UO-LISP NEWSLETTER

| January 1984 | Vol. 1. No. 1 |

Premier Edition

The growing number of UO-LISP users has prompted Far West to publish this newsletter to acquaint users with new software being offered, provide tips on use of the system, and broadcast bugs and fixes. Each issue will provide news of upcoming events of interest to LISP programmers. A section will be devoted to answering questions from users and as a special feature, each issue will have a complete LISP program designed to illuminate to some aspect of the UO-LISP system.

We encourage you to submit programs, questions, articles, and news of coming events. Our close connection with Lisp means that we are not always in touch with needs of beginning users. Your problem may be one which has not occurred to us and an explanation of its solution will be of benefit to the entire community. Or perhaps you have a program or utility which is of use to everyone. You will be a much more productive LISP programmer if you can build on the tools provided by others and do not have to "reinvent the wheel" for each new program.

New Software

Many improvements have been made in UOLISP since the release of version 1.5 for CP/M and the TRS-80. The interpreter has been augmented, the number of support packages has doubled, and a large set of example programs in LISP and RLISP have been implemented. The following changes have occurred since version 1.5.

The Manual

Perhaps the biggest change has been to the UO-LISP User's Guide. The new manual is well over 300 pages long and is divided into 12 chapters. It includes new sections on the document formatter, big and fixed numbers, the revised structure editor, the utility packages, the macro packages, and the Little Meta Translator Writing system. In addition, nearly every function in the entire manual is now described with an example of its operation. Since the manual has gotten so large, we have also created a handbook which includes all of the functions in the system, a synopsis of commands for the editors, text processor,

and other systems, and a list of all packages in the system. To keep the manual size reasonable, we have also created a report series for the application and demonstration programs.

Version 1.14c Interpreter

1. COMPRESS, EXPLODE, and EXPLODE2 have been implemented. COMPRESS takes a list of single character identifiers and builds a lisp expression out of it. EXPLODE and EXPLODE2 create lists of characters from S-expressions.

2. The N2I function converts integers into single character identifiers. I2N converts single character identifiers into their corresponding numbers.

3. MACRO type functions have been implemented. MACRO functions permit implementing WHILE...DO..., FOR loops, data structuring in LISP without overhead (when compiled).

4. DIGIT and LITER return T when their arguments are single characters which are digits and letters of the alphabet.

5. Up to 4 disk files may be open at any time in any combination of input or output. Two other channels are connected to the CP/M print and read devices. The user can also install his own I/O drivers on any channel through the use of the INSTALL function.

6. The FLUID variable binding mechanism is implemented. FLUID variables are like GLOBAL variables, but can be used as function parameters or as PROG variables. They also permit variable name communication between interpreted and compiled code.

7. A backtrace on error mechanism is implemented. When an error occurs, the contents of the ALIST and frame stack are displayed.

8. The BPS!$ function displays the number of available identifiers, free code pointers, string space available, and the amount of remaining binary program space.

9. The loader and compiler check for overrunning the binary program space.

10. Packages which have been loaded by the fast loader can be unloaded and their binary program space and code pointers can be used for other packages.

11. A Coroutine mechanism has been implemented. This permits executing up to 5 processes concurrently.

*12. The disk I/O routines permit random access files to be used.

13. Many bugs have been fixed and many of the basic interpreter functions have been recoded for speed and size improvement.

14. A short call mechanism permits the compiler to generate two byte function calls for many functions with significant savings in the size of compiled code (10%).

15. The garbage collector removes unreferenced identifiers from the symbol table.

16. COND and LAMBDA now support an implied PROGN. The antecedent of a COND or the body of a LAMBDA can be a set of S-expressions without the necessity of adding a PROGN.

17. The PROG1 function has been implemented. It is like PROGN, but the value returned is the first of its statements rather than the last.

18. The support functions EQCAR (equivalent to (EQ (CAR ..) ..)), REVERSIP (like REVERSE but does so in place destroying the original list structure), LEQ, NEQ, and GEQ.

19. The COMMENT function has been implemented as a way of saving annotation in a LISP file.

20. The vector data type has been moved inside the interpreter.

21. Read macros and a read table are now supported.

The Compiler and Optimizer

1. The basic compiler optimizes calls to LIST that have 1, 2, and 3 arguments to the equivalent functions NCONS, LIST2, and LIST3.

2. Implied PROGN is supported in COND's and LAMBDA expressions.

3. The PROG1 function is open compiled.

4. Improvements have been made to the LAP pretty printer and LAP interface.

5. Lambda expresssions as functions are compiled. Thus: ((LAMBDA (X) (ADD1 X)) 12) is compiled correctly.

6. Improvements have been made in the code generated by the basic compiler.

7. The FLUID variable type is supported in compiled code.

8. The two byte call mechanism is supported by the compiler as an option.

9. The optimizer does argument reordering to take advantage of registers.

10. Functions are listed as they are compiled during the fast load file generation process.

11. The compiler and interpreter have been fixed to allow compilation of functions with more than 3 arguments (up to 63 are permissible).

RLISP

1. The terse printer is interfaced to RLISP.

2. Some improvements have resulted in a reduction in the size of the RLISP code.

3. Big numbers and fixed numbers are supported in source code.

The Trace Package

1. The basic package has been simplified by the removal of the BREAK function and the output improved. The Terse printer is interfaced to the output facility. The size of a traced function has been reduced.

2. An extended trace package implements tracing of FEXPR's as well as assigments to variables. A program BREAK facility permits selective tracing and environment examination at run time.

3. An execution profiling package has been implemented. This package lists the number of times a function gets called during the execution of a program.

The Document Formatting Program LISPTEX

The document formatting package implements LISP based word processing. The formatter does text justification, centering, page numbering, index and table of contents maintenance, table formatting, and switching between different fonts. This document was formatted and printed by LISPTEX.

The Structure Editor

The structure editor has been completely rewritten. A file package constructs and maintains multiple copies of the source file by attaching a version number to the file name. More than one file can be edited at any time. The structure editor contains many new commands including the ability to perform editing on more than one function or structure at a time. There is a limited ability to maintain comments in the source file. The editor also contains a character editor, a facility for editing the individual characters of an atom or structure.

New Packages

Several new support packages have been implemented and many old ones have been updated.

1. CP/M operating system support. This package includes a number of routines which perform calls on the CP/M disk and basic I/O routines. A second package implements more complex calls and permits interaction with the CP/M filing system for deletion and renaming of files as well as directory search.

2. Fast arithmetic, bit-logical operations, and random number generator. This package contains routines for doing fast addition, and subtraction as well as shifting and logical operations on integers. A pseudo-random number generator is implemented.

3. History saving read loop. This package maintains a list of the commands that have been previously entered. These may be examined, edited, and rexecuted.

4. Terse printer. This package complements the pretty printer and can be used with the structure editor and the trace package. It limits the amount of output from PRINT by displaying only the top few levels of a tree structure and the first few elements of an array or list.

5. Internal GLOBAL variables. This package implements named access to many of the internal global variables used in the system.

6. Macro packages. These packages contain a number of compiled macros for advanced control and data structures in LISP. This includes CASE, IF, FOR, REPEAT, and WHILE macros in the control package, structure definition and assignment functions in the data structures package, and some useful I/O macros in a third package. The fourth package implements the backquote facility for easy construction of macros.

7. Terminal drivers. This is a collection of routines to be loaded with programs that require CRT screen operations. Each terminal type has their own driver program. The source code for the Televideo and ADM22 terminal drivers are included.

8. Sort packages. These two packages implement a list insertion sort and a disk based merge sort so that large numbers of items can be sorted into a disk file.

9. The LSED Screen Editor. This program edits LISP source program files on a character rather than structure basis.

10. Z80 assembler package. This package is an extension to LAP to permit all the Z80 instructions to be used.

*11. File Transfer Program (FTP). This package permits you to communicate with other UOLISP installations and with Far West. The program includes an electronic mail facility, talk

facility, and the file transfer mechanism.

*12. Distributed LISP. This package implements synchronization and communication of process between two or more LISP systems.

*13. Franz LISP compatibility package. This package permits Franz LISP programs to be written and tested. These can then be moved to UNIX version of Franz LISP. Not all of Franz LISP is supported.

14. Auto loading. This package implements automatic loading of functions within the lisp system. A second package implements package swapping.

15. Low level debugging. This package is a byte and address level DDT which permits examination of compiled LISP functions and contents of buffers.

16. The Little Meta Translator Writing System is now available for CP/M systems.

*17. PROLOG. The programming language PROLOG implemented in LISP (contributed by Rabbe Fogelholm of the Royal Institute of Technology, Stockholm, Sweden). This package permits the user to create and debug simple PROLOG programs.

18. A package has been written to support the construction of read macros.

*19. Cross reference program. This program takes a UO-LISP source file and displays information about which functions are called from where, and what GLOBAL and FLUID variables are used.

20. The 'hunk' data structure is a fast byte vector. It is useful for storing single bytes in a fixed length vector and retrieving the value without a list search as in the UO-LISP vector structure. This will speed up many applications by an order of magnitude.

Educational Software

Far West is now offering two packages for those learning LISP. These make UO-LISP look like the LISP presented in various text books. Currently packages exist for 'LISP' by Winston and Horn, and the 'LISP 1.5 Primer' by Clark Weissman.

* item is to be released in the near future.

Demonstration Programs

A number of demonstration programs have been implemented. The

source code for these are distributed together with a document describing each one and examples of its use (where applicable).

1. The SNAKE game. A game for CRT's: the snake gobbles the random numbers that appear on the screen and gets longer. The operator controls movement with characters from the terminal. The game ends when the snake runs into itself or the wall.

2. Othello. A game program which demonstrates some uses of vectors. The program is not very smart though it has been known to win. The documentation describes the rules and the strategy used by the program.

*3. The NLARGE computer algebra system. This program accepts equations and performs operations on them. For example, (X+1)^2 is expanded into (X^2 + 2*X + 1). Bignums permit (X+1)^20 to be expanded (we don't know what the limit is). Polynomials can be added, subtracted divided, multiplied, and differentiated with respect to any variable. The package also includes some matrix manipulations to do computation of determinants, matrix inversion and multiplication. The program treats the algebraic operators as "objects" and is a simple example of "object oriented" programming in LISP. A demonstration program which runs about 10 minutes is included.

4. Fruit world. A simple intelligent system shows how the Little Meta Translator Writing System can be interfaced to a program that knows about fuit and how to make inferences based on this information. The input and output are English sentences.

5. Your Program. Far West is actively soliciting contributions from users. These will either be published in the newsletter or offered as part of the growing UOLISP program library.

UO-LISP Bugs & Complaints

We hope to keep this section small but we won't be foolish enough to deny that there aren't any such things around. The following have come to our attention:

Version 1.5a,b only (TRS-80 Model I, III). The Little META Translator Writing system has a problem running under version 1.5a (not version 1.5 or before). The use of the - sign in the test-x- construct causes problems and the sample distributed programs don't work. To solve this problem, edit these files and place at least one blank after every minus sign.

Version 1.13-1.14b (CP/M system). Characters with a code less than 32 are ignored by the input reader. Consequently assigning entries for them in the read table will not work. This has been fixed in subsequent versions.

Other News

FOLLK (Friends of LISP/Logo & Kids) is a recently formed non-profit Educational & Scientific Corporation based in San Francisco. It publishes a quarterly newsletter with articles describing LISP, Logo, and PROLOG. It also sponsors computer camps, a hot-line for answering questions about LISP, Logo and other AI languages, and monthly FOLLK-Meets. A subscription to the newsletter is $7.50 and a FOLLK regular membership is $25.00 and a student membership is $15.00. FOLLK can be reached at 254 Laguna Honda Boulevard, San Francisco, California 94116, (415)-753-6555. FOLLK is not connected with Far West in any way.

Coming Events

The 1984 ACM Composium on LISP and Functional Programming will be held at the University of Texas in Austin on August 5-8, 1984.

EUROSAM '84 International Symposium on Symbolic and Algebraic computation will be held in Cambridge, England on July 9-11, 1984.

