Specialist

To make a specialist, use the function Specialist, which has the form:

```
(Specialist <name> <comment>
    (declare <declaration1> <declaration2> ...)
    (kgs <kg1> <kg2> ...)
    (messages <message1> <message2> ...))
```

The declare, kgs, and messages sections are optional, as well as the comment.

Exercise 7: Creating the House Specialist

To make an specialist called House do:

```
←(Specialist House (* House is a new specialist))
```

You should also create a browser to facilitate future additions.

```
←(←New $CSRLBrowser Show '(House Specialist))
```

4.2. Adding Subspecialists

To add subspecialists to the expert system, it is easiest to edit the declarations of the existing specialists.

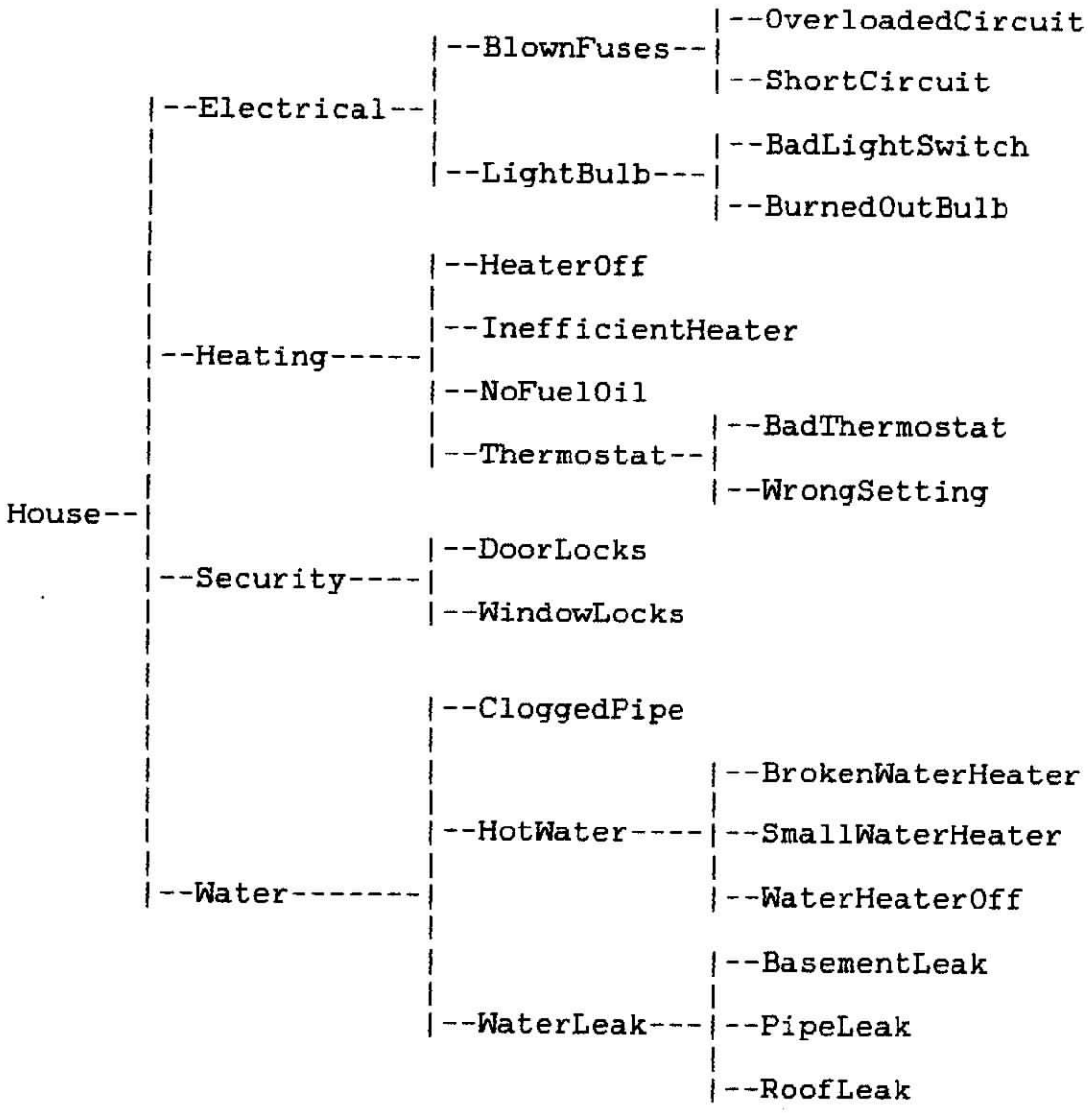
Exercise 8: Adding Subspecialists to House

Use the Edit command (or left shift/middle button) to edit the House specialist. Add declarations after the comment (change the comment if you wish) which look like:

```
(declare
    (subspecialists Electrical Heating
                    Security Water))
```

Be sure that you have spelled "subspecialists" correctly. Now Exit the editor and Recompute the browser. The specialists should have been automatically created.

Now add more specialists so that the browser looks something like:



Of course, feel free to restructure the specialists any way you like. You should be forewarned, though, that the "perfect" hierarchy does not exist. Below we will only be concerned with embedding knowledge within the Heating subhierarchy of this structure, so it is not necessary to create the whole structure.

4.3. Adding Knowledge to the Specialists

In CSRL, most of the knowledge of a specialist takes the form of knowledge groups. A knowledge group (hereafter abbreviated to kg) maps a list of expressions to a confidence value (or some other measure). This can be the confidence value of the specialist or perhaps the confidence value in some intermediate hypothesis. For example, there might be a kg in the Security specialist called badNeighborhood, which could measure the likelihood that you are in a bad neighborhood or measure of the "badness" of the area. This value and values of other kgs (which measure other facets of security) could be combined by a "summary" kg to arrive at a confidence value for the specialist.

One type of kg is called a Table kg. Its form is:

```
(⟨name⟩ Table ⟨comment⟩
     (match ⟨expression⟩ ⟨expression⟩ ...
       with (if ⟨test⟩ ⟨test⟩ ...
                then ⟨value⟩
              elseif ⟨test⟩ ⟨test⟩ ...
                then ⟨value⟩
              else ⟨value⟩)))
```

For example, the BadGas specialist of Auto-Mech has this kg:

```
(relevant Table
   (match
      (AskYNU? "Is the car slow to respond")
      (AskYNU? "Does the car start hard")
      (And
        (AskYNU? "Do you hear knocking or pinging
                                          sounds")
        (AskYNU? "Does the problem occur while
                                          accelerating"))
     with
       (if T ? ?
           then -3
         elseif ? T ?
           then -3
         elseif ? ? T
           then 3
         else 1)))
```

If the first expression is T (true), then the value of the kg is -3. Else, if the second expression is T, then -3. Else, if the third expression is T, then 3. Otherwise, its value is 1. Note that the number of tests following the "if" or "elseif" is the

same as the number of expressions of the table. Also note that each test must be true for the "row" of the table to match. The "?" test matches any value. The syntax for expressions and tests are discussed below. You are encouraged to look at other Table kgs in Auto-Mech.

The other type of kg which will be discussed is the Rule kg. Its form is:

```
(⟨name⟩ Rules ⟨comment⟩
    (match ⟨expression⟩
      with (if ⟨test⟩
               then ⟨value⟩
             elseif ⟨test⟩
               then ⟨value⟩
             else ⟨value⟩))
    (match ⟨expression⟩
      with ...)
    ...)
```

A example from the Carburetor specialist of Auto-Mech is:

```
(other Rules
  (match (AskYNU? "Is there fuel leaking around the
                                        carburetor")
    with (if T then 3))
  (match (Or
           (And
             (AskYNU? "Do you hear knocking or
                                        pinging sounds")
             (AskYNU? "Does the engine idle fast"))
           (And
             (AskYNU? "Does the car hesitate")
             (AskYNU? "Does the problem occur while
                                        decelerating")
             (AskYNU? "Does the engine idle fast"))
           (AskYNU? "Does the engine idle slow")
           (AskYNU? "Does the car run rough"))
    with (if T then 3)))
```

Each rule (a match-with form) is tried in succession until one "matches". A rule matches if it returns a value, i.e., the ⟨expression⟩ satisfies a test in the if-then part, or if there is an else clause within it. (As a consequence, it only makes sense to have an else clause in the last rule.) The value of the matched rule becomes the value of the kg.

4.3.1. Expressions in CSRL

Names of knowledge groups, numbers, CSRL variables, LISP expressions (e.g., AskYNU? is a LISP function), "self", and the logical constants T, F, and U are the base expressions of CSRL. CSRL variables will not be discussed here (but see the Refine message procedure of Specialist for an example). Any list which is not mistaken for a CSRL expression is assumed to be a LISP expression. The literal "self" evaluates to the name of the current specialist. Note that CSRL uses a three-valued logic.

CSRL provides the normal set of logical and numerical comparison operators, as well as a small set of arithmetic functions.

(And ⟨exp⟩ ⟨exp⟩ ...)
: Returns T if every expression is T, F if any expression is F, and U otherwise.

(Or ⟨exp⟩ ⟨exp⟩ ...)
: Returns T if any expression is T, F if every expression is F, and U otherwise.

(Not ⟨exp⟩)
: Returns T if the expression is F, F if it is T, and U otherwise.

LT, LE, GE, GT
: Numerical comparison operators with the obvious interpretation. If one or both expressions are not numbers, U is returned.

(Range ⟨exp⟩ ⟨exp⟩ ⟨exp⟩)
: Returns T if the first expression is within the range (closed interval) of the second and third expressions, and F if it is not. U is returned if any expression is not a number.

EQ
: Equivalent to EQ in LISP.

Plus, Subtract, Minus, Times, Divide
: Arithmetic functions with obvious interpretations. If any of the expressions are not numbers, U is returned.

4.3.2. Tests

(And ⟨test⟩ ⟨test⟩ ...)
: Returns T if every test is T, F otherwise.

(Or ⟨test⟩ ⟨test⟩ ...)

Returns T if any test is T, F otherwise. test is F, and U otherwise.

(Not <test>) Returns T if the test is F, F if it is T, and U otherwise.

(LT <number>), (LE <number>), (GE <number>), (GT <number>),
Numerical comparison with the obvious interpretation, e.g., for LT, returns T if the expression is less than the number.

(Range <number> <number>)
Returns T if the expression is within the range (closed interval) of the numbers, and F otherwise.

(EQ <atom or number>)
Returns T if the expression is EQ to the atom or number, and F otherwise.

<atom or number>
If not embedded in a LT, Range, or some other comparison test, an equality test is implied.

4.4. Writing Procedures for Establish Messages

In general, a specialist sets its confidence value within its procedure for the Establish message. This section explains some of the syntax for these procedures. A simplified form for an Establish message procedure is:

```
(Establish <comment>
   <statement>
   <statement>
   ...)
```

CSRL also has facilities for creating local variables and passing parameters, but these will not be necessary for the final exercise. As always, you are encouraged and exhorted to look at the Auto-Mech specialists as examples for specialists that you write.

4.4.1. Statements

(if <exp> then <st> elseif <exp> then <st> else <st>)
Evaluates each expression until one is T. The corresponding clause is executed. If no

expression is T, the else clause (if any) is executed. More than one statement can follow a then, elseif, or else.

(SetConfidence <expression> <expression>)
The first expression should evaluate to the name of a specialist. In general, you will only use "self" here. The value of the second expression becomes the confidence value of the specialist.

(DoLisp <form> ((<lisp var> <CSRL exp>)
(<lisp var> <CSRL exp>)
...)
Uses Lisp EVAL to evaluate the form. Before that, each of the Lisp variables are bound to the corresponding CSRL expression. This allows an escape to Lisp, and a way to get at values in CSRL. This may also be used as an expression.

5. Final Exercise

Exercise 9: Implementing Part of the House Expert System

Add knowledge groups and establish procedures to the specialists in the Heating hierarchy. Use AskYNU? to get information from the user about his heating system. You should consider the following questions in your implementation.

"Does your house get too cold"

"Does your house get too hot"

"Is your heating bill too high"

"Are you out of fuel oil"

"Is your furnace old"

"Has your furnace been checked recently"

"Is the fuel oil igniting in your furnace"

"Is your thermostat set low"

"Is your thermostat set high"

"Does changing the thermostat affect the temperature"

Also, examine the following questions, consider what additional hypotheses should be considered, and modify the Heating hierarchy accordingly. In addition to adding more specialists, you will probably need to reorganize the hierarchy so that similar specialists are grouped, e.g., having a FuelOilDelivery specialist with EmptyFuelOilTank and ClosedFuelOilValve as subspecialists.

"Is the fuel oil valve open"

"Are your heating vents open"

"Does the furnace fan turn on"

"Are your windows open"

"Do you have a fireplace"(heat can escape up the chimney)

"Do you feel air coming through the windows"

"Is your attic insulated"

Table of Contents

List of Exercises

XIV

a, b & c

filed on: {indigo}<kbvlsi>loopscourse>truckin.bravo
Last Edited: SM: July 11, 1983

Truckin

A teaching game for expert systems

by the Loops Design Team
Daniel Bobrow, Sanjay Mittal, and Mark Stefik

Abstract. *Truckin* is a knowledge system used for teaching knowledge representation techniques in Loops. *Truckin* provides an environment for creating, testing, and evaluating small bodies of knowledge interactively. Much of the knowledge in *Truckin* is represented as rules for controlling an automated truck in a simulation. The rules enable a truck to plan its journey along a road as it buys and sells commodities. A *Truckin* knowledge base can be evaluated in terms of the ability it gives an automated truck to make a profit while avoiding hazards of the highway. *Truckin* knowledge bases can be extended incrementally, so that a new Loops user can begin by extending existing sets of rules. *Truckin* contains illustrative examples and idioms for access-oriented, object-oriented, and rule-oriented programming.

Introduction

Loops is a knowledge representation system designed for use in building expert systems. It augments the Interlisp-D environment with object-oriented programming, access-oriented programming, and rule-oriented programming. Loops was developed by members of the Knowledge Systems Area at Xerox PARC.

In January 1983, the Loops system was ready for *beta*-testing outside of Xerox PARC. To help beta-test users to learn and evaluate Loops, the Loops Group decided to offer a short intensive course. This course was intended to provide hands-on experience in using Loops. Because Loops was designed for expert systems applications, it was believed that ***the best way to teach Loops would be to organize the course around a mini-expert system.***

It was important that the mini-expert system for the course not be too technically specialized, because people taking the Loops course would come with a variety of different backgrounds. The mini-expert system should be based on knowledge from common experience. The expert system needed to be engaging enough and open-ended enough to draw people into developing fairly elaborate knowledge bases. This led to the idea of a simulation game around which we could have "knowledge competitions".

This document describes the *Truckin* world as a teaching and simulation game around which the Loops course is organized.

The *Trucker's Handbook*

A "player" in *Truckin* is qualitatively different from a player of computer and video games, such as those that are popular in arcades and on home computers. In most computer games, a player is a person. In *Truckin*, a player is a knowledge base. In this way, personal competition in *Truckin* is one level removed from the game. The objective is to create a knowledge base that can effectively guide a truck in the situations it encounters in the simulation environment.

Truckin provides a set of commodities, producers, consumers, hazards, road stops, and trucks. The *Truckin* world is intended to be complex enough to be interesting in the Loops course, but too complex for a simple mathematical model. The online data base of facts about this world is, metaphorically, called the *Trucker's HandBook*. A wise *Truckin* player will consider the facts in the *Trucker's Handbook* in planning its route.

The simulation in *Truckin* is controlled by a program called the *GameMaster*. In different contexts, the term "Game Master" refers to different combinations of programs, knowledge bases, processes, and computers. For simplicity, we refer to the whole thing as "the" *GameMaster*. The *GameMaster* chooses the initial configuration of the highway, sometimes called the game board, decides on the legality of the requests made by the players, updates the GameWorld and maintains the display.

The players talk to the *GameMaster*, who also decides which player gets the next turn. Currently, players get their turn on a time-robin basis, i.e., the player who has used the least amount of time gets the next turn. During a turn a player can buy or sell commodities at any *RoadStop* at which it is parked. These transactions are governed by practical considerations of how much money is in the truck's *cashBox*, whether there is cargo room in the truck for the goods, and whether the *RoadStop* advertises an interest in buying or selling the particular commodities. During a single move a truck can also drive to one other *RoadStop*. The distance that the truck can travel in a move is governed by a variable reflecting traffic conditions, as well as the maximum speed of the truck and the amount of gasoline remaining in the fuel tank.

Profits and Risks

The goal of a player in *Truckin* is to maximize profit during the game. The game ends after a predetermined number of turns. At the end of a game, the winning player is the one with the most cash. *AlicesRestaurant* is a special roadstop because any player who is parked there when the game ends, gets a hefty bonus.

Players compete for a fixed supply of goods and parking places. Just as with real trucks, there are a number of things that are important to know about the world. For the details of this, a player should consult the *Truckin Handbook (*database of relevant Loops classes). Here is a summary of the elements of the *Truckin* world:

Kinds of Trucks. Players start the game at *UnionHall* with an empty *truck* and an allotment of fuel and cash. Trucks come in different varieties, with different speeds, different fuel efficiencies, and different capacities for carrying merchandise. During each turn, the speed of the truck is measured as the ratio of the number of roadstops actually moved and the maximum allowed for that class of truck.

RoadStops. *RoadStops* are the positions along the highway. In the standard version of the game board, there are sixty six RoadStops along the highway. Neighboring *RoadStops* are separated by one mile. Up to two trucks can be parked at a *RoadStop.*

Producers. *Producers* are *RoadStops* at which players can purchase goods. A given producer will sell only a fixed kind of item, for example televisions, shirts, or apples. A *Producer* has only a fixed inventory of items for sale, and this inventory is used up as the simulation runs. The game board display shows the quantity of items for sale, and a price ratio which can be multiplied times the *averagePrice* of a *Commodity* to determine the purchase price. An icon is displayed on the game board to show the kind of *Commodity.* There are usually about 30 *Producers* distributed along the highway.

Consumers. *Consumers* are *RoadStops* at which players can sell goods. In general, *Consumers* are interested in generic kinds of goods, such as sporting goods, office supplies, or groceries. The capacity for a *Consumer* to buy goods decreases as items are purchased. The game board display shows the quantity of items that will be purchased, and a price ratio. The name of the generic class of *Commodities* to be purchased is displayed on the game board. There are usually about *23 Consumers* distributed along the highway.

Commodities. *Commodities* are the things that are bought and sold along the highway. The kinds of Commodities that are available are shown in the *Trucker's Handbook.* Some Commodities have special features, such as being fragile or perishable. *PerishableCommodities* have a *lifetime* (expressed in turns) which determines how long the *Commodities* remain salable. *Fragile Commodities* have a *fragility* which determines how likely they are to break when you go past *RoughRoads. Commodities* also have a volume and a weight which means that a truck can carry commodities limited by the available volume and weight on the truck.

Gasoline. Driving a truck uses up gasoline. Gasoline can be purchased at *GasolineStations* along the highway. Running out of gasoline results in a towing and a fine. There are usually about 5 *GasolineStations* along the highway.

WeighStations. *WeighStations* represent the arm of the government in *Truckin.* If a player goes by a *WeighStation* without stopping, he risks some chance of receiving a stiff fine and a towing back to the *WeighStation.* If he stops, he must pay a small toll (and use up a turn).

Rough Roads. Some *RoadStops* correspond to rough places on the road. Driving past a *RoughRoad* entails some risk to any *FragileCommodities* that are on board. If a player stops at a *RoughRoad,* no damage will result.

Bandits. *Bandits* in *Truckin* do not sit still. They can park at various *RoadStops* as controlled by the *GameMaster* and can intercept trucks. If a bandit intercepts a truck, or is parked at the same roadstop as a truck, it will take all of the *LuxuryGoods* that it has room for and one fifth of the money in the *cashBox.*

The CityDump. In general, an attempt to sell perished or damanged goods results in a stiff fine.

However, such goods can be unloaded for a fee at the *CityDump.* (In the simulation, these goods are sold for a modest "negative price".)

The Union Hall. If a player runs out of gas, he will be towed to *Union Hall.* There he will be given a new allotment of cash, but his truck will be emptied. This happens to a player whether he goes to *UnionHall* on his own request, or whether he is towed there for violating some rule.

Alice's Restaurant. At the end of the game, all of the trucks try to make it to *Alice's Restaurant.* Players ending the game at any of the Alice's get their cash doubled. There may be more than one *Alice's Restaurant* on the highway, and any one of them will do. If there are more trucks in a game than parking places at the restaurant, then there will be competition for the places. To preclude the strategy of just going to *Alice's Restaurant* and parking, any player who parks there for more than some specified time will be towed away to Union Hall.

Advice for Independent *Truckers*

To succeed at *Truckin,* a player must be responsive to the configuration of the highway and to changing conditions. To make a profit, a player must consider the spread between price ratios and the convenience of the relative locations for buying and selling commodities. A player must not exceed the capacity of his truck in either weight or volume.

We have a few final suggestions for players. Don't buy goods that you can't sell at a profit. Don't buy *PerishableCommodities* if you can't deliver them on time. If your goods spoil or are damaged, take them to the *CityDump.* Keep an eye on your fuel gauge. Don't drive too quickly with *FragileCommodities* over *RoughRoads.* Don't spend all of your cash on *Commodities*; you may need some for incidentals along the way. Watch out for bandits, rough roads, and weigh stations. And try to be at *Alice's Restaurant* when the game ends.

filed on: {indigo}<kbvlsi>loopscourse>truckinmanual.bravo
Last Edited: SM: July 11, 1983

Truckin MANUAL

by the Loops Design Team
Daniel Bobrow, Sanjay Mittal, and Mark Stefik

This document gives the basic instructions for creating game boards, starting, stopping, and continuing a game, interrupting a game in the middle, and attaching gauges to monitor the internal state of *Truckin* players.

[NB: *Truckin* now has versions which run on both single machines as well as multiple-machine configurations. The following instructions are written for the single machine version. Any differences for the multi-machine version are indicated in smaller print. Otherwise the instructions apply to both versions.]

A. Creating a new game

Send the message *New* to *$Truckin* as follows:

(← $Truckin New)

this creates a new game board and the lisp variable *PlayerInterface* is set to the instance of *TruckinPlayerInterface*. **All commands sent by your player or you go to *PlayerInterface*.** You can play any number of times on this basic game board as follows.

[On a *RemoteMasterMachine*:
(← $MasterTruckin New) creates a new game.
On a *RemoteSlaveMachine*:
(← $SlaveTruckin New) sets up the Truckin world and links your machine to the MasterMachine. You will be asked for a *unique name* to identify your machine and the address of the *PostMaster*. Please ask the game coordinator for this address. A new game cannot be created from a slave machine - the slave machine will run the game created by the master machine.
PlayerInterface is set as above in both these cases as well].

[[[[In all these cases, upto 4 arguments can be given to the New message to select the game configuration you want. The description above is for the default case.
Arg 1: Type of Game--
This specifies what kind of *DecisionMaker and PlayerInterface* you want. Currently the only value is **TimeTruckinDM** (the appropriate player interface is automatically selected). Later, we may put in other versions of the game.
Arg 2: Type of Game Board--
This specifies what kind of game board you want: **BWTruckin** or **ColorTruckin**. The former is the default. In order to use ColorTruckin option, you need a color monitor attached to your machine.
Arg 3: Type of Simulator--
This specifies whether you want the game board to be displayed or not: **DisplayTruckinS** or **NoDisplayTruckinS**. The former is the default. The NoDisplay version of the simulator maintains an upto date version of the game but does not display the game board.
Arg 4: Broadcast List--
This is a list of objects who want to receive a copy of all game messages which change the world. These objects must be capable of responding to the messages described in the *MultiMachineTruckin* document. These objects will get the messages after the world has already been updated.]]]]

