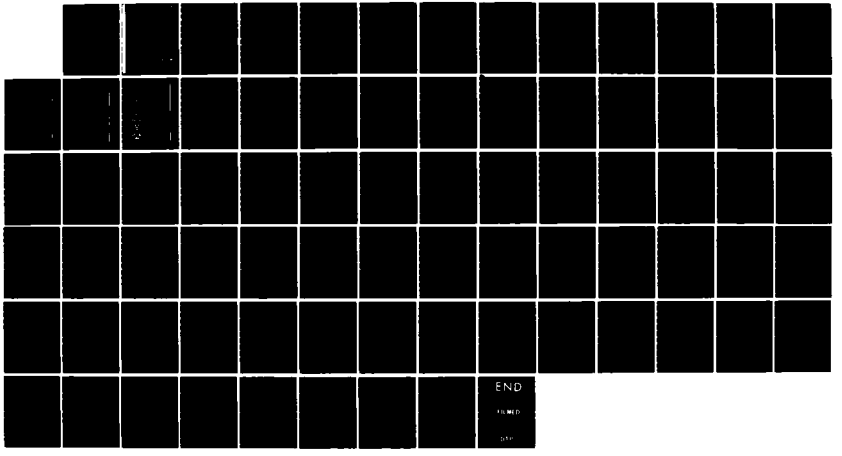
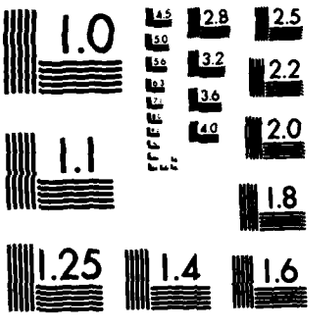
AD-A148 963 AN OBJECT-ORIENTED SIGNAL PROCESSING ENVIRONMENT: THE KNOWLEDGE-BASED SIG.. (U) MASSACHUSETTS INST OF TECH CAMBRIDGE RESEARCH LAB OF ELECTRON.. W P DOVE ET AL. 1/1

UNCLASSIFIED OCT 84 TR-502 N00014-81-K-0742 F/G 17/2 NL





(12)

An Object-Oriented Signal Processing Environment: The Knowledge-Based Signal Processing Package

Software authors:
Webster P. Dove and Cory Myers

Document author:
Evangelos E. Milios

Technical Report 502

October 1984

Massachusetts Institute of Technology
Research Laboratory of Electronics
Cambridge, Massachusetts 02139

An Object-Oriented Signal Processing Environment: The Knowledge-Based Signal Processing Package

Software authors:
Webster P. Dove and Cory Myers

Document author:
Evangelos E. Milios

Technical Report 502

October 1984

Massachusetts Institute of Technology
Research Laboratory of Electronics
Cambridge, Massachusetts 02139

This work has been supported in part by the Advanced Research Projects Agency monitored by ONR under contract N00014-81-K-0742 NR-049-506, in part by Sanders Associates Inc., and in part by an Amoco Foundation Fellowship.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION (6b. OFFICE SYMBOL (If applicable)) Research Laboratory of Electronics Massachusetts Institute of Technology	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research Mathematical and Information Scien. Div.
6c. ADDRESS (City, State and ZIP Code) 77 Massachusetts Avenue Cambridge, MA 02139	7b. ADDRESS (City, State and ZIP Code) 800 North Quincy Street Arlington, Virginia 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION (8b. OFFICE SYMBOL (If applicable)) Advanced Research Projects Agency	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-81-K-0742
8c. ADDRESS (City, State and ZIP Code) 1400 Wilson Boulevard Arlington, Virginia 22217	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. NR 049-506 / WORK UNIT NO.
11. TITLE (Include Security Classification) An Object-Oriented Signal Processing Environment: KBSP Package	
12. PERSONAL AUTHOR(S) W.P. Dove, C. Myers, E. E. Milios	
13a. TYPE OF REPORT Technical	13b. TIME COVERED FROM _____ TO _____
14. DATE OF REPORT (Yr., Mo., Day) October 1984	15. PAGE COUNT 72
16. SUPPLEMENTARY NOTATION This is M.I.T., R.L.E. Technical Report No. 502	
17. COSATI CODES FIELD / GROUP / SUB. GR.	18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

A LISP-based signal processing package for integrated numeric and symbolic manipulation of discrete-time signals is described. The package is based on the concept of "signal abstraction" in which a signal is defined by its non-zero domain and by a method for computing its samples. Most common signal processing operations are defined in the package and the package provides simple methods for the definition of new operators. The package provides facilities for the manipulation of infinite duration signals and periodic signals, for the efficient computation of signals over intervals, and for the caching of signal values. The package is currently being expanded to provide for manipulation of continuous-time signals and symbolic signal transformations, such as the Fourier transform, to form the basis of knowledge-based signal processing systems.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Kyra M. Hall RLE Contract Reports	22b. TELEPHONE NUMBER (Include Area Code) (617) 253-2569 / 22c. OFFICE SYMBOL

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

SECURITY CLASSIFICATION OF THIS PAGE

TABLE OF CONTENTS

INTRODUCTION

This document presents the philosophy and usage of the Knowledge-Based Signal Processing Package (KBSP).

The purpose of the KBSP software is to provide a solid and easy-to-use signal processing software facility on the M.I.T. Lisp machine, in the form of a very-high-level language for programming signal processing operations. It was designed to support the Knowledge-based Signal Processing projects at the M.I.T. Digital Signal Processing Group. This is the reason why its name includes the words "knowledge-based", in spite of the fact that the KBSP package is not a Knowledge-based system in the usual sense of the term.

The KBSP package is a growing system. This document describes the system as of July 1984. New signal processing operations for one-dimensional signals are being continuously added to the system. Future expansions of the system will include multidimensional signal processing, treatment of analog signals through a combination of numeric and symbolic processing and digital filter design. These expansions may make it necessary to revise some of the original design decisions of the KBSP package. Thus, this document may not completely apply to future versions of the KBSP package.

The philosophy of the system is loosely based on [Kopec], which pioneered the use of the concept of *data abstraction* as suitable for signal representation in programs. The advent of the M.I.T. Lisp machine made possible an efficient implementation of these ideas.

The experience from using the KBSP software on the Lisp machine so far has demonstrated that the combination of data abstraction ideas in signal processing and the unique programming environment of the Lisp machine can indeed give a powerful signal processing facility supporting incremental programming and powerful graphics all in one and the same machine.