Bibliography

In this section we will list recently published articles and books of interest to the LISP programmer. We would also like to print book and article reviews.

Marti, J., 'The Little Meta Translator Writing System', Software Practice and Experience, October 1983, pp. 78-xx.
Describes the Little Meta Translator Writing System by presenting a syntax checker, an interpreter and a compiler for a small programming language. Recommended reading for Little Meta users (reprints are available on a first come first serve basis from Far West).

Questions and Complaints from the Users

In this section we will present questions we have received from users both over the phone and by letter as well as the best answers we can give.

Can I reconfigure the data spaces for my own applications?
Answer: No. Z80 code is not very relocatable and not every CP/M or TRS-80 system has a relocating linker. We have spent considerable time tuning the data spaces so that most applications will run without reassembling the system. The 8086 version of UO-LISP will be statically reconfigurable. The Z80 system can be easily be reconfigured by Far West upon request (please call us to discuss your needs). For example we could create a version which permitted you to have 20 files open at one time, or a version with 32k binary program space (at the expense of stack and dotted-pair space), or most any other special request you might have, as long as it will fit into 64k.

I'm getting really sick of seeing the CAR and CDR of NIL error

message.
There are two solutions to this problem. The easiest is not to take CAR or CDR of an atom, usually this error is a symptom of some other error. Some LISP dialects (Interlisp and MacLisp for instance) do permit you to take CAR and CDR of NIL, but not other atoms. If this style of programming appeals to you, simply create a special CAR and CDR which check for this special condition (don't forget the composites too):

```
(DE CAR!* (x) (AND (PAIRP x) (CAR x)))
(DE CDR!* (x) (AND (PAIRP x) (CDR x)))
```

Programs

In this first issue we include the following interesting algorithm programmed by Julian Padget of the University of Bath in England. It computes PI to as many decimal places as one wishes to wait for using a method called "continued fractions". The following program can be run as is. The last lines computed PI to 10 and 20 decimal places respectively. We have used this routine to compute PI to 100 decimal places, but it takes about 30 minutes.

```
% Load the packages required.
(FLOAD "USEFUL")
(LOADF "MACROS" "RTABLE" "FIXED")

% !*PREC is precision to compute to, default=10.
(GLOBAL '(!*PREC))
(SETQ !*PREC 10)

% Define a read macro for big numbers.
(DRM !#
  (WHILE (DIGIT (SETQ C (R!$)))
         (WITH C L)
         (INITIALLY (READCH))
         (RETURNS (MKQUOTE L))
         (DO (SETQ L (CONS (I2NO C) L))
             (READCH))))
(DE I2NO (C) (DIFFERENCE (I2N C) 48))


% Compute PI for N iterations.
(DE PI (N)
  (FOR (WITH AN2 AN1 AN BN2 BN1 BN TMP1 TMP2)
       (INITIALLY (SETQ AN2 #0)
                  (SETQ AN1 #1)
                  (SETQ BN2 #1)
                  (SETQ BN1 #1))
       (FROM I 2 N)
       (DO
         (SETQ TMP1 (BEXPT (BIGNUM (SUB1 I)) 2))
         (SETQ TMP2 (BSUB1 (BTIMES2 #2 (BIGNUM I))))
         (SETQ AN (BPLUS2 (BTIMES2 TMP1 AN2)
                          (BTIMES2 TMP2 AN1)))
         (SETQ BN (BPLUS2 (BTIMES2 TMP1 BN2)
```

```
                        (BTIMES2 TMP2 BN1)))
          (SETQ AN2 AN1) (SETQ AN1 AN)
          (SETQ BN2 BN1) (SETQ BN1 BN) )
        (RETURNS (!$QUOTIENT
                   (BTIMES2 #4 (CONS 0 AN))
                   (CONS 0 BN)))))

% Gets about .75 digits/iteration.
(!$PRINT (PI 16))
(SETQ !*PREC 20)
(!$PRINT (PI 28))
```

This program uses both the FIXED and BIGNUM packages as well as the MACROS, RTABLE, and USEFUL packages. Note that FIXED automatically causes the BIGNUM package to be loaded if it is not already so. The read macro for defined by the call to DRM is used to all big numbers to be used in the source. Any number prefixed by an # will be converted into big number format. The # read macro uses must of the features of the WHILE macro which is defined in the MACROS package. The argument of the PI function is the number of iterations to perform. The algorithm computes slightly less than .75 digits per iteration.

Next Issue

The next issue will appear in April 1984 and features articles on the structure editor and a Little Meta translator for a subset of LOGO.

UO-LISP NEWSLETTER

| January 1985 | Vol. 2 No. 1 |

IBM PC Version Announced

Northwest Computer Algorithms is pleased to announce the immediate availability of UO-LISP Version 3.0 for the IBM Personal Computer. Two versions are being distributed: the **Learn Lisp for $95.00** and the basic **compiler Version 3.0 for $150.00.** The minimum requirements are 128k of main memory, at least one 320k double sided double density floppy disk drive, and PC-DOS 1.1.

Basic Compiler Version

Version 3.0 is an extended version of UO-LISP Version 1.16a. It supports a full 8k free pairs, 2k identifiers, 1k compiled functions, 8k string space, and up to 500k or more of binary program space permitting execution of very large programs. The package includes:

1. The Version 3.0 UO-LISP interpreter
2. The UO-LISP compiler for the Intel 8086 microprocessor
3. The trace packages, the execution profiler
4. RLISP
5. The structure editor
6. LISPTEX
7. BIGNUMs, FIXED numbers
8. Basic PC-DOS interface routines
9. Basic screen driver permitting setting of all screen charactersistics and color graphics primitives
10. History saving read loop
11. Sort packages
12. Terse and pretty printers
13. All the macro packages

Little Meta is available as a separate package.

IBM-PC Learn Lisp System

The IBM-PC version of the **Learn Lisp** system is identical to that of the Z80 system with the exception of larger data spaces and built-in access to the system read table.

Differences between the Z80 and 8086 Versions

There are very few differences between the two systems. Listed below are the major changes you would expect to find when moving from Z80 Version 1.16a to 8086 Version 3.0.

1. More data space is available.

2. Read macro support is built-in, not part of the RTABLE file.

3. Most system functions are defined in Lisp. With the !*FLINK flag set to NIL, most of the system functions can be safely redefined at any time.

4. Error messages have been improved and more bad conditions cause errors.

5. Print macros are defined for output of special data types.

FUTURE PLANS

Northwest Computer Algorithms plans several new product releases and revisions in both the Z80 CP/M and the 8086 system in the coming monthes.

1. A revised version of the Z80 CP/M manual over 400 pages long including all new packages and more examples.

2. An optimizer and assembler for the 8086 as well as the rest of the packages from the Z80 system.

3. Version 3.1 of the 8086 version (to be released this spring) will support 32k free pairs, 16k string space, 4k identifiers, 4k compiled function pointers, and 300k compiled code space on large machines. The new system will also include a fast floating point package with 8087 capability. All previous programs will be upwardly compatible with this new version.

4. Common Lisp. Version 3.1 will feature a large subset of Common Lisp, the proposed ARPA standard for Lisp.

5. Objects (flavors). An objects package is currently being polished for release on all systems.

6. Application programs. Several large AI application programs are being prepared for distribution. This includes the complete REDUCE algebra system and an expert system writing tool.

Version 1.16a Z80 (CP/M)

There have been a few minor bug fixes to version 1.16 of the interpreter and some of the packages as follows:

The Interpreter

1. The FLUID function has been changed so that the initial binding is created on the bottom of the association list rather than the top. The previous version exhibited the behavior of losing the top level FLUID binding if it was created inside an interpreted PROG or ERRORSET.

2. The PUTD function has been changed to do a REMD before the actual function definition. The previous version would not allow you to change the type of a function (from EXPR to FEXPR for instance), but rather would leave both definitions on the property list and use the EXPR one during evaluation.

The Compiler

1. The compiler was modified so that function definition occurs only after the compilation process has completed. This permits compilation of a function which is redefining a previous function which is also being used during the compilation process.

2. The FASLOUT function can be used by application programs that define functions at read time (such as Little Meta and the objects package) to place function definitions and other forms in fast load files.

A SIMPLE OBJECT SYSTEM

In this issue we present a simple object oriented programming system that you can implement and run on any basic CP/M, PC-DOS, or MS-DOS system. It requires on the MACROS and STRUCT packages (the Z80 version), or the MACROS package (8086 version). In the next issue we will present a simple simulation program (with graphics) that uses the system to simulate the behavior of several flying airplanes.

"Object Oriented Programming" is not a new idea. The designers of SIMULA and SMALLTALK pursued this style of programming in the 60's. It is embodied in contemporary Lisp system technology as "flavors". This article presents a small subset of the flavor system capabilities designed specifically for simulation. There are many other applications for object based systems that we will not relate here.

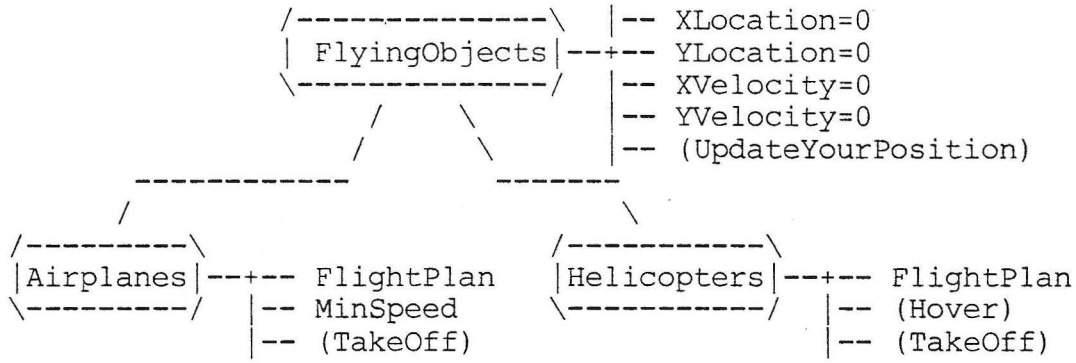
A small collection of Lisp functions forms an "object system". This includes functions for creating a class hierarchy of conceptual objects, functions for creating objects (members of the classes),

functions for describing behaviors (called "methods") of the objects, and functions for accessing the behaviors and values associated with the objects and classes. We examine each of these as they relate to a simulation of one or more airplanes flying in an airspace.

In the boxes in the following text is a complete object programming system. Examples of the use of the system are contained in code not in boxes.

CLASSES

A simulation class is a place holder for a collection of objects with common characteristics. A simulation program arranges classes in a tree structured hierarchy with the most general at the top and the most specific at the bottom. To understand this concept, consider two different types of flying objects and a possible classification scheme.



There are two important types of information in this structure: the relationships between classes and the quantities and functions associated with each class. Here we see that airplanes and helicopters are both "FlyingObjects" but that airplanes have a minimum safe speed and helicopters do not. Both objects have a current 'XLocation' and 'YLocation' (initially 0), velocities 'XVelocity' and 'YVelocity' and a procedure, 'UpdateYourPosition', to move the object to a new location. However, both objects have a procedure called Takeoff, but that associated with airplanes is different than that of helicopters.

The most important aspects of a class are: its superclasses, that is, classes higher in the hierarchy, its variables and their default settings, and a set of procedures that define operations on all members of the class and its subclasses. In our small system classes are declared by the CLASS macro which collects the variables and superclass information.

CLASS accepts two or more arguments. The first is the unquoted name of the class being defined. The second is a list of one or more classes from which this class will inherit varibles and methods. The remaining arguments, by convention, are elements of an association list of variables and default values for the class. We have defined CLASS to simply put this information on the property list of the

class name under the indicators SUPERCLASSES and VARIABLES without performing any evaluation of its arguments. The macro returns the name of the class being defined.

