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.

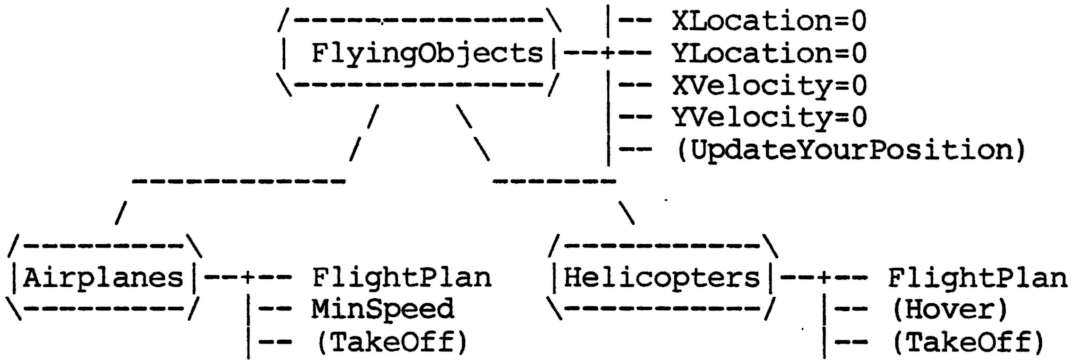
A small collection of Lisp functions forms an "object system". This includes functions for creating a class hierarchy of conceptual objects, functions for creating objects (members of the classes),

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.