Chapter 1 entitled "An example session with the KBSP software", attempts to give an idea of the signal processing environment that KBSP offers. Chapter 1 serves as the motivation of the

reader for learning about the KBSP system. However, the best motivation would be a live demonstration, so users with access to a Lisp machine are urged to try the example session.

Chapter 2 entitled "Lisp, Abstraction and Object-oriented Programming" gives an overview of the concepts in modern programming languages that are useful in giving an overall perspective of the KBSP environment. The exposition is intended for people with programming experience, but no previous contact with Lisp.

Chapter 3 entitled "The basic signal processing software" presents most of the available KBSP operators that are provided with the KBSP system. A short explanation is given with each operator, which is enough to enable the reader to use it. Very few references to the underlying implementation are made in this chapter. The chapter serves as the user's manual if used together with the alphabetical index of the KBSP operators.

Chapter 4 entitled "The KBSP implementation" is an explanation of the basic design decisions in the implementation of KBSP. This chapter is useful to the users who wish to extend or modify KBSP for their own use.

Chapter 5 entitled "the KBSP Graphics" offers a brief explanation of the KBSP graphics window and screen frame. The reader with minimal background in the Lisp Machine Window system should be able to make simple modifications to the Lisp code in order to generate different simple configurations of KBSP windows.

1. AN EXAMPLE SESSION WITH THE KBSP SOFTWARE.

The following session is an example session in which the interaction with the KBSP package is shown at the Lisp command level. A sequence corresponding to a Hamming window of length 32 is created, named, its values over an interval are computed and either returned to the user or placed into an array. The Hamming window is plotted over a specified interval and then its FFT and cepstrum are computed and plotted. An example with a signal stored in a file is also shown. A sequence is created from a file and plotted, and then a section of it is plotted together with the log-magnitude of its FFT and its real cepstrum.

In reading through the session, the following points are important:

- A * denotes the most recently created object (an array, a sequence etc.). Thus the last object is referred to with a * instead of the full command that created it.
- An indented comment is written above each command that the user types, explaining the command that follows. Whatever is between a command and the next comment is something that was printed by the system as a response to the command.
- The message "mouse a window" is printed by the system as a response to a plot command and prompts the user to place the mouse cursor inside a desired KBSP graphics window and click left. Then the window is selected and the plot command plots the object in this window.
- The default KBSP screen configuration, which is used in this session, consists of 4 KBSP graphics windows and a KBSP Lisp Listener, where the commands are typed. If a user wants his own KBSP screen configuration, he can either use the Edit Screen facility of the Lisp machine or define his configuration in a Lisp file, in the same way the default KBSP screen was defined.
- #<something> is the Lisp object that is returned by the corresponding command. If "something" starts with ART, the object is an array. Anything else is the name of a flavor type.

- The screen copies that follow give an idea how the actual display looks like. Note that the default KBSP display contains 4 KBSP graphics windows and a KBSP Lisp listener. Each KBSP window contains three panes, the top label, the bottom label and the graphics pane. The top label pane contains the range of the x-axis of the plot and the bottom label pane contains the name of the window (KBSP-WINDOW0,1,2 or 3), the range of the y-axis of the plot (minimum and maximum value) and the name of the plotted object (or the command that generated it). If the object is a complex sequence, the graphics pane is split into two, with a label pane in the middle and another one on the bottom.

The example session with KBSP

```
   ; take a hamming window of length 32
(hamming 32)
#<HAMMING 27732413>
   ; name it test
(seq-setq test *)
#<HAMMING 27732413>
   ; fetch its values over the interval [0 10]
(fetch-interval test [0 10])
#<ART-Q-10 27732542>                    ; result is an array of length 10
   ; list the values of the array
(listarray *)
(1.0 0.99058384 0.96272063 0.9175512 0.8569248 0.78332347 0.69976044
0.60965675 0.5167014 0.42469984)
   ; make an array of size 20 and name it "array"
(setq array (make-array 20))
#<ART-Q-20 27732653>
   ; fetch the values of test over [0 20] and place them into "array"
(fetch-interval test [0 20] array)
```

```
#<ART-Q-20 27732653>

   ; list the array

(listarray *)

(1.0 0.99058384 0.96272063 0.9175512 0.8569248 0.78332347 0.69976044
0.60965675 0.5167014 0.42469984 0.33741868 0.25843123 0.19097129 0.13780051
0.101095915 0.08236012 0. 0. 0. 0.)

   ; find the domain of test

(domain test)

(INTERVAL -16 16)

   ; find all KBSP functions containing the string "fft" in their names

(kbsp-apropos 'fft)

USER:SEQ-FFT-CONVOLVE - Function (X H), Flavor
USER:IFFT - Function (SEQUENCE &OPTIONAL (LENGTH (NEXT-POWER-OF-2
    ($LENGTH SEQUENCE)))), Flavor
USER:FFT-COMPLEX - Function (SEQUENCE LENGTH), Flavor
USER:FFT-REAL - Function (SEQUENCE LENGTH), Flavor
USER:FFT - Function (SEQ &OPTIONAL
    (LENGTH (NEXT-POWER-OF-2 ($LENGTH SEQ)))), Flavor
(FFT FFT-REAL FFT-COMPLEX IFFT SEQ-FFT-CONVOLVE)

   ; plot "test" over its domain (see screen copy 1, window 0)

(plot test nil)

mouse a window
#<HAMMING 27732413>

   ; plot test over the interval [-30 30] (see screen copy 1, window 1)

(plot test nil [-30 30])

mouse a window
#<HAMMING 27732413>

   ; take the fft of test

(fft test)

#<FFT 27743544>

   ; plot it (see screen copy 1, window 2)

(plot * nil)
```

mouse a window
#<FFT 27743544>

; take the real cepstrum of test

(cepstrum test)

#<CEPSTRUM 30031337>

; plot it (see screen copy 1, window 3)

(plot * nil)

mouse a window
#<CEPSTRUM 30031337>

; take the logarithm of the magnitude of the fft of test over the positive
frequencies.