The class hierarchy of airplanes and helicopters would be defined as follows.

```
*(CLASS FlyingObjects (ROOT)
*   (XLocation . 0)
*   (YLocation . 0)
*   (XVelocity . 0)
*   (YVelocity . 0)
*   (FlightPlan . NIL))
FlyingObjects

*(CLASS Airplanes (FlyingObjects)
*  (MinSpeed . 0))
Airplanes

*(CLASS Helicopters (FlyingObjects))
Helicopters
```

Please note that we haven't really created any objects yet, just classes of objects and their relationships. The first CLASS declaration creates the class of FlyingObjects. It is a subclass of the special ROOT class, the global class from which all objects inherit the most basic methods.

The CLASS macro seen below merely places the values of its arguments on the property list of the CLASS name. The superclass list goes under the indicator SUPERCLASSES and the variable list under the indicator VARIABLES.

```
(DEFMACRO CLASS (cname supers . vars)
% Declare class 'cname' with the superclasses list
% 'supers' and the alist of variables 'vars'.
  `(PROGN (PUT ',cname 'SUPERCLASSES ',supers)
          (PUT ',cname 'VARIABLES ',vars)
          ',cname))
```

INSTANCES

We call the simulation of a real world object an "instance". This instance involves both a data structure and operations on it. We first examine the creation of the data structure and then the declaration of "methods" for operating on it. The INSTANCE macro creates a data structure corresponding to the state of a particular simulation object. The INSTANCE declaration contains first the class name of which the INSTANCE is a member. INSTANCE then requires an unquoted name with which the data structure will be associated. This is followed by an association list of variables and values. For example, the following INSTANCE declaration creates the data structure for a particular airplane and assigns an initial location, minimum speed, and flight plan to it.

```
*(INSTANCE Airplanes N7374D
*   (XLocation . 34)
*   (YLocation . 75)
*   (MinSpeed . 45)
*   (FlightPlan . '((MOVE 45 0) (MOVE 0 45))))
N7374D
```

You can create as many instances as you have storage provided that each has a special name. The variables of an instance are derived from its CLASS declaration and all variables higher up the hierarchy. Thus, MinSpeed is a variable which only the Airplanes class has, while XLocation, YLocation, XVelocity, YVelocity, and FlightPlan are shared by all FlyingObjects. More precisely: all variables are inherited from higher levels of the hierarchy. If two or more declarations of a variable exist, the lowest level value takes precedence over the higher. Thus the XLocation value of 34 takes precedence over the default value of 0 provided by the FlyingObject CLASS declaration.

The INSTANCE macro performs several operations. It first collects all variables into a large association list by wandering up the hierarchy. The pair of functions COLLECTVARS and CVARS1 builds an alist with the following rules:

1. INSTANCE variable declarations have the highest precedence.

2. The variables of the class from which the instance is derived superseed variables with the same name of higher level classes.

3. The final list of variables will have one occurrence of every variable from the instance declaration and the class hierarchy starting at the class of the instance.

```
(DEFMACRO INSTANCE (cname iname . vars)
% Declare instance 'iname' with the variables from
% the class 'cname' and set the variables in the
% alist 'vars' to new values. Drag down all variables
% and behaviors from the class structure.
  `(PROGN (PUT ',iname 'VARIABLES
               (COLLECTVARS (LIST ',cname) ',vars))
          (PUT ',iname 'CLASS '(,cname))
          ',iname))

(DE COLLECTVARS (classes vars)
% Collect the variables from the list of classes
% augmenting the list as we go higher in the
% structure. Lowest level variable declarations
% have the highest precedence.
  (IF (NULL classes) THEN vars
   ELSE (COLLECTVARS
          (APPEND (CDR classes)
                  (GET (CAR classes) 'SUPERCLASSES))
          (CVARS1 (GET (CAR classes) 'VARIABLES)
                  vars))))

(DE CVARS1 (nl vars)
% 'nl' is an alist of variables and values to merge
% into the old list 'vars'. Variables are added to the
% list only if they are not already there.
(IF (NULL nl) THEN vars
 ELSEIF (ATSOC (CAAR nl) vars) THEN
         (CVARS1 (CDR nl) vars)
 ELSE (CVARS1 (CDR nl) (CONS (CAR nl) vars))))
```

METHODS

To manipulate the data of the INSTANCE structure we provide a mechanism for declaring functions. This process is complicated by the desire to allow multiple definitions of the same function, perhaps one for each class. For example, the function FlyTo might have different definitions for different classes: one FlyTo specific to Airplanes and one specific to Helicopters. We declare functions with the METHOD macro. The first argument of METHOD is the CLASS name with which the function is to be associated. The second is the name of the method followed by an argument list, and a body. METHOD creates a function definition whose name is formed from the CLASS name and the method name.

Before we look at a particular method, we must examine the means for invoking a method. For historical reasons, the invocation of a method is called "sending a message", consequently the procedure for calling a method is named SEND. The first argument of SEND is the name of the instance containing the data structure on which the

method is to operate. This argument can be computed at run time, while fixed instance names must be quoted. The second argument of SEND is the name of the method to be invoked. Subsequent arguments are actual parameters of the method.

So that methods can access the particular instance they are to operate on, the METHOD function creates an implied parameter, SELF. This parameter is aways bound to the real name of the instance being operated upon. SELF can be used as a local variable inside the method.

Methods are inherited through the hierarchy in the same manner as variables, though the process occurs at run time. The special class ROOT, contains two very useful methods named SET!-YOUR and GET!-YOUR. Since most classes are subclasses of ROOT, these two functions are always accessible. SET!-YOUR corresponds to SETQ and is used to change the value of an instance variable. For example, to change the XVelocity of the instance N7374D we created earlier, I would perform the following:

```
*(SEND 'N7374D 'SET!-YOUR 'XVelocity 32)
32
```

The GET!-YOUR function performs the opposite task of retrieving the value of an instance variable. To update the XLocation of N7374D I would enter the following.

```
*(SEND 'N7374D 'SET!-YOUR 'XLocation
*     (PLUS (SEND 'N7374D 'GET!-YOUR 'XVelocity)
*           (SEND 'N7374D 'GET!-YOUR 'XLocation)))
66
```

Now, for an entire method: let us create a function which updates the position of a FlyingObject by adding its X and Y velocity components to its current X and Y locations. The method has no arguments other than the implied name of the instance being modified.

```
*(METHOD FlyingObject UpdateYourPosition ()
*  (SEND SELF 'SET!-YOUR 'XLocation
*        (PLUS (SEND SELF 'GET!-YOUR 'XVelocity)
*              (SEND SELF 'GET!-YOUR 'XLocation)))
*  (SEND SELF 'SET!-YOUR 'YLocation
*        (PLUS (SEND SELF 'GET!-YOUR 'YVelocity)
*              (SEND SELF 'GET!-YOUR 'YLocation))) )
!{FlyingObject!}UpdateYourPosition
*(METHOD FlyingObject PrintPosition ()
*  (PRIN1 (SEND SELF 'GET!-YOUR 'XLocation))
*  (PRIN2 "x")
*  (PRINT (SEND SELF 'GET!-YOUR 'YLocation))
*  NIL)
!{FlyingObject!}PrintPosition
```

Since UpdateYourPosition must operate on any FlyingObject, we use the implied SELF formal parameter to tell SEND what instance XLocation,

YLocation, and the other variables are to be taken from. The PrintPosition method illustrates a simpler use of the functions to display the current X and Y locations of the object specified. Consider the following use of PrintPosition and UpdateYourPosition.

```
*(INSTANCE Airplanes N5445E
*  (XVelocity . 3)
*  (YVelocity . -2))
N5445E

*(SEND 'N5445E 'PrintPosition)
0x0
NIL

*(SEND 'N5445E 'UpdateYourPosition)
-2

*(SEND 'N5445E 'PrintPosition)
3x-2
NIL
```

I've implemented METHOD as a macro so that its arguments need not be quoted. The PUTD inside the METHOD macro first creates a name for the method composed of the class for which the method is defined (enclosed in curly brackets) and the method name. In addition to the normal parameter names of the method, the LAMBDA expression formal parameter list contains the parameter SELF. The SEND function (presented later) always includes the name of the instance being operated on for this parameter. Thus, METHOD, is essentially a DE function call with a funny name for the function and an extra parameter. METHOD returns the constructed name of the method.

```
(DEFMACRO METHOD (cname bname vars . body)
% Define the behavior 'bname' for class 'cname'.
% 'vars' is a list of local parameters, and 'body'
% is a list of expressions to evaluate.
  `(PROGN
     (PUTD (MAKENAME ',cname ',bname) 'EXPR
           '(LAMBDA (SELF @vars) @body))
     (MAKENAME ',cname ',bname) ))

(DE MAKENAME (cname bname)
% Construct a funny name to hide the function.
  (COMPRESS
    (APPEND '(!! !{)
       (NCONC (EXPLODE cname)
          (APPEND '(!! !})
             (EXPLODE bname))))))
```

The SEND function complements METHOD by generating an APPLY to call the appropriate method. The GETB function returns the definition of the method that the message is being sent to (the function to be called). Like the situation for variables, the function must be located in the class hierarchy. Notice that the second argument of

APPLY includes the first argument of SEND, the instance name. This becomes the value of the SELF parameter of every method.

```
(DEFMACRO SEND (inst bname . args)
% Invoke the behavior 'bname' for instance 'inst',
% with evaluated arguments 'args'.
  `(APPLY (GETB (GET ,inst 'CLASS) ,bname)
          (LIST ,inst @args)))

(DE GETB (hierarchy bname)
% Get Behavior definition for 'bname' starting with
% the list of classes 'hierarchy'.
  (IF (NULL hierarchy) THEN
      (ERROR 0 (LIST bname "is not a behavior"))
   ELSE (LET ((df (GETD (MAKENAME (CAR hierarchy) bname))))
         (IF df THEN (CDR df)
           ELSE (GETB
                  (APPEND
                    (CDR hierarchy)
                    (GET (CAR hierarchy) 'SUPERCLASSES))
                  bname)))))
```

The final touches on this objects package are the basic methods for the ROOT class, SET!-YOUR and GET!-YOUR. SET!-YOUR merely does a RPLACD on an element of the alist under the VARIABLES indicator for the instance named in the SEND. GET!-YOUR just returns the value from the alist.

```
(METHOD ROOT SET!-YOUR (var val)
% Root behavior to assign a value to an instance
% variable (no error checking).
  (RPLACD (ATSOC var (GET SELF 'VARIABLES)) val)
  val)

(METHOD ROOT GET!-YOUR (var)
% Root behavior to retrieve the value of an instance
% variable (no error checking).
  (CDR (ATSOC var (GET SELF 'VARIABLES))) )
```

Conclusions

This simple objects package has a large number of deficiencies. In particular, the following improvements would need to be made for use in larger applications:

1. Error checking (there isn't much)
2. Compilation of methods
3. Direct calls to methods rather than a lookup through the hierarchy
4. Faster access to variables
5. Manipulation of inheritance
6. More versions of SEND for FEXPR and MACRO style methods
7. More primitive ROOT methods

In the next newsletter we will present a simple simulation using this objects package.

UO-LISP NEWSLETTER

| April 1985 | Vol. 2 No. 2 |

Please accept our apologies for the lateness of this issue. Your July issue should be on time. We hope you like the new print of the newsletter, as we have just purchased a Hewlett Packard Laserjet printer with which we will be augmenting the print quality of all our documentation.

Version 3.1 Announced

Northwest Computer Algorithms takes pleasure in announcing Version 3.1 of UO-LISP for the IBM-PC computer and compatibles. Version 3.1 is a major enhancement of Version 3.0 featuring larger data spaces and additional functionality. This includes:

- 16k free pairs (8k in V3.0)
- 16k bytes of string space (8k in V3.0)
- Small integers in the range -8192 and 8191
- The Little Meta Translator Writing System
- A compiler option for open coding commonly used functions such as the CAR and CDR composites, small integer arithmetic, and some predicates.
- The MAPOBL function scans the identifier table applying a user supplied function to each symbol.

UO-LISP Version 3.1 requires a minimum of 256k bytes of main storage and PC-DOS 1.1 and higher, or MS-DOS 2.0 and later. The system is source code compatible with Version 3.0 and Version 1.16 (the CP/M version).