B. Starting a game

Send the message *BeginGame* to *PlayerInterface* to start a game as follows:

(← PlayerInterface BeginGame)

This message refreshes the game board created earlier and prompts you for the players you want in this game. You can either create new players from among the existing player classes (via an interactive menu) or use any players created earlier. [The menu appears next to the prompt window at the left top of the screen]. The menu for the players offers you a choice of both player classes and existing player names. You can opt for all existing players by choosing the *ALL-EXISTING* menu option. Select *NO* when you are done selecting players.

You can pass one optional arguments in the BeginGame message.

Arg : If T then all existing players will be used for the game and you will not be asked for players. This might be convenient during debugging when you want to use the same game board and same set of players for debugging your player.

[[If you are running *SlaveTruckin*, BeginGame will let you select your local players, but the game will only start when the Master Machine decides - which it does when a BeginGame is done on the Master Machine.]]

C. Interrupting a game in the middle.

In addition to the rule exec and the break/trace facility of the rule language (see Rule Language manual), there is another way to temporarily stop a game in the middle and bring up the lisp user exec. Hold the **CTRL** and **LEFT SHIFT** keys simultaneously when one of the trucks is moving. This will put you into the Lisp User Exec, where you can examine things and/or edit your rule sets and functions. Type **OK** in the Exec to resume the game. On a dorado, the trucks move pretty fast, so if the above does not work the first time, try again.

C.2 Interrupting a player any time

Left of the Status Window, you will notice a menu which lists the players running on your machine. Selecting any player in the menu, allows you to interrupt that player and bring up the Rule Exec. Remember that the game time continues to click while you are in the Rule Exec.

D. Suspension/Premature Termination of the Game

You can suspend, resume, or kill the game by using the **Game Control Menu**, which normally appears left of the Status Window. Selecting *Suspend* will suspend the game (but remember that the time allocated for the game continues to tick, so when you resume, the intervening time will be deducted from the game time). Selecting *Awake* will resume the game and *Kill Game* will kill the game.

E. Attaching gauges to your player's truck

You can attach gauges to *Instance Variables* (IVs) of objects under your control such as your player or truck in order to monitor important internal state during the game. When you first create a player, the game master will offer to put gauges on your truck, i.e., to the IVs *cashBox, fuel, weight,* and *volume.* You have several options. **NO** will not put any gauges. **YES** response will lead to the system asking you whether you want gauges on each of the four IVs listed above. For each IV for which you respond with *YES* the system will offer a choice of gauges. **DEFAULT** response will put default gauges on fuel.

Once you put gauges on a player, they can be reused when you use the same game board for a new game or create new game boards. Thus, if you expect to use a player many times, it pays to attach the desired gauges once and continue to use the player.

F. Attaching gauges to other IVs of your player

When you create a player, the instance object is given the same name as the driver name you enter. Thus, if you name some player **Joe** you can access the object as **$Joe.**

You will often find it useful to attach gauges to IVs of your player. For example, if your player is an instance of **Peddler**, you might want to monitor IVs such as *destination, stoppingPlace,* and *goal.* The way to attach gauges on your player is to send it the **AddGauges** message. For example,

(← $Joe AddGauges '(destination goal)

will attach gauges to destination and goal IVs of $Joe if $Joe is an instance of Peddler. The AddGauges method will prompt you for the type of gauge. The most suitable gauge for arbitrary values is **LCD.**

The AddGauges message can be used to select default gauges on the instance variables indicated, instead of having to select gauges yourself each time. In order to do this, you have to specify additional information in the object class as shown in the following simplified description of the class **Truck.**

```
(DEFCLASS Truck
   (MetaClass ..)
   (Supers ..)
   (ClassVariables ..)
   (InstanceVariables (cashBox 10000 DefaultGauge LCD GaugeLimit (0 10000))
                      (fuel 80 DefaultGauge Dial GaugeLimit (0 80))))
```

Thus, suppose, you wanted an LCD gauge to be the default gauge on *destination,* you can specify this for use by the AddGauges method by adding the property *DefaultGauge* to the instance variable *destination* with *LCD* as the value. Then pass T as the *second argument* in the above AddGauges message. This will result in a LCD gauge being installed on destination and you will be prompted only for goal. [You can do the same for *goal* or any other IVs also]. If the default gauge you have specified is being used for numbers, you also should specify the default limits. For this, put under the *GaugeLimit* property a list containing the two numbers which indicate the lower and upper limits.

G. Adding gauges under program control

You can also attach gauges under full program control by specializing the method **SetUpGauges** in the class **Player.** The description given above is carried out by this method. You could write your own SetUpGauges method in your player class and make it attach gauges by using the method *AddGauges* described earlier. Both **Truck** and **Player** respond to the message AddGauges. This way you could build into your SetUpGauges method, your choice of gauges, which then will be carried out by the system each time you create a player of that class.

H. Selecting trucks under program control

You can also select the truck you want for your player automatically, instead of being prompted for it. In order to do this specialize the method **SelectTruck** for your player class. This method will be called when your player class is instantiated. This method should return the name of one of the truck classes currently allowed in the game. Currently, the allowed trucks are: **MacTruck, GMCTruck, FordTruck,** and **PeterBiltTruck.**

I. Summarizing the truck data at a glance

You can get a summary report of your players truck by sending your player (say **Joe**) the **Show** message as follows:

(← $Joe Show)

This will print out the cashBox, fuel, weight, and volume, as well show you the cargo your truck is carrying. This summary may be useful during debugging.

J. Clearing up the screen

If your screen gets messed up for some reason, you can restore it to the initial state by buttoning the LoopsLogo in the middle top of your screen and selecting the command *SetUpScreen*. You can also do this in the middle of the game when you are in any of the rule exec, user exec or break exec. Even though the game board and gauges will disappear temporarily, they will come back as those windows are written to.

K. When players get control

A player gets control when his/her turn comes and the game master sends a **TakeTurn** message to the instance of your player object. Your top-level rule-set must be written to respond to this message.

You can also write your player in such a way that the top-level rule set never returns, i.e., the *TakeTurn* rule-set uses *whileAll* control structure. The *PlayerInterface* will suspend you when you make a *Buy, Move,* or *Sell* request and reschedule you when your turn comes again.

L. Legal requests by players during game

A player can make three kinds of requests during the game: **Move, Buy, Sell.** After each request, the player is suspended until the request is completed and your turn comes again (i.e., all other players have used up the same amount of time).

1. (← PlayerInterface Move player numOrLoc)
This is a request to move *player* from the current location to a location determined by *numOrLoc*. If numOrLoc is a number, then it is the relative offset from the current location. It can be positive or negative. It can also be the actual instance object representing the particular roadStop in the game.

2. **(← PlayerInterface Buy player qty)**
This is request to buy *qty* of the commodity at the location at which *player* is currently parked.

3. **(← PlayerInterface Sell player commodityInstance qty)**
This is a request to sell sell *qty* of the commodity *commodityInstance* owned by the player in their truck's cargo, at their current location. If qty is not specified, then the qty in the commodityInstance will be used.

The standard value of *player* in all the three above messages is *self* which is bound to the player executing the rule-set.

filed on: {indigo}<kbvlsi>loopscourse>truckinvocabulary.bravo
Last Edited: SM: July 11, 1983

Truckin Query Functions

by the Loops Design Team

Daniel Bobrow, Sanjay Mittal, and Mark Stefik

This document summarizes the functions and methods you will find useful in writing the rules for your *Truckin* players. These functions allow you to select and filter roadstops satisfying different constraints as well as conveniently access other information about the current status of the *Truckin* world. Many of the following functions are also available as methods attached to the class **Player**, allowing you to easily specialize them if you so desire.

In the following summary, functions marked with an asterisk (*) are also implemented as methods on **Player** with the same name as the function and taking the exact same arguments. For more details about these functions see the listing of the file TRUCKINV in your folder.

A. Selection functions

The following functions return a list of roadstops based on certain constraints.

AnyRoadStop (roadStopType numMoves direction roomToParkFlg)*
Randomly picks one of the roadstops of type *roadStopType* where *roadStopType* is one of the RoadStop classes. If *numMoves* is provided, it returns only those roadstops within that distance. If *direction* is F then only those in the forward direction, if B then only in the backward direction, if NIL then in either direction. If *roomToParkFlg* is T then only those roadstops where there is room to park.

Buyers (commodityClass numMoves includeCDFlg)*
Returns all of the Buyers (i.e. *Consumer* roadstops) able to purchase a commodity of type *commodityClass*. If *numMoves* is provided, returns only those within that distance. A common case is to use the instance variable *maxMove* of your player as this argument. If *includeCDFlg* is T then includes *CityDumps* also, otherwise not.

NthRoadStop (numMoves direction fromRoadStop roomToParkFlg)*
Returns the Nth roadstop in the given direction from *fromRoadStop*. If *fromRoadStop* is NIL, the current location of the player is used. If *direction* is NIL, **Forward** is assumed. If there are fewer than *numMoves* roadstops in the specified direction, that is if the request would go off the board, this function returns the farthest roadstop in that direction.

RoadStops (roadStopType numMoves direction roomToParkFlg)*
Returns all of the roadstops of type *roadStopType* reachable within *numMoves* in the direction specified by *direction* taking into account room to park if *roomToParkFlg* is T.

Sellers (commodityClass numMoves)*
Returns all the roadstops which are Sellers (i.e. *Producer* roadstops) of *commodityClass* and are located within *numMoves.*

B. Filter functions
The following functions take a set of roadstops as one argument and prune that set based on other criteria specified by other arguments. Some of the following functions are very general and can be used to filter (or order) any set of objects of the same class and are not limited to working on roadstops only. These are: **FilterObjs, PickHiObj, PickLowObj,** and **SortObjs.**

FilterObjs (self selector objects)
Sends a *selector* msg to *self* for each of the object in *objects* and returns all of the objects for which the rule set returned a non-NIL value. This is the basic function for doing filtering based on your knowledge encoded as rules.

FurthestRoadStop (roadStops fromRoadStop)*
Returns the roadstop in *roadStops* which is furthest from *fromRoadStop* excluding *fromRoadStop.* If *fromRoadStop* is NIL, assumes the current location of the player.

NearestRoadStop (roadStops fromRoadStop)*
Same as **FurthestRoadStop** except returns the nearest roadstop.

PickHiObj (self selector objects)
Sends a *selector* msg to *self* for each object in *objects* to determine a numeric rating for each of the objects. It returns the object with the highest numeric rating. When the value returned is non-numeric for an object, then that object is automatically excluded.

PickLowObj (self selector objects)
Same as **PickHiObj** except returns the one with the lowest numeric rating.

SortObjs (self selector objects)
Sends a *selector* msg to *self* for each object in *objects* to determine a numeric rating for each of them. It returns a list of objects in the **descending** order of their numeric rating. It also excludes the ones with non-numeric ratings.

C. Miscellaneous functions

AnyBanditsP (toRoadStop fromRoadStop)
Returns T if there are any bandits parked between *toRoadStop* and *fromRoadStop,* NIL otherwise.

DirectionOf (toRoadStop fromRoadStop)*
Returns the direction of travel for going from *fromRoadStop* to *toRoadStop.* If the *fromRoadStop* is not given, then the current location of the player is assumed.

Distance (toRoadStop fromRoadStop)*

Computes the distance between *fromRoadStop* and *toRoadStop.* If the *fromRoadStop* is not given, then the current location of the player is assumed.

PricePerUnit (producerRoadStop)
Returns the buying price per unit of the commodity being sold at the *producerRoadStop.* If the argument is not *a Producer* roadstop, then complains and returns 1.

RoomToParkP (roadStop)
Returns T if there is room to park at *roadStop.*

ISA (instance className)
Returns T if *instance* is an instance of *className.*

Nth (list index)
Returns the *index* element of *list.*

SUBCLASS (class superClass)
Returns T if *class* is same as or a subclass of *superClass.*

The following are available only as methods on **Player** class.

(← player Range)
Computes how far the *player* can move based on the amount of fuel carried on the player's truck.

(← player Range1)
Computes how far the *player* can move in a single turn. This depends on the fuel in the truck and the maximum distance allowed by the game master for that turn.

(← player TimeAtStop)
Returns the time spent by player at the stop where currently parked. Useful when parked at one of the Alice's.

(← player TurnsAtStop)
Returns the number of turns *player* has been parked at the stop where currently parked. Useful when parked at one of the Alice's.

D. Useful Global Variables

1. PlayerInterface (you can also use **PI**)
After doing (← $Truckin New), *PlayerInterface* is bound to the instance of the class *TruckinPlayerInterface* and is used to send messages to the GameMaster for making moves and starting game. You can also get some game information such as *roadStops* and *localPlayers* from this object.

2. Simulator

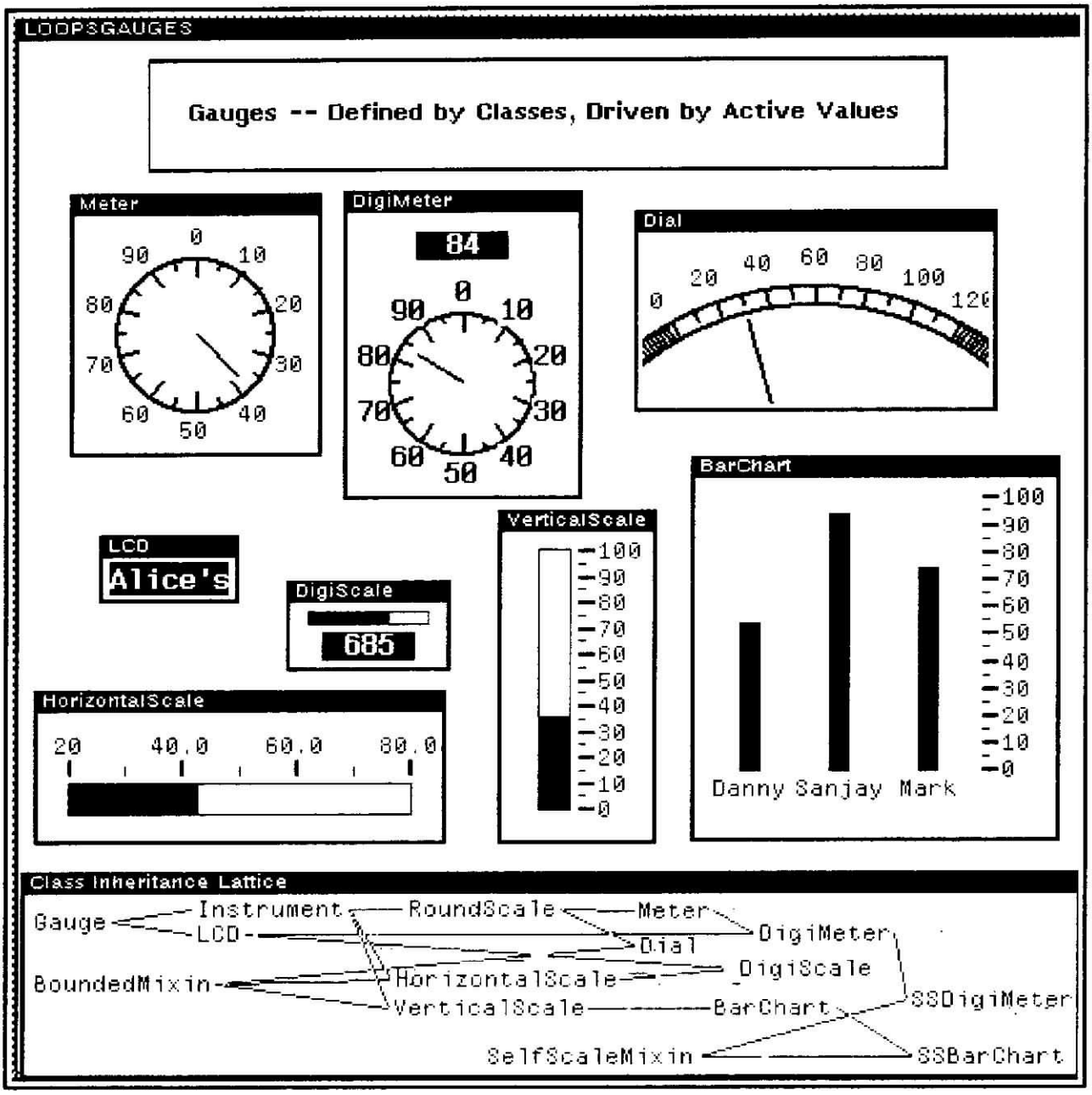
Once the game is set up, *Simulator* is bound to the instance of *TruckinSimulator* and can be used to access important game information such as *roadStops, players, beginTime, endTime, timeLeft.*

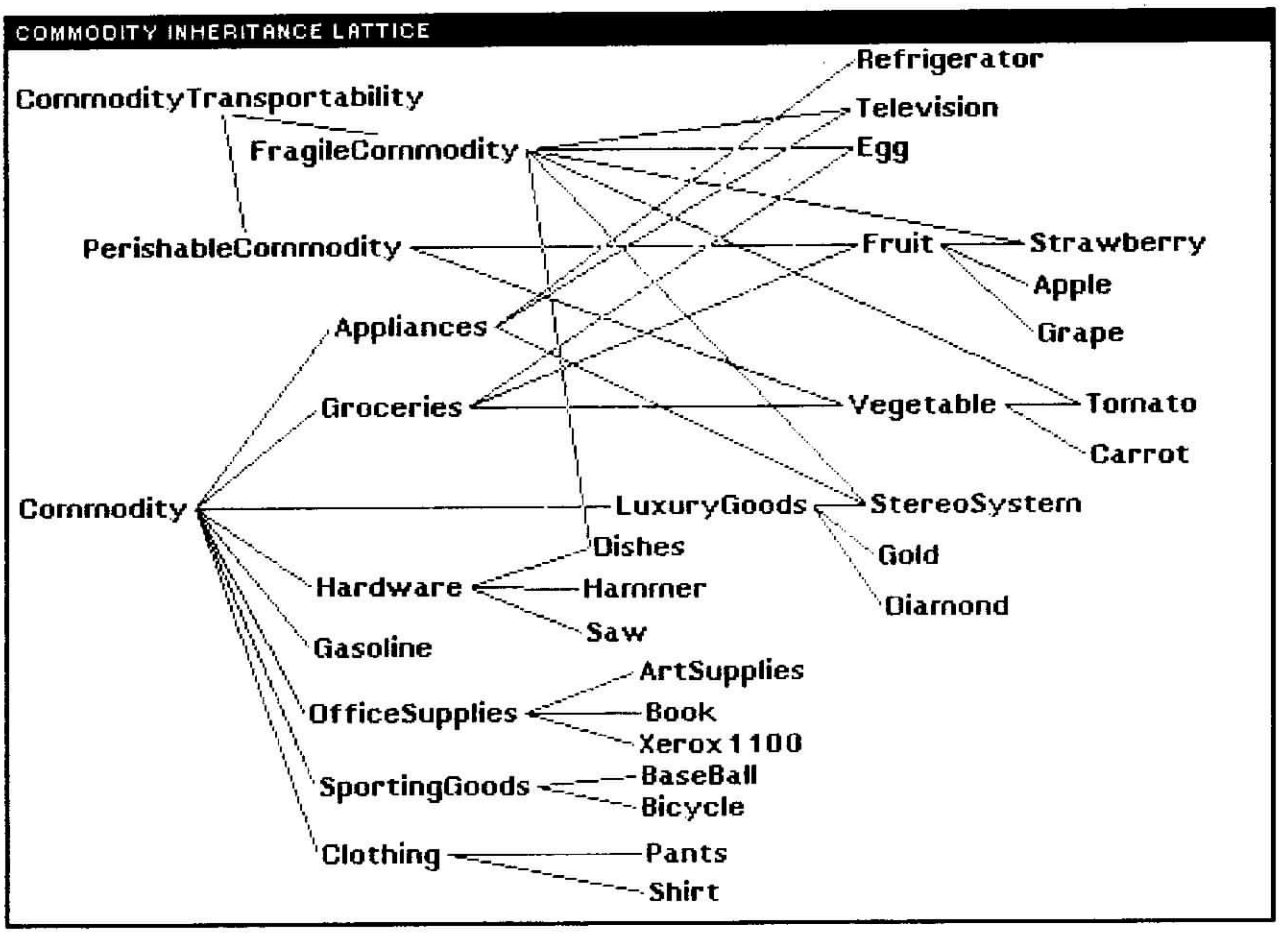
3. debugMode

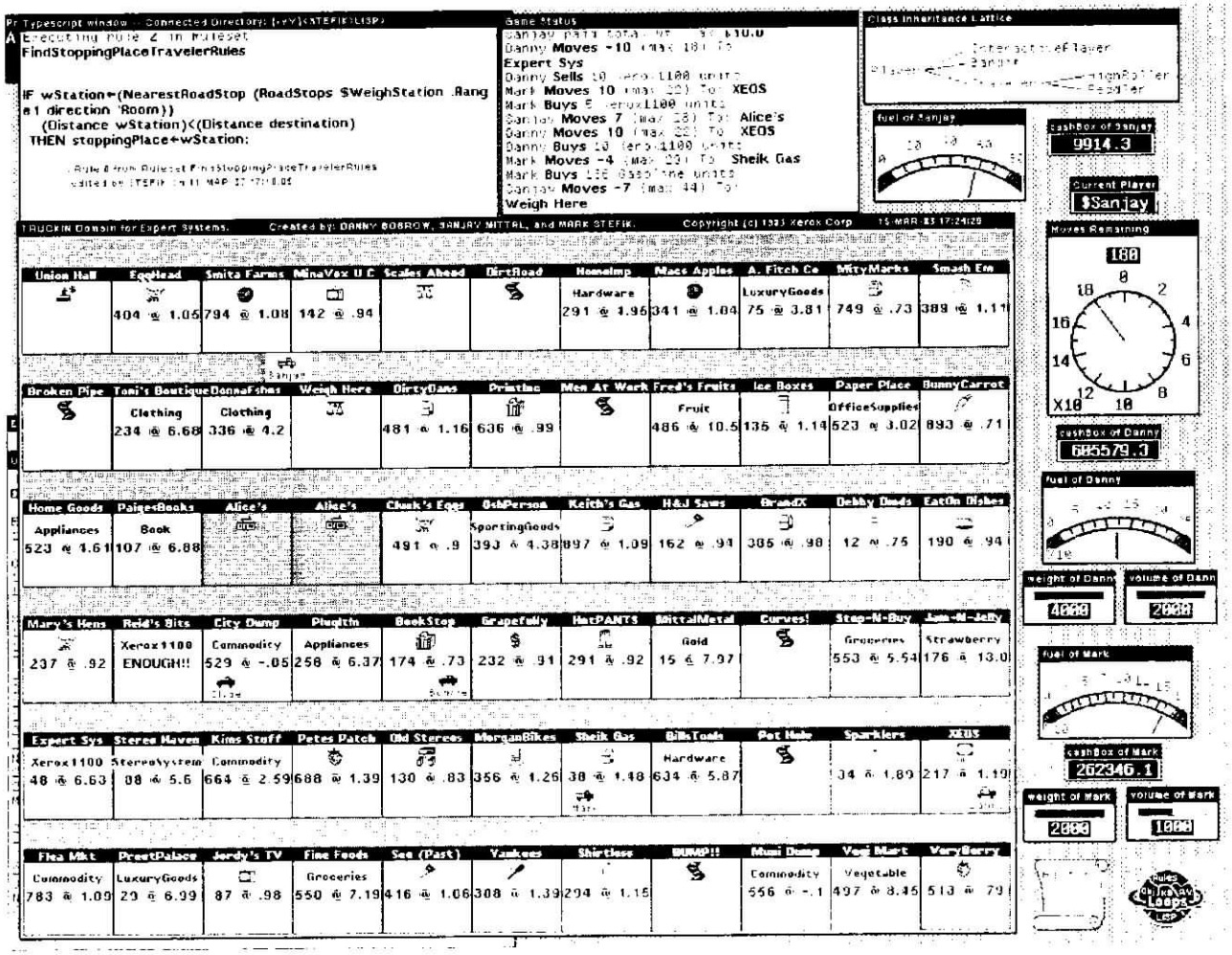
If set to *T*, then each time a rule is violated, the *RuleExec* is automatically brought up. Useful while debugging your rulesets. If set to *NIL*, then the RuleExec is not entered for each rule violation. Also, the GameMaster traps all errors. Initially set to *T*.