(log-mag (fft test))

#<LOG-MAG 30201376>

; plot it. (see screen copy 2, window 2)

(plot * nil)

mouse a window
#<LOG-MAG 30201376>

; take the log-mag of the fft of size 512

(log-mag (fft test 512))

#<LOG-MAG 30201757>

; plot it (see screen copy 2, window 3)

(plot * nil)

mouse a window
#<LOG-MAG 30201757>

; define a sequence corresponding to the preemphasized version of
; a speech file stored on another computer

(seq-setq eyes (preemphasize (file "dspg://usr//lib//speech//dat//eyes.s02")))

Enter user name for host DSPG:
Password for logging in to DSPG as eem (or Escape to change user id):
#<PREEMPHASIZE 30211126>

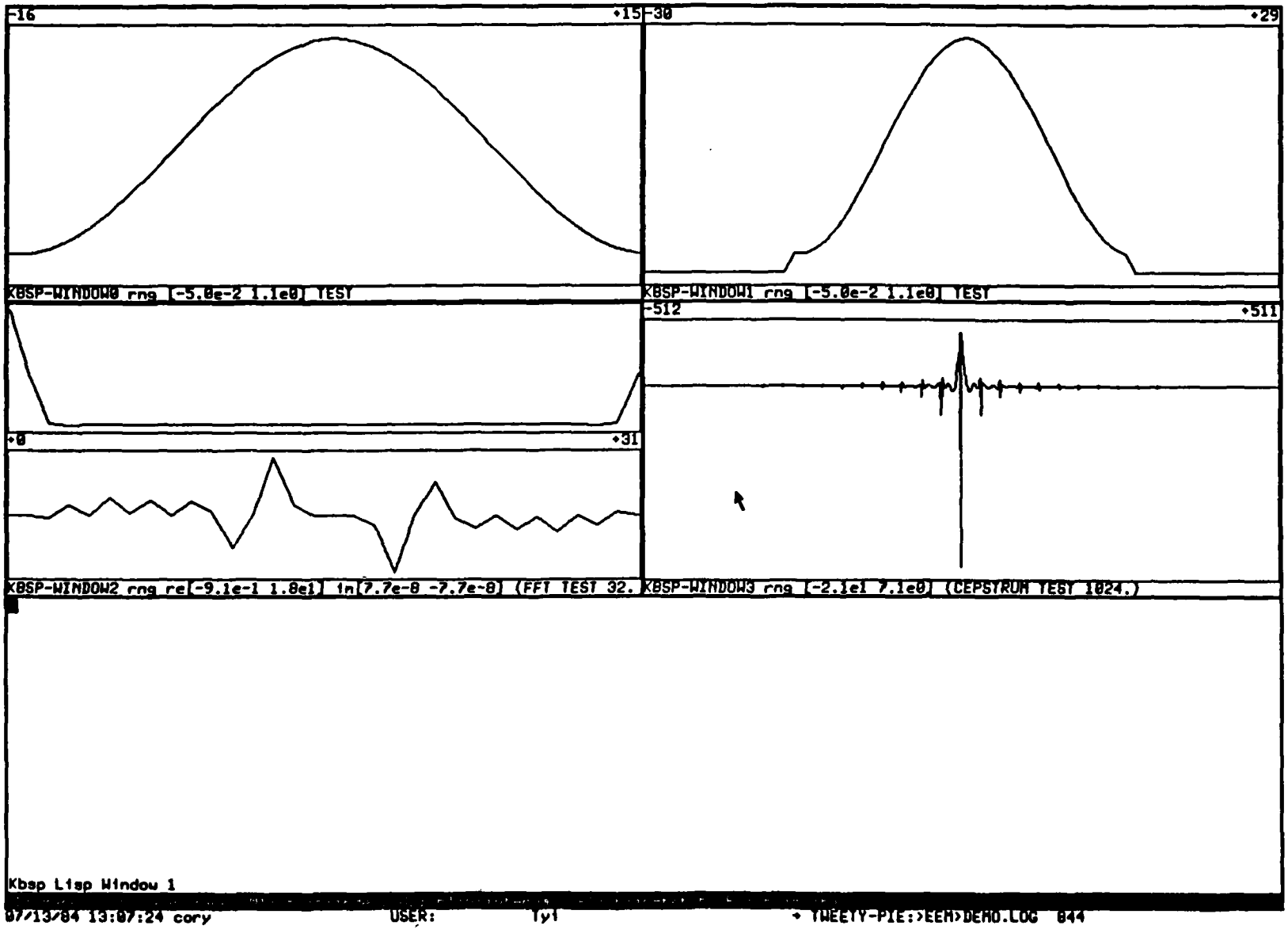
```
   ; plot the preemphasized speech file (see screen copy 3, window 0)

(plot eyes nil)

mouse a window
#<PREEMPHASIZE 30211126>

   ; plot eyes over [8500 10500] (see screen copy 3, window 1)

(plot eyes nil  [8500 10500])

mouse a window
#<PREEMPHASIZE 30211126>

   ; name the section over the interval [8500 10500]

(seq-setq piece (section eyes [8500 10500]))

#<SECTION 31164662>

   ; plot the log-mag of the fft of piece (see screen copy 3, window 2)

(plot (log-mag (fft piece)) nil)

mouse a window
#<LOG-MAG 31165021>

   ; plot the real cepstrum of piece with fft size 2048
   ; over the interval [-64 64] (see screen copy 3, window 3)

(plot (cepstrum piece 2048) nil [-64 64])

mouse a window
#<CEPSTRUM 31175243>
```

The reader should notice that the graphics windows are mouse-sensitive. Clicking on a particular point in the waveform pane gives a vertical line at this point, while the coordinates of the intersection of corresponding point on the waveform are shown on the top label pane of the window. See screen copy 3.