Simulation Using the Simple Objects Package

In last months news letter, we presented a simple object based programming system. In this issue we add additional functionality to the objects package and demonstrate its use with a simple simulation of a number of moving interacting objects.

From the last issue, recall that there are a number of built-in behaviors: GET!-YOUR, SET!-YOUR and so on. We now add a behavior to destroy an instance of an object. This function simply removes any properties associated with the object system.

```
(METHOD ROOT KILLYOURSELF ()
% Remove all object properties associated with SELF.
  (REMPROP SELF 'SUPERCLASSES)
  (REMPROP SELF 'VARIABLES)
  (REMPROP SELF 'CLASS))
```

During the course of a long simulation hundreds of objects may be created. KILLYOURSELF frees up storage allocated to an object when it is no longer in use. Since KILLYOURSELF resides at the ROOT of the hierarchy, the behavior can remove any object.

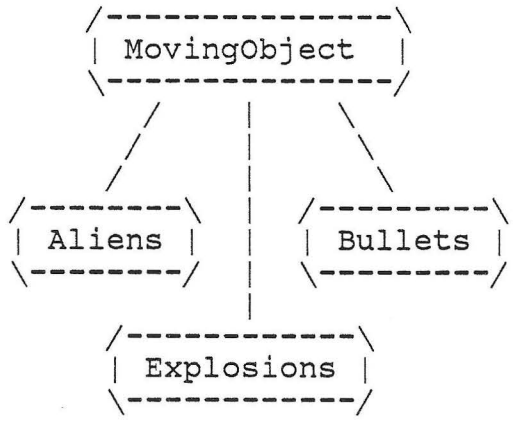
Most simulations require dynamic object creation. The INSTANCE declaration is ill suited to this purpose, hence we introduce an analogous construct, MAKE, for constructing objects at run time. Like INSTANCE, MAKE constructs an object instance but with the added twist that the object's class and name can be computed at run time. The INSTANCE declaration automatically quotes the objects name and class.

```
(DEFMACRO MAKE (cname iname . vars)
% Construct an object instance. This function has the exact
% same arguments as INSTANCE, but cname and iname are
% computed at run time.
  '(PROG (tmp)
     (SETQ tmp ,iname)
     (PUT tmp 'VARIABLES
       (COLLECTVARS (LIST ,cname)
             (FOR (IN x ',vars)
                  (COLLECT (CONS (CAR x) (EVAL (CDR x)))))))
     (PUT tmp 'CLASS (LIST ,cname))
     (RETURN tmp)))
```

You should add these two procedures to the objects package listed in the previous issue.

The following program simulates interactions among some moving objects of different types. So that you can understand

the interactions better, I've called these Aliens and Bullets. Each of these is a type of MovingObject the topmost class is our hierarchy. Basically:



An Explosion is another moving object. Though its behavior is different than that of Aliens and Bullets we attach it to MovingObjects for convenience. This class hierarchy is coded as follows:

```
(CLASS MovingObjects (ROOT)
% MovingObjects have a location and velocity.
   (Xloc . 0)              % Initial X location * 10.
   (Yloc . 0)              % Initial Y location * 10.
   (Xvel . 0)              % Initial X velocity * 10.
   (Yvel . 0) )            % Initial Y velocity * 10.

(CLASS Explosion (MovingObjects)
% An explosion has a location, state (changes with time)
% and picture.
   (Xloc . 0)              % Y Location * 10.
   (Yloc . 0)              % Y Location * 10.
   (State . 0)             % Explosion state.
   (Picture . NIL))        % Explosion picture.

(CLASS Aliens (MovingObjects)
% An alien is a MovingObject with a 4 character picture.
   (Picture . ((-1 0 <) (0 0 *) (1 0 >) (0 1 ^))))

(CLASS Bullets (MovingObjects)
% A bullet is a moving @ sign.
   (Picture . ((0 0 !@))))
```

MovingObjects have only one common procedure. The object erases its previous screen image by writing blanks at all its locations and then gets drawn at a new location. As you can see from

above, each object has a picture associated with it. The picture is a list of 3 element lists. The first two elements of each picture element are x and y coordinates relative to the center of the object at which to display the third element of the list; a character. The CLIPW routine displays a single character at coordinates x and y provided that x and y lie within the screen boundary.

```
% Load the terminal package if not present.
(COND ((NOT (GETD 'CURSOR)) (FLOAD "TERMINAL")))

(GLOBAL '(TERM!-MAXX TERM!-MAXY))

(DE CLIPW (x y c)
% Write a character c at location x y unless it's off
% the screen.
  (IF (NOT (OR (MINUSP x) (GREATERP x TERM!-MAXX)
               (MINUSP y) (GREATERP y TERM!-MAXY))) THEN
      (CURSOR x y)
      (PRIN2 c)))
```

We now provide two routines: one to erase an object by drawing blanks over it, and the other to display it in a new position.

```
(METHOD MovingObjects EraseYourself ()
% Clear an objects screen representation.
  (PROG (lx ly)
     (SETQ lx (QUOTIENT (SEND SELF 'GET!-YOUR 'Xloc) 10))
     (SETQ ly (QUOTIENT (SEND SELF 'GET!-YOUR 'Yloc) 10))
     (FOR (IN pos (SEND SELF 'GET!-YOUR 'Picture))
          (DO (CLIPW (PLUS lx (CAR pos))
                     (PLUS ly (CADR pos)) " "))))))

(METHOD MovngObjects DrawYourself ()
% Draw an object on the screen.
  (PROG (lx ly)
     (SETQ lx (QUOTIENT (SEND SELF 'GET!-YOUR 'Xloc) 10))
     (SETQ ly (QUOTIENT (SEND SELF 'GET!-YOUR 'Yloc) 10))
     (FOR (IN pos (SEND SELF 'GET!-YOUR 'Picture))
          (DO (CLIPW (PLUS lx (CAR pos))
                     (PLUS ly (CADR pos))
                     (CADDR pos))))))
```

The controlling routine, MoveYourself, first erases the object currently on the screen, then updates its position based on its current velocity, and then draws the object at its new location. The key fact to notice is that the MoveYourself routine can be used by all moving objects: in this program, both Aliens and Bullets both use this routine to update their positions.

```
(METHOD MovingObjects MoveYourself ()
% Move an object by clearing it from the screen,
% updating its position, and then redrawing it.
  (SEND SELF 'EraseYourself)
  (SEND SELF 'SET!-YOUR 'Xloc
        (PLUS (SEND SELF 'GET!-YOUR 'Xloc)
              (SEND SELF 'GET!-YOUR 'Xvel)))
  (SEND SELF 'SET!-YOUR 'Yloc
        (PLUS (SEND SELF 'GET!-YOUR 'Yloc)
              (SEND SELF 'GET!-YOUR 'Yvel)))
  (SEND SELF 'DrawYourself))
```

While both Bullets and Aliens move around on the screen, they have slightly different behaviors. A bullet disappears when it goes off the screen. We let the Alien decide what happens when it's hit by a bullet rather than the other way around. To keep track of what bullets are active, we keep a list of active object names for each type of object. Thus, when a bullet goes off the screen, it both deletes its instance data structure (using the KILLYOURSELF behavior) and removes the instance name from the list of active bullets.

```
% Lists of currently active objects.
(GLOBAL '(Bullets Aliens Explosions))

(METHOD Bullets ChangeYourself ()
% If the bullet goes off the screen, erase its object from
% the list of active bullets.
  (PROG (lx ly)
     (SETQ lx (QUOTIENT (SEND SELF 'GET!-YOUR 'Xloc) 10))
     (SETQ ly (QUOTIENT (SEND SELF 'GET!-YOUR 'Yloc) 10))
     (IF (OR (MINUSP lx) (GREATERP lx TERM!-MAXX)
             (MINUSP ly) (GREATERP ly TERM!-MAXY)) THEN
         (SEND SELF 'EraseYourself)
         (SEND SELF 'KILLYOURSELF)
         (SETQ Bullets (DELETE SELF Bullets)) )))
```

In addition to running off the screen and disappearing, Aliens are destroyed by bullets. The collission results in an explosion and disappearance of both objects. In the following code segment, if the alien being examined is sufficiently close to any bullet on the screen, it destroys both objects and signals that an explosion should take place at the point of intersection.

```
(METHOD Aliens ChangeYourself ()
% If an alien runs off the screen, then remove it. If it
% runs into a bullet then start an explosion.
  (PROG (lx ly blowup)
     (SETQ lx (QUOTIENT (SEND SELF 'GET!-YOUR 'Xloc) 10))
     (SETQ ly (QUOTIENT (SEND SELF 'GET!-YOUR 'Yloc) 10))
     (IF (OR (MINUSP lx) (GREATERP lx TERM!-MAXX)
             (MINUSP ly) (GREATERP ly TERM!-MAXY)) THEN
          (SEND SELF 'EraseYourself)
          (SEND SELF 'KILLYOURSELF)
          (RETURN (SETQ Aliens (DELETE SELF Aliens))))
     (FOR (IN b Bullets)
          (WHEN
            (AND
             (EQUAL lx
               (QUOTIENT (SEND b 'GET!-YOUR 'Xloc) 10))
             (EQUAL ly
               (QUOTIENT (SEND b 'GET!-YOUR 'Yloc) 10))))
          (DO (SEND b 'EraseYourself)
              (SEND b 'KILLYOURSELF)
              (SETQ Bullets (DELETE b Bullets))
              (SETQ blowup T)))
     (IF blowup THEN
          (SEND SELF 'EraseYourself)
          (SETQ Explosions
                (CONS
                  (MAKE 'Explosion (SETQ blowup (GENSYM)))
                  Explosions))
          (SEND blowup 'SET!-YOUR 'Xloc
                (SEND SELF 'GET!-YOUR 'Xloc))
          (SEND blowup 'SET!-YOUR 'Yloc
                (SEND SELF 'GET!-YOUR 'Yloc))
          (SEND SELF 'KILLYOURSELF)
          (SETQ Aliens (DELETE SELF Aliens))) ))
```

An explosion is scheduled by the Alien ChangeYourself behavior. An explosion is a sequence of two different pictures that are done during each simulation step. The object variable State associated with each explosion tells which of the two pictures to display and when to terminate the explosion.

```
(METHOD Explosion NextState ()
% Change an explosion to the next state.
  (PROG (state)
     (SETQ state (SEND SELF 'GET!-YOUR 'State))
     (IF (ZEROP state) THEN
        (SEND SELF 'SET!-YOUR 'Picture
           '((-1 0 -) (-1 1 !.) (0 1 |) (1 1 !.) (1 0 -)
             (1 -1 !.) (0 -1 |) (-1 -1 !.)))
        (SEND SELF 'SET!-YOUR 'State 1)
        (SEND SELF 'DrawYourself)
      ELSEIF (ONEP state) THEN
        (SEND SELF 'EraseYourself)
        (SEND SELF 'SET!-YOUR 'Picture
           '((-2 0 -) (0 2 |) (2 0 -) (0 -2 |)))
        (SEND SELF 'SET!-YOUR 'State 2)
        (SEND SELF 'DrawYourself)
      ELSEIF (EQUAL state 2) THEN
        (SETQ Explosions (DELETE SELF Explosions))
        (SEND SELF 'EraseYourself)
        (SEND SELF 'KILLYOURSELF))))
```

The top level driver completes the simulation. It clears the screen and runs the simulation until no more objects are active and then stops.