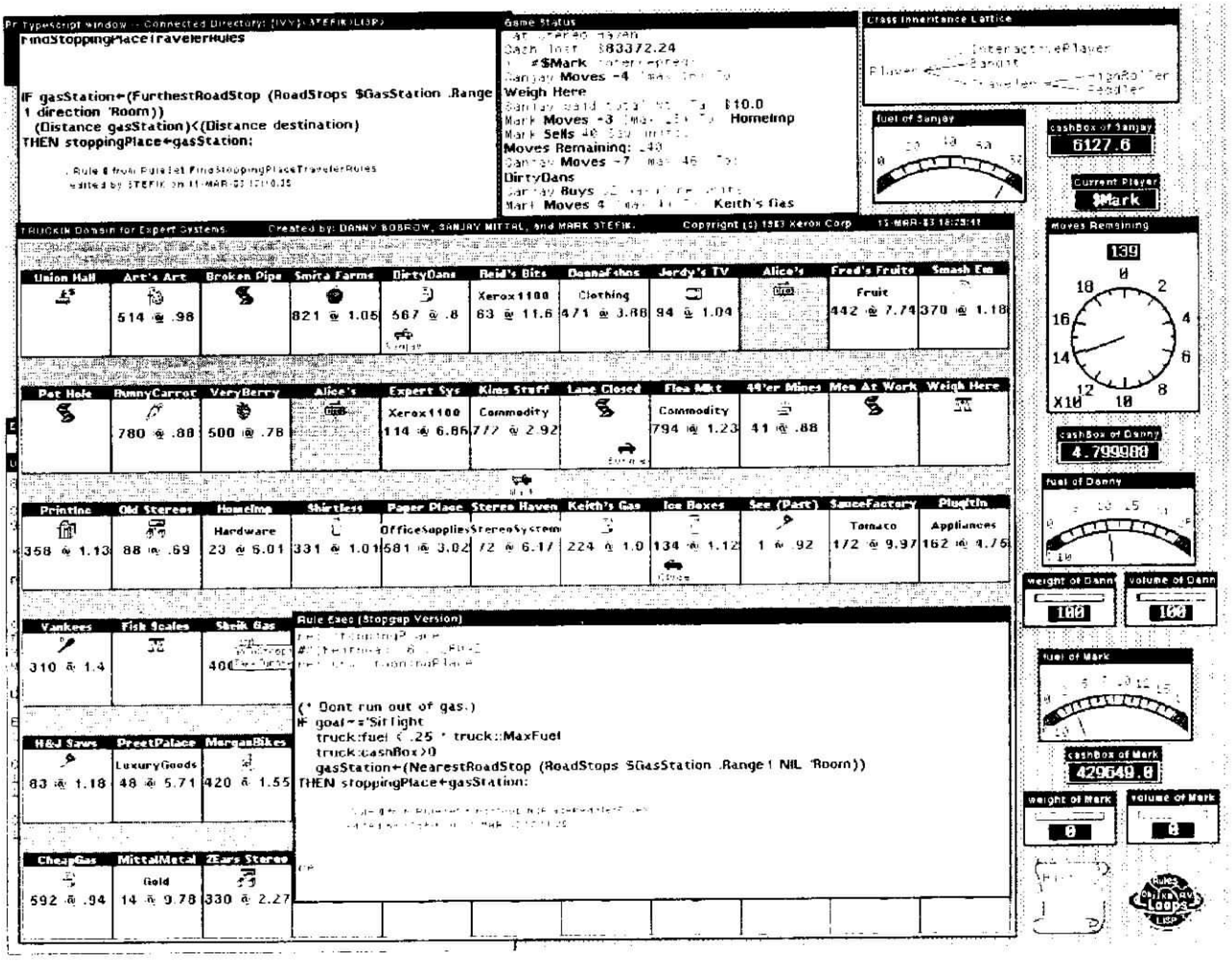
4. truckinLogFlg

If set to *T*, before game is started, then prepares a log file of all important game messages in a file called *TRUCKINLOG*. This log file may be useful during the debugging of your players. Set this variable to NIL, if you dont want any log file. Initially set to NIL.


LOOPSGAUGES
Gauges -- Defined by Classes, Driven by Active Values
Meter
DigiMeter
84
Dial
LCD
Alice's
DigiScale
685
VerticalScale
HorizontalScale
BarChart
Danny Sanjay Mark
Class Inheritance Lattice
Gauge
Instrument
LCD
RoundScale
Meter
Dial
DigiMeter
BoundedMixin
HorizontalScale
VerticalScale
DigiScale
BarChart
SSDigiMeter
SelfScaleMixin
SSBarChart


COMMODITY INHERITANCE LATTICE
CommodityTransportability
FragileCommodity
PerishableCommodity
Appliances
Groceries
Commodity
Hardware
Gasoline
OfficeSupplies
SportingGoods
Clothing
Refrigerator
Television
Egg
Fruit
Strawberry
Apple
Grape
Vegetable
Tomato
Carrot
LuxuryGoods
StereoSystem
Gold
Diamond
Dishes
Hammer
Saw
ArtSupplies
Book
Xerox1100
BaseBall
Bicycle
Pants
Shirt


Pr Typescript window -- Connected Directory: [rrV]<STEFIK.LISP>
Executing Rule 2 in RuleSet
FindStoppingPlaceTravelerRules
IF wStation←(NearestRoadStop (RoadStops $WeighStation .Rang
e1 direction 'Room))
(Distance wStation)<(Distance destination)
THEN stoppingPlace←wStation:
Game Status
Danny Moves -10
Expert Sys
Danny Sells
Mark Moves 10
Mark Buys
Sanjay Moves 7
Danny Moves 10
Danny Buys
Mark Moves -4
Mark Buys
Sanjay Moves -7
Weigh Here
Class Inheritance Lattice
Fuel of Sanjay
CashBox of Sanjay
9914.3
Current Player
$Sanjay
TRUCKIN Domain for Expert Systems. Created by: DANNY BOBROW, SANJAY MITTAL, and MARK STEFIK. Copyright (c) 1983 Xerox Corp
Moves Remaining
180
CashBox of Danny
685579.3
Fuel of Danny
Weight of Danny
40000
Volume of Danny
2000
Fuel of Mark
CashBox of Mark
262346.1
Weight of Mark
2000
Volume of Mark
1000
Union Hall
Eggshead
404 @ 1.05
Smith Farms
794 @ 1.08
MinaVox U.C.
142 @ .94
Scales Ahead
DirtRoad
Homelmp
Hardware
291 @ 1.95
Mace Apples
341 @ 1.04
A. Fitch Co.
LuxuryGoods
75 @ 3.81
MittyMarks
749 @ .73
Smash Em
388 @ 1.11
Broken Pipe
Toni's Boutique
Clothing
234 @ 6.68
DonnaFisher
Clothing
336 @ 4.2
Weigh Here
DirtyDans
481 @ 1.16
Printho
636 @ .99
Men At Work
Fred's Fruits
Fruit
486 @ 10.5
Ice Boxes
135 @ 1.14
Paper Place
OfficeSupplies
523 @ 3.02
BunnyCarrot
893 @ .71
Home Goods
Appliances
523 @ 1.61
PaigesBooks
Book
107 @ 6.88
Alice's
Alice's
Chunk's Eggs
491 @ .9
GoMPerson
SportingGoods
393 @ 4.38
Keith's Gas
897 @ 1.09
H&J Saws
162 @ .94
BrandX
385 @ .98
Debby Deeds
12 @ .75
EatOn Dishes
190 @ .94
Mary's Hens
237 @ .92
Reid's Bits
Xerox1100
ENOUGH!!
City Dump
Commodity
529 @ -.05
Pluglin
Appliances
258 @ 6.37
BookStop
174 @ .73
Grapefruity
232 @ .91
HotPANTS
291 @ .92
MittalMetal
Gold
15 @ 7.97
Curves!
Stop-N-Buy
Groceries
553 @ 5.54
Jam-N-Jelly
Strawberry
176 @ 13.0
Expert Sys
Xerox1100
48 @ 6.63
Stereo Haven
StereoSystem
08 @ 5.6
Kims Stuff
Commodity
664 @ 2.59
Petes Patch
688 @ 1.39
Old Stereos
130 @ .83
MorganBikes
356 @ 1.26
Sheik Gas
38 @ 1.48
BikTools
Hardware
634 @ 5.87
Pet Mike
Sparklers
34 @ 1.89
XEUS
217 @ 1.19
Flea Mkt
Commodity
783 @ 1.09
PrestPalace
LuxuryGoods
29 @ 6.99
Jordy's TV
87 @ .98
Fine Foods
Groceries
550 @ 7.19
Sea (Past)
416 @ 1.06
Yankees
308 @ 1.39
Shirtless
294 @ 1.15
BUMP!!
Miami Dunes
Commodity
556 @ -.1
Veg Mart
Vegetable
497 @ 8.45
VeryBerry
513 @ .79


Pr Typescript Window -- Connected Directory: [VVV]:STEFIK.LISP
FindStoppingPlaceTravelerRules
IF gasStation←(FurthestRoadStop (RoadStops $GasStation .Range
1 direction 'Room))
(Distance gasStation)<(Distance destination)
THEN stoppingPlace←gasStation;
Game Status
Cash Lost $83372.24
Sanjay Moves -4
Weigh Here
Mark Moves -3
Mark Sells
Moves Remaining: -40
Sanjay Moves -7
DirtyDans
Sanjay Buys
Mark Moves 4
Keith's fias
Class Inheritance Lattice
fuel of Sanjay
cashBox of Sanjay
6127.6
Current Player
$Mark
TRUCKIN Dossier for Expert Systems
Created by: DANNY BOBROW, SANJAY MITTAL, and MARK STEFIK.
Copyright (c) 1983 Xerox Corp
Moves Remaining
139
cashBox of Danny
4.799980
fuel of Danny
weight of Danny
100
volume of Danny
100
fuel of Mark
cashBox of Mark
429640.0
weight of Mark
0
volume of Mark
0
Union Mall
Art'e Art
Broken Pipe
Smith Farms
DirtyDans
Reid's Bits
Dannel'shos
Jordy's TV
Alice's
Fred's Fruits
Smash Em
514 @ .98
821 @ 1.05
567 @ .8
Xerox1100
63 @ 11.6
Clothing
471 @ 3.88
94 @ 1.04
Fruit
442 @ 7.74
370 @ 1.18
Pot Hole
RunnyCarrot
VeryBerry
Alice's
Expert Sys
Kims Stuff
Lane Closed
Flea Mkt
49'er Mines
Men At Work
Weigh Here
780 @ .88
500 @ .78
Xerox1100
114 @ 6.86
Commodity
777 @ 2.92
Commodity
794 @ 1.23
41 @ .88
Printine
Old Stereos
Homelmp
Shirtless
Paper Place
Stereo Haven
Keith's Gas
Ice Boxes
See (Part)
SauceFactory
Plugthn
358 @ 1.13
88 @ .69
Hardware
23 @ 6.01
331 @ 1.01
581 @ 3.02
OfficeSupplies
72 @ 6.17
StereoSystem
224 @ 1.0
134 @ 1.12
1 @ 92
Tomato
172 @ 9.97
Appliances
162 @ 4.76
Yankees
Fish Scales
Sheik Gas
310 @ 1.4
Rule Exec (Stopgap Version)
(* Dont run out of gas.)
IF goal~='SitTight
truck:fuel < .25 * truck::MaxFuel
truck:cashBox>0
gasStation←(NearestRoadStop (RoadStops $GasStation .Range1 NIL 'Room))
THEN stoppingPlace←gasStation;
H&J Saws
PrestPalace
MarsanBikes
LuxuryGoods
83 @ 1.18
48 @ 5.71
420 @ 1.55
CheapGas
MittalMetal
Zeus Stereo
Gold
592 @ .94
14 @ 9.78
330 @ 2.27

XVIII

(1)

LOOPS Summary

LOOPSNames (← obj SetName 'FOO) gives obj the Loops name FOO

$FOO evaluates to object named FOO #$FOO reads in as object named FOO

Variable Access Read macros and their translation

form	*translation*	*form*	*translation*
@X	(GetValue self 'X)	@X←newValue	(PutValue self 'X newValue)
@(Obj X)	(GetValue Obj 'X)	@(Obj X)←newValue	(PutValue Obj 'X newValue)
@(Obj X P)	(GetValue Obj 'X 'P)	@(Obj X P)←newValue	(PutValue Obj 'X newValue 'P)
@@X	(GetClassValue self 'X)	@@X←newValue	(PutClassValue self 'X newValue)
@@(Obj X)	(GetClassValue Obj 'X)	@@(Obj X)←newValue	(PutClassValue Obj 'X newValue)
@@(Obj X P)	(GetClassValue Obj 'X 'P)	@@(Obj X P)←newValue	(PutClassValue Obj 'X newValue 'P)
		@X←+newValue	(PushValue self 'X newValue)

Defining and Editing Classes

DC(className New *classNameList*) e.g. DC (StudentEmployee (Student Employee))

(← *class* New *classNameList*) e.g. (← $Class New 'StudentEmployee '(Student Employee))

(← $StudentEmployee Edit) or EC(StudentEmployee)

Sending Messages

(← *object Selector arg1 ... argn*)

e.g. (← $PayRoll PrintOut 'payFile)

(←Super object selector arg1 ...)

e.g. (←Super self Edit commands)

(DoMethod *object selector class arg1 arg2 ...*)

e.g. (DoMethod X PP $Object 'file1)

Creating, Editing,, Inspecting Instances

(← *class* New) e.g (← $Transistor New)).

propName)

(← *object* Edit) e.g EI(*myInstance*)

(← *object* Inspect) e.g (← *$Foo Inspect*)

Defining and Editing Methods

DM(*className selector*)

edit template definion of method .

DM(*className selector fnName*)

fnName is unction implementing the method.

DM(*className selector argsOrFnName form*)

e.g. DM(Number Increment (self)

((* add1 to myValue)

@myValue←(ADD1 @myValue)))

EM (*className selector*)

edit method used in className

Active Values

#(*localState getFn putFn*)

e.g. #(*37* PrintFetcher StopSmasher)

DefAVP(getFn) edit template for getFn named getFnName

args: (*self varName oldValue propName activeVal type*)

DefAVP(putFnName T) edit template for putFnName

args: (*self varName newValue propName activeVal type*)

GetLocalState (*activeValue self varName propName*)

PutLocalState (*activeValue newValue self varName*

Examples of Useful Active Value Forms

#((RANDOM 1 10) FirstFetch)

Replace me by a random number when first fetched

#(Initial (DATE))

Replace me by today's date on initialization

#((self rightPoint) GetIndirect PutIndirect)

Put and get my value from my rightPoint

Debugging

BreakMethod (*className selector*)

TraceMethod (*className selector*)

BreakIt (*self varName propName type breakOnGetAlsoFlg*)

TraceIt (*self varName propName type breakOnGetAlsoFlg*)

UnBreakIt (*self varName propName type*)

Saving Classes and Instances

(CLASSES * *classNameList*) Saves class definitions on files

(INSTANCES * *instanceNameList*) Saves named Instances on file, and instances pointed to by them.

(← $KB New *'KBName 'environmentName newVersionFlg*) Create new KB attaached to Environemnt

(← $*environmentName* Open) Open Environment for reading and writing

(← $*environmentName* Cleanup) *Save instance and class data in a KB*

(← $*environmentName* Close) *Close Environment and release attached KB*

(← $KB Old *'KBName 'environmentName*) Connect an old KB to new environment.

(← $*environmentName* Erase) Cancel an entire session

Give *CURRENTX CURRENTY* and *HEADING* default values of 0.
Give *ERASED?* a default value of T.

Now add the class variable *WINDOW*:

(ClassVariables (WINDOW NIL doc (* Turtle graphics window shared by all instances of this class.)))

This variable will cause all the turtles created to share a common window. In the next step, the function InitializeTurtleIcons will initialize this variable.

Note: Contrary to the usual Loops convention for naming variables, these variables are all in upper case because they were translated that way from the Lisp program.

Step 2. *Initialize the Turtle class.*

The function *InitializeTurtleIcons* installs the methods needed to make *Turtles* display themselves. Run this function by typing:

(InitializeTurtleIcons)

Step 3. *Translate the Turtle lisp functions into Loops methods.*

Note: For this exercise make the names of the methods be all upper case letters. This isn't the standard LOOPS convention but is simplest for the purposes of this exercise.

After you have added the instance variables to the class, the next thing to do is translate the following functions:

CENTER POINT FORWARD TURN JUMP and CLEAR

into method form. This list of functions is bound to the variable *SimpleTurtleFns*. The function *MakeLoopsMethod* is provided to do the translation. It takes the arguments *CLASS FUNCTIONNAME* and *SELECTOR*.

If the *SELECTOR* is not specified the name of the function will be used as a *SELECTOR*. For example to make a method *FORWARD* by translating the function *FORWARD*, type:

(MakeLoopsMethod $Turtle 'FORWARD)

Hint: You can save yourself some typing by using a *for* loop to repeat this step for all of the functions ie.

```
(for Fn in SimpleTurtleFns
     do (MakeLoopsMethod $Turtle Fn))
```

Step 4. *Compare the Lisp and Loops programs.*

At this point you should have a complete Turtle *class*, with all of the necessary variables and methods. Using the editor, compare it with the TURTLE *record*. What do the structures have in common?

Hint: (RECLOOK 'TURTLE) will print the definition of the TURTLE record into the typescript window.

Use PP to look at the lisp functions FORWARD and Turtle.FORWARD. What do you notice about how variables are set and fetched?

Step 5. *Make an instance of Turtle.*

You can now make an instance of the Turtle class by sending the class a *New* message. Give the name *MyTurtle* to the new instance.

(← $Turtle New 'MyTurtle)

To initialize the postion of the Turtle with respect to the window, send the turtle the message CENTER:

(← $MyTurtle CENTER)

You can try out your turtle by sending it messages, such as *FORWARD* and *Turn*. For example, sending it a *FORWARD* message should make it draw a line (as the *FORWARD* function did in the original Turtle Grahpics program). To do this, type:

(← $MyTurtle FORWARD 100 20)

Step 6. *Making some more methods.*

In this step we will convert the functions *PRETTY0, PRETTY1, PRETTY2, and PRETTY3* into methods on *Turtle.*

Use the function *MakeLoopsMethod* to convert *PRETTY0* into a method of *Turtle.* Using the browser, edit this method to examine its structure. Compare the original LISP code with that generated by the translator. Notice how calls to functions like *FORWARD* amd *TURN*, which are now methods, have been converted to messages sent by the *Turtle* to itself. Test out the translation as follows:

(← $MyTurtle PRETTY0)

Repeat this step for *PRETTY1, PRETTY2,* and *PRETTY3.* You may find it interesting to carry out the translation yourself instead of using *MakeLoopsMethod.*

Part II. *Specializing a Turtle.*

Now that you have succeeded in making an object that behaves as the original turtle did, the next step is to specialize the Turtle by creating a history turtle with the class *Turtle* as its only super class.

Step 1. *Make a class for the History Turtle.*

Use the browser to specialize *Turtle* to create a new class called *HistTurtle*with. *Specialize* is a suboption of *Add*.*

HistTurtle should have an instance variable named *HISTORY.* Make this change with the editor.

This turtle will be like the one in another of your previous exercises, in that it remembers the commands it has executed. Bound to *HISTORY* will be a list of operations. e.g. ((FORWARD 10 5) (POINT 30) ..).

Step 2. *Specialize methods in HistTurtle.*

Using the browser, add new methods each of the selectors *FORWARD CENTER JUMP POINT TURN.* These methods should use ←Super to carry out the methods inherited from *Turtle,* after putting the received event on the history list.

Hint: DM is a suboption of Add* in the browser.

Each of these methods will have the general form of the one for *FORWARD* shown below:

```
(LAMBDA (self dist width)
    (←@ HISTORY
            (CONS (LIST 'FORWARD dist width ) (@ HISTORY))
    (←Super self FORWARD dist width))
```

Be sure to test out the *FORWARD* method, before doing the others. You will need to make an instance of *HistTurtle* first, in order to check the method.

```
(← $HistTurtle New 'MyHistTurtle)
(← $MyHistTurtle CENTER)
(← $MyHistTurtle FORWARD 200 10)
```

Step 3. *Inspecting your HistTurtle.*

After you have sent your *HistTurtle* a few commands, check to see if it is remembering its moves by inspecting it. To invoke the inspector, type:

```
(← $MyHistTurtle Inspect)
```

Step 4. *Making your HistTurtle use its history list.*

In this last step, you are more on your own. This step is to add to the *HistTurtle* the cabability to use its history list in additional commands.

Implement the following methods for *HistTurtle.*

> *REDO* receives one argument specifying the how many operations to repeat. It begins that far back on the *HISTORY* list and executes the operations in the original order. If no count is passed REDO the entire history list is repeated.
>
> *RESET* clears the *HISTORY* list.
>
> *METHOD* takes as its arguments the *NAME* of a new method for the *Turtle* class and the number *N* of preceeding events that should be taken from the *HISTORY* list. It constructs a new method containing these steps and installs it in the *Turtle* class.
>
> Hint: These methods could use a function that extracts a copy of the last N events from the history list. Write a recursive function that takes as arguments the *IIISTORY* list and the number N of events requested. Since the *IIISTORY* list has the most recent event is nearest the head of the list, your list of events will come out in the reverse order. Creating the sequence by recursive CONSing should reverse that order again.

Specialization is one of the important capabilities provided by objects. What would it have taken to create a *HistTurtle* using records alone?.

BROWSERS REVISITED

In Day 2, the first thing you learned about LOOPS was the notion of the ClassBrowser. At this point, after completing lots of exercises with the aid of the ClassBrowser, you are aware of the central role it plays in using LOOPS.

Today, we'll do three things to increase your understanding of browsers and to practice using your LOOPS skills.

- understand how the browsers work by using and poking around in other system browsers,
- build a browser that will make the standard class browser easier to use for very large systems, and
- give our new browser class some "advanced features".

For the first exercise today, these notes will be relatively detailed to lead you through understanding browsers. For the second, an outline will be given of how to construct the browser we will describe. For the third, only a description of what we want our "advanced" browser to do will be given. Consider the last exercise as a kind of self test on your current LOOPS abilities.

Other LOOPS Browsers

In addition to the ClassBrowser, there are several other browsers that come as a part of our LOOPS loadup.