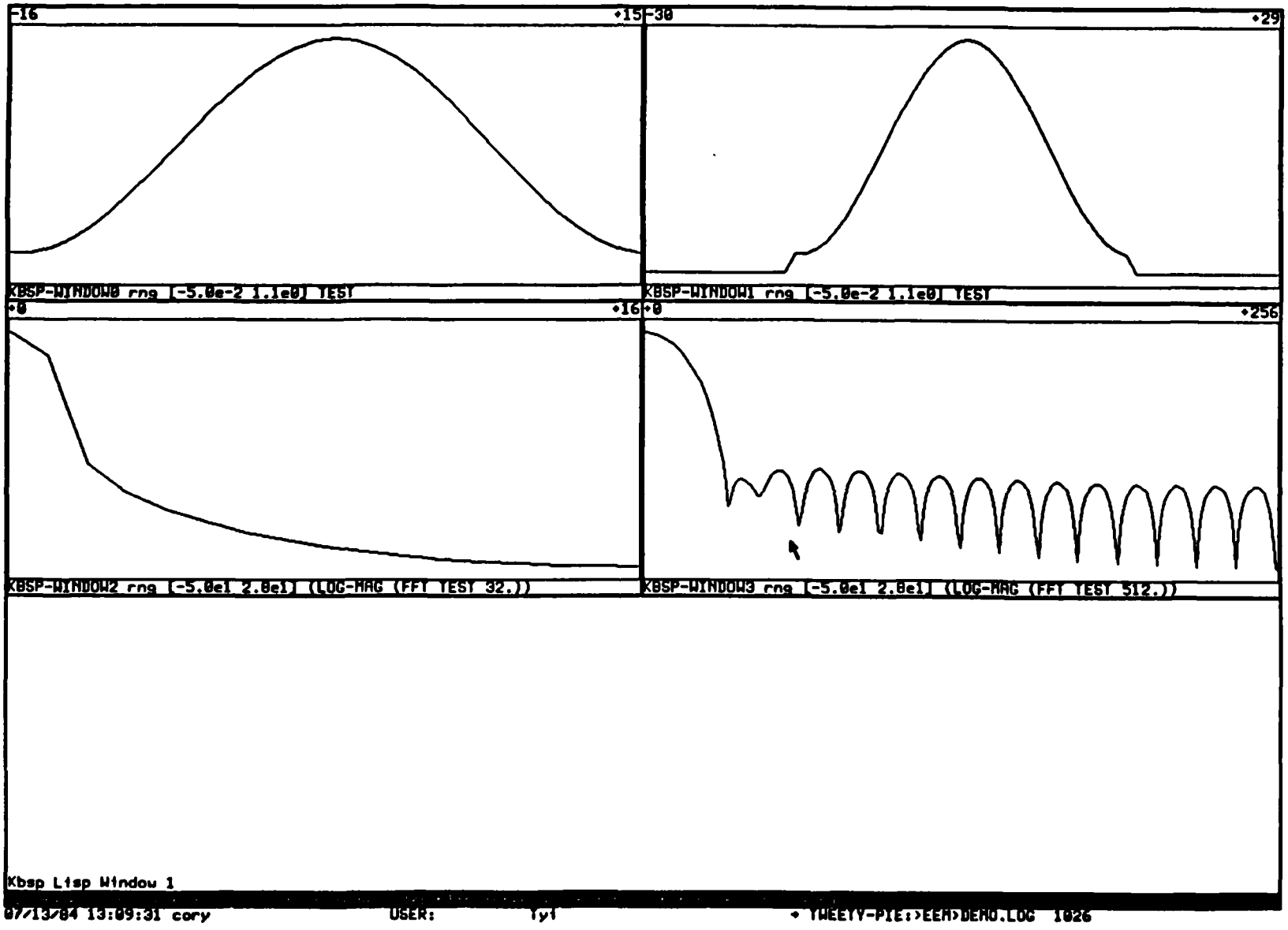
The bottom pane of a kbsp graphics window is also mouse-sensitive. If one clicks left on it, the history of computations that led to the waveform in the window is shown on the kbsp lisp window. The example on screen copy 3 shows the result of clicking left on kbsp windows 0, 2 and 3.

This example session reveals only a part of the usefulness of the KBSP package. It shows nothing about the ease with which new operators can be programmed, a topic which is explained

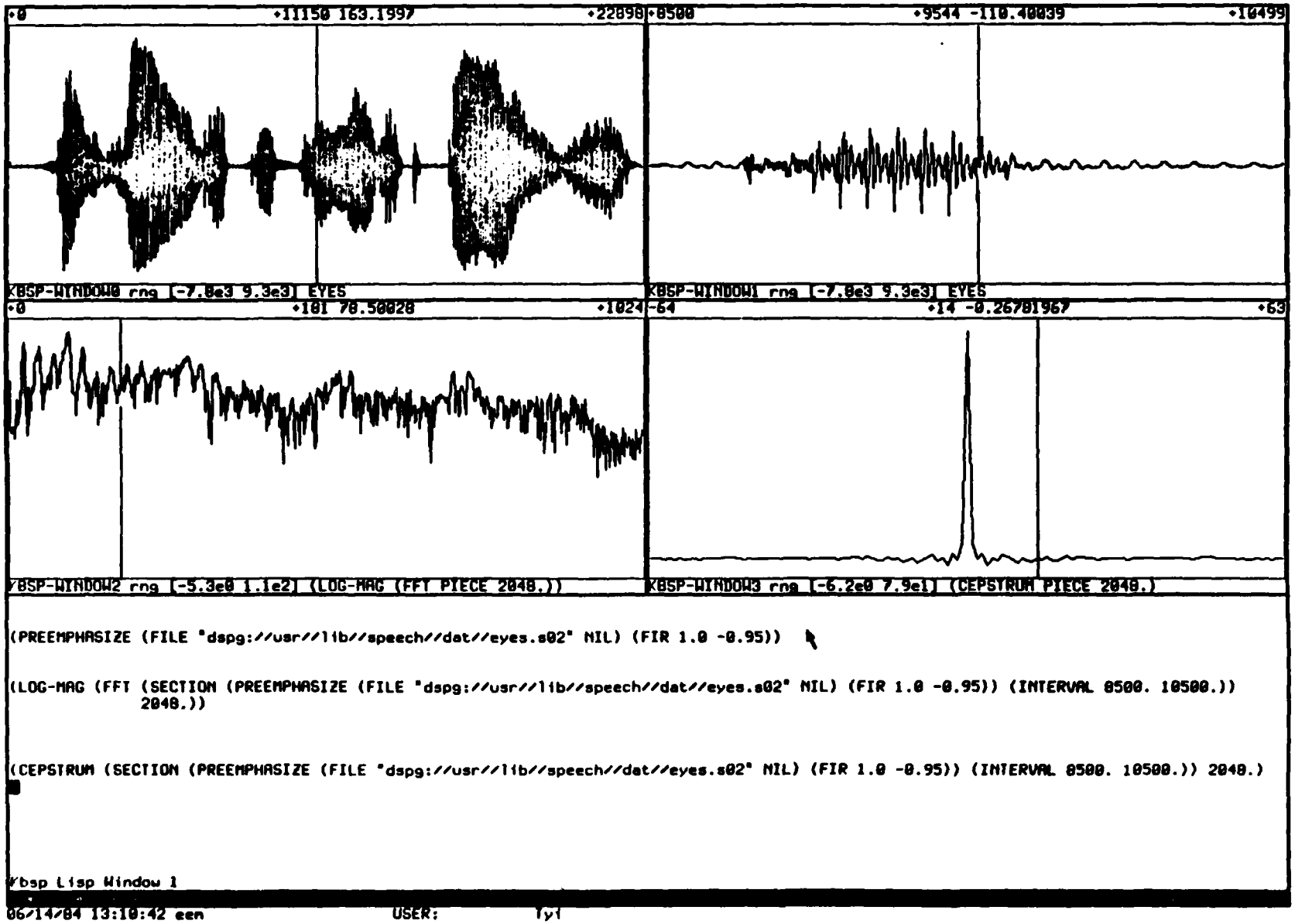
in Chapter 4 on the KBSP implementation.