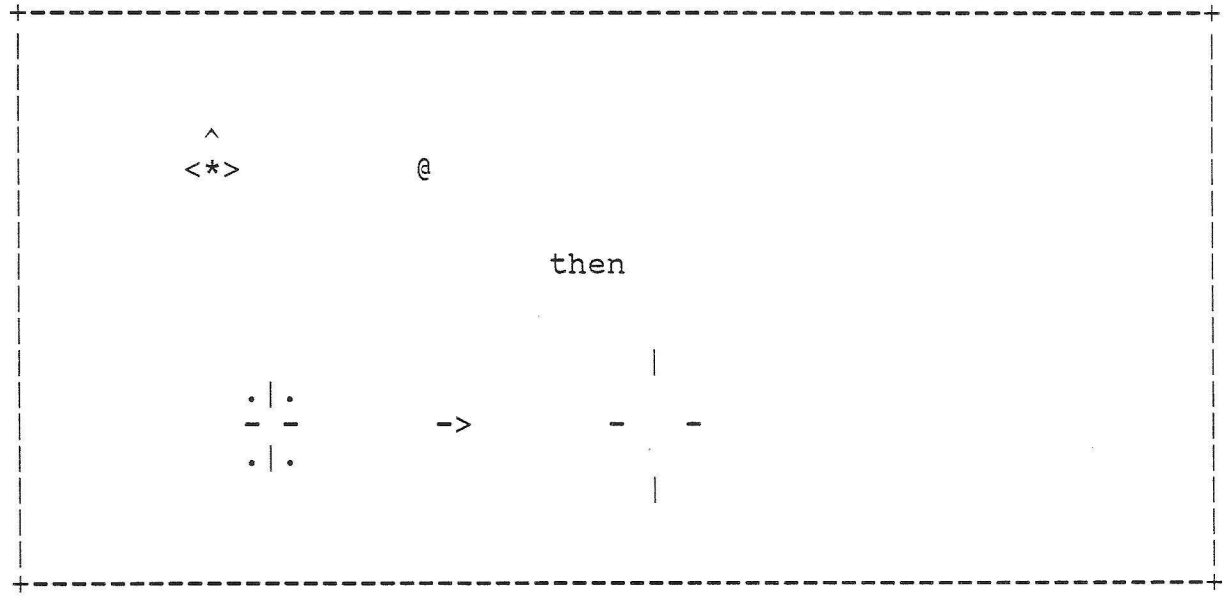
```
(DE RunTheScreen ()
% Run the screen until no more objects are visible.
  (LINELENGTH 0)
  (CLEAR)
  (WHILE (OR Aliens Bullets Explosions)
         (DO (FOR (IN o (APPEND Bullets Aliens))
                  (DO (SEND o 'MoveYourself)
                      (SEND o 'ChangeYourself)))
             (FOR (IN o Explosions)
                  (DO (SEND o 'NextState))))))
```

To run the simulation we create an instance of an alien and a bullet headed at each other. We put the names of these instances on the appropriate lists so the top level loop changes their states at the appropriate times.

```
% Put two objects on the screen and let them go at it.
(SETQ Aliens
  (LIST (INSTANCE Aliens A1
            (Xloc . 300)
            (Yloc . 120)
            (Xvel . 5)
            (Yvel . 0))))
(SETQ Bullets
  (LIST (INSTANCE Bullets B1
            (Xloc . 400)
            (Yloc . 120)
            (Xvel . -6)
            (Yvel . 0))))

(RunTheScreen)
```

The initial screen configuration should resemble the following. The two objects "rush" at each other and disappear in an explosion, only, if you haven't compiled the program, they won't be in any great hurry.



Finally, we demonstrate a more complex example, one with four initial objects aimed at each other but with anhilation scheduled at different times.

```
% Now Put 4 objects on the screen.
(SETQ Aliens (LIST
  (INSTANCE Aliens A1
     (Xloc . 100) (Yloc . 100) (Xvel . 5) (Yvel . 0))
  (INSTANCE Aliens A2
     (Xloc . 200) (Yloc . 200) (Xvel . 6) (Yvel . -6))))
(SETQ Bullets (LIST
  (INSTANCE Bullets B1
     (Xloc . 205) (Yloc . 100) (Xvel . -5) (Yvel . 0))
  (INSTANCE Bullets B2
     (Xloc . 250) (Yloc . 150) (Xvel . -5) (Yvel . 5))))
(RunTheScreen)
```

Institute of Artificial Intelligence

The Institute of Artificial Intelligence is sponsoring a summer training program for workers in the field of Artificial Intelligence. Their brochure states:

> "The Institute of Artificial Intelligence is a permanent, fully independent repository of AI experience, learning and research. Its primary goal is in-depth education and training of functional AI practitioners, such as knowledge engineers, project managers, and AI system programmers. The Institute is affiliated with Harvey Mudd College, the prestigious engineering and science school of the Claremont Colleges."

The Institute can be reached at (213)-201-0106 or

The Institute of Artificial Intelligence
1888 Century Park East, Suite 1207
Los Angeles, California
90067-1716

Classes commence June 24, 1985.

UO-LISP Version Number Changes

Northwest Computer Algorithms has renumbered the versions of UO-LISP to simplify ordering and communication with our users. UO-LISP Version 1 is for the older TRS-80 systems, Version 2 for the Z80 CP/M systems, and Version 3 for the 8086 family. Each version is futher specified by a release number, and, for the Version 3 group, still further by a modification number. The following table gives the current versions and numbers:

Version Name	Current Release	CPU Family	Operating System(s)
V1	V1.5B	Z80	TRSDOS
V2	V2.16	Z80	CP/M 2.2
V3	V3.0.03	8086	PC-DOS, MS-DOS

The old version CP/M system V1.16 is now called V2.16. NOTE: The (BLS.1) style configuration is no longer sold. However, purchasers of this configuration can buy the new Version 2 at less than the total price of the add-ons of the previous system.

New Product Configurations

To streamline mail-order distribution, we have simplified UO-LISP packaging and ordering. The following package pricing applies as of July 1, 1985:

UO-LISP V1	$80.00
Little Meta V1	$40.00
UO-LISP V2	$125.00
Learn Lisp V2	$85.00
Little Meta V2	$80.00
UO-LISP V2 with Lisp Tutorial Support	$160.00
UO-LISP V3	$150.00
Learn Lisp V3	$85.00
Little Meta V3	$80.00
UO-LISP V3 with Lisp Tutorial Support	$185.00
Reference Manual V3	$30.00
Reference Manual V2	$30.00
Reference Manual V1	$20.00
Tutorial Guide	$15.00
Newsletter 1 Year	$12.00
Back issues	$3.00

UO-LISP NEWSLETTER

July 1985 Vol. 2 No. 3

Hard on the heels of the last late newsletter is this Summer issue. We switch from programming with objects to customizing the LSED screen editor by including the complete source code for an EMACS style interface.

Feature: EMACS Style Interface for LSED

LSED is a completely user programmable editor. The system provides the basic interface functions and the user configures the system to perform in whatever manner desired. This can take the form of rebinding the control character sequences to perform different functions thus permitting the special keys on some systems to operate correctly. Alternatively, the system can be configured to look like some other editor. This is particularly useful if you are used to something like Wordstar or Emacs.

The user's guide provides some simple examples of rebinding key sequences for different terminal types. In this exposition we examine a complete rework of the editor to make it look like the popular Emacs editor.

There are 4 undocumented functions that need to be examined for this effort. We include them as a bonus to newsletter subscribers.

(eFARM CHAR:integer KEYMAP:alist REPT:integer):tri-boolean

Type: EXPR.

eFARM calls the function associated with character CHAR based on the current KEYMAP. KEYMAP is the association list formed by the BIND function. REPT is the number of times that the command should be repeated. If the command succeeds, the value of the function OK is returned, if it fails, the value of the function FAIL is returned and if the command is to terminate LSED execution, the value of TERM is returned (these are checked for by the OKP, FAILP and TERMP functions). If CHAR is the first of a multiple character command, more characters will be read by eFARM to complete the command.

(eGET):integer

Type: EXPR.

eGET returns the next "raw" character from whatever input is currently selected. This includes keyboard input, input from a keyboard macro, and input from the file copy on the screen. eGET should be used for any non-echoing input of characters.

(eGETV N:integer):{list,tri-boolean}

Type: EXPR.

eGETV returns a list of characters for line N of the file being edited. It does all appropriate swapping to and from disk files and so on. It returns FAIL if there is no such line in the file.

(eREAD MSG:string):list

Type: EXPR.

eREAD causes MSG to appear in the middle window and to return a list of characters that the user types following this prompt. eREAD should be called for any user input of text information.

Emacs is more than a set of keybindings and extensions. There are many additional functions not bound to any keys that are supported. The following is about 1/3 of the total available functions in the Unix EMACS system. The following text contains very brief descriptions of functions and bindings that follow in boxes.

To assure that the file we are creating is compilable without loading LSED and the screen driver, we declare global variables we need from the basic system.

```
% Some globals from LSED and the terminal drivers.
(GLOBAL '(
    !#C           % The current key map.
    TERM!-MAXX    % The last screen column.
    TERM!-MAXY    % The top most row (starting at 0).
    !#FY          % The current file line number.
    !#FX          % The rest of the current line.
    !#SX          % Current screen column number.
    !#SY          % Current screen row number.
   ))
```

The first function allows us to have multiple keymaps: the mapping between control key sequences and functions that do things. This function would enable us to switch between the Emacs mode and the regular LSED mode, invaluable during debugging the package. We will call the standard key map UO!-LISP and the EMACS set EMACS. We switch to the (currently empty) EMACS key map before we start rebinding keys.

```
% Keep around the current key map name.
(GLOBAL '(!#KEYMAPNAME))
(SETQ !#KEYMAPNAME 'UO!-LISP)

(DE USEKEYMAP (name)
% Switch to keymap name. The current command list is saved
% under its name (in !#KEYMAPNAME) and the new keymap is
% selected.
  (PUT !#KEYMAPNAME 'LSEDKEYMAP !#C)
  (SETQ !#C (GET name 'LSEDKEYMAP))
  (SETQ !#KEYMAPNAME name))

(USEKEYMAP 'EMACS)        % Switch to emacs keymap.
```

We now work through the control keys, binding them to their appropriate functions and defining new ones when there are no equivalents in the base system. Recall that BIND has two arguments, the first a list of the ASCII codes of the control characters that trigger the evaluation of the second argument, the name of a function. The first set, ^A through ^I (tab), are bound to built-in functions. LSED doesn't work well with tabs, so we use the tab key to evaluate expressions in the top window, something that basic Emacs has no command for.

```
(BIND '(1) 'BOL)          % ^A beginning of line
(BIND '(2) 'BACKC)        % ^B backwards character
(BIND '(3) 'QUITNS)       % ^C quit without saving
(BIND '(4) 'DELFC)        % ^D delete forward character
(BIND '(5) 'EOL)          % ^E end of line
(BIND '(6) 'FORC)         % ^F forward character
(BIND '(8) 'DELBC)        % ^H backwards character
(BIND '(9) 'EEXP)         % tab eval expression
```

The function bound to ^J (line feed) has no counterpart in LSED. The line feed key operates like the return key except that it automatically indents the next line the same number of blanks that the previous line has. We accomplish this by retrieving the current line of characters (using eGETV and the current line number !#FY). We insert a carriage return and as many blanks as we find in the previous line. The LBLS function counts the number of blanks at the beginning of the line.

```
(BIND '(10) 'NLIND)       % <lf> new line and indent.
(DE NLIND ()
% Terminate the current line, indent the next to where
% ever this one starts.
  (PROG (cl)
     (SETQ cl (eGETV !#FY))
     (CINS 13)
     (FOR (FROM i 1 (LBLS cl)) (DO (CINS 32)))
     (RETURN (OK))))

(DE LBLS (l)
% Returns the number of blanks at the beginning of line l.
  (IF (NULL l) THEN 0
   ELSEIF (EQCAR l 32) THEN (ADD1 (LBLS (CDR l)))
   ELSE 0))
```