The Supers Browser

To look DOWN the class hierarchy, we have been making heavy use of the ClassBrowser. To look UP the class hierarchy, you may use the SupersBrowser.

```
EX:     (<leftArrow>New $SupersBrowser Show 'LuxuryGoods)
```

Play with your new SupersBrowser to be sure you are

understanding how such functions as WHEREIS are working here.

The Meta Browser

To look from a class to its meta class, then on up the "meta class link", you can use the MetaBrowser.

EX: ((leftArrow>New $MetaBrowser Show 'LuxuryGoods)

Play with this as well. One particularly interesting thing to figure out is highlighted by doing WHEREIS on the Method New in objects in your new MetaBrowser.

The Instance Browser

Besides the one mentioned above, there is also a system browser for instances. There is also a fairly well developed instance browser for named instances that one of your instructors has built (jon). We will not be looking at either of these in todays class, but after the class, you may want to poke around in the system version, and if you want to poke in jon's version, he will make it available.

How Browsers Work

All of the browsers in LOOPS share some top level functionality. Note what that functionality is.

- instances of browsers amount to windows with some kind of lattice drawn in the window where nodes in the lattice are LOOPS objects, and the lattice links are some link between LOOPS objects and
- mouse operations can be done on the nodes in the lattice (ie the LOOPS objects represented by the named nodes).

In this section we'll get a top level perspective on how this functionality is achieved.

```
EX:     (<leftArrow>New $ClassBrowser
                Show 'LatticeBrowser)
```

Here you can see that all the browsers have a common linkage back to the LOOPS object $LatticeBrowser.

The important slots of a browser for us are

1. the CV LeftButtonItems,
2. the CV MiddleButtonItems,
3. the CV LocalCommands, and
4. the method called GetSubs.

The CVs hold information for setting both the items that will appear in the pop up menus of the browser, and for information that will point to the method to process a selection from those menus. If the menu item is a simple (ie, no sub menu associated with it) then the form that of the CV entry is

```
(<menuItemThatWillAppear>
 '<nameOfMethodToProcessIt>
  <descriptiveStringToAppearInPromptWindow>)
```

If the item has an associated sub menu, then the structure is a little more complicated to allow embedding of the sub menu information as well as default information.[1]

---

[1] As you've already seen, when you see a stared menu item, it means there is an associated submenu. If you button the stared item with the middle button, you get the sub menu, but if you button it with the right button, you'll get a default action.

EX: poke around in $LatticeBrowser and $ClassBrowser to see how the CVs for left and middle button items look

LocalCommands

The CV LocalCommands tells the browser processing mechanism whether the method associated with a menu item is understood by the browser itself or by the object you have buttoned. If a method name is contained in the CV LocalCommands, then when you button a menu item, the message associated with that menu item will be sent to the browser itself with two arguments

1. obj - a binding to the object that you have buttoned in the browser (ie binding to the POINTER to the LOOPS object) and
2. objName - a binding to the name of the object you have buttoned.

In addition self is bound to the instance of the browser you are working in.

If on the other hand the menu item you chose is NOT contained in LocalCommands, then the browser mechanism will send the associated message to the LOOPS object you have buttoned, with self bound to the LOOPS pointer for that object.

EX: poke around in the LocalCommands of $ClassBrowser

From the above you can see that if a browser item is a local command, then the method that responds must be of two arguments: obj and objName.

EX: look at the local command GetSubs of $ClassBrowser

1. When To Make Something Local

Although not something that LOOPS imposes on you, I find the following programming disipline useful when constructing browsers.

> In addition to any other considerations, make any menu item which interacts with the user be a local browser command. Make methods which run in the objects themselves do no interactive operations.

If you follow this disipline, then you will factor cleanly any LOOPS system you build into "program control component" and "user interactive component". I find such a factoring useful especially for debugging purposes.[2]

GetSubs

The GetSubs method of a browser does just what the name implies, it finds the "sub objects" of the current object. Note "sub objects" can be defined however you want to define it.

- ClassBrowser - look down the class hierarchy
- SupersBrowser - look up the class hierarchy
- MetaBrowser - look along the meta link
- etc.

---

[2] There are special considerations when you want to interact with LOOPS under a mouse process. See the INTERLISP manual, or talk to me about this if you want to someday build "interactive" operations under from a browser.

The important point is that $LatticeBrowser has methods which send a GetSubs message to self. The methods in $LatticeBrowser do a lot of the work of any browser. But they use a method that we may easily specialize for our own purposes.

EX:
```
do WHEREIS
on $ClassBrowser
and find how many of its
methods are up higher
in LatticeBrowser

try to find an example
of GetSubs being used
in some method in
$LatticeBrowser
```

Building A KeyClassBrowser

Suppose we have a very large set of LOOPS classes that is hard to view all at one time in a browser.

EX: (<leftArrow>New $ClassBrowser Show 'Object)

Now that you believe that you really do sometime have TOO many LOOPS classes for getting a gestalt view, lets decide how to build a browser to help us.

Here is one possibility to accomplish what we want.

- a browser that will only show those LOOPS classes which we have designated as "Key Classes"
- but will allow us to "expand" the view to show a normal class browser starting from any key class we choose in our KeyClassBrowser

Designation of KeyClass

The first thing we have to do is decide how we are going to mark, or keep track of key classes.

```
EX:     add a new CV to
        $Object called
        KeyClasses
```

Now make three new methods for $Class:

```
EX:     MakeKeyClass
          makes self be a
          key class by
          using PutClassValue

        UnMakeKeyClass
          removes self
          from the list
          of KeyClasses

        KeyClass?
          a predicate function
          to test if self is
          a key class
```

Now lets make some of the objects in the class hierarchy to be on the list of KeyClasses using the functions we just made.

```
EX:     pp(KeyClasses)
          this is a list i have
          provided. You are to
          make each Class on the
                                  3
          list into a key class
```

Now lets make a specialization of ClassBrowser.

---

3 Remember that sending a message to Object for example doesn't make sense. You can however send a message to the pointer $Object.

```
EX:     specialize $ClassBrowser
          to a new Class called
          $KeyClassBrowser
```

All that we have to do now to get our new browser to make an instance and display in the way we want is to specialize the method GetSubs from $ClassBrowser.

```
EX:     copy the method
        GetSubs from $ClassBrowser
        to $KeyClassBrowser

        look at the method and decide
        what we have to do next
```

GetSubs in $ClassBrowser sends a message off to obj to fetch the subs of obj.[4] Lets simply change the name of the method that goes to obj, then go back and create that method.

```
EX:     do EM on GetSubs
        in $KeyClassBrowser

        change SubClasses
        to SubKeyClasses
```

Now we have to make sure that obj will understand the method SubKeyClasses. What we want is a method that will

- fetch its subs in the class hierarchy (it can do this by issuing a SubClasses message to self)
- collect the subs that are KeyClasses

---

[4] I think they should have called it "OBI-ONE".

- for those subs that are not key classes, continue searching down the class hierarchy looking for key classes, stopping when one is found

Be sure you understand what i am saying in this logic. It may be helpful to draw pictures of the class hierarchy.

EX: DefMethod a new
method in $Class
called SubKeyClasses
to accomplish the above

test your method to
make sure it works

If you get just plain stuck in this, then look at the function jonsClass.SubKeyClasses in your loadUP.

CONGRATULATIONS!

EX: (<leftArrow>New $KeyClassBrowser Show 'Object)

Now most of the battle is done. But there are still a few things that are needed in our KeyClass browser.

EX: make any instances of
$KeyClassBrowser come up
with "Key Class Browser"
in the title bar instead
of "Class Browser"

EX: now add new
functionality to
the $KeyClassBrowser
by adding left button
functions:
MakeRegularClassBrowser
MakeSupersBrowser

MakeMetaBrowser

this involves adding
things to LeftButtonItems
and creating new methods
for $KeyClassBrowser

Question: WHY was building this new browser such a "relatively" easy operation using LOOPS?

Advanced Features For $KeyClassBrowser

In the regular notion of browsers, there is no real idea that browsers can "cooperate" with each other to present a consistent view of the LOOPS world to the user. In this section, i'll suggest ways you might further develop your $KeyClassBrowser so that you use it not only to get a gestalt view of all of the LOOPS objects, but also to "manage" the view of LOOPS objects (as shown in OTHER browsers) that is presented.

THIS SECTION IS OPTIONAL!

One basic problem with browsers is that you have to constantly run about closing old browsers. The reason is that if you have say two browsers up, and you alter one of them is some way (i mean alter the class hierarchy (eg)) in one of them, then you could be in trouble if you try to use the other one. Suppose you destroy a class in one, then try to specialize it in the other.

Solution: make your KeyClassBrowser a sort of index you use into the others. Only pull up a class browser by using the KeyClassBrowser AND keep track of all the browsers "spawned" from the key class browser. Any time you generate a NEW ClassBrowser, close the OLD one.

Further modification: Instead of just killing it, close it, cash its pointer on a stack, and build some functionality that will allow you to "pop back up" to it.

Another mild problem is doing WHEREIS using a ClassBrowser, and the object that has the method you want to find is not IN the current browser. Of course, the browser will let you know in the PromptWindow where the object is, but sometime you want more.

Solution: specialize WHEREIS so that if the object you want to highlight is not in the current browser, that a SupersBrowser will pop up AND blink the appropriate object for you.

This could go on and on. There are MANY useful things that could be done to extend the notion of cooperative browsers. As you think more about this, even if you don't implement your idea, please tell me about it.

X

Darken @ 98%

Filed on {INDIGO}<Loops>LoopsCourse>ExerciseRules.bravo
Edited: MJS November 10, 1983 5:00 PM

Editing and Debugging RuleSets

by the Loops Design Team

Daniel Bobrow, Sanjay Mittal, and Mark Stefik

Purpose of these Exercises

This set of exercises explores the programming and debugging facilities for using rules in Loops. In this session we will extend the behavior of a rule-driven "player" of the *Truckin* game. The techniques introduced in this session will provide experience in representing and debugging knowledge representation that will be needed in the later sessions.

Notation

In the dialogs below, regular font is used to show what the system types, and bold font is used to show what the user types.

Preliminaries

Go to your workstation for the course and LOGIN according to the instructions in the handout titled "How to Start Loops".

Part I. Basic Debugging Tools

This exercise is to use the auditing, tracing, and breaking facilities of the rule language to understand the rules behind the behavior of an automated player of *Truckin* called *Traveler.* A *Traveler* is a class for players that commute between *UnionHall* and *AlicesRestaurant*, stopping for gas and *WeighStations* along the way.

There is a listing of *Traveler* in your course notebook. You may want to refer to it as we continue.

Step 1. Getting Started

In the discussion which follows, we will use the term *gameMaster* to refer to that which runs the game. Depending on which version of *Truckin'* is being used, the gameMaster may be one or more programs, databases, and processes, or even more than one computer. Some variations on this will be discussed later in this class. For the purposes of this discussion, it is convenient to refer to it all as simply the gameMaster.

The first step is to set up a new *Truckin* simulation. Do this as follows:

(← $Truckin New)
(← PI BeginGame)

PI is an abbreviation for *PlayerInterface.* It's short, since you may type it a lot.

The gameMaster will then ask you in the prompt window to choose a kind of player. Use the mouse to select *Traveler* in the menu next to the prompt window, and type the name "Trav1" for the name of the driver. The gameMaster will then ask you for the kind of truck. Select *FordTruck* (or any other brand that you prefer). Then the gameMaster will ask you about gauges on *Trav1* and it is recommended that you choose DEFAULT and that you place the fuel gauge to the right of the game board.

When the gameMaster asks you for the second player, select "No" with the cursor.

The gameMaster will put up an inspector window of game parameters for you to set. These parameters control such things as the length of the game, the number of bandits, and so forth. You can set the values of the parameters by selecting a parameter and using the *PutValue* option. To see a description of a particular parameter, select the value of a parameter and select the *Properties* option. It is recommended that you set the value of the parameter gameDuration to 60 for this exercise.

When you are finished, click DONE in the menu next to the inspector. The gameMaster will then put up a bar chart for you player's cash.

Then the simulation will begin. *Trav1* will begin commuting back and forth on the board. If you depress the CTRL key, the *Traveler* will pause in its travels. The *fuel* gauge should go down at each move, and the *cashbox* bar chart will change whenever the *Traveler* spends or receives money. Every few turns, the bandits (*Bonnie* and *Clyde*) will make their moves.

The typescript window will type a rule before each move of *Trav1*. This is because the *Traveler* was compiled in the system with rule tracing turned on (compiler options T).

Step 2. Using the Rule Executive

Find the menu labeled *Interrupt* at the top center of the display. This menu should have an entry for the player *Trav1*. Clicking *Trav1* in this menu will invoke the Rule Exec window. Do this now. (It may take a few seconds for the request to be noticed -- it waits until the current player finishes its move.) The player interrupt will disappear.

The Rule Exec will prompt you with "re:". ***Notice that time keeps marching on, even though Trav1 is suspended.*** You can resume the simulation by typing "OK" to the rule exec as follows:

re: **OK**

Do this, and click *Trav1* again, to get the feel of using the Rule Exec to interrupt execution. Inside the rule exec, you can type expressions in the rule language. For example:

re: **destination**

will cause the current destination of your *Travl* to be typed. You can look at the values of other variables (such as *truck* or *stoppingPlace*) as well. You can also use compound terms as in

re: **truck:fuel**

The number you get back should correspond to the value shown by *Travl's* fuel gauge. You can also look at class variables, as in

re: **truck::MaxFuel**

You can make adjustments to your player, as in

re: **truck:cashBox←20000**

This should cause the *cashBox* bar chart to get updated immediately. (Of course, this would be cheating if done in a RuleSet, since your player should only acquire *"cash"* by selling merchandise.) Lastly, you can send messages, as in

re: **(← self Show)**

Note: Be sure to use rounded parentheses in these expressions, not brackets. In the Rule langauge, brackets are used to control precedence of operators, and parentheses are used for function calls and messages.

or, if you are not at *UnionHall*,

re: **(← truck:location:prev PP)**

This illustrates that *self* is set to the work space, and also shows off the *Show* method that *Traveler* inherits from *Player.*

Step 3. Changing Compilation Options

One way of changing compilation options is to send a *TurnOn* message to a player class. For example,

re: **(← $Traveler TurnOn 'A)**

To edit a RuleSet, enter the rule exec and create a browser for players as follows:

re: **(Browse $Player)**

Using the browser, select *Traveler* (with the middle button). Using the left mouse button, select EM* (for EditMethod), and you will be shown a list of the Methods for *Traveler*: *BuyGas*, *FindStoppingPlace*, *GoToStoppingPlace*, and *TakeTurn*. Select the *FindStoppingPlace* method. The TTYIN edit window will come up automatically -- ready for you to examine or make changes to the RuleSet.

WARNING!!! Do not use "Reprint" on a RuleSet. This will do terrible things to its readability. The rule editor that your are using is our 'StopGap" version. In the next version of the Rule Language, we will use a structural editor and a more Lisp-like notation.

Verify that the RuleSet has auditing turned on. If not, change the compiler options declaration to read as follows:

Compiler Options: A ;

When you have finished editing the RuleSet, type ↑X to exit. A pop-up menu will then appear by the cursor. The *Help* option in the menu can be used to see a description of the other options.

Most of these options are intended for use in debugging the Rule Compiler. You may find it interesting to use them to examine the LISP code currently generated by the rule compiler under different compiler options.

To compile the rules and quit, select OK.

Step 4. Asking *Why*

Leave the rule exec and let the simulation run for a while. (The audit trail is created incrementally, as the program runs, so you need to wait for it to complete another turn.) Then interrupt *Travl* and use the audit trail to answer "why" questions as follows:

re: **why stoppingPlace**

Using the rule exec in combination with auditing is a handy way of discovering which rules were responsible for particular decisions. (Perhaps this should be called *how* instead of *why*.) Try asking *why* for other variables such as destination. You can ask *why* for compound variables as in

re: **why truck:fuel**

in this case you should get back the message "Rule not known." because the *fuel* variable is not set by a RuleSet compiled with auditing turned on.

If "Why" is typed without arguments, the Rule Exec uses the previously entered expression. For example

re: **stoppingPlace**
(AlicesRestaurant 123.45)
re: **why**
--- that is, why stoppingPlace

IF (Distance destination)<=.Range1 · (RoomToParkP destination)
THEN stoppingPlace←destination;
Rule 4 from FindStoppingPlaceTravelerRules

Step 5. Suspending and Waking *Truckin'*

Another menu at the top of the screen is labeled *GameControl.* This menu has options for suspending, killing, and waking the current simulation. The gameMaster consists of several processes for the different players, the clock, and scheduling. If you enter a Lisp break, or are editing, you may want to stop the frenzy of activity on the screen so that you can work. This menu is for that purpose. Practice suspending and waking the gameMaster. (But don't *kill* the game yet!)

Step 6. Breaking on Rule Invocation

In this section we will see how to step through the execution of a RuleSet. Using the Browser and rule editor, change the compiler options for the *FindStoppingPlace* method of *Traveler* to read as follows:

Compiler Options: BT;

This indicates that the rule should "break" to the rule exec whenever a rule is tested or executed. Exit the rule editor and rule executive and let the simulation run.

As the simulation continues, you will see a rule print out in the Typescript window, and then the rule executive will pop up. By typing "OK" to the rule exec as in

re: **OK**

You may find it useful to suspend the game while you are here.

You can step through the execution of the rules. Note that the break occurs *before* executing the left or right hand sides of the rules. Step through the execution of *FindStoppingPlace* a few times to see how it works. You may want to use this feature for some subtle case of debugging particular RuleSets.

When you are tired of typing ok, try

re: **(← $FindStoppingPlaceTravelerRules Off 'BT)**

and let the *Traveler* run. You may have to type OK a few more times, until the Lisp interpreter let's go of the function with the "break code" in it.

Step 7. Debugging with Gauges

In this section we will see how to create extra gauges to help with debugging. We will begin by putting a gauge on *stoppingPlace.* Enter the User Exec by depressing CTRL-LeftShift.

The User Exec is an alternative to the Rule Exec. The User Exec expects Lisp expressions and provides the Interlisp-D environment (e.g., the history list). The Rule Exec expects rule language expressions and provides rule facilities (e.g., why questions). The User Exec can also be entered by typing UE as a command to the Rule Exec.

To create a gauge on *stoppingPlace,* type:

```
← (←New $LCD Attach $Travl 'stoppingPlace)
← OK
```

For obscure and temporary reasons, the ←New syntax doesn't currently work in the StopGap Rule language.

The *Truckin' Manual* describes more automatic ways of installing gauges on the instance variables of a player.

Step 8. Listing RuleSets

To get a hardcopy listing of the RuleSets associated with a class, use the function *ListRuleSets.* The listings will appear on the local printer. Get one of the course instructors to show you where it is. To make a listing, type the following to the user exec.

```
← (ListRuleSets 'Traveler)
```

Part II. Creating A New Player

The purpose of this exercise is to practice making a new kind of *Player* called a *BigMac,* which is a revised version of the *Traveler.* A *BigMac* is a class of player that commutes between two of the *AlicesRestaurants* in the simulation. *A BigMac* (a hungry driver of a Mac truck) will presumably eat a lot, visiting *UnionHall* only when it runs out of money and gets towed there. A *BigMac* always drives a *MacTruck.*

Step 1. Setting Up

You may want to restart the game before continuing with this exercise. Kill the game using the GameControl menu. Using the mouse, close a few of the windows at the top of the screen until you find the Loops Logo (Saturn). Depress the left mouse button, and a menu should pop up. Select the *SetUpScreen* option to restore the screen to its original state. You may want to create a new browser for players as:

```
← (Browse $Player)
```

To create a new player that is a specialization of *Traveler* use the *Specialize* option in the Browser. When you are prompted for a class name in the prompt window, type:

Class Name: **BigMac**

The Browser will now indicate a new class for *BigMac.*

You may need to shape the browser window to see *BigMac.*

Step 2. Replacing a Method

To insure that *BigMac* always drives a *MacTruck*, we need to replace its *SelectTruck* method. Initially, the method for *SelectTruck* is inherited from *Player*, and prompts for a truck.

To replace this method use the EM! option in the Browser.

Either approach will put you into a Lisp editor on the Lisp function that implements the method. Edit the function so that it just returns *'MacTruck*.

Step 3. Adding an Instance Variable to the WorkSpace

To change the behavior of *BigMac*, it will first be necessary to add an instance variable to record the *nextDestination*. Edit *BigMac* with a Lisp Editor of your choice. (Hint: You may access it through the *Edit* option of the Player Browser.) When you have added the instance variable, that portion of *BigMac's* definition should look as follows:

(InstanceVariables (nextDestination NIL doc (* Next destination. A different AlicesRestaurant.)))

You may want to add some documentation to *BigMac* itself. If you do, the relevant portion of *BigMac's* definition should look approximately like this:

(MetaClass PlayerMeta
doc (* A Player that commutes between AlicesRestaurants, eating hamburgers.)
Edited: (edited: *MyName* "21-Feb-83: 15:31"))

BigMac inherits other instance variables from *Traveler*, but they don't show in the source because they are not introduced at this level of the inheritance lattice. To see them, you can prettyPrint a summary of it through the browser (using the *PrintSummary* option).

Step 4. Specializing the Rules of *Traveler*

In this section, you should go back and edit the rules for the *TakeTurn* method and modify them for a *BigMac* player. In debugging your rules, the techniques introduced in Part 1 for auditing and breaking RuleSets and adding gauges, will be of use.

Hints:

1. The course handout "*Truckin'* Query Functions" describes a set of functions for accessing information in the world of the *Truckin'* simulation. These functions will be discussed later in the course, but you may find it helpful to browse this document when you are trying to understand the *Traveler* rules.

2. You will probably want to replace the *TakeTurn* method of *Traveler* with one specialized for *BigMac*. You can use the EM! option in a player browser to do this.

3. *BigMac* should initialize his *destination* and *nextDestination* on the first call. In the method for *TakeTurn*, the following rule may be a useful substitution for some of the existing rules:

```
(* On first call, initialize destination and nextDestination.)
IF ~destination
THEN alices←(RoadStops 'AlicesRestaurant)
     destination←(CAR alices) direction←(DirectionOf destination)
     nextdestination←(CADR alices);
```

This rule assumes that *alices* is defined as a temporary variable of the rule set.

4. In addition, a new rule like the following may be appropriate in the method for *GoToStoppingPlace*:

```
(* Switch destination and nextDestination when you arrive.)
IF truck:location=destination
THEN temp←destination
     destination←nextDestination
     nextDestination←temp
     direction←(DirectionOf destination);
```

5. If you have trouble with the behavior of *BigMac*, use the auditing, breaking, and gauging facilities you have learned about to understand the behavior.

Step 4. Saving the Rules on a File

To save your RuleSet on a file, type the following to a User Exec or at Top-level lisp.

(FILES?)

Lisp will ask you whether to save various instances and functions. Type BIGMAC for all of the things that you want to save. Type] (a right square bracket) for all of the things that you don't want to save, such as *Bonnie, Clyde,* and other things not related to your file. You can type LINEFEED (LF key) to mean *same as previous.*

Then make a file containing your *BigMac* player as follows:

MAKEFILE(BIGMAC)

This file can later be retrieved by typing

LOAD(BIGMAC).

[Optional] Part III. How Auditing Works

This section is intended for those who finish their *BigMac* player early, and would like to learn more about how the auditing works in the rule language. This section is a tour of the auditing facility.