SCREEN COPY #1



SCREEN COPY #2



SCREEN COPY #3

2. LISP, ABSTRACTION AND OBJECT-ORIENTED PROGRAMMING.

This section attempts to convey some of the concepts of modern programming languages as they were incorporated in Zeta Lisp, the Lisp dialect running on the MIT Lisp machine [Weinreb], and used by the designers of the KBSP software. It does not attempt to be complete or detailed in any sense. The interested reader is referred to the bibliography for further study.

Zeta Lisp historically evolved from the MIT Maclisp. It combines features of the original Lisp of the 60's, ideas about *data abstraction* in programming languages and characteristics of the Smalltalk *object-oriented* programming environment. Familiarity with these concepts will prove very helpful in the understanding and use of the signal processing software on the Lisp machine and of the Lisp machine in general.

From a broad perspective, the Lisp Machine programming environment represents one effort to improve software productivity by providing advanced facilities to support incremental and interactive programming. The importance of such efforts derives from the fact that conventional programming languages have shortcomings that make them inadequate for building and maintaining large software systems [Barstow, Chapter 25]. John Backus states [Backus]:

> *Conventional Programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor, the Von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.*

Whether the Lisp machine environment in general and the KBSP package in particular are a step in the right direction will be proven by experience. They certainly are a step away from some

of the problems that Backus attributes to conventional programming.

THE ORIGINAL LISP

The original Lisp, developed at M.I.T. by John McCarthy in the early 60's [McCarthy] is best described by the term *functional or applicative language* [Backus]. Programming in a functional language consists of writing procedures or functions which resemble the concept of mathematical functions (mappings) more than that of programming language procedures (the term *mapping* will be used for mathematical functions, because the term "function" means "procedure" in certain programming languages. Under a mapping *f* over a domain *D*, the *image* of an element *x* in *D* is denoted by *(f x)*, using the Lisp prefix notation). Lisp functions can be viewed as mappings of the domain of definition of their arguments. For specific values of their arguments, they return the image of these values under the mapping they define. In general, for *pure* Lisp functions the returned values (1) depend on the values of the arguments only (2) are new copies rather than modified versions of the arguments and (3) more generally, Lisp functions are not supposed to have *side effects*, i.e. they are not supposed to modify their arguments or other existing data.

In a functional language like Lisp, the programmer does not think in terms of variables and their side-effects through execution of procedures but thinks in terms of mappings. A program in Lisp is a mapping obtained from simpler mappings by using certain simple composition rules.

As an example of how operators are composed in a functional language consider the real cepstrum operation. Assuming that we have already defined Lisp functions for computing the logarithm of the magnitude of a complex sequence, the Discrete Fourier Transform of a sequence and the real part of a complex sequence, we can write another Lisp function to compute the real cepstrum of the sequence by applying the previous three Lisp functions on the input sequence in the right order pretty much like the composition of mappings in mathematics.

Original Lisp was basically an interpreted programming language, in which the programmer interacts with a Lisp *environment*, as opposed to traditional compiled languages, in which the

programming activity consists of the Edit-Compile-Run-Debug loop. Lisp allowed incremental programming, in which the programmer incrementally builds his programs as functions made of simpler functions. Previously defined functions are "remembered" by the Lisp environment. The programmer can either apply them to arguments or combine them in order to create new Lisp functions, which are in turn remembered by the Lisp environment. Because the programmer can apply the intermediate functions easily, debugging is nicely integrated into the programming activity, as opposed to the conventional Edit-Compile-Debug loop, where three very different programs must be used: the operating system interpreter, the compiler and the debugger.

An important feature that Lisp introduced is that of treating programs like data. This derives from the interpreted nature of Lisp and allows Lisp programs to examine, generate or modify other Lisp programs.

One may wonder of course why Lisp has not had much appeal to the programming community at large. There are many reasons, one being that the programming concept it introduced was not easily implementable efficiently on conventional von Neumann architectures. Memory management is implicit in the Lisp environment. Once an object is not accessible anymore, it is declared garbage, and a special, complicated and costly program, called the *garbage collector*, is needed to identify and free the storage garbage occupies [Steele].

ABSTRACTION IN PROGRAMMING

The motivation behind the work in very-high-level languages is to ease the programming task by providing the programmer with a language containing primitives or abstractions suitable to his problem area. The programmer is then able to spend his effort in the right place; he concentrates on solving his problem, and the resulting program will be more reliable as a result [Liskov].

Functions in Lisp and procedures in conventional high-level languages are extremely helpful in programming, because they allow the partitioning of the task into different levels of abstraction. One builds the cepstrum Lisp function not out of additions and multiplications of numbers, but out

of higher level procedures, which are in turn implemented in terms of arithmetic operations. This process is known as *procedural abstraction*. In the mid-70's, however, it was realized that another kind of abstraction, *data abstraction*, could improve the productivity of programmers by allowing the partitioning of the complexity of programs in a direction orthogonal to that of procedural abstraction. Pascal records and C structures are examples of attempts to assist in data abstraction: Related pieces of data are packaged together and in many instances the package is treated as a whole. As an example, consider a piece of digitized speech data. An array of numbers can be used to hold the actual numerical values. However, additional information needs to be stored together with the data, such as the sampling rate, the age and sex of the speaker, the duration of the segment, day, time and place of the recording. Thus one could imagine a data structure for speech signals with the following components: a floating precision array for storing the data, an alphabetic string for storing the age and sex of the speaker and so on.

A key issue with data structures as described above is whether the user is allowed to access the components in any way he wants. For example, is he allowed to access the array storing the data and take it apart using the ordinary array operations? Accessing the internal implementation of a data structure may not be desirable, because the whole program must change if the implementation changes. Hence a discipline is useful according to which usage of the data structure is separated from its internal implementation by means of a clean interface defined in terms of the problem area. The next question is whether such disciplined use of data structures is left to the good will of the programmer or whether is enforced by the programming language itself. Pascal and C, for example, encourage the disciplined use of data structures but they do not enforce it.

A language supports data abstraction if it has the following features:

1. It allows definition of data types by packaging together conceptually related pieces of data, called elements of the data type.

2. It automatically provides a set of functions that allow certain primitive operations on the data type without directly accessing the internal representation. Such functions are functions for

accessing the elements, modifying the elements and initializing the elements when the data type is instantiated (for example, "real number" is a data type. Real variable x is an instance of the data type "real number". Defining the real variable x is equivalent to instantiating the data type "real number" once. Of course, "real number" is a trivial data type with a single element).