LSED normally irretrievably expunges deleted text from the file. The kill to end of line command (connected to ^K) places deleted text in a special buffer for retrieval at a later date. The command takes the part of the current text line following the cursor, erases it from the screen, and appends it to the kill buffer variable !#KBUFFER. The function also intercepts subsequent ^K's and causes their text to be added to the buffer. The COPYONE function copies the first element of the tail of the list as we destructively replace the CAR of this element with a

carraige return (13) character to terminate the line. This is an important fact to remember when constructing LSED functions: all lines must end with a carriage return.

```
(BIND '(11) 'KEOL)          % ^K kill to end of line
(GLOBAL '(!#KBUFFER))
(DE KEOL ()
% Delete the rest of the current line (do this by RPLACing).
% Append all this stuff to the !#KBUFFER buffer and keep
% doing so as long as ^K is pressed. Note that if we are
% already at the end of the line, don't add to kill buffer,
% but just do a DELFC.
  (PROG (c)
     (SETQ !#KBUFFER NIL)
11   (IF (EQUAL !#FX '(13)) THEN (DELFC)
      ELSE (SETQ !#KBUFFER (NCONC !#KBUFFER
                                  (LIST (COPYONE !#FX))))
           (CLEAR!-EOL)
           (RPLACA !#FX 13)
           (RPLACD !#FX NIL))
     (SETQ c (eGET))
     (IF (EQUAL c 11) THEN (GO 11)
                      ELSE (RETURN (eFARM c !#C NIL)))))

(DE COPYONE (l)
% Creates a new list l with the first element replaced.
  (CONS (CAR l) (CDR l)))
```

The ENTER (or sometimes RETURN) key is bound to SELF. SELF causes the character to be entered as is into the file, in this case, causing a new line. The OPENL function (^O) is similar in flavor except that it backs up before the inserted character efectively opening a blank line (or with the remainder of the current one) following the one the cursor is on.

```
(BIND '(12) 'REDRAW)        % ^L redraw screen
(BIND '(13) 'SELF)          % ^M new line
(BIND '(14) 'DOWNL)         % ^N next line
(BIND '(15) 'OPENL)         % ^O open new line
(DE OPENL ()
% Insert a new line at the current position but leave the
% cursor where it is by backing up one character.
  (CINS 13)
  (BACKC))
```

The ^Q key causes one more character to be read and inserted as is into the input file. This is most useful for entering control characters (such as ^A into LISPTEX document files).

```
(BIND '(16) 'UPL)          % ^P previous line
(BIND '(17) 'QC)           % ^Q quote next character
(DE QC ()
% Insert next character as is into the file.
  (CINS (eGET)))
```

The transpose character function TCHAR switches the next two characters in the file if there are any and one of them is not a carriage return. The cursor is left in its original position. Note that CINS moves the cursor ahead

```
(BIND '(19) 'FIND)         % ^S find forward
(BIND '(20) 'TCHAR)        % ^T transpose next 2 characters
(DE TCHAR ()
% Read the next two characters, but quit if they
% are anything funny.
  (PROG (c1 c2)
     (SETQ c1 (CURC))
     (IF (FAILP (FORC)) THEN (BACKC) (RETURN (FAIL)))
     (SETQ c2 (CURC))
     (IF (OR (EQN c1 13) (EQN c2 13)) THEN
        (BACKC) (RETURN (FAIL)))
     (BACKC) (DELFC) (DELFC)
     (CINS c2) (CINS c1) (BACKC)
     (RETURN (BACKC))))
```

When we implemented LSED we decided that the 80x24 character screens common on most small computers were not sufficient to support multiple window editing. Instead we adopted the strategy of editing files simultaneously. The following commands implement this facility and switching between the files. Note that some of these commands require regular characters as modifiers for ^X (for example ^X E). In this case we provide a binding for both upper and lower case versions of the character.

```
(BIND '(21) 'REPT)          % ^U repeat next command
(BIND '(22) 'FORP)          % ^V forward page

(BIND '(24 3) 'QUITNS)      % ^X ^C exit emacs
(BIND '(24 5) 'ECOM)        % ^X ^E middle window eval
(BIND '(24 6) 'SAVEX)       % ^X ^F save and exit
(BIND '(24 9) 'INSF)        % ^X ^I insert file
(BIND '(24 17) 'SAVEF)      % ^X ^Q save but no exit
(BIND '(24 22) 'VISITF)     % ^X ^V visit file
(BIND '(24 26) 'SW)         % ^X ^Z shrink window
(BIND '(24 40) 'DKM)        % ^X (  define keyboard macro
(BIND '(24 41) 'EKM)        % ^X )  end of keyboard macro
(BIND '(24 69) 'KM)         % ^X E  execute keyboard macro
(BIND '(24 101) 'KM)
(BIND '(24 78) 'OTHERF)     % ^X N  other file
(BIND '(24 110) 'OTHERF)
(BIND '(24 80) 'OTHERF)     % ^X P  other file
(BIND '(24 112) 'OTHERF)
```

The yank from kill buffer command copies the kill buffer into the file at the current position. This feature is normally used as in cut and paste editing or to move blocks of text between two files. Since the kill buffer is not destroyed by the command it can be inserted in more than one place or carried between files. The FINALLY clause removes the final carriage return ending the last line of the kill buffer.

```
(BIND '(25) 'YFKB)          % ^Y yank from kill buffer.
(DE YFKB ()
% Yank the stuff from the kill buffer and place in the file.
% This kludgy version updates the screen. This probably
% shouldn't happen unless the kill buffer is small. The kill
% buffer is not cleared.
  (FOR (IN l !#KBUFFER)
       (DO (FOR (IN c l) (DO (CINS c))))
       (FINALLY (DELBC))
       (RETURNS (OK))))
```

The LTTOP function (bound to escape !) rolls the line the cursor is on (!#SY on the screen) to the top of the screen. It does this by rolling up the screen a line at a time until either the end of file is encountered (ROLLUP returns FAIL) or the line is at the top of the screen. Note that ROLLUP causes !#SY to be

incremented for every successful roll.

```
(BIND '(26) 'ROLLUP)        % ^Z roll screen up
(BIND '(27 3) 'QUITNS)      % esc ^C quit no save
(BIND '(27 27) 'ECOM)       % esc esc execute expression
(BIND '(27 33) 'LTTOP)      % esc ! line to top of window
(DE LTTOP ()
% Move the current line to the top of the window by
% scrolling.
  (PROG ()
loop (IF (EQN !#SY TERM!-MAXY) THEN (RETURN (OK)))
     (IF (FAILP (ROLLUP)) THEN (RETURN (FAIL)))
     (GO loop)))
```

We now branch into some word processing functions, those dealing with words, and sentences. For example, the esc ^ command causes the cursor to skip over all non-printing characters and then invert the case of all alphabetic characters until some non-alphabetic character is encountered. The function CINV inverts the case of a single character and returns NIL when a non-alphabetic is encountered. CWINV scans over control characters and blanks and then the word inverting case. Notice that one must always check for FAILP of a cursor movement operation. Not doing this will generally cause an infinite loop when the cursor reaches end of file.

```
(BIND '(27 44) 'BOP)      % esc , beginning of window
(BIND '(27 46) 'EOP)      % esc . end of page
(BIND '(27 60) 'BOF)      % esc < beginning of file
(BIND '(27 62) 'EOF)      % esc > end of file
(BIND '(27 94) 'CWINV)    % esc ^ case word invert
(DE CINV (c)
% Invert the case of character c. Returns NIL if c is not
% an alphabetic character.
  (IF (AND (GEQ c 65) (LEQ c 90)) THEN (PLUS c 32)
   ELSEIF (AND (GEQ c 97) (LEQ c 122)) THEN (DIFFERENCE c 32)
   ELSE NIL))

(DE CWINV ()
% Invert the case of a word.
  (PROG (c)
l1    (IF (GREATERP (CURC) 32) THEN (GO l2))
      (IF (FAILP (FORC)) THEN (RETURN (FAIL))
       ELSE (GO l1))
l2    (IF (SETQ c (CINV (CURC))) THEN
         (DELFC) (CINS c)
         (IF (FAILP (FORC)) THEN (RETURN (FAIL))
                            ELSE (BACKC) (GO l2)))
      (RETURN (OK))))
```

Moving backwards a word is a bit more difficult as the CURC function alwyas returns the next character in the input line. As before we bind both esc B and esc b to the command. This command checks for failure of BACKC so that the result of backing up to the beginning of file des not result in an infinite loop.

```
(BIND '(27 66) 'BW)       % esc B backwards word
(BIND '(27 98) 'BW)
(DE BW ()
% Move backwards a word.
  (PROG ()
l1    (IF (MEMBER (CURC) '(32 13)) THEN
        (IF (OKP (BACKC)) THEN (GO l1)
                          ELSE (RETURN (FAIL))))
l2    (IF (MEMBER (CURC) '(32 13)) THEN (RETURN (OK))
       ELSEIF (OKP (BACKC)) THEN (GO l2)
                            ELSE (RETURN (FAIL)))))
```

Delete forward word is a function in the same fashion.

Notice that many of the key bindings for commands operating on words are bound to key sequences that are difficult to remember. If you are in the habit of moving the cursor through words and sentences you might consider binding these sequences to single character codes or to the special function keys.

```
(BIND '(27 68) 'DFW)      % esc D delete next word
(BIND '(27 100) 'DFW)
(DE DFW ()
% Delete all the blanks up to and including the next word.
  (PROG ()
l1    (IF (MEMBER (CURC) '(32 13)) THEN
       (IF (OKP (DELFC)) THEN (GO l1)
                         ELSE (RETURN (FAIL))))
l2    (IF (MEMBER (CURC) '(32 13)) THEN (RETURN (OK))
       ELSEIF (OKP (DELFC)) THEN (GO l2)
                            ELSE (RETURN (FAIL)))))
```

Forward sentence and forward word are more of the same.

```
(BIND '(27 69) 'FS)       % esc E forward sentence
(BIND '(27 101) 'FS)
(DE FS ()
% Move forward a sentence. A sentence is terminated with a
% . ! or ? and followed by a blank, EOF, or end of line.
  (PROG ()
s1    (IF (MEMBER (CURC) '(46 33 63)) THEN (GO s2)
       ELSEIF (OKP (FORC)) THEN (GO s1)
       ELSE (RETURN (FAIL)))
s2    (IF (FAILP (FORC)) THEN (RETURN (OK))
       ELSEIF (MEMBER (CURC) '(32 13)) THEN (RETURN (OK))
       ELSE (GO s1))))

(BIND '(27 70) 'FW)       % esc F forward word
(BIND '(27 102) 'FW)
(DE FW ()
% Move forward over blank characters and A-Z characters to
% the end of this word.
  (PROG ()
l1      (IF (MEMBER (CURC) '(32 13)) THEN
          (IF (OKP (FORC)) THEN (GO l1)
                           ELSE (RETURN (FAIL))))
l2      (IF (NOT (MEMBER (CURC) '(32 13))) THEN
          (IF (OKP (FORC)) THEN (GO l2)
                           ELSE (RETURN (FAIL))))
       (RETURN (OK))))
```

Delete backward word is a bit different than just backing up as we don't want to delete extra characters. Changing the case of a word to lower is, of course, the analog of changing it to upper case.

```
(BIND '(27 72) 'DBW)      % esc H delete backward word
(BIND '(27 104) 'DBW)
(DE DBW ()
% Delete the blanks before and the previous word.
  (PROG ()
11    (IF (MEMBER (CURC) '(32 13)) THEN
          (DELFC)
          (IF (OKP (BACKC)) THEN (GO 11)
                            ELSE (RETURN (FAIL))))
12    (IF (MEMBER (CURC) '(32 13)) THEN (RETURN (OK)))
      (DELFC)
      (IF (OKP (BACKC)) THEN (GO 12)
                        ELSE (RETURN (FAIL)))))

(BIND '(27 76) 'CWL)      % esc L case word lower
(BIND '(27 108) 'CWL)
(DE CWL ()
% Lower the case of the next word. Skip over any blanks in
% the way.
  (PROG (c)
11    (IF (MEMBER (CURC) '(32 13)) THEN
       (IF (OKP (FORC)) THEN (GO 11) ELSE (RETURN (FAIL))))
12    (SETQ c (CURC))
      (IF (AND (GEQ c 65) (LEQ c 90)) THEN
          (DELFC) (CINS (PLUS c 32))
       ELSEIF (MEMBER c '(32 13)) THEN (RETURN (OK))
       ELSEIF (OKP (FORC)) THEN (GO 12)
       ELSE (RETURN (OK)))))
```