To see how an audit trail works interrupt your player with ↑F. Use the left mouse button to

select the top item of the "trace back" menu to the left of the rule exec window. This will create an inspector for the workSpace.

If there is no trace back window, type **(← self Inspect)** instead.

Select the value of the destination variable with the left mouse button. (It will turn black). Depress the middle mouse button and a menu should pop up. Select the *Properties* option. This will spread out the properties of destination. The interesting part of this is the *reason* property of destination. The value of the *reason* property should be an instance of a *StandardAuditRecord.* If you inspect that record, you can "inspect" all the way to the *rule object* which prints out when you type the *why* question.

The Lisp code generated for the RuleSet must not only save values, but must also create the audit records and link them to the *reason* properties when it executes. To see this auditing code, you may want to invoke the rule editor on the *FindStoppingPlace* method of *Traveler,* exit the editor with ↑X, and select the EF menu option to examine the Lisp code. You should be able to find the code that makes the audit trail. Say DE to the Lisp TTY editor if you want DEDIT. After looking at the code, exit the editors.

The next step is to look at the Audit Class declaration for the RuleSet. Select the *EditAllDecls* option in the Rule Compiler menu. This will put you in the editor again, except that several additional "default" declarations will now be made explicit. In particular, you can now see the declaration for the audit class. Exit the rule editor.

To see where the meta-assignment statement for saving the rule in the audit record came from, edit the class *StandardAuditRecord.*

A *StandardAuditRecord* saves only a pointer to a rule. The specification of what to save in an audit record is made by meta-assignment statements - either in the class for the audit record, or in the RuleSet. The class for the audit record must have instance variables for all of the values to be saved. This facility can be used for experimenting with belief revision systems. See the Rules Manual for details. This material is beyond the scope of the 3-day Loops course.

Filed on {INDIGO}<Loops>LoopsCourse>ExerciseCompetition.bravo
Edited: MJS August 3, 1983 7:37 PM

Knowledge Programming

by the Loops Design Team

Daniel Bobrow, Sanjay Mittal, and Mark Stefik

Purpose of these Exercises

These exercises provide experience in building a small knowledge system in Loops: a *Truckin'* player. This *Truckin'* player will be your "entry" in the knowledge competition at the end of the course. The exercises will help you to prepare your player, and tell you how to enter the knowledge competition, and what to expect.

Notation

In the dialogs below, regular font is used to show what the system types, and bold font is used to show what the user types.

Preliminaries

Go to your workstation for the course and LOGIN according to the instructions in the handout titled "How to Start Loops".

Part I. Specializing an Existing Player

This exercise is to augment the knowledge of a modest knowledge system incrementally. For this purpose, we have provided a sample *Truckin'* player as follows:

Peddler (50 rules): - a player created for the second Loops course with a structure that makes it convenient to create *evaluation functions* that prioritize player decisions.

Find the listing of *Peddler* in your course notebook. You may want to refer to it as we continue.

Step 1. Creating A Player

Create a player that is a specialization of *Peddler.* As in the *BigMac* exercises, either a browser to create your specialized player.

You will need to name your player. Here are some names that have been used before:

HappyHauler	PeterLorry
Routier	Maverick
SafeSam	
SpendThrift	
Toyota	
TravelingSalesman	

When you pick a name, register it with us and we will be alert for name conflicts before the knowledge competition.

Step 2. Hints About *Peddler*

Peddler is a reasonably good player, but its knowledge base lacks some fine points. Here are some known weaknesses which you may choose to remedy. (You may discover other weak spots.)

1. *Peddler* uses "rating RuleSets" to pick *Producers*, *Consumers*, and stopping places near *AlicesRestaurant*. But it shows little flexibility in its selection of which *AlicesRestaurant* to go to at the end, or which *GasStations* to visit. You may want to extend the rating idea to these other decisions.

2. *Peddler* looks for low prices when it picks a *Producer*, rather than computing its probable profit (by looking ahead for a possible *Consumer*). Late in the game, it may even buy goods for which there are no attractive *Consumers*.

3. *Peddler's* gasoline logic is poorly organized. When *Peddler* is low on gas, it can completely forget about other things, like stopping at *WeighStations* or getting to *AlicesRestaurant*. If *Peddler* gets in a tight loop between a *Producer* and a *Consumer*, and there is no gas station in-between, it will go back and forth content on making a tidy profit but forgetting to go outside that range for gas.

4. *Peddler* avoids *PerishableCommodities* and *FragileCommodities*, because it lacks knowledge for dealing with them. It needs to know about *RoughRoads*, and that perished commodities can spoil, and that damaged merchandise must be taken to the dump to avoid fines. There is a lot of money to be made on such merchandise.

5. *Peddler's* rating schemes using a numeric method for combining factors. You may choose to use a symbolic approach to combine the factors more rationally.

Step 3. Strategy for Preparing Your Knowledge System

Here are some suggestions for building and debugging your player:

Keep it simple!

You don't have a great deal of time (6-8 hours) to build your system, and you will be learning Loops (and perhaps coping with bugs) at the same time. In the previous Loops

courses, the most successful entrees were created by people who fixed bugs and made small improvements to the players rather than starting a complete re-design.

Ask Why!

During the first Loops course, people sometimes used only conventional programming techniques and struggled to find bugs that the auditing facilities could easily have pinpointed for them. Part of what makes rules special are the special facilities for auditing rules. Use this! It will help you pinpoint shortcomings in your rules! Also use the gauges and breaking and tracing facilities. These tools for understanding your player really help.

Save your player frequently!

Computer systems are subject to occasional crashes. Think conservatively. Save your files every 15 - 30 minutes, and make listings every hour or two. Be sure to type **(FILES?)** before doing your MAKEFILE. (Or use the lisp function CLEANUP).

Try a crowded board!

Some important phenomena only show up during a crowded game, as will be the case during the knowledge competition. By analogy with ecological systems, the stiffest competition for a player is often itself. Identical players, like members of the same species, compete for the same ecological niche. Be sure to exercise your player on such a board before the competition. A good way to test your player is to compete it against several copies of itself and several *Peddlers*.

Try different trucks!

Trucks have different storage capacities, speeds, ranges, etc. The performance of your rules is likely to be different for the different classes of trucks. Use a browser to determine the characteristics of the different trucks and try your player with different trucks so that you can understand how your rules interact with the truck characteristics.

Lisp is legitimate!

If you find that the rule language is too confining for you, you can always express some of the knowledge in a Lisp function, called directly or installed as a method on your player or some other object. This is completely "legitimate" in *Truckin'*, and may be appropriate since the *StopGap* version of the Rule Language has a somewhat awkward interface. You may also find it useful to define some new classes of Loops objects.

Step 4. Technical Note -- Creating, Copying, and Installing RuleSets

In working on your player, you may want to define some new RuleSets as methods. To define a RuleSet as a method, use the DefRSM suboption under DM* in the browser.

You may want to create a new version of one of the Rating RuleSets. One way to do this is to use the CopyRules method of RuleSets. For example, to create a rating RuleSet of GasStations for a player named *RoadRunner*, you may proceed as follows:

← (← $PeddlerRateConsumers CopyRules RoadRunnerRateGasStations)

We recommend that you use your player name as a prefix on your RuleSetNames, to avoid name conflicts during the knowledge competition.

Step 5. Save Your Player Periodically

After naming your player, inform the file system by typing:

← FILES?)

to the Lisp Executive. When it prompts you for a file for the class of your player, use the player name for the file name. When you have finished with this, create a file for your player:

←MAKEFILE(*MyPlayerName*)

All computers and systems are subject to failure. Be conservative. Save your player every half hour or so, to avoid losing your work.

Part II. Entering the Knowledge Competition

The knowledge competition is the usually exciting finale to our course. Here are the things you need to do to participate.

9:15 am ---

1. *Register and file your player.* Your player should be ready, loaded in your machine, and backed-up in the file system. You should see the course instructors to register your player name, and the name of a driver for it. (This is to insure that the names are unique and short enough for the competition.) Make sure that your player loads correctly from a file. You should also make sure that you have saved everything your player needs (using *FILES?*), and that all rule tracing and breaking has been turned off.

9:30 am

2. *Competition debriefing.* Before starting the competition, we will spend a few minutes talking about the idea of a knowledge competition, and will ask each group to spend 1-2 minutes talking about their player. One person in your group should be ready to tell us the main ideas you are trying.

10:00 am

4. *Starting the competition.* We will then bring the distributed game master on line and tell you the network address of the *PostMaster.*

Note: The game clock will start so that the competition starts *automatically* at 10:15. If you fail to complete the following instructions on time, you can still enter your player, but the competition will have already started and you will be behind.

5. *Go to your workstation.* Type:

(SlaveTruckin)

This will start the "slave" to the game master, and will also run the competition with the "simulation display" turned off. The game master will then ask you the following questions:

Name of your machine: ***JonesMachine***
What is the address of the PostMaster: ***12365***

For the machine name, you should append "Machine" to your last name. For example, if your name is "Jones", then you should enter "JonesMachine" as above. The address of the Postmaster will be the number that we gave you in step 3.

You will then be prompted to start your players in the usual way. Use your registered *driver* name. The promptwindow will begin to print messages about the competition starting soon.

6. It's a lot more fun to watch the game with everyone else, so you can hear the cheers and groans and gossip as everybody watches and comments on the performance and luck of the road!

Good Luck!

SYSTEMS BUILT ON TOP OF LOOPS

Thus far in the class, you have been doing exercises to build your INTERLISP/LOOPS expertise. Today, we'll be looking at several systems which have been built on top of LOOPS.

The first system we'll look at is the SIS tool, a tool constructed to allow some of the generic ideas of browsers to be applied and extended. Then we'll go on to look briefly at a medical diagnosis system called MDX/MYCIN, a system built following the OSU-AI paradigm of diagnostic expert system design, and which is implemented on top of both LOOPS and the SIS. Finally, we'll end the course by looking at some of the capabilities of CSRL, a language which has been developed to allow the expression of diagnostic systems in a straightforward way.

THE SIS TOOL

The Structured Instance System is a LOOPS tool which has been designed to extend the notions of "environment control" to the realm of named instances. One of the important design goals of the system defined ClassBrowser is that the browser should be an aid to the LOOPS user for organizing and dealing with the world of LOOPS classes she is developing. The SIS applies that same top level idea to LOOPS named instances.

The SIS has its root in the distinguished LOOPS class called StructuredInstanceObject. At the class level it includes a tool called the StructuredInstanceObjectClassBrowser, a specialization of the normal ClassBrowser. Going to the instance level, the SIS includes a browsing tool called the InstanceBrowser that allows manipulation of the relations between the instances of a subclass of StructuredInstanceObject. Finally, the SIS includes another graphical tool for showing the current setting for a user chosen IV of a collection of instances, the ValueLattice.

StructuredInstanceObjects

The SIS includes acts on any LOOPS class which has StructuredInstanceObject on its supers list. There are two properties that such LOOPS classes have that are important to understand.

- First, any instantiation of such a class is a named LOOPS object that can be viewed (along any number of user defined relations) in relation to other members of the same class. These relationships that may exist between the instances of the StructuredInstanceObject are called "InstanceLinks".

- Second, there is a special kind of IV that can be defined for any SturcturedInstanceObject: the "UserIV". The notion here is the SIS system has been constructed to allow knowledge engineers to build systems for potential end users. At the end user level, many IVs are of no interest at all - they may exist in a system only in terms of "making the system work", but the end user may not need to know of their existence. To hide such IVs from the end user, the SIS distinguishes a special sort of IV (the UserIV) for express interactions that may be necessary with the end user.

The StructuredInstanceObjectClassBrowser

The starting point for jumping into a number of SIS defined objects is the StructuredInstanceObjectClassBrowser.

EX: do
    ssi]

The function ssi brings up the a normal class browser starting from the root SturcutredInstanceMeta, and a StructuredInstanceObjectClassBrowser starting from the root StructuredInstanceObject.

Within the SIS system, there are many new menu items that you find defined at each of the three level of browser type tools. Whenever you see a menu item called "StardardFunctions*" you can expect to find a submenu of more normally defined browser operations.

EX: mouse the object $organization
with the left button, then
mouse the StandardFunctions*
with the middle button and
look at the items in the submenu

do the same sort of operation
with the middle mouse menu item
StandardFunctions*

In addition to the standard functions, there are a number of new operations that you will find too. The middle mouse sub menu for example, includes extensive operations for storing and retrieving objects from secondary storage.

EX: conn (FLOPPY)
dir()

now middle mouse $organization
and select Save* with the middle button
the select the item SaveClassAndInstances

now do dir() again

now middle mouse $jonsTopLevel,
select LoadSubObject* with the
middle button, and then select
LoadSubObject

this will bring a menu from which
you could select a file to load that
would restore any object inferior to
jonsTopLevel that has been stored

mouse any button outside the selection
menu to tell the system you don't want to
load any file right now

On the left selection menu for the StructuredInstanceObjectClassBrowser, you'll find a item BrowseInstances that will bring up an InstanceBrowser on the current instances of the object buttoned.

Before going on to look at the IB, you may want to play a little bit with some of the other menu items for the StructuredInstanceObjectClassBrowser, but do NOT do CompileMethods (from the middle button menu).

The InstanceBrowser

The heart of the SIS facilities is the InstanceBrowser.

EX: select the left button item
BrowseInstances on $organization

when it asks for your starting list
enter]

when it asks for the relation you want
to show select (partOfSuper partOfSub)

place the IB in the normal way

You now have up an InstanceBrowser for the instances of $organization as viewed along a certain relation. A design goal for the SIS was to allow the user many facilities for graphically editing the relation between his instances. These graphical facilities are located on the middle menu items for the IB.

EX: mouse $xerox with the middle button
select MoveInstance

(at this point the IB goes into
"gather mode" to gather up
all the supers you want for
the new location for $xerox.)

follow the instructions you will see
in the PromptWindow to move $xerox
to have a super of $battelle and
no subs

select MoveInstance again to
put $xerox back where it was
to start with.

EX: experiment with the other
middle button items in the IB

The IB can be brought up for any define relation for a StructuredInstanceObject.

EX: go back to the class level
and bring up an IB for
the instances of $organization
along the relation
(typeOfSuper typeOfSub)

place the new IB next to the
old one

In addition to the facilities that you have seen so far, the SIS system was a vehicle for some initial experimentation into the notion of "co-operating browsers". At one level, there is cooperation between IBs.

EX: select $xsis with
with the left button
in the IB showing the
relation (typeOfSuper typeOfSub)

select TotallyKillInstance
from the menu

note what happens to the
other IB you have up

EX: select $battelle
with the left button
in the IB showing
(partOfSuper partOfSub)

select SpawnNewBrowser

note that the old one
(the one from which you
spawned) disappears

The Value Lattice

In addition to being able to graphically edit the relations that exist between instances, the SIS also provides a way to access (in this case either display or change) the values of a selected UserIV in all the instances. Suppose for example that you want to have a UserIV for $organization call projects, and that you'd like to store in that IV the names of ongoing project at each level in an organization.

EX: select the title menu item
SetIVofInterest in the IB
showing (partOfSuper partOfSub)
for $organization

select the IV projects

select the title menu item
MakeValueLattice in the same IB

select the ValueOnly mode of
presentation

place the ValueLattice

The structure of the ValueLattice is exactly the same as the IB from which it was invoked, but the print characters for each node show the current setting for the IVofInterest that is currently set.

The action of the mouse is very simple for the ValueLattice.

- the left button is just a display device that will highlight a node you select, and highlight the corresponding node in the parent IB. This is to help not get lost in the ValueLattice.

EX: select any node in the
ValueLattice with the
left mouse button, and
note the action in the
parent IB.

- The middle mouse button is for resetting the IVofInterest for a selected node.

EX: select the node for
$columbus-ai in the
ValueLattice with the
middle button

when asked, type in
(DARPA)

note the effect on the
ValueLattice

close the parent IB
and see what happens
(what would you
WANT to happen?)

Discussion

The SIS is a system that is designed to extend programming environment offered by LOOPS to the realm of named, structured instances of LOOPS objects. In so doing, the idea of cooperating browsers was introduced. As was hinted at in yesterday's exercises, this notion can be very powerful if viewed in a light of trying to help the user manage his thinking. If you have suggestions, comments, or criticisms of the few parts of the SIS that you've seen, don't hesitate to make them known.

MDX/MYCIN

MDX/MYCIN is a medical diagnosis system operating in a subdomain of the MYCIN system: bacterial meningitis. At the end of the notebook, you'll find a paper describing the system and the motivation behind construcing it. For purposes of this exercise, what we will do is

- bring the system up and run it on a typical case
- imagine that we are participating in a knowledge engineering seesion whose purpose is to debug a portion of MDX/MYCIN

EX: button mmConcept
with the middle mouse
button and select
BringUpCommandMenu

The command menu is a permanent menu (as opposed to a pop-up menu such as you have seen before) that allows the user to just ask for a desired function.

EX: button Diagnose
in the command menu
that you have up for
mmConcept

select the case
mycinCase232
for running

The system will now bring up an IB, and proceed to diagnosis the case following an MDX approach. Note how clear cut it is to follow the action of what the system is doing at any one time.

When the job is done, the IB will show in inverted video the MDX/MYCIN specialists which have been established.

MDX/MYCIN Debugging Session

At this point we can imagine that you are a knowledge engineer. At your side is your resident medical expert. You have just run case 232 and your expert is requesting to see the establishing numbers for each specialist.

EX: set the IVofInterest
for the IB that is
shown to MostRecentResult

bring up a ValueLattice

Now the medical expert points out that the establishing value say for diploPneumonia is too low. Instead of 1, it should have been 2.

Now you as the knowledge engineer must (with the help of the medical expert) figure out what part of the domain knowledge is incorrect.

The first thing to do is to look at the individual knowledge groups inside the specialist diploPneumonia.

EX: left mouse $diploPneumonia
middle mouse TruthTableFunctions*
select BrowseTruthTables

Now you are looking at the individual TruthTables (ie knowledge groups) that reside in the diploPneumonia specialist.

EX: set the IVofInterest
for the IB you have up
on the knowledge groups
to MostRecentResult

bring up a ValueLattice
for the knowledge group
IB

Note we are now playing the same game that we did before for the specialist level except one level down; ie, at the knowledge group level.

Now your resident medical expert says (from looking at the establishing results for the individual knowledge groups) that the problem is in the diploPneumoia.headInjury knowledge group.

At this point you have pinpointed a potential problem and you can now revise the domain knowledge for that ONE knowledge group that you have found to be in error.

EX: left mouse the
diploPneumonia specialist

select TruthTableFunctions*
with the middle mouse button

select EditTruthTable

select diploPneumonia.headInjury

Now you will find the truth table you have fingered as the culprit in a DEdit window, ready for you to alter.

Imagine you have done the necessary modifications, and exit DEdit.

Now you have made the necessary change to your domain knowledge. Consider what portions of the system must now be re-tested in order to verify that the change is acting as you want it to.

Comments

There are several points to be made about the expert system you have just seen. First, the system is easily extensible. (Why?)

Second, the system is easy to debug because knowledge is factored in a relatively clean way on two levels.

Third, the principle of NAMING the knowledge groups and the specialists themselves allows easy of use by the medical expert.

USING CSRL IN INTERLISP-D

CSRL is a language for implementing diagnostic expert systems. This chapter emphasizes the details of loading and interacting with CSRL on a Xerox 1108 (hereafter called a Dandelion), rather than describing the language and motivating it. It assumes that the you, the reader, have some familiarity with using a Dandelion and the LOOPS language.

Conventions in this document: Since the printer does not have a true backarrow character, a "←" is used instead in this document. Examples showing user interaction indicate what the user enters by underlining it.

1. Loading CSRL

Before you can load CSRL, your Dandelion must be in Interlisp running LOOPS. A fresh version of LOOPS is recommended. CSRL with the Auto-Mech expert system takes up about 700 pages.

To load CSRL, obtain the relevant floppy from Tom Bylander (his office is Caldwell 408), insert the floppy in into the Dandelion, and type in:

←LOAD({FLOPPY}LOADCSRL)

Before any files are loaded, you will be asked 2 questions. The first question finds out if you want to load the source code for CSRL, and the second asks you if you want to load the Auto-Mech expert system which is written in CSRL. If you are doing this for the first time, answer the first question "n" and the second question "y". The following sections are written in the context that you have loaded the Auto-Mech system.

2. The CSRL Browser

The CSRL browser allows you examine, modify, and run a CSRL expert system.

Exercise 1: Creating a CSRL Browser

To get a CSRL browser for Auto-Mech, enter:

```
←(←New $CSRLBrowser Show '(Auto-Mech Specialist))
```

> The cursor will prompt you (by changing to a box shape) to place the browser on the display. You will probably need to Recompute the browser (using the title menu) in order to display the whole lattice. Warning: Do not select the ShowValues item in the title menu until you have run a case.

The lattice that is displayed shows you the "specialist" structure of the expert system. FuelSystem, for example, is a "subspecialist" of Auto-Mech and a "superspecialist" of Vacuum, Delivery, and other specialists. Each specialist of Auto-Mech is associated with a hypothesis about the state of an automobile engine, e.g., FuelSystem is associated with the hypothesis that something is wrong with the fuel system (the subsystem that delivers a mixture of fuel and air to the cylinders of the engine). The hypotheses of FuelSystem's subspecialists are (as you might expect) sub-hypotheses of FuelSystem's hypothesis.

The remainder of this section briefly describes the commands available to you on the browser. Following sections describe more details about using the browser and creating your own expert system.

2.1. Left Button Commands

Print	Prints a specialist or some part of it on the PPDefault window. Selecting this item brings up the following menu.	
	Specialist	Prints the whole specialist.
	Declarations	Prints the declarations of the specialist. These indicate its super- and subspecialists.
	Knowledge Group	Prints a knowledge group of the specialist. Another submenu is displayed for selecting which knowledge group to print. Knowledge groups correspond to

major decisions to be made by the specialist.

Message — Prints a procedure that responds to a particular CSRL message (not the same as a LOOPS message). Another submenu is displayed for selecting the procedure to print. The Specialist class contains the default procedures for messages.

Doc — Retrieve documentation on the specialist or some part of it. It has the same submenu structure as the Print command. This command also lets you retrieve documentation for parts which are inherited by the specialist. Currently all the specialists in Auto-Mech are subclasses of the Specialist class, which contains default information for all specialists.

WhereIs — Find out where a part of the specialist is inherited from.

Unread — Unread the specialist into the current display stream.

Diagnose — Do diagnosis starting with this specialist. Submenus for selecting what case to diagnose, and what message to send are displayed. This command is covered in more detail in the next section.

2.2. Middle Button Commands

Add — Add a new item to the specialist. Brings up the following submenu.

Specialist — Lets you edit the specialist. This is no different from using the Edit command to edit the specialist.

Declarations — Lets you edit the declarations. This is no different from using the Edit command to edit the declarations.

Knowledge Group — Add a knowledge group to the specialist. You are prompted to

type in the name of the knowledge group in the prompt window and then you are sent to the editor. If you type in the name of a previously defined knowledge group, that is what you will edit.

Message — Add a message to the specialist. A submenu is displayed for selected what message you want to add (what messages can be sent is predefined), and then you are sent to the editor. If you select a previously defined message, that is what you will edit.

BoxNode — Boxes the node. This is useful (in fact necessary) to use the Copy command.

Copy — Allows you to copy a knowledge group or message from the specialist that was selected to the specialist which is currently boxed. Submenus let you select the knowledge group or message of your pleasure.