3. It provides two distinct views of a data type: the *concrete* (for implementing the data type as a package of related pieces of data) and the *abstract* (for using the data type). The abstract view consists of a class of objects (all possible instances of the data type) and a set of operations that can be performed on these objects.

4. It provides facilities for defining new operations on the objects as part of the abstract view of the data type.

5. It provides facilities for building a new data type by combining old ones, where the abstract view of the component types becomes part of the new data type automatically by inheritance.

Examples of abstract data types:

(1) A *stack* can be viewed as a data type on which two operations can be performed: *push* and *pop*. This is the abstract view of the stack: push(x,y) pushes the value x onto the stack y. Pop(y) returns the value last pushed into the stack and removes the value from the stack. Pop(y) signals "error" if the stack is empty. A stack can be implemented in different ways:

a. As a pair of an array and an index variable, where the index variable contains the index of the top of the stack, push(x,y) increases the index variable by 1 and stores x in the corresponding array index and pop(y) returns the value of the array index equal to the index variable and decreases the index variable by 1. If the index variable is 0, an error is signalled.

b. As a linked list, i.e. a set of cells with two entries, the first being the value stored in the cell and the second being the pointer to the next cell. A pointer points to the cell containing the top of the stack, while the pointer of the last cell points to "error". Push(x,y) grabs a free

cell, sets its value to x and its pointer to the current top of the stack, and sets the pointer to the stack to point to the newly allocated cell. Pop(x) returns the value of the top of the stack, deallocates the corresponding cell and sets the pointer to the stack to point to the cell pointed to by the cell just deallocated.

A higher level program that uses the stack data type is independent of which of the two implementations is actually used and is written in terms of push and pop, which are meaningful operations to the user, as opposed to operations on with indexes of arrays or pointers of linked lists.

(2) A basic signal can be viewed as a data type with two operations: getting the domain of definition and fetching the value at a point of the domain [Kopec]. A speech signal can be viewed as another data type with components other abstract data types: a basic signal, time and date of recording, sampling rate and age and sex of the speaker. Notice that mechanisms for defining new abstract data types by combining already defined abstract data types is a useful feature of a programming language.

In an abstract data type, the operations of its abstract view are the only means of using the data type and serve as the "contract" for its use. In the previous example, if the user wants to know the sex of the speaker, he should not be able to directly access the string that stores this information. He must use the operation that gets the sex of the speaker, which is part of the abstract view of the data type. It is this operation that will access the string, not the user. Thus the top level program and the implementation of the data structure are totally separated from each other. This has several advantages: the top level program is more meaningful and independent of the data structure implementation. The programmer of the top level only needs to look at the "contract" and he will be able to write his program. If the implementor of the data structure wants to change the data structure implementation, he can do so without the need for changes at the top level, as long as his new contract is compatible with the old one (i.e. it provides at least the functions promised by the old one).

Zeta Lisp provides the necessary facilities for defining abstract data types and operations on them. The concepts of abstract data types in Zeta Lisp are basically as explained above, but the terminology is different. The term *flavor* is equivalent to the term "abstract data type", the term *object* denotes a flavor instance (the term "instance" is used in the same sense that a real variable x is an instance of the type "real number". A flavor corresponds to "real number", a flavor instance (object) corresponds to the memory cell holding the value of x, and the name of the object corresponds to x) and an operation on a flavor is called *method*. The act of invoking a method on a flavor instance (in general with arguments like a Lisp function) is called *sending or passing a message to an object*.

The concept of procedural and data abstraction is central to the signal processing software on the Lisp machine. Procedural abstraction is achieved by extending the Lisp function mechanism to include *systems*. Systems are a generalization of functions and deal with *sequences*, which is the abstract data type for signals. The basic abstract operations on a sequence are finding the domain of its definition and computing its numerical values over a specified interval which is a subset of its domain. Sequences are implemented using flavors. Sequences are *immutable* objects, i.e. no operation modifies its input sequences, but instead it returns a modified copy. Thus, systems implement side-effect-free operations. Each system has a flavor type associated with it and all objects that it outputs are instances of this flavor type. The system/sequence mechanism is part of Lisp. Conceptually, systems can be treated just like Lisp functions, which accept sequences as inputs. They work like Lisp functions, except for the extra bookkeeping they perform related to the output sequence.

A central design decision was the idea of *delayed* or *lazy evaluation* [Kopec]. Applying a system to its arguments does not cause any computation to happen. The mechanism for the computation is set up (the output flavor is defined and instantiated and the functions that perform the computation are set up - remember that Lisp can treat programs as data). Only when a request is issued such as plotting the sequence or getting the numerical values over an interval does real computation happen. Moreover, the minimum amount of computation occurs. For example, only the

values over the specified interval are computed, not the values over the full domain of the sequence. An elaborate mechanism for achieving delayed evaluation exists as part of the underlying signal processing language.

The idea of delayed evaluation reflects a shift of focus in signal representation: a signal is not viewed as the collection of its numeric values, but as a symbolic entity, described by the sequence of operations that were applied to generate it. The view of a signal as a symbolic entity has enormous potential for operating on infinite-duration discrete signals and on analog signals and for reasoning about signals based on their symbolic description.

OBJECT-ORIENTED ENVIRONMENT

The concept of abstract data type as a programming language feature was introduced in the previous section. Languages like Modula-2 or ADA support abstract data types in the context of a *strongly-typed* programming language, which performs type checking at compile-time. In knowledge-based programming, however, compile-time type checking is not always possible, because of the existence of *dynamic data types*, whose type is determined at run time [Barstow]. An *Object-oriented environment* supports dynamic abstract data types [Barstow Ch. 8, Byte]. The central theme in an Object-oriented environment is the concept of object as instance of an abstract data type. According to this view, an object has operations which belong to its abstract view or its *interface*, "private memory" for storing information, which can only be manipulated by operations in the object's interface.

In a pure object-oriented environment, objects are *the only* structuring mechanism, around which the software is built. The concept of procedure is replaced by the concept of *message*, according to which an object carries out one of its operations when another object sends it a message to do so. This is *the only* way that action can occur in an object-oriented environment and it is called *object communication through message passing*. Thus the programming activity in an object-oriented environment is centered around choosing the appropriate data abstractions and providing

them with suitable operations (messages). Program "execution" consists of message sending between objects.

In Zeta Lisp, data abstraction and the existence of objects and message passing is viewed as complementary to procedural abstraction, hence it is not the only structuring mechanism. Zeta Lisp provides the facilities supporting a wide range of programming styles, including functional programming, object-oriented programming and combinations of them in various ways.