The replace command requires collecting two strings and then performing replacements throughout the rest of the file. We first check the strings for the presence of ^G. We use this character to signal the command to be aborted. Once both strings have been read, the FND function locates the first occurrence of the string. If none are found the routine returns success or failure based on the number of replacements performed. If an occurrence of the !#ROLD string is found, its characters are deleted and replaced with those of the new string and the search and replacement continues.

```
(BIND '(27 82) 'RPL)      % esc R replace
(BIND '(27 114) 'RPL)
(GLOBAL '(!#ROLD !#RNEW))
(DE RPL ()
% Search for the string !#ROLD and replace it them with
% !#RNEW. Display the count of replacements in the
% message window.
  (PROG (rplcnt)
     (SETQ rplcnt 0)
     (SETQ !#ROLD (eREAD "Search for:"))
     (IF (MEMBER 7 !#ROLD) THEN (MSG '("aborted"))
                                (RETURN (FAIL)))
     (SETQ !#RNEW (eREAD "Replace with:"))
     (IF (MEMBER 7 !#RNEW) THEN (MSG '("aborted"))
                                (RETURN (FAIL)))
loop (IF (FAILP (FND !#ROLD)) THEN
       (IF (ZEROP rplcnt) THEN (MSG '("no replacements"))
                               (RETURN (FAIL))
        ELSE (MSG (LIST rplcnt "occurrences replaced"))
             (RETURN (OK))))
     (FOR (FROM i 1 (LENGTH !#RNEW)) (DO (DELBC)))
     (FOR (IN c !#ROLD) (DO (CINS c)))
     (SETQ rplcnt (ADD1 rplcnt))
     (GO loop)))
```

The final commands and functions are mere repetitions of earlier ones.

```
(BIND '(27 85) 'CWU)      % esc U case word upper.
(BIND '(27 117) 'CWU)
(DE CWU ()
% Convert the next word to all upper case.
  (PROG (c)
l1    (IF (MEMBER (CURC) '(32 13)) THEN
        (IF (OKP (FORC)) THEN (GO l1) ELSE (RETURN (FAIL))))
l2    (SETQ c (CURC))
      (IF (AND (GEQ c 96) (LEQ c 122)) THEN
         (DELFC) (CINS (DIFFERENCE c 32)) (GO l2)
       ELSEIF (MEMBER c '(32 13)) THEN (RETURN (OK))
       ELSEIF (FAILP (FORC)) THEN (RETURN (FAIL))
       ELSE (GO l2))))

(BIND '(27 86) 'BACKP)    % esc V backwards page
(BIND '(27 118) 'BACKP)
(BIND '(27 88) 'ECOM)     % esc X execute lisp in mini
(BIND '(27 120) 'ECOM)
(BIND '(27 90) 'ROLLDN)   % esc Z roll down
(BIND '(27 122) 'ROLLDN)
(BIND '(27 127) 'DELBC)   % del delete backward character
```

To compile this batch of functions you must first load the MACROS package. To run the system, you must first load LSED and then the emacs file.

In the next Issue:

The next issue of the UO-LISP newsletter will present details of the upcoming release of version 2.17 for Z80 CP/M systems, and version 3.2 for IBM PC users. The feature article will be a rule based expert system for generating musical scores.

UO-LISP NEWSLETTER

October 1985 Vol. 2 No. 4

Version 3.2 Learn Lisp Announced

Version 3.2 of the UO-LISP Learn Lisp system is now available from Northwest Computer Algorithms. This newest version contains many long awaited features:

1. **Generic Arithmetic.** Bignumber arithmetic is now automatic when the small integer magnitude is exceeded. Small number arithmetic is accomplished with the usual functions, but each is prefixed with an I: IADD1, IPLUS2, etc.. Bignumber-only arithmetic is accomplished by the same functions prefixed with a B: BADD1, BPLUS2, etc.. On the other hand, the PLUS2 function both verifies numeric operands are present and decides whether IPLUS2, BPLUS2, or FPLUS2 (for floating point) is to be called.

2. **Floating Point.** 7 digits of accuracy, arbitrary exponents. Floating point numbers are entered in the usual format. Output is in standard notation for reasonable sized numbers and scientific notation when a fixed exponent range is exceeded. The generic arithmetic package converts fixed operands to floating point whenever at least on argument is floating.

3. **Space Sizes** Dotted-pair space has been expanded to 16384 possible pairs. String space has expanded to accomodate about 2000 strings with an aggregate length of 32k bytes.

4. **New Functions.** Several new functions have been added: TRINTS: turn on saving the name of interpreted functions on the alist for traceback. UNTRINTS Turn off the interpeted function tracing. CLOSEALL Close all open files. MAPOBL Apply a function to every identifier in the symbol table.

Blackjack

In place of the promised music generation program, we offer this program submitted by Andrew Parker. The program is divided into two parts, the first is a general program for shuffling a deck

of cards. The second plays the game. We present this pretty much as we received it.

```
%*****************************************************
%
%  This program shuffles a deck of cards. The shuffled
%  deck is returned as a list. The deck of cards is
%  maintained in the global variable "deck". The deck
%  is a standard 52 card deck with no JOKERs. The user
%  can modify "deck" by adding the additional elements
%  required. For example to add two JOKERs you would do
%  the following:
%
%       (SETQ deck (APPEND deck '(JOKER JOKER)))
%
%  Two decks can be shuffled together by APPENDING the
%  global variable "deck" to itself.


% create  the deck of cards
(GLOBAL '(deck))

(SETQ deck '(!2H !3H !4H !5H !6H !7H !8H !9H !1!0H JH
             QH KH AH !2D !3D !4D !5D !6D !7D !8D !9D
             !1!0D JD QD KD AD !2C !3C !4C !5C !6C !7C
             !8C !9C !1!0C JC QC KC AC !2S !3S !4S !5S
             !6S !7S !8S !9S !1!0S JS QS KS AS))

%*****************************************************
%  remove!-nth - removes the nth element of a list.
%
%  ARGUMENTS:  l -  a list (the deck of cards)
%              element!-number - the position of the
%                   nth element
%
%  RETURNS:    the list l with the nth element removed
%
%  SIDE EFFECTS: none

(DE remove!-nth (l element!-number)
   (COND ((NULL l) NIL)
         ((EQ element!-number 1) (CDR l))
         (T (CONS (CAR l)
                  (remove!-nth (CDR l)
                               (SUB1 element!-number))))))

%*****************************************************
% return!-nth: returns the nth element of the list l as
% an atom.
%
% ARGUMENTS:  l - a list (the deck of cards)
%             element!-number: the position of the
%                 nth element of the list
%
% RETURNS:    the nth element of the list l as an atom
```

```
%
% SIDE EFFECTS: none

(DE return!-nth (l element!-number)
   (COND ((NULL l) NIL)
         ((EQ element!-number 1) (CAR l))
         (T (return!-nth (CDR l)
                         (SUB1 element!-number)))))

%*****************************************************
%  shuffle
%
% ARGUMENTS: l - a list (the deck of cards)
%
% RETURNS:   the list l randomly rearranged.
%
% SIDE EFFECTS: none

(DE shuffle (l)
  (PROG (card!-position length!-l)
    (COND ((NULL l) (RETURN NIL))
          ((EQN 1 (SETQ length!-l (LENGTH l)))
           (SETQ card!-position 1))
          (T (SETQ card!-position
              (ADD1 (QUOTIENT (RANDOM)
                              (QUOTIENT 8191
                                        length!-l))))))
    (RETURN (CONS (return!-nth l card!-position)
                  (shuffle
                   (remove!-nth l card!-position))))))

%*****************************************************
%  get!-deck
%
%  ARGUMENTS:  none
%
%  RETURNS:    a list that represents a shuffled deck
%              of cards
%
%  SIDE EFFECTS: reassigns the global variable "deck"
%              to the order

(DE get!-deck ()
  (SETQ deck (shuffle deck)))

STOP
```