Delete — Allows you to delete a knowledge group, message, or the specialist itself if it is a tip specialist, i.e., a specialist with no subspecialists. Submenus let you select the knowledge group or message. An additional submenu of one item is displayed to ok the deletion. Clicking the mouse outside the menu cancels the deletion. Warning: Undoing a knowledge group or message deletion is not possible. To undo a specialist deletion you need to add the specialist back as a subspecialist of the appropriate specialist, and edit the declarations of the specialist.

Edit — Allows you to edit the specialist or some previously defined part. Its submenu is the same as the Add command. The difference is that for the Knowledge Group and Message subcommands, you select what you want to edit from another submenu.

Rename — Allows you to rename a knowledge group or the specialist itself.

2.3. Title Menu Commands

ShowValues — Brings up a sub-browser which shows the confidence values for the specialist in the current case. See next section for more details.

Recompute, AddRoot, DeleteRoot, SaveInIT — Same as for class browsers. Recompute is not automatically called when a specialist's declarations are changed, so you will need to Recompute the browser "manually".

2.4. Shift Commands

If the left shift key is depressed when you click the **left** mouse button, a summary of the specialist is printed in the PPDefault window. This will be referred to as the Print Summary command.

If the left shift key is depressed when you click the **middle** mouse button, you will be sent to the editor to edit the specialist. You can use the Edit command to do the same thing.

Exercise 2: Operating the Browser

If you haven't brought up the browser for Auto-Mech yet, it is suggested that you do so now.
Try using the Print, Print Summary, and Doc commands on the specialists.
Use the Edit command to bring up a specialist or a part of one in the editor. If you make no changes before you exit the editor, no processing will be done. If you make changes, and exit the editor, then if an error is discovered, you will go back to the editor; otherwise the changes that you have made will take effect (and are undoable). Selecting the Stop item from the Exit submenu (use the middle button to display the submenu) will let you exit the editor without making any changes. Use the Print command to confirm this.
Use the Copy command to copy the summary knowledge group of Specialist to Choke. You will need to use the BoxNode command on Choke first. Confirm the copy with the Print command. Use the Delete command to remove the summary knowledge group from Choke.
Rename the Choke specialist to UsedToBeChoke. Rename it back to Choke.

3. Running a Case

Exercise 3: Running Auto-Mech

Left button the Auto-Mech specialist in the browser, select the Diagnose command, "new case?" and "Establish-refine". Auto-Mech will now present questions for you to answer in the TTY window. Also, some information about what the specialists are doing is displayed. The specialist that is currently executing is boxed in the browser. For anyone who doesn't want to think of answers to the questions, use the following:

```
Do you have problems starting your car? n

Does the car stall? n

Does the car run rough? y

Does the problem occur while idling? n

Does the problem occur on loading? y

Does the problem occur while the engine is both
        hot and cold? y

Have you eliminated ignition as a possible cause
        of the problem? y

Is any fuel delivered to the carburetor? y

Have you been getting bad gas mileage? n

Are there any cracked, punctured or loose vacuum hoses? u

Can you hear hissing while the engine is running? n

Are the vacuum hoses old? y

Can you see cracks in the carburetor gasket? y

Has FuelSystem completed diagnosis? n

Is the air filter old? n

Has FuelSystem completed diagnosis? n

Have you tried a higher grade of gas? y
```

3.1. What Happens When a Case is Run

What you just did was to send an Establish-refine message to the Auto-Mech specialist in the context of a new case. Upon receiving this message, the Auto-Mech specialist sent itself an Establish message, some questions were asked, and then the specialist sent itself a Refine message, which then called Auto-Mech's subspecialist, FuelSystem, with Establish and Refine messages. As this "establish-refine" process was applied further down the hierarchy, some of the specialists were sent Establish messages, but not Refine messages. This happened either because the specialist had no subspecialists or because the specialist did not have a high enough confidence value.

In CSRL, a seven-point confidence value scale is used. For convenience, we use the integers from -3 to +3, which can be loosely interpreted as:

3	Hypothesis is confirmed.
2	Hypothesis is very likely.
1	Hypothesis is mildly likely.
0	Evidence for the hypothesis is inconclusive.
-1	Hypothesis is mildly unlikely.
-2	Hypothesis is very unlikely.
-3	Hypothesis is disconfirmed.

If the confidence value is 2 or 3, the specialist is said to be established. For -2 or -3, the specialist is said to be rejected. Otherwise the specialist is suspended.

A sub-browser is available to display the confidence values in a graphical manner by selecting the ShowValues item from the title menu.

Exercise 4: Creating a Confidence Value Browser

> Select the ShowValues items in the Auto-Mech browser. You will be prompted to display the browser window on the screen. Try out the menus in this browser. Note: The confidence value browser won't work unless the top node, Auto-Mech, has a confidence value in the current case.

Since you can use the main browser to send any message to any

specialist, you are able to "explore" any diagnosis that was not originally done.

Exercise 5: Diagnosing From Inside the Hierarchy

> Left button one the specialists that has a confidence value but was not refined, and select Diagnose, "current case?", and Refine from the menus. After this processing is finished, go to the confidence value browser, and select Recompute from the title menu.

3.2. The Current Case

CSRL remembers what the current case is by setting the variable currentCase. Cases correspond to instances of the class CSRLCase, which has methods for implementing a simple question-asking facility. CSRLCase also keeps track of old cases. Presently there is no facility (but one is planned) for renaming or saving cases.

3.3. Tracing CSRL

The trace information that the diagnose provides you is done using the functions TraceCSRL and UntraceCSRL. Both of them are NLambda-NoSpread functions, and take arguments corresponding to the following forms:

Specialist	Trace all specialists (TraceCSRL Specialist)
‹specialist›	Trace the specialist (TraceCSRL Mixture)
Message	Trace all messages (TraceCSRL Message)
(Message to ‹specialist›)	Trace messages to this specialist (TraceCSRL (Message to ValveOpen))
(Message from ‹specialist›)	Trace messages sent from this specialist (TraceCSRL (Message from FuelSystem))
Rule	Trace all rules. This will only trace the rules of specialists that are currently traced. (TraceCSRL Rule)

(Rule of <specialist>)
Trace rules in the specialist. This will only trace the rules if the specialist is traced.
(TraceCSRL (Rule of Carburetor))

(Rule of <name> kg)
Trace rules of knowledge groups with this name
(TraceCSRL (Rule of summary kg))

For example, the present tracing level was done by (TraceCSRL Specialist Message). UntraceCSRL removes a previous TraceCSRL.

Exercise 6: Tracing Rules in Selected Specialists

Trace the rules of each of FuelSystem's immediate subspecialists, and then diagnose a new case.

4. Making Your Own Expert System

This section gives a brief incomplete account of how to build a simple expert system in CSRL. In particular, it contains no information on how to change the default Refine procedure. It also depends on your ability to figure out how the Auto-Mech system. The section also includes exercises on building part of a expert system for diagnosing problems in a house (or apartment or mansion if you like).

4.1. Making the First Specialist

To make a specialist, use the function Specialist, which has the form:

```
(Specialist <name> <comment>
    (declare <declaration1> <declaration2> ...)
    (kgs <kg1> <kg2> ...)
    (messages <message1> <message2> ...))
```

The declare, kgs, and messages sections are optional, as well as the comment.

Exercise 7: Creating the House Specialist

To make an specialist called House do:

```
←(Specialist House (* House is a new specialist))
```

You should also create a browser to facilitate future additions.

```
←(←New $CSRLBrowser Show '(House Specialist))
```

4.2. Adding Subspecialists

To add subspecialists to the expert system, it is easiest to edit the declarations of the existing specialists.

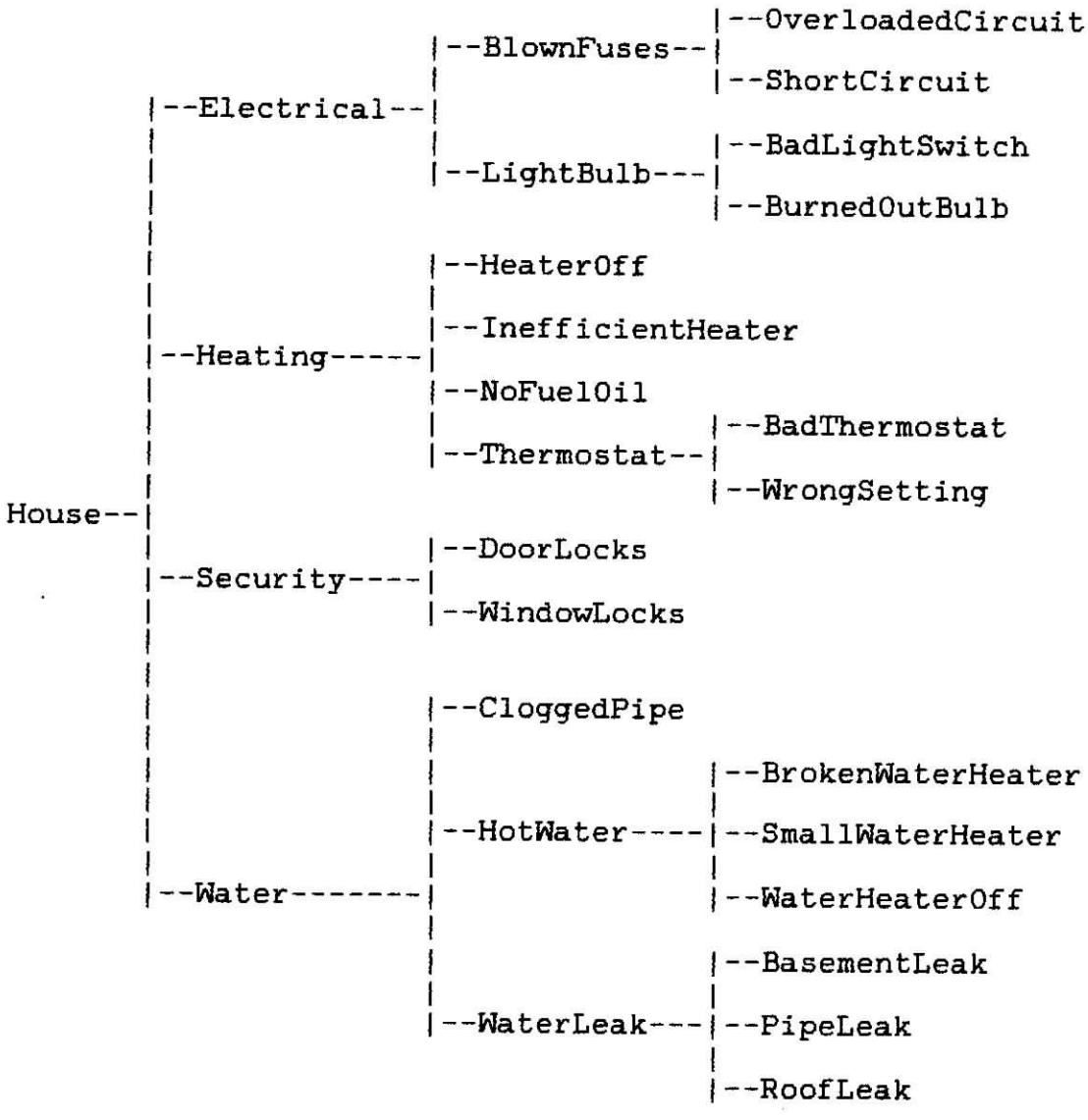
Exercise 8: Adding Subspecialists to House

Use the Edit command (or left shift/middle button) to edit the House specialist. Add declarations after the comment (change the comment if you wish) which look like:

```
(declare
    (subspecialists Electrical Heating
                    Security Water))
```

Be sure that you have spelled "subspecialists" correctly. Now Exit the editor and Recompute the browser. The specialists should have been automatically created.

Now add more specialists so that the browser looks something like:



Of course, feel free to restructure the specialists any way you like. You should be forewarned, though, that the "perfect" hierarchy does not exist. Below we will only be concerned with embedding knowledge within the Heating subhierarchy of this structure, so it is not necessary to create the whole structure.

4.3. Adding Knowledge to the Specialists

In CSRL, most of the knowledge of a specialist takes the form of knowledge groups. A knowledge group (hereafter abbreviated to kg) maps a list of expressions to a confidence value (or some other measure). This can be the confidence value of the specialist or perhaps the confidence value in some intermediate hypothesis. For example, there might be a kg in the Security specialist called badNeighborhood, which could measure the likelihood that you are in a bad neighborhood or measure of the "badness" of the area. This value and values of other kgs (which measure other facets of security) could be combined by a "summary" kg to arrive at a confidence value for the specialist.

One type of kg is called a Table kg. Its form is:

```
(⟨name⟩ Table ⟨comment⟩
     (match ⟨expression⟩ ⟨expression⟩ ...
       with (if ⟨test⟩ ⟨test⟩ ...
                then ⟨value⟩
              elseif ⟨test⟩ ⟨test⟩ ...
                then ⟨value⟩
              else ⟨value⟩)))
```

For example, the BadGas specialist of Auto-Mech has this kg:

```
(relevant Table
   (match
      (AskYNU? "Is the car slow to respond")
      (AskYNU? "Does the car start hard")
      (And
        (AskYNU? "Do you hear knocking or pinging
                                          sounds")
        (AskYNU? "Does the problem occur while
                                          accelerating"))
     with
       (if T ? ?
           then -3
         elseif ? T ?
           then -3
         elseif ? ? T
           then 3
         else 1)))
```

If the first expression is T (true), then the value of the kg is -3. Else, if the second expression is T, then -3. Else, if the third expression is T, then 3. Otherwise, its value is 1. Note that the number of tests following the "if" or "elseif" is the

same as the number of expressions of the table. Also note that each test must be true for the "row" of the table to match. The "?" test matches any value. The syntax for expressions and tests are discussed below. You are encouraged to look at other Table kgs in Auto-Mech.

The other type of kg which will be discussed is the Rule kg. Its form is:

```
(⟨name⟩ Rules ⟨comment⟩
    (match ⟨expression⟩
      with (if ⟨test⟩
               then ⟨value⟩
             elseif ⟨test⟩
               then ⟨value⟩
             else ⟨value⟩))
    (match ⟨expression⟩
      with ...)
    ...)
```

A example from the Carburetor specialist of Auto-Mech is:

```
(other Rules
  (match (AskYNU? "Is there fuel leaking around the
                                        carburetor")
    with (if T then 3))
  (match (Or
           (And
             (AskYNU? "Do you hear knocking or
                                        pinging sounds")
             (AskYNU? "Does the engine idle fast"))
           (And
             (AskYNU? "Does the car hesitate")
             (AskYNU? "Does the problem occur while
                                        decelerating")
             (AskYNU? "Does the engine idle fast"))
           (AskYNU? "Does the engine idle slow")
           (AskYNU? "Does the car run rough"))
    with (if T then 3)))
```

Each rule (a match-with form) is tried in succession until one "matches". A rule matches if it returns a value, i.e., the ⟨expression⟩ satisfies a test in the if-then part, or if there is an else clause within it. (As a consequence, it only makes sense to have an else clause in the last rule.) The value of the matched rule becomes the value of the kg.

4.3.1. Expressions in CSRL

Names of knowledge groups, numbers, CSRL variables, LISP expressions (e.g., AskYNU? is a LISP function), "self", and the logical constants T, F, and U are the base expressions of CSRL. CSRL variables will not be discussed here (but see the Refine message procedure of Specialist for an example). Any list which is not mistaken for a CSRL expression is assumed to be a LISP expression. The literal "self" evaluates to the name of the current specialist. Note that CSRL uses a three-valued logic.

CSRL provides the normal set of logical and numerical comparison operators, as well as a small set of arithmetic functions.

(And ⟨exp⟩ ⟨exp⟩ ...)
: Returns T if every expression is T, F if any expression is F, and U otherwise.

(Or ⟨exp⟩ ⟨exp⟩ ...)
: Returns T if any expression is T, F if every expression is F, and U otherwise.

(Not ⟨exp⟩)
: Returns T if the expression is F, F if it is T, and U otherwise.

LT, LE, GE, GT
: Numerical comparison operators with the obvious interpretation. If one or both expressions are not numbers, U is returned.

(Range ⟨exp⟩ ⟨exp⟩ ⟨exp⟩)
: Returns T if the first expression is within the range (closed interval) of the second and third expressions, and F if it is not. U is returned if any expression is not a number.

EQ
: Equivalent to EQ in LISP.

Plus, Subtract, Minus, Times, Divide
: Arithmetic functions with obvious interpretations. If any of the expressions are not numbers, U is returned.

4.3.2. Tests

(And ⟨test⟩ ⟨test⟩ ...)
: Returns T if every test is T, F otherwise.

(Or ⟨test⟩ ⟨test⟩ ...)

Returns T if any test is T, F otherwise. test is F, and U otherwise.

(Not <test>) Returns T if the test is F, F if it is T, and U otherwise.

(LT <number>), (LE <number>), (GE <number>), (GT <number>),
Numerical comparison with the obvious interpretation, e.g., for LT, returns T if the expression is less than the number.

(Range <number> <number>)
Returns T if the expression is within the range (closed interval) of the numbers, and F otherwise.

(EQ <atom or number>)
Returns T if the expression is EQ to the atom or number, and F otherwise.

<atom or number>
If not embedded in a LT, Range, or some other comparison test, an equality test is implied.

4.4. Writing Procedures for Establish Messages

In general, a specialist sets its confidence value within its procedure for the Establish message. This section explains some of the syntax for these procedures. A simplified form for an Establish message procedure is:

```
(Establish <comment>
   <statement>
   <statement>
   ...)
```

CSRL also has facilities for creating local variables and passing parameters, but these will not be necessary for the final exercise. As always, you are encouraged and exhorted to look at the Auto-Mech specialists as examples for specialists that you write.

4.4.1. Statements

(if <exp> then <st> elseif <exp> then <st> else <st>)
Evaluates each expression until one is T. The corresponding clause is executed. If no

expression is T, the else clause (if any) is executed. More than one statement can follow a then, elseif, or else.

(SetConfidence <expression> <expression>)
The first expression should evaluate to the name of a specialist. In general, you will only use "self" here. The value of the second expression becomes the confidence value of the specialist.

(DoLisp <form> ((<lisp var> <CSRL exp>)
(<lisp var> <CSRL exp>)
...)
Uses Lisp EVAL to evaluate the form. Before that, each of the Lisp variables are bound to the corresponding CSRL expression. This allows an escape to Lisp, and a way to get at values in CSRL. This may also be used as an expression.

5. Final Exercise

Exercise 9: Implementing Part of the House Expert System

Add knowledge groups and establish procedures to the specialists in the Heating hierarchy. Use AskYNU? to get information from the user about his heating system. You should consider the following questions in your implementation.

"Does your house get too cold"

"Does your house get too hot"

"Is your heating bill too high"

"Are you out of fuel oil"

"Is your furnace old"

"Has your furnace been checked recently"

"Is the fuel oil igniting in your furnace"

"Is your thermostat set low"

"Is your thermostat set high"

"Does changing the thermostat affect the temperature"

Also, examine the following questions, consider what additional hypotheses should be considered, and modify the Heating hierarchy accordingly. In addition to adding more specialists, you will probably need to reorganize the hierarchy so that similar specialists are grouped, e.g., having a FuelOilDelivery specialist with EmptyFuelOilTank and ClosedFuelOilValve as subspecialists.

"Is the fuel oil valve open"

"Are your heating vents open"

"Does the furnace fan turn on"

"Are your windows open"

"Do you have a fireplace"(heat can escape up the chimney)

"Do you feel air coming through the windows"

"Is your attic insulated"

Table of Contents

List of Exercises

XIV

a, b & c

filed on: {indigo}<kbvlsi>loopscourse>truckin.bravo
Last Edited: SM: July 11, 1983

Truckin

A teaching game for expert systems

by the Loops Design Team
Daniel Bobrow, Sanjay Mittal, and Mark Stefik

Abstract. *Truckin* is a knowledge system used for teaching knowledge representation techniques in Loops. *Truckin* provides an environment for creating, testing, and evaluating small bodies of knowledge interactively. Much of the knowledge in *Truckin* is represented as rules for controlling an automated truck in a simulation. The rules enable a truck to plan its journey along a road as it buys and sells commodities. A *Truckin* knowledge base can be evaluated in terms of the ability it gives an automated truck to make a profit while avoiding hazards of the highway. *Truckin* knowledge bases can be extended incrementally, so that a new Loops user can begin by extending existing sets of rules. *Truckin* contains illustrative examples and idioms for access-oriented, object-oriented, and rule-oriented programming.

Introduction

Loops is a knowledge representation system designed for use in building expert systems. It augments the Interlisp-D environment with object-oriented programming, access-oriented programming, and rule-oriented programming. Loops was developed by members of the Knowledge Systems Area at Xerox PARC.

In January 1983, the Loops system was ready for *beta*-testing outside of Xerox PARC. To help beta-test users to learn and evaluate Loops, the Loops Group decided to offer a short intensive course. This course was intended to provide hands-on experience in using Loops. Because Loops was designed for expert systems applications, it was believed that ***the best way to teach Loops would be to organize the course around a mini-expert system.***

It was important that the mini-expert system for the course not be too technically specialized, because people taking the Loops course would come with a variety of different backgrounds. The mini-expert system should be based on knowledge from common experience. The expert system needed to be engaging enough and open-ended enough to draw people into developing fairly elaborate knowledge bases. This led to the idea of a simulation game around which we could have "knowledge competitions".

This document describes the *Truckin* world as a teaching and simulation game around which the Loops course is organized.

The *Trucker's Handbook*

A "player" in *Truckin* is qualitatively different from a player of computer and video games, such as those that are popular in arcades and on home computers. In most computer games, a player is a person. In *Truckin*, a player is a knowledge base. In this way, personal competition in *Truckin* is one level removed from the game. The objective is to create a knowledge base that can effectively guide a truck in the situations it encounters in the simulation environment.

Truckin provides a set of commodities, producers, consumers, hazards, road stops, and trucks. The *Truckin* world is intended to be complex enough to be interesting in the Loops course, but too complex for a simple mathematical model. The online data base of facts about this world is, metaphorically, called the *Trucker's HandBook*. A wise *Truckin* player will consider the facts in the *Trucker's Handbook* in planning its route.

The simulation in *Truckin* is controlled by a program called the *GameMaster*. In different contexts, the term "Game Master" refers to different combinations of programs, knowledge bases, processes, and computers. For simplicity, we refer to the whole thing as "the" *GameMaster*. The *GameMaster* chooses the initial configuration of the highway, sometimes called the game board, decides on the legality of the requests made by the players, updates the GameWorld and maintains the display.

The players talk to the *GameMaster*, who also decides which player gets the next turn. Currently, players get their turn on a time-robin basis, i.e., the player who has used the least amount of time gets the next turn. During a turn a player can buy or sell commodities at any *RoadStop* at which it is parked. These transactions are governed by practical considerations of how much money is in the truck's *cashBox*, whether there is cargo room in the truck for the goods, and whether the *RoadStop* advertises an interest in buying or selling the particular commodities. During a single move a truck can also drive to one other *RoadStop*. The distance that the truck can travel in a move is governed by a variable reflecting traffic conditions, as well as the maximum speed of the truck and the amount of gasoline remaining in the fuel tank.

Profits and Risks

The goal of a player in *Truckin* is to maximize profit during the game. The game ends after a predetermined number of turns. At the end of a game, the winning player is the one with the most cash. *AlicesRestaurant* is a special roadstop because any player who is parked there when the game ends, gets a hefty bonus.