LEVELS OF USER INTERACTION IN THE KBSP PACKAGE

The user can interact with the KBSP package at three different levels:

1. The **top level**, where the user does not define new operations that generate signals, but uses only the existing ones. The top level of the KBSP package is the same as that of Lisp. The Lisp language has been enriched with one more abstract data type corresponding to a signal, that of *a sequence*. The Lisp functions that have sequences as outputs are called *systems*. Systems are extended Lisp functions that take the burden of bookeeping associated with sequences off the user. A mechanism is provided for abstractly combining existing systems to define new systems (SYS-ALIAS). Chapter 3 presents the top level view of the KBSP package, namely all signal operations (Lisp functions and systems) that constitute the core of the KBSP package. Chapter 5 explains the KBSP graphics facilities at the top level.

2. the **system definition level**, where the user not only uses the existing systems at the top level or combines them abstractly, but also uses them as building blocks for creating his own systems by operating on the internal representation of sequences. At the system definition level, the user comes in touch with the undelying object-oriented philosophy of the package, which has been used at the implementation level. A system definition expands into an abstract data type definition and instantiation. The various forms in a system definition translate into method definitions for the sequence type being defined. This viewpoint is explained in Section 4.3.

3. the **KBSP maintenance and modification level**, where the user changes the undelying KBSP language, namely the Lisp facilities that enable the definition of sequences and signals easily at the top level. A user may want to change the internals of KBSP, if he finds that the current KBSP is inadequate for his specialized needs. However, this requires a thorough understanding of the implementation that can only be acquired through detailed study of the code.

2.5 SUMMARY

The ideas mentioned in this chapter are an outcome of the research activity in the area of programming languages and software methodologies during the 70's. They are gradually becoming accepted practice during our decade mainly because the concepts are now better understood and also because advances in the VLSI technology and computer architectures have made possible efficient implementations of the ideas. One such example is the M.I.T. Lisp machine [Weinreb], which provides an integrated programming environment including facilities supporting the previous ideas together with an integration of the programming language with the operating system (not only is most of the Lisp machine operating system written in Lisp, but it is also part of Lisp).

The top level view that the KBSP package offers to the user is that of a functional programming environment, in which systems are treated like mappings (functions) and sequences are primitive objects. Systems can thus be considered as generalized Lisp functions and sequences as primitive data types in the KBSP environment.

The lower level implementation view of the KBSP package approximates that of an object-oriented implementation in that most operations are translated into messages that are passed between objects. However, the top-level user is not required to use message sending, because most message sending operations have been repackaged as Lisp function calls to provide a uniform Lisp-function-oriented top-level view (message sending in Zeta Lisp has different, and for some people, confusing syntax).

3. THE BASIC SIGNAL PROCESSING SOFTWARE

3.1 SETS AND INTERVALS

Intervals are a simple and very important concept in the KBSP software and the user should become familiar with the way intervals are represented and used. The notation [a b] denotes an interval starting at "a" and ending at "b". "a" is included in the interval but "b" is not, i.e. the interval is closed on the left but open on the right (in fact, the notation [a b] is equivalent to a Lisp function that creates an interval and it can be typed in instead of (INTERVAL a b)). A variety of functions is provided for using intervals, such as constructing an interval from its bounds, finding an interval's start and end, testing whether an interval is empty and finding covers and intersections of intervals. In the context of the previous chapter, intervals can be viewed as an abstract data type (although it has not been implemented as a flavor, but as a Lisp structure).

Supports are used to describe possibly noncontiguous regions of the number line. Each support contains one or more intervals. Each interval describes a contiguous region of the number line closed on the left and open on the right.

This section presents generic operations on numbers introduced to accommodate infinite values as well and basic operations on intervals and sets. The prefix "$" indicates a generic operator, i.e. an operator that applies to extended numbers, or, more generally, to objects of a variety of types. Note that the Zeta Lisp notation for function arguments is being used throughout. The notation is valid for systems, as well, and has the following form:

FOO (A1 A2 ... An &OPTIONAL O1 O2 ... On &REST RESTARGLIST)

FOO is the name of the function or system. A1, A2, ..., An are the required arguments, O1, O2, ..., On are the optional arguments and RESTARGLIST is a list bound to the rest arguments. See [Weinreb] for more explanation of this format.

Extended number system

The extended number system contains the real numbers and $\pm\infty$. The basic operations and predicates on numbers are extended to take into account the case in which one of the arguments may be $\pm\infty$.

INF, MINF constants

These constants have symbolic values "INFINITY" and "MINUS-INFINITY", respectively and their mathematical properties are taken into account when they appear in generic arithmetic operations.

EXTENDED-NUMBERP (X)

Predicate for testing whether object X is an extended number.

$= (A &REST OTHERS)

Predicate for testing whether its arguments are all equal (eq if they are not real numbers) (In Lisp, there are two kinds of equality: Two variables are *equal* if they are names for two objects that look the same, but may occupy different pieces of storage. Two variables are *eq* if they are two different names, i.e. aliases, for the same object).

$>, $<, $>=, $<= (A B)

Two-argument predicates for extended numbers

$MAX, $MIN (&REST ARGLIST)

Return the maximum or minimum of a list of extended numbers.

$MINUS (A)

Returns the negative of an extended number A.

$+, $*, $-, $// (A B)

Return the sum, product, difference, quotient of A and B, where A and B are extended numbers.

$1+, $1- (X)

Increase or decrease the extended number X by 1.

Intervals

Intervals are implemented as Lisp structures. The interval has two components, START and END. The functions that access the components are provided automatically by the Lisp structure and they are INTERVAL-START and INTERVAL-END. They take an interval instance as an argument and return the corresponding component. In addition, there are generic operations $START and $END, that apply to intervals as well as to other objects with a start and an end

(such as signals). A predicate function is also provided, INTERVAL-P, for testing whether an arbitrary object is an interval. The constant NULL-INTERVAL is the undefined interval [nil, nil].

INTERVAL (START END)

INTERVAL (START END)

Create and return an interval from START up to, but not including, END. If START is greater than or equal to END, NULL-INTERVAL is returned.

[START END]

Shorthand notation for (INTERVAL START END) which can be typed in instead of the longer expression.

NULL-INTERVAL-P (INTERVAL)

NON-EMPTY-INTERVAL-P (INTERVAL)

Returns the start of the interval if the interval is nonempty, otherwise it returns nil.

INTERVAL-LENGTH (INTERVAL)

Returns the length covered by this interval.

FINITE-INTERVAL-P (INTERVAL)

Returns T if the length interval is greater than or equal to 0 and less than infinity.