The second program section plays the game. There are a few machine dependent parameters at the beginning that must be fiddled for non-IBM compatible machines (this worked on the Tandy Model 2000 without change).

```
(COND ((NOT (GETD 'CURSOR)) (FLOAD "TERMINAL")))


(GLOBAL '(
 my!-deck            % deck of cards to be delt
 dimond              % diamonds
 club                % clubs
 heart               % hearts
 spade               % spades
 joker               % joker
 upper!-left         % upper left corner of card
 upper!-right        % upper right corner of card
 lower!-left         % lower left corner of card
 lower!-right        % lower right corner of card
 top!-edge           % top edge of card
 side!-edge          % side edge of card
 card!-back          % back of card
 your!-current!-bet % amount you are betting
 your!-hand          % A-list representing your hand
 your!-total         % amount of money you have left
 computer!-hand      % A-list of the computers hand
 double!-hand))      % A-list of your double hand

%
% The following values must be set for the type of
% computer you are using. The default is for the IBM PC
% and the COMPAQ deskpro.

(SETQ dimond 4)
(SETQ club   5)
(SETQ heart  3)
(SETQ spade  6)
(SETQ joker  2)
(SETQ upper!-left  218)
(SETQ upper!-right 191)
(SETQ lower!-left  192)
(SETQ lower!-right 217)
(SETQ top!-edge    196)
(SETQ side!-edge   179)
(SETQ card!-back   178)



%
% your!-hand and the computer!-hand are A-list
% representing the cards that have been dealt. The
% A-list looks as follows:
%
%  ( ((x!-position . y!-position) (card . suit))
%    ((x!-position . y!-position) (card . suit)).....)
%
% There is one entry for each card a player has been
% dealt. At the beginning of each hand the lists are
```

```
% set to NIL and the totals to zero.
%


(SETQ deck '(
 1  2  3  4  5  6  7  8  9 10 11 12 13     % hearts
 14 15 16 17 18 19 20 21 22 23 24 25 26    % diamonds
 27 28 29 30 31 32 33 34 35 36 37 38 39    % clubs
 40 41 42 43 44 45 46 47 48 49 50 51 52))  % spades

%*****************************************************
% get!-normal!-card -
%
% ARGUMENTS - card!-number: numeric value representing
%                           one playing card.  If the
%                           number is greater then 52
%                           it is assumed to be a joker
%
% RETURNS - a dotted pair in which the CAR is the
%           card value 2-10, J-A and the CDR is the
%           suit 3-hearts, 4-dimonds, 5-clubs,
%           and 6-spades.

(DE get!-normal!-card (card!-number)
  (PROG (suit card)
     (COND ((LESSP card!-number 14)
                (SETQ suit heart)
                (SETQ card card!-number))
           ((LESSP card!-number 27)
                (SETQ suit dimond)
                (SETQ card (DIFFERENCE card!-number 13)))
           ((LESSP card!-number 40)
                (SETQ suit club)
                (SETQ card (DIFFERENCE card!-number 26)))
           ((LESSP card!-number 53)
                (SETQ suit spade)
                (SETQ card (DIFFERENCE card!-number 39)))
           (T   (SETQ suit joker)
                (SETQ card "J") ))
     (COND ((EQ card 11) (SETQ card "J"))
           ((EQ card 12) (SETQ card "Q"))
           ((EQ card 13) (SETQ card "K"))
           ((EQ card 1)  (SETQ card "A")))
     (RETURN (CONS card suit)) ))

%*****************************************************
%
%                     OUTPUT SECTION
%
%*****************************************************
% print!-card!-id - displays the card value 2-10, J-A
% and the graphic representation of the suit.

(DE  print!-card!-id (card!-frame)
    (TERPRI)
```

```
    (CURSOR (PLUS (CAAR card!-frame) 4)
            (DIFFERENCE (CDAR card!-frame) 2))
    (PRIN2 (CADR card!-frame))
    (!$PA (CDDR card!-frame)) )

%*******************************************************
% print!-card!-back - if the card is delt face down,
% fill in the back

(DE print!-card!-back (card!-frame)
  (PROG (x y)
    (TERPRI)
    (SETQ x (ADD1 (CAAR card!-frame)))
    (SETQ y (CDAR card!-frame))
    (FOR (FROM I 1 6)
      (DO
        (CURSOR x (DIFFERENCE y I))
        (FOR
          (FROM counter 1 8)
           (DO (!$PA card!-back) )) )) ))

%*******************************************************
%  print!-card - displays one card on the screen. The
% x,y position is the upper left hand corner of the card.
%
% ARGUMENTS: card!-frame - an A-list representing the
%                          card and its position
%                      ((x!-position . y!-position)
%                       (card!-value . suit))
(DE print!-card (card!-frame)
  (PROG (x y card)
    (SETQ x (CAAR card!-frame))
    (SETQ y (CDAR card!-frame))
    (SETQ card (CDR card!-frame))
    (TERPRI)
    (CURSOR x y)
    (!$PA upper!-left)
    (FOR (FROM I 1 8) (DO (!$PA top!-edge)))
    (!$PA upper!-right)
    (FOR (FROM I 1 6)
      (DO
        (CURSOR x (DIFFERENCE y I))
        (!$PA side!-edge)
        (PRIN2 "        ")
        (!$PA side!-edge) ))
    (CURSOR x (DIFFERENCE y 7))
    (!$PA lower!-left)
    (FOR (FROM I 1 8) (DO (!$PA top!-edge)))
    (!$PA lower!-right) ))

%*******************************************************
% erase!-card - removes a card from the screen. It does
% not repaint any underlying cards.

(DE erase!-card (card!-frame)
```

```
  (PROG (x y)
    (SETQ x (CAAR card!-frame))
    (SETQ y (CDAR card!-frame))
    (FOR (FROM i 0 7)
      (DO
        (TERPRI)
        (CURSOR x (DIFFERENCE y i))
        (PRIN2 "          ") )) ))

%*******************************************************
% redraw!-hand - re-paints a hand on the screen.

(DE redraw!-hand (hand)
   (COND ((NOT (NULL hand))
          (print!-card (CAR hand))
          (print!-card!-id (CAR hand))
          (redraw!-hand (CDR hand)) )) )

%*******************************************************
% get!-card - get the first card off the deck
% The card returned is an A-list (card value .  suit)

(DE get!-card ()
 (PROG (card)
   (COND ((NULL my!-deck)
          (CURSOR 1  2)
          (CLEAR!-EOP)
          (PRIN2T "I'm shuffling a new deck, please wait")
          (SETQ my!-deck (get!-deck)) ))
     (SETQ card (get!-normal!-card (CAR my!-deck)))
     (SETQ my!-deck (CDR my!-deck))
     (RETURN card) ))

%*******************************************************
% card!-position
% ARGUMENTS hand -  A-list  representing one hand
% RETURNS: a dotted pair contianing the x!-position and
%          y!-position to display the next card

(DE card!-position (hand)
  (COND ((LESSP (DIFFERENCE (CDAAR hand) 4) 14)
          (CONS (PLUS2 (CAAAR hand) 8) 22))
        (T (CONS (CAAAR hand)
                 (DIFFERENCE (CDAAR hand) 4)))))

%*******************************************************
%
%                        BLACK JACK
%
%*******************************************************
% deal - deals out the first two cards for black jack.

(DE deal ()
  (PROG (card)
    (CLEAR)
```

```
    (TERPRI)
    (CURSOR 5 24)
    (PRIN2T "The House")
    (CURSOR 58  24)
    (PRIN2T "Your Hand")
    (SETQ computer!-hand
           (SETQ double!-hand
                  (SETQ your!-hand NIL)))
    (FOR (FROM i 1 2)
      (DO
       (SETQ your!-hand
          (APPEND
           (LIST
            (CONS (CONS 58
                        (DIFFERENCE 26 (TIMES i 4)))
                  (get!-card)))
           your!-hand))
       (print!-card (CAR your!-hand))
       (print!-card!-id (CAR your!-hand))
       (SETQ computer!-hand
          (APPEND
           (LIST
            (CONS (CONS 5
                        (DIFFERENCE 26 (TIMES i 4)))
                  (get!-card)))
           computer!-hand))
       (print!-card (CAR computer!-hand))
       (COND ((NEQ i 1)
              (print!-card!-id (CAR computer!-hand)))
             (T (print!-card!-back
                 (CAR computer!-hand)))))))


%*****************************************************
% total!-cards
% ARGUMENTS: hand - A-list containing all the cards in
%         a hand.
% RETURNS: a numeric value representing the total of
%         the hand.

(DE total!-cards (hand)
  (PROG (aces total card!-value)
    (SETQ total (SETQ aces 0))
    (WHILE (NOT (NULL hand))
     (DO
      (COND ((NUMBERP (CADAR hand))
             (SETQ card!-value (CADAR hand)))
            ((EQ (CADAR hand) "A")
             (SETQ card!-value 11)
             (SETQ aces (ADD1 aces)))
            (T (SETQ card!-value 10) ))
      (SETQ total (PLUS total card!-value))
      (SETQ hand (CDR hand)) ))
    (WHILE (AND (GREATERP total 21)
                (GREATERP aces 0))
```

```
      (DO
        (SETQ total (DIFFERENCE total 10))
        (SETQ aces (SUB1 aces)) ))
    (RETURN total) ))

%*****************************************************
% deal!-your!-cards
% ARGUMENTS: your!-hand - A-list representing a players
%     hand

(DE deal!-your!-cards ()
  (PROG (command ender)
    (SETQ ender (SETQ command 'H))
    (SETQ PROMPT!* "(H)it, (S)tay, (D)double, (Q)uit: ")
    (WHILE (AND (NOT (MEMQ command '(!S !s)))
                (LESSP (total!-cards your!-hand) 21)
                (NOT (MEMQ command '(!Q !q))) )
     (DO
      (TERPRI)
      (CURSOR 1 2)
      (CLEAR!-EOL)
      (SETQ command (READ))
      (COND ((MEMQ command '(!H !h))
             (SETQ your!-hand (deal!-card your!-hand)))
            ((AND (MEMQ command '(!D !d))
                  (EQ (LENGTH your!-hand) 2)
                  (EQ (CADAR your!-hand)
                      (CADADR your!-hand)))
                    (do!-double)
                    (SETQ command 'S) )) ))
    (CURSOR 1 2)
    (CLEAR!-EOL)
    (RETURN command) ))

%*****************************************************
% deal!-card
% ARGUMENTS  hand - A-list representing one hand
% RETURNS    an A-list with the additional card in it.

(DE deal!-card (hand)
 (PROG ()
    (SETQ hand (APPEND (LIST (CONS (card!-position hand)
                                   (get!-card)))
                       hand))
    (print!-card (CAR hand))
    (print!-card!-id (CAR hand))
    (RETURN hand) ))

%*****************************************************
% do!-double - control function for when you double.
% The computer hand will never double.
%
% ARUGMENTS - none we will always double your!-hand

(DE do!-double ()
```

```
  (PROG (command)
    (SETQ double!-hand (LIST (CAR your!-hand)))
    (SETQ your!-hand (CDR your!-hand))
    (erase!-card (CAR double!-hand))
    (print!-card (CAR your!-hand))
    (print!-card!-id (CAR your!-hand))
    (SETQ double!-hand
      (LIST (CONS (CONS
                   (DIFFERENCE (CAAAR double!-hand) 26)
                   (PLUS (CDAAR double!-hand) 4) )
                  (CDAR double!-hand) )))
    (CURSOR (CAAAR double!-hand) 24)
    (PRIN2 "Double Hand")
    (print!-card (CAR double!-hand))
    (print!-card!-id (CAR double!-hand))
    (SETQ your!-hand (deal!-card your!-hand))
    (SETQ double!-hand (deal!-card double!-hand))
    (SETQ PROMPT!* "YOUR HAND - (H)it, (S)tay: ")
    (WHILE (AND (NOT (MEMQ command '(!S !s)))
                (LESSP (total!-cards your!-hand) 21))
     (DO
      (TERPRI)
      (CURSOR 1 2)
      (CLEAR!-EOL)
      (SETQ command (READ))
      (COND ((MEMQ command '(!H !h))
             (SETQ your!-hand
                   deal!-card your!-hand)) )) ))
    (SETQ command 'H)
    (SETQ PROMPT!* "DOUBLE HAND - (H)it, (S)tay: ")
    (WHILE (AND (NOT (MEMQ command '(!S !s)))
                (LESSP (total!-cards double!-hand) 21))
     (DO
      (TERPRI)
      (CURSOR 1 2)
      (CLEAR!-EOL)
      (SETQ command (READ))
      (COND ((MEMQ command '(!H !h))
             (SETQ double!-hand
                   (deal!-card double!-hand)) )) )) ))

%*****************************************************
% who!-won - determines who won the game.  It displays
% the results on the screen and adjusts your!-total
% winnings.
%
% ARGUMENTS:  hand -       A-list representing the hand
%                          to compare against the
%                          computer.
%             hand!-type - discription for the display
%                          'double or 'first

(DE who!-won (hand hand!-type)
  (COND ((AND (LEQ (total!-cards hand) 21)
              (LEQ (total!-cards computer!-hand) 21))
```

```
                (COND ((LESSP
                         (total!-cards hand)
                         (ADD1
                          (total!-cards computer!-hand)))
                          (PRIN2 "Computer wins ")
                          (PRIN2 hand!-type)
                          (PRIN2 " hand")
                          (SETQ your!-total
                                (DIFFERENCE
                                  your!-total
                                  your!-current!-bet)))
                      (T (PRIN2 "You win ")
                         (PRIN2 hand!-type)
                         (PRIN2 " hand")
                         (SETQ your!-total
                               (PLUS
                                your!-total
                                your!-current!-bet)))))
        ((LEQ (total!-cards hand) 21)
           (PRIN2 "You win ")
           (PRIN2 hand!-type)
           (PRIN2 " hand")
           (SETQ your!-total
                 (PLUS your!-total your!-current!-bet)))
        ((LEQ (total!-cards computer!-hand) 21)
           (PRIN2 "Computer wins ")
           (PRIN2 hand!-type)
           (PRIN2 " hand")
           (SETQ your!-total
                 (DIFFERENCE your!-total
                             your!-current!-bet)))
        (T (PRIN2 "We're both over, no winner") )) )

%*****************************************************
% BLACKJACK - main controller loop for playing
%      blackjack.

(DE BLACKJACK ()
   (PROG (command)
      (CLEAR)
      (SETQ your!-total 2000)
      (TERPRI)
      (CURSOR 1 12)
      (PRIN2 "Type any character to start....")
      (WHILE (EQ (DIRECTIO 255) 0)
             (DO (SETQ SEED!* (QUOTIENT (RANDOM) 819))))
      (WHILE (AND (NOT (MEMQ command '(!Q !q)))
                  (GREATERP your!-total 0))
        (DO
          (SETQ your!-current!-bet 0)
          (WHILE (OR (EQ your!-current!-bet 0)
                     (NOT (NUMBERP your!-current!-bet)))
           (DO
            (SETQ PROMPT!* " ")
            (TERPRI)
```

```
        (CURSOR 1 2)
        (CLEAR!-EOP)
        (PRIN2
"Please enter your bet, you can wager up to $")
        (PRIN2 your!-total)
        (PRIN2 " :")
        (SETQ your!-current!-bet (READ)) ))
      (deal)
      (COND ((NOT (MEMQ
                  (SETQ command (deal!-your!-cards))
                  '(!Q !q)))
        (redraw!-hand (REVERSE computer!-hand))
        (COND ((OR (LEQ (total!-cards your!-hand) 21)
                   (AND double!-hand
                        (LEQ
                         (total!-cards double!-hand)
                         21)))
               (WHILE (LEQ
                       (total!-cards computer!-hand)
                       16)
                (DO
                 (SETQ computer!-hand
                    (deal!-card computer!-hand))))))
        (CURSOR 1 4)
        (CLEAR!-EOP)
        (who!-won your!-hand 'first)
        (CURSOR 1 3)
        (CLEAR!-EOP)
        (COND (double!-hand
               (who!-won double!-hand 'double))) )) ))
    (CLEAR)
    (TERPRI)
    (CURSOR 1 12)
    (PRIN2 "Your total is $")
    (PRIN2 your!-total)
    (CURSOR 1 2)
    (SETQ PROMPT!* "*") ))
```

The program is executed by simply calling the function BLACKJACK.