Players compete for a fixed supply of goods and parking places. Just as with real trucks, there are a number of things that are important to know about the world. For the details of this, a player should consult the *Truckin Handbook (*database of relevant Loops classes). Here is a summary of the elements of the *Truckin* world:

Kinds of Trucks. Players start the game at *UnionHall* with an empty *truck* and an allotment of fuel and cash. Trucks come in different varieties, with different speeds, different fuel efficiencies, and different capacities for carrying merchandise. During each turn, the speed of the truck is measured as the ratio of the number of roadstops actually moved and the maximum allowed for that class of truck.

RoadStops. *RoadStops* are the positions along the highway. In the standard version of the game board, there are sixty six RoadStops along the highway. Neighboring *RoadStops* are separated by one mile. Up to two trucks can be parked at a *RoadStop.*

Producers. *Producers* are *RoadStops* at which players can purchase goods. A given producer will sell only a fixed kind of item, for example televisions, shirts, or apples. A *Producer* has only a fixed inventory of items for sale, and this inventory is used up as the simulation runs. The game board display shows the quantity of items for sale, and a price ratio which can be multiplied times the *averagePrice* of a *Commodity* to determine the purchase price. An icon is displayed on the game board to show the kind of *Commodity.* There are usually about 30 *Producers* distributed along the highway.

Consumers. *Consumers* are *RoadStops* at which players can sell goods. In general, *Consumers* are interested in generic kinds of goods, such as sporting goods, office supplies, or groceries. The capacity for a *Consumer* to buy goods decreases as items are purchased. The game board display shows the quantity of items that will be purchased, and a price ratio. The name of the generic class of *Commodities* to be purchased is displayed on the game board. There are usually about *23 Consumers* distributed along the highway.

Commodities. *Commodities* are the things that are bought and sold along the highway. The kinds of Commodities that are available are shown in the *Trucker's Handbook.* Some Commodities have special features, such as being fragile or perishable. *PerishableCommodities* have a *lifetime* (expressed in turns) which determines how long the *Commodities* remain salable. *Fragile Commodities* have a *fragility* which determines how likely they are to break when you go past *RoughRoads. Commodities* also have a volume and a weight which means that a truck can carry commodities limited by the available volume and weight on the truck.

Gasoline. Driving a truck uses up gasoline. Gasoline can be purchased at *GasolineStations* along the highway. Running out of gasoline results in a towing and a fine. There are usually about 5 *GasolineStations* along the highway.

WeighStations. *WeighStations* represent the arm of the government in *Truckin.* If a player goes by a *WeighStation* without stopping, he risks some chance of receiving a stiff fine and a towing back to the *WeighStation.* If he stops, he must pay a small toll (and use up a turn).

Rough Roads. Some *RoadStops* correspond to rough places on the road. Driving past a *RoughRoad* entails some risk to any *FragileCommodities* that are on board. If a player stops at a *RoughRoad,* no damage will result.

Bandits. *Bandits* in *Truckin* do not sit still. They can park at various *RoadStops* as controlled by the *GameMaster* and can intercept trucks. If a bandit intercepts a truck, or is parked at the same roadstop as a truck, it will take all of the *LuxuryGoods* that it has room for and one fifth of the money in the *cashBox.*

The CityDump. In general, an attempt to sell perished or damanged goods results in a stiff fine.

However, such goods can be unloaded for a fee at the *CityDump.* (In the simulation, these goods are sold for a modest "negative price".)

The Union Hall. If a player runs out of gas, he will be towed to *Union Hall.* There he will be given a new allotment of cash, but his truck will be emptied. This happens to a player whether he goes to *UnionHall* on his own request, or whether he is towed there for violating some rule.

Alice's Restaurant. At the end of the game, all of the trucks try to make it to *Alice's Restaurant.* Players ending the game at any of the Alice's get their cash doubled. There may be more than one *Alice's Restaurant* on the highway, and any one of them will do. If there are more trucks in a game than parking places at the restaurant, then there will be competition for the places. To preclude the strategy of just going to *Alice's Restaurant* and parking, any player who parks there for more than some specified time will be towed away to Union Hall.

Advice for Independent *Truckers*

To succeed at *Truckin,* a player must be responsive to the configuration of the highway and to changing conditions. To make a profit, a player must consider the spread between price ratios and the convenience of the relative locations for buying and selling commodities. A player must not exceed the capacity of his truck in either weight or volume.

We have a few final suggestions for players. Don't buy goods that you can't sell at a profit. Don't buy *PerishableCommodities* if you can't deliver them on time. If your goods spoil or are damaged, take them to the *CityDump.* Keep an eye on your fuel gauge. Don't drive too quickly with *FragileCommodities* over *RoughRoads.* Don't spend all of your cash on *Commodities*; you may need some for incidentals along the way. Watch out for bandits, rough roads, and weigh stations. And try to be at *Alice's Restaurant* when the game ends.

filed on: {indigo}<kbvlsi>loopscourse>truckinmanual.bravo
Last Edited: SM: July 11, 1983

Truckin MANUAL

by the Loops Design Team
Daniel Bobrow, Sanjay Mittal, and Mark Stefik

This document gives the basic instructions for creating game boards, starting, stopping, and continuing a game, interrupting a game in the middle, and attaching gauges to monitor the internal state of *Truckin* players.

[NB: *Truckin* now has versions which run on both single machines as well as multiple-machine configurations. The following instructions are written for the single machine version. Any differences for the multi-machine version are indicated in smaller print. Otherwise the instructions apply to both versions.]

A. Creating a new game

Send the message *New* to *$Truckin* as follows:

(← $Truckin New)

this creates a new game board and the lisp variable *PlayerInterface* is set to the instance of *TruckinPlayerInterface*. **All commands sent by your player or you go to *PlayerInterface*.** You can play any number of times on this basic game board as follows.

[On a *RemoteMasterMachine*:
(← $MasterTruckin New) creates a new game.
On a *RemoteSlaveMachine*:
(← $SlaveTruckin New) sets up the Truckin world and links your machine to the MasterMachine. You will be asked for a *unique name* to identify your machine and the address of the *PostMaster*. Please ask the game coordinator for this address. A new game cannot be created from a slave machine - the slave machine will run the game created by the master machine.
PlayerInterface is set as above in both these cases as well].

[[[[In all these cases, upto 4 arguments can be given to the New message to select the game configuration you want. The description above is for the default case.
Arg 1: Type of Game--
This specifies what kind of *DecisionMaker and PlayerInterface* you want. Currently the only value is **TimeTruckinDM** (the appropriate player interface is automatically selected). Later, we may put in other versions of the game.
Arg 2: Type of Game Board--
This specifies what kind of game board you want: **BWTruckin** or **ColorTruckin**. The former is the default. In order to use ColorTruckin option, you need a color monitor attached to your machine.
Arg 3: Type of Simulator--
This specifies whether you want the game board to be displayed or not: **DisplayTruckinS** or **NoDisplayTruckinS**. The former is the default. The NoDisplay version of the simulator maintains an upto date version of the game but does not display the game board.
Arg 4: Broadcast List--
This is a list of objects who want to receive a copy of all game messages which change the world. These objects must be capable of responding to the messages described in the *MultiMachineTruckin* document. These objects will get the messages after the world has already been updated.]]]]

B. Starting a game

Send the message *BeginGame* to *PlayerInterface* to start a game as follows:

(← PlayerInterface BeginGame)

This message refreshes the game board created earlier and prompts you for the players you want in this game. You can either create new players from among the existing player classes (via an interactive menu) or use any players created earlier. [The menu appears next to the prompt window at the left top of the screen]. The menu for the players offers you a choice of both player classes and existing player names. You can opt for all existing players by choosing the *ALL-EXISTING* menu option. Select *NO* when you are done selecting players.

You can pass one optional arguments in the BeginGame message.

Arg : If T then all existing players will be used for the game and you will not be asked for players. This might be convenient during debugging when you want to use the same game board and same set of players for debugging your player.

[[If you are running *SlaveTruckin*, BeginGame will let you select your local players, but the game will only start when the Master Machine decides - which it does when a BeginGame is done on the Master Machine.]]

C. Interrupting a game in the middle.

In addition to the rule exec and the break/trace facility of the rule language (see Rule Language manual), there is another way to temporarily stop a game in the middle and bring up the lisp user exec. Hold the **CTRL** and **LEFT SHIFT** keys simultaneously when one of the trucks is moving. This will put you into the Lisp User Exec, where you can examine things and/or edit your rule sets and functions. Type **OK** in the Exec to resume the game. On a dorado, the trucks move pretty fast, so if the above does not work the first time, try again.

C.2 Interrupting a player any time

Left of the Status Window, you will notice a menu which lists the players running on your machine. Selecting any player in the menu, allows you to interrupt that player and bring up the Rule Exec. Remember that the game time continues to click while you are in the Rule Exec.

D. Suspension/Premature Termination of the Game

You can suspend, resume, or kill the game by using the **Game Control Menu**, which normally appears left of the Status Window. Selecting *Suspend* will suspend the game (but remember that the time allocated for the game continues to tick, so when you resume, the intervening time will be deducted from the game time). Selecting *Awake* will resume the game and *Kill Game* will kill the game.

E. Attaching gauges to your player's truck

You can attach gauges to *Instance Variables* (IVs) of objects under your control such as your player or truck in order to monitor important internal state during the game. When you first create a player, the game master will offer to put gauges on your truck, i.e., to the IVs *cashBox, fuel, weight,* and *volume.* You have several options. **NO** will not put any gauges. **YES** response will lead to the system asking you whether you want gauges on each of the four IVs listed above. For each IV for which you respond with *YES* the system will offer a choice of gauges. **DEFAULT** response will put default gauges on fuel.

Once you put gauges on a player, they can be reused when you use the same game board for a new game or create new game boards. Thus, if you expect to use a player many times, it pays to attach the desired gauges once and continue to use the player.

F. Attaching gauges to other IVs of your player

When you create a player, the instance object is given the same name as the driver name you enter. Thus, if you name some player **Joe** you can access the object as **$Joe.**

You will often find it useful to attach gauges to IVs of your player. For example, if your player is an instance of **Peddler**, you might want to monitor IVs such as *destination, stoppingPlace,* and *goal.* The way to attach gauges on your player is to send it the **AddGauges** message. For example,

(← $Joe AddGauges '(destination goal)

will attach gauges to destination and goal IVs of $Joe if $Joe is an instance of Peddler. The AddGauges method will prompt you for the type of gauge. The most suitable gauge for arbitrary values is **LCD.**

The AddGauges message can be used to select default gauges on the instance variables indicated, instead of having to select gauges yourself each time. In order to do this, you have to specify additional information in the object class as shown in the following simplified description of the class **Truck.**

```
(DEFCLASS Truck
   (MetaClass ..)
   (Supers ..)
   (ClassVariables ..)
   (InstanceVariables (cashBox 10000 DefaultGauge LCD GaugeLimit (0 10000))
                      (fuel 80 DefaultGauge Dial GaugeLimit (0 80))))
```

Thus, suppose, you wanted an LCD gauge to be the default gauge on *destination,* you can specify this for use by the AddGauges method by adding the property *DefaultGauge* to the instance variable *destination* with *LCD* as the value. Then pass T as the *second argument* in the above AddGauges message. This will result in a LCD gauge being installed on destination and you will be prompted only for goal. [You can do the same for *goal* or any other IVs also]. If the default gauge you have specified is being used for numbers, you also should specify the default limits. For this, put under the *GaugeLimit* property a list containing the two numbers which indicate the lower and upper limits.

G. Adding gauges under program control

You can also attach gauges under full program control by specializing the method **SetUpGauges** in the class **Player.** The description given above is carried out by this method. You could write your own SetUpGauges method in your player class and make it attach gauges by using the method *AddGauges* described earlier. Both **Truck** and **Player** respond to the message AddGauges. This way you could build into your SetUpGauges method, your choice of gauges, which then will be carried out by the system each time you create a player of that class.

H. Selecting trucks under program control

You can also select the truck you want for your player automatically, instead of being prompted for it. In order to do this specialize the method **SelectTruck** for your player class. This method will be called when your player class is instantiated. This method should return the name of one of the truck classes currently allowed in the game. Currently, the allowed trucks are: **MacTruck, GMCTruck, FordTruck,** and **PeterBiltTruck.**

I. Summarizing the truck data at a glance

You can get a summary report of your players truck by sending your player (say **Joe**) the **Show** message as follows:

(← $Joe Show)

This will print out the cashBox, fuel, weight, and volume, as well show you the cargo your truck is carrying. This summary may be useful during debugging.

J. Clearing up the screen

If your screen gets messed up for some reason, you can restore it to the initial state by buttoning the LoopsLogo in the middle top of your screen and selecting the command *SetUpScreen*. You can also do this in the middle of the game when you are in any of the rule exec, user exec or break exec. Even though the game board and gauges will disappear temporarily, they will come back as those windows are written to.

K. When players get control

A player gets control when his/her turn comes and the game master sends a **TakeTurn** message to the instance of your player object. Your top-level rule-set must be written to respond to this message.

You can also write your player in such a way that the top-level rule set never returns, i.e., the *TakeTurn* rule-set uses *whileAll* control structure. The *PlayerInterface* will suspend you when you make a *Buy, Move,* or *Sell* request and reschedule you when your turn comes again.

L. Legal requests by players during game

A player can make three kinds of requests during the game: **Move, Buy, Sell.** After each request, the player is suspended until the request is completed and your turn comes again (i.e., all other players have used up the same amount of time).

1. (← PlayerInterface Move player numOrLoc)
This is a request to move *player* from the current location to a location determined by *numOrLoc*. If numOrLoc is a number, then it is the relative offset from the current location. It can be positive or negative. It can also be the actual instance object representing the particular roadStop in the game.

2. **(← PlayerInterface Buy player qty)**
This is request to buy *qty* of the commodity at the location at which *player* is currently parked.

3. **(← PlayerInterface Sell player commodityInstance qty)**
This is a request to sell sell *qty* of the commodity *commodityInstance* owned by the player in their truck's cargo, at their current location. If qty is not specified, then the qty in the commodityInstance will be used.

The standard value of *player* in all the three above messages is *self* which is bound to the player executing the rule-set.

filed on: {indigo}<kbvlsi>loopscourse>truckinvocabulary.bravo
Last Edited: SM: July 11, 1983

Truckin Query Functions

by the Loops Design Team

Daniel Bobrow, Sanjay Mittal, and Mark Stefik

This document summarizes the functions and methods you will find useful in writing the rules for your *Truckin* players. These functions allow you to select and filter roadstops satisfying different constraints as well as conveniently access other information about the current status of the *Truckin* world. Many of the following functions are also available as methods attached to the class **Player**, allowing you to easily specialize them if you so desire.

In the following summary, functions marked with an asterisk (*) are also implemented as methods on **Player** with the same name as the function and taking the exact same arguments. For more details about these functions see the listing of the file TRUCKINV in your folder.

A. Selection functions

The following functions return a list of roadstops based on certain constraints.

AnyRoadStop (roadStopType numMoves direction roomToParkFlg)*
Randomly picks one of the roadstops of type *roadStopType* where *roadStopType* is one of the RoadStop classes. If *numMoves* is provided, it returns only those roadstops within that distance. If *direction* is F then only those in the forward direction, if B then only in the backward direction, if NIL then in either direction. If *roomToParkFlg* is T then only those roadstops where there is room to park.

Buyers (commodityClass numMoves includeCDFlg)*
Returns all of the Buyers (i.e. *Consumer* roadstops) able to purchase a commodity of type *commodityClass*. If *numMoves* is provided, returns only those within that distance. A common case is to use the instance variable *maxMove* of your player as this argument. If *includeCDFlg* is T then includes *CityDumps* also, otherwise not.

NthRoadStop (numMoves direction fromRoadStop roomToParkFlg)*
Returns the Nth roadstop in the given direction from *fromRoadStop*. If *fromRoadStop* is NIL, the current location of the player is used. If *direction* is NIL, **Forward** is assumed. If there are fewer than *numMoves* roadstops in the specified direction, that is if the request would go off the board, this function returns the farthest roadstop in that direction.

RoadStops (roadStopType numMoves direction roomToParkFlg)*
Returns all of the roadstops of type *roadStopType* reachable within *numMoves* in the direction specified by *direction* taking into account room to park if *roomToParkFlg* is T.

Sellers (commodityClass numMoves)*
Returns all the roadstops which are Sellers (i.e. *Producer* roadstops) of *commodityClass* and are located within *numMoves.*

B. Filter functions
The following functions take a set of roadstops as one argument and prune that set based on other criteria specified by other arguments. Some of the following functions are very general and can be used to filter (or order) any set of objects of the same class and are not limited to working on roadstops only. These are: **FilterObjs, PickHiObj, PickLowObj,** and **SortObjs.**

FilterObjs (self selector objects)
Sends a *selector* msg to *self* for each of the object in *objects* and returns all of the objects for which the rule set returned a non-NIL value. This is the basic function for doing filtering based on your knowledge encoded as rules.

FurthestRoadStop (roadStops fromRoadStop)*
Returns the roadstop in *roadStops* which is furthest from *fromRoadStop* excluding *fromRoadStop.* If *fromRoadStop* is NIL, assumes the current location of the player.

NearestRoadStop (roadStops fromRoadStop)*
Same as **FurthestRoadStop** except returns the nearest roadstop.

PickHiObj (self selector objects)
Sends a *selector* msg to *self* for each object in *objects* to determine a numeric rating for each of the objects. It returns the object with the highest numeric rating. When the value returned is non-numeric for an object, then that object is automatically excluded.

PickLowObj (self selector objects)
Same as **PickHiObj** except returns the one with the lowest numeric rating.

SortObjs (self selector objects)
Sends a *selector* msg to *self* for each object in *objects* to determine a numeric rating for each of them. It returns a list of objects in the **descending** order of their numeric rating. It also excludes the ones with non-numeric ratings.

C. Miscellaneous functions

AnyBanditsP (toRoadStop fromRoadStop)
Returns T if there are any bandits parked between *toRoadStop* and *fromRoadStop,* NIL otherwise.

DirectionOf (toRoadStop fromRoadStop)*
Returns the direction of travel for going from *fromRoadStop* to *toRoadStop.* If the *fromRoadStop* is not given, then the current location of the player is assumed.

Distance (toRoadStop fromRoadStop)*

Computes the distance between *fromRoadStop* and *toRoadStop.* If the *fromRoadStop* is not given, then the current location of the player is assumed.

PricePerUnit (producerRoadStop)
Returns the buying price per unit of the commodity being sold at the *producerRoadStop.* If the argument is not *a Producer* roadstop, then complains and returns 1.

RoomToParkP (roadStop)
Returns T if there is room to park at *roadStop.*

ISA (instance className)
Returns T if *instance* is an instance of *className.*

Nth (list index)
Returns the *index* element of *list.*

SUBCLASS (class superClass)
Returns T if *class* is same as or a subclass of *superClass.*

The following are available only as methods on **Player** class.

(← player Range)
Computes how far the *player* can move based on the amount of fuel carried on the player's truck.

(← player Range1)
Computes how far the *player* can move in a single turn. This depends on the fuel in the truck and the maximum distance allowed by the game master for that turn.

(← player TimeAtStop)
Returns the time spent by player at the stop where currently parked. Useful when parked at one of the Alice's.

(← player TurnsAtStop)
Returns the number of turns *player* has been parked at the stop where currently parked. Useful when parked at one of the Alice's.

D. Useful Global Variables

1. PlayerInterface (you can also use **PI**)
After doing (← $Truckin New), *PlayerInterface* is bound to the instance of the class *TruckinPlayerInterface* and is used to send messages to the GameMaster for making moves and starting game. You can also get some game information such as *roadStops* and *localPlayers* from this object.

2. Simulator

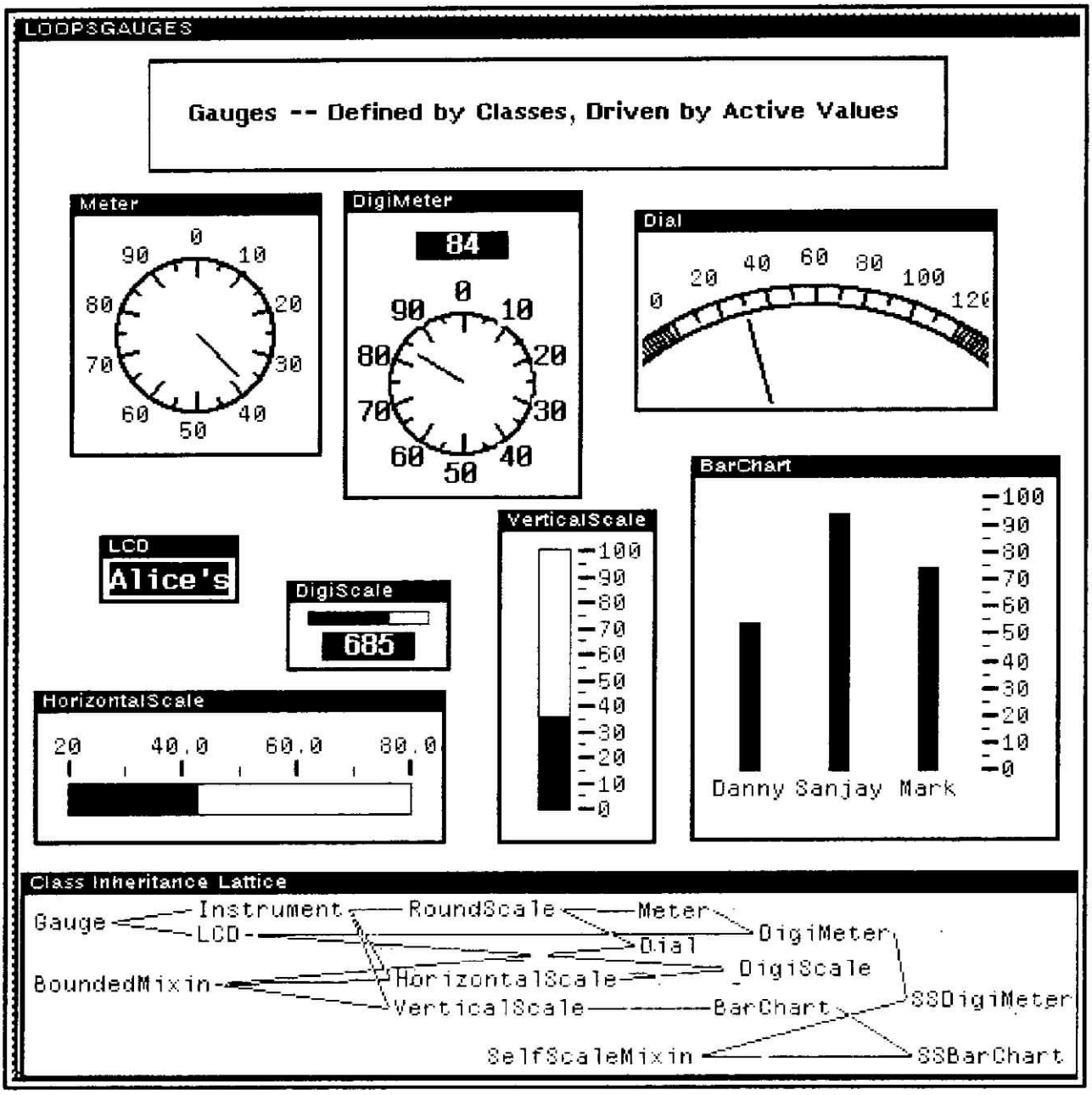
Once the game is set up, *Simulator* is bound to the instance of *TruckinSimulator* and can be used to access important game information such as *roadStops, players, beginTime, endTime, timeLeft.*