INTERVAL-INTERSECT (&REST INTERVALS)

Returns the interval which is the intersection of INTERVALS. If INTERVALS do not intersect, NULL-INTERVAL is returned.

INTERVAL-ADJOINING-P (&REST INTERVALS)

Predicate testing whether all the intervals are neighbors of at least one point. For example,

(interval-adjoining-p [0 3] [4 5]) -> nil
(interval-adjoining-p [0 3] [0 5]) -> T
(interval-adjoining-p [0 3] [3 5]) -> T

INTERVAL-COVER (&REST INTERVALS)

Returns the smallest interval that completely covers INTERVALS. For example,

(interval-cover [0 2] [1 3]) -> [0 3]
(interval-cover [0 2] [3 4]) -> [0 4]

INTERVAL-COVERS-P (A B)

Predicate testing whether interval A completely covers B. B can be a number or an interval.

INTERVAL-EQ (A B)

Predicate testing whether intervals A and B are identical.

INTERVAL-INTERSECT-P (&REST INTERVALS)

Returns the intersection of the intervals. If INTERVALS do not intersect, it returns NIL. Compare with INTERVAL-INTERSECT.

INTERVAL-DELAY (INTERVAL DELAY)

Returns INTERVAL shifted to the right by DELAY.

INTERVAL-SAMPLE (INTERVAL SAMPLING-RATE)

Return a new interval by sampling INTERVAL at the sampling rate. It basically divides start and end of INTERVAL by the sampling rate and returns the resulting interval.

Generic operations involving intervals

$GET-INTERVAL, $START, $END, $LENGTH (OBJECT)

Returns the interval, the start, the end and the length of the interval associated with OBJECT. If no interval is associated, return NIL. These operations apply to many object types, such as signals and intervals.

Supports

Supports are Lisp lists containing nonadjoining non-empty intervals in ascending order. The first element of the list is the atom ': support. NULL-SUPPORT is a list with only one element, the atom ': support. Supports have not been used in the existing KBSP software, so the rest of this section can be skipped at first reading without loss of continuity.

SUPPORT-P (OBJECT)

Predicate for testing whether OBJECT is a support.

SUPPORT (&REST ARGLIST)

The support which completely covers all elements of ARGLIST. ARGLIST is a list of intervals and/or supports.

NULL-SUPPORT-P (SUPPORT)

Predicate testing whether SUPPORT is empty.

FINITE-SUPPORT-P, NON-EMPTY-SUPPORT (SUPPORT)

Predicate testing whether SUPPORT has at least one interval.

SUPPORT-COVERS-P (A B)

Predicate testing whether support A completely covers B. B is an interval, support or number.

Generic set operations

$NULL (OBJECT)

Predicate testing whether OBJECT is null. OBJECT is interval or support.

$COVERS-P (A B)

Predicate for testing whether A completely covers B. A and B are numbers, intervals or supports.

$INTERSECT (&REST ARGS)

Returns the support which is the intersection of the arguments (which are intervals or supports).

$COVER (&REST ARGLIST)

Returns the interval which completely covers ARGLIST. ARGLIST is a list of intervals, supports or numbers. Note that $COVER returns an interval, as opposed to $INTERSECT, which returns a support.

$COMPLEMENT (SET UNIVERSE)

Returns the support which is the complement of SET (an interval) with respect to UNIVERSE (an interval or support).

$INTERSECT-P (&REST ARGLIST)

If the intersection of arguments is non-empty, it is returned. Otherwise return NIL.

3.2 BASIC FUNCTIONS FOR DEALING WITH SEQUENCES.

3.2.1 FETCHING SEQUENCE INTERVALS

The following functions enable the user to fetch the values of a sequence over a given interval. If the value of CACHED? is ':NO, the fetched values are just returned. Otherwise, the values are saved away in a buffer, which is part of the sequence data structure (a flavor instance). The buffer is an array that uses space, unless steps are taken to prevent it. In the case of caching, if a fetch operation is performed later, requesting some of these values, they will not be computed again (this is the default behavior of the system). It should be noted that the FETCH operations do cause computation to happen, in contrast to the application of a system to its arguments. For more explanation of buffering (caching) of sequences the reader is referred to the chapter on the underlying implementation mechanisms. The reader should be cautioned that the caching philosphy just mentioned and more fully described in section 4.4 may change in future implementations.

FETCH-INTERVAL (SEQ INTERVAL &OPTIONAL OUTPUT-ARRAY CACHED?)

Fetch a sequence over an interval. Return an array, if OUTPUT-ARRAY is not provided, or return the values into the OUTPUT-ARRAY. If CACHED? is given and it is equal to ':NO, then the resulting sequence values are not saved away and the buffer of SEQ does not change. Otherwise, the resulting values are saved (in the buffer of SEQ).

FETCH-IMAGINARY-INTERVAL
(SEQ INTERVAL &OPTIONAL OUTPUT-ARRAY CACHED?)

Fetch the imaginary part of a sequence over an interval. Return an array, if OUTPUT-ARRAY is not provided, or return the values into the OUTPUT-ARRAY.

FETCH-COMPLEX-INTERVAL
(SEQ INTERVAL &OPTIONAL OUTPUT-REAL-ARRAY OUTPUT-IMAG-ARRAY CACHED?)

See above.

FETCH (SEQ INDEX)

Fetch the value of SEQ corresponding to INDEX. A loop of FETCH operations can accomplish the same as a FETCH-INTERVAL operation, but it is much slower because of the function call that occurs with every FETCH. This comment applies to FETCH-IMAGINARY and FETCH-COMPLEX.

FETCH-IMAGINARY (SEQ INDEX)

Fetch the imaginary value of SEQ corresponding to INDEX.

FETCH-COMPLEX (SEQ INDEX)

Fetch a pair of real values of SEQ corresponding to INDEX. The pair is returned via the multiple value mechanism.

FETCH-UNCACHED-INTERVAL (SEQ INTERVAL &OPTIONAL OUTPUT-ARRAY)

Same as FETCH-INTERVAL but uncached, meaning that the resulting sequence values are not buffered and that the sequence buffers do not change as a result of FETCH-UNCACHED-INTERVAL.

FETCH-UNCACHED-IMAGINARY-INTERVAL
(SEQ INTERVAL &OPTIONAL OUTPUT-ARRAY)

Same as FETCH-IMAGINARY-INTERVAL but uncached.

FETCH-UNCACHED-COMPLEX-INTERVAL
(SEQ INTERVAL &OPTIONAL OUTPUT-REAL-ARRAY OUTPUT-IMAG-ARRAY C[