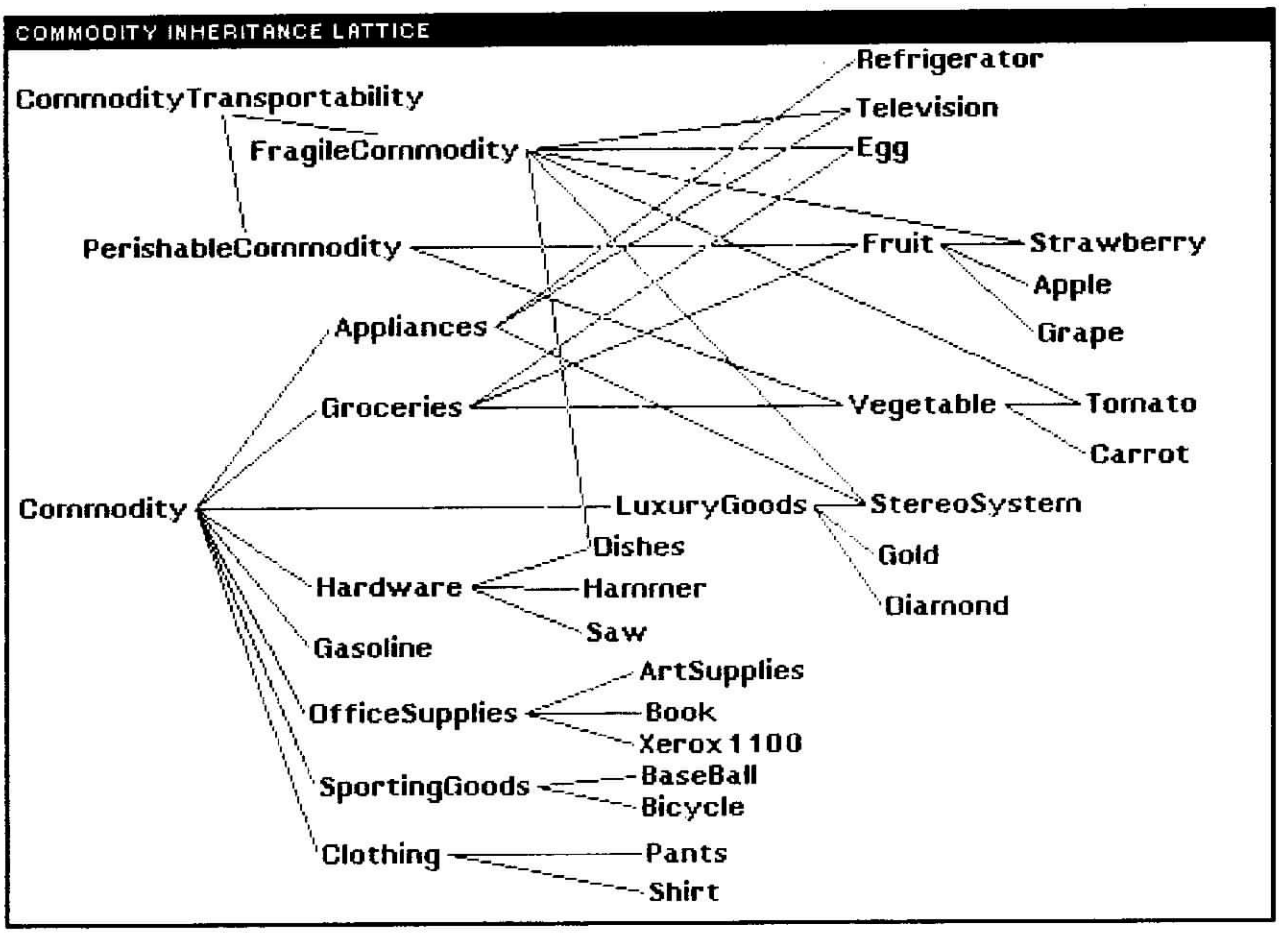
3. debugMode

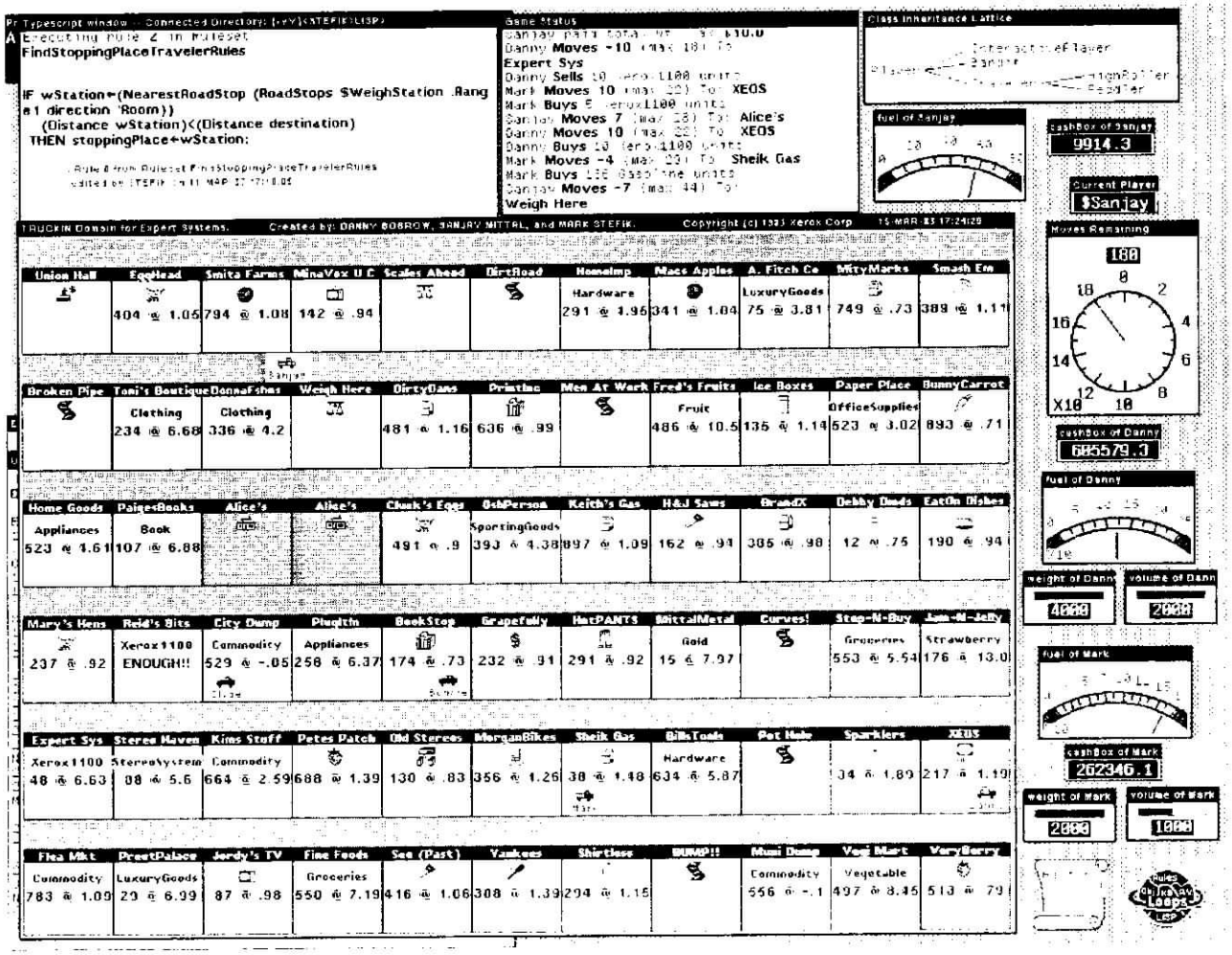
If set to *T*, then each time a rule is violated, the *RuleExec* is automatically brought up. Useful while debugging your rulesets. If set to *NIL*, then the RuleExec is not entered for each rule violation. Also, the GameMaster traps all errors. Initially set to *T*.

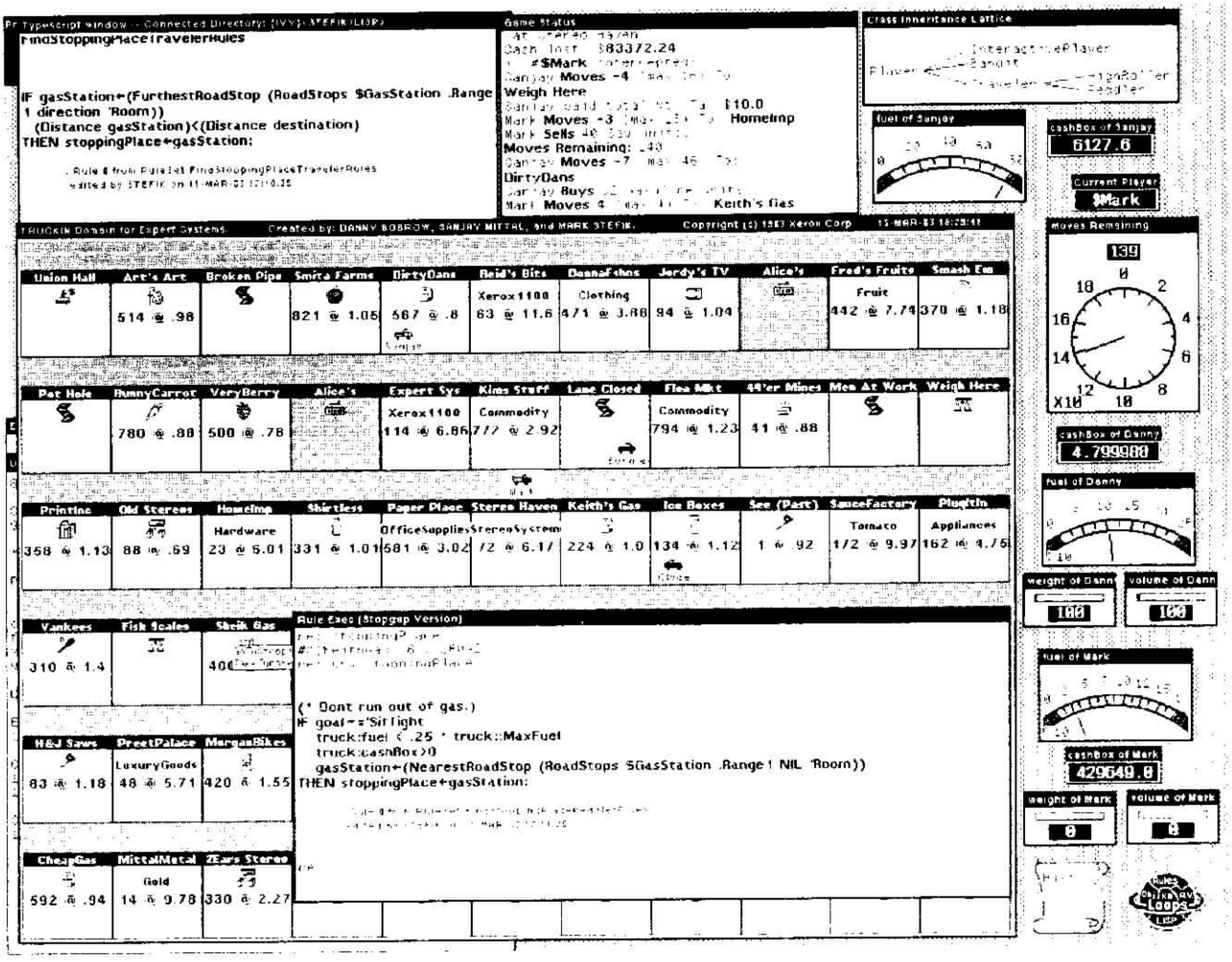
4. truckinLogFlg

If set to *T*, before game is started, then prepares a log file of all important game messages in a file called *TRUCKINLOG*. This log file may be useful during the debugging of your players. Set this variable to NIL, if you dont want any log file. Initially set to NIL.


LOOPSGAUGES
Gauges -- Defined by Classes, Driven by Active Values
Meter
DigiMeter
84
Dial
LCD
Alice's
DigiScale
685
VerticalScale
HorizontalScale
BarChart
Danny Sanjay Mark
Class Inheritance Lattice
Gauge
Instrument
LCD
RoundScale
Meter
Dial
DigiMeter
BoundedMixin
HorizontalScale
VerticalScale
DigiScale
BarChart
SSDigiMeter
SelfScaleMixin
SSBarChart


COMMODITY INHERITANCE LATTICE
CommodityTransportability
FragileCommodity
PerishableCommodity
Appliances
Groceries
Commodity
Hardware
Gasoline
OfficeSupplies
SportingGoods
Clothing
Refrigerator
Television
Egg
Fruit
Strawberry
Apple
Grape
Vegetable
Tomato
Carrot
LuxuryGoods
StereoSystem
Gold
Diamond
Dishes
Hammer
Saw
ArtSupplies
Book
Xerox1100
BaseBall
Bicycle
Pants
Shirt


Pr Typescript window -- Connected Directory: [rrV]<STEFIK.LISP>
Executing Rule 2 in RuleSet
FindStoppingPlaceTravelerRules
IF wStation←(NearestRoadStop (RoadStops $WeighStation .Rang
e1 direction 'Room))
(Distance wStation)<(Distance destination)
THEN stoppingPlace←wStation:
Game Status
Danny Moves -10
Expert Sys
Danny Sells
Mark Moves 10
Mark Buys
Sanjay Moves 7
Danny Moves 10
Danny Buys
Mark Moves -4
Mark Buys
Sanjay Moves -7
Weigh Here
Class Inheritance Lattice
Fuel of Sanjay
CashBox of Sanjay
9914.3
Current Player
$Sanjay
TRUCKIN Domain for Expert Systems. Created by: DANNY BOBROW, SANJAY MITTAL, and MARK STEFIK. Copyright (c) 1983 Xerox Corp
Moves Remaining
180
CashBox of Danny
685579.3
Fuel of Danny
Weight of Danny
40000
Volume of Danny
2000
Fuel of Mark
CashBox of Mark
262346.1
Weight of Mark
2000
Volume of Mark
1000
Union Hall
Eggshead
404 @ 1.05
Smith Farms
794 @ 1.08
MinaVox U.C.
142 @ .94
Scales Ahead
DirtRoad
Homelmp
Hardware
291 @ 1.95
Mace Apples
341 @ 1.04
A. Fitch Co.
LuxuryGoods
75 @ 3.81
MittyMarks
749 @ .73
Smash Em
388 @ 1.11
Broken Pipe
Toni's Boutique
Clothing
234 @ 6.68
DonnaFisher
Clothing
336 @ 4.2
Weigh Here
DirtyDans
481 @ 1.16
Printho
636 @ .99
Men At Work
Fred's Fruits
Fruit
486 @ 10.5
Ice Boxes
135 @ 1.14
Paper Place
OfficeSupplies
523 @ 3.02
BunnyCarrot
893 @ .71
Home Goods
Appliances
523 @ 1.61
PaigesBooks
Book
107 @ 6.88
Alice's
Alice's
Chunk's Eggs
491 @ .9
GoMPerson
SportingGoods
393 @ 4.38
Keith's Gas
897 @ 1.09
H&J Saws
162 @ .94
BrandX
385 @ .98
Debby Deeds
12 @ .75
EatOn Dishes
190 @ .94
Mary's Hens
237 @ .92
Reid's Bits
Xerox1100
ENOUGH!!
City Dump
Commodity
529 @ -.05
Pluglin
Appliances
258 @ 6.37
BookStop
174 @ .73
Grapefruity
232 @ .91
HotPANTS
291 @ .92
MittalMetal
Gold
15 @ 7.97
Curves!
Stop-N-Buy
Groceries
553 @ 5.54
Jam-N-Jelly
Strawberry
176 @ 13.0
Expert Sys
Xerox1100
48 @ 6.63
Stereo Haven
StereoSystem
08 @ 5.6
Kims Stuff
Commodity
664 @ 2.59
Petes Patch
688 @ 1.39
Old Stereos
130 @ .83
MorganBikes
356 @ 1.26
Sheik Gas
38 @ 1.48
BikTools
Hardware
634 @ 5.87
Pet Mike
Sparklers
34 @ 1.89
XEUS
217 @ 1.19
Flea Mkt
Commodity
783 @ 1.09
PrestPalace
LuxuryGoods
29 @ 6.99
Jordy's TV
87 @ .98
Fine Foods
Groceries
550 @ 7.19
Sea (Past)
416 @ 1.06
Yankees
308 @ 1.39
Shirtless
294 @ 1.15
BUMP!!
Miami Dunes
Commodity
556 @ -.1
Veg Mart
Vegetable
497 @ 8.45
VeryBerry
513 @ .79


Pr Typescript Window -- Connected Directory: [VVV]:STEFIK.LISP
FindStoppingPlaceTravelerRules
IF gasStation←(FurthestRoadStop (RoadStops $GasStation .Range
1 direction 'Room))
(Distance gasStation)<(Distance destination)
THEN stoppingPlace←gasStation;
Game Status
Cash Lost $83372.24
Sanjay Moves -4
Weigh Here
Mark Moves -3
Mark Sells
Moves Remaining: -40
Sanjay Moves -7
DirtyDans
Sanjay Buys
Mark Moves 4
Keith's fias
Class Inheritance Lattice
fuel of Sanjay
cashBox of Sanjay
6127.6
Current Player
$Mark
TRUCKIN Dossier for Expert Systems
Created by: DANNY BOBROW, SANJAY MITTAL, and MARK STEFIK.
Copyright (c) 1983 Xerox Corp
Moves Remaining
139
cashBox of Danny
4.799980
fuel of Danny
weight of Danny
100
volume of Danny
100
fuel of Mark
cashBox of Mark
429640.0
weight of Mark
0
volume of Mark
0
Union Mall
Art'e Art
Broken Pipe
Smith Farms
DirtyDans
Reid's Bits
Dannel'shos
Jordy's TV
Alice's
Fred's Fruits
Smash Em
514 @ .98
821 @ 1.05
567 @ .8
Xerox1100
63 @ 11.6
Clothing
471 @ 3.88
94 @ 1.04
Fruit
442 @ 7.74
370 @ 1.18
Pot Hole
RunnyCarrot
VeryBerry
Alice's
Expert Sys
Kims Stuff
Lane Closed
Flea Mkt
49'er Mines
Men At Work
Weigh Here
780 @ .88
500 @ .78
Xerox1100
114 @ 6.86
Commodity
777 @ 2.92
Commodity
794 @ 1.23
41 @ .88
Printine
Old Stereos
Homelmp
Shirtless
Paper Place
Stereo Haven
Keith's Gas
Ice Boxes
See (Part)
SauceFactory
Plugthn
358 @ 1.13
88 @ .69
Hardware
23 @ 6.01
331 @ 1.01
581 @ 3.02
OfficeSupplies
72 @ 6.17
StereoSystem
224 @ 1.0
134 @ 1.12
1 @ 92
Tomato
172 @ 9.97
Appliances
162 @ 4.76
Yankees
Fish Scales
Sheik Gas
310 @ 1.4
Rule Exec (Stopgap Version)
(* Dont run out of gas.)
IF goal~='SitTight
truck:fuel < .25 * truck::MaxFuel
truck:cashBox>0
gasStation←(NearestRoadStop (RoadStops $GasStation .Range1 NIL 'Room))
THEN stoppingPlace←gasStation;
H&J Saws
PrestPalace
MarsanBikes
LuxuryGoods
83 @ 1.18
48 @ 5.71
420 @ 1.55
CheapGas
MittalMetal
Zeus Stereo
Gold
592 @ .94
14 @ 9.78
330 @ 2.27

XVIII

(1)

LOOPS Summary

LOOPSNames (← obj SetName 'FOO) gives obj the Loops name FOO

$FOO evaluates to object named FOO #$FOO reads in as object named FOO

Variable Access Read macros and their translation

form	*translation*	*form*	*translation*
@X	(GetValue self 'X)	@X←newValue	(PutValue self 'X newValue)
@(Obj X)	(GetValue Obj 'X)	@(Obj X)←newValue	(PutValue Obj 'X newValue)
@(Obj X P)	(GetValue Obj 'X 'P)	@(Obj X P)←newValue	(PutValue Obj 'X newValue 'P)
@@X	(GetClassValue self 'X)	@@X←newValue	(PutClassValue self 'X newValue)
@@(Obj X)	(GetClassValue Obj 'X)	@@(Obj X)←newValue	(PutClassValue Obj 'X newValue)
@@(Obj X P)	(GetClassValue Obj 'X 'P)	@@(Obj X P)←newValue	(PutClassValue Obj 'X newValue 'P)
		@X←+newValue	(PushValue self 'X newValue)

Defining and Editing Classes

DC(className New *classNameList*) e.g. DC (StudentEmployee (Student Employee))

(← *class* New *classNameList*) e.g. (← $Class New 'StudentEmployee '(Student Employee))

(← $StudentEmployee Edit) or EC(StudentEmployee)

Sending Messages

(← *object Selector arg1 ... argn*)

e.g. (← $PayRoll PrintOut 'payFile)

(←Super object selector arg1 ...)

e.g. (←Super self Edit commands)

(DoMethod *object selector class arg1 arg2 ...*)

e.g. (DoMethod X PP $Object 'file1)

Creating, Editing,, Inspecting Instances

(← *class* New) e.g (← $Transistor New)).

propName)

(← *object* Edit) e.g EI(*myInstance*)

(← *object* Inspect) e.g (← *$Foo Inspect*)

Defining and Editing Methods

DM(*className selector*)

edit template definion of method .

DM(*className selector fnName*)

fnName is unction implementing the method.

DM(*className selector argsOrFnName form*)

e.g. DM(Number Increment (self)

((* add1 to myValue)

@myValue←(ADD1 @myValue)))

EM (*className selector*)

edit method used in className

Active Values

#(*localState getFn putFn*)

e.g. #(*37* PrintFetcher StopSmasher)

DefAVP(getFn) edit template for getFn named getFnName

args: (*self varName oldValue propName activeVal type*)

DefAVP(putFnName T) edit template for putFnName

args: (*self varName newValue propName activeVal type*)

GetLocalState (*activeValue self varName propName*)

PutLocalState (*activeValue newValue self varName*

Examples of Useful Active Value Forms

#((RANDOM 1 10) FirstFetch)

Replace me by a random number when first fetched

#(Initial (DATE))

Replace me by today's date on initialization

#((self rightPoint) GetIndirect PutIndirect)

Put and get my value from my rightPoint

Debugging

BreakMethod (*className selector*)

TraceMethod (*className selector*)

BreakIt (*self varName propName type breakOnGetAlsoFlg*)

TraceIt (*self varName propName type breakOnGetAlsoFlg*)

UnBreakIt (*self varName propName type*)

Saving Classes and Instances

(CLASSES * *classNameList*) Saves class definitions on files

(INSTANCES * *instanceNameList*) Saves named Instances on file, and instances pointed to by them.

(← $KB New *'KBName 'environmentName newVersionFlg*) Create new KB attaached to Environemnt

(← $*environmentName* Open) Open Environment for reading and writing

(← $*environmentName* Cleanup) *Save instance and class data in a KB*

(← $*environmentName* Close) *Close Environment and release attached KB*

(← $KB Old *'KBName 'environmentName*) Connect an old KB to new environment.

(← $*environmentName* Erase) Cancel an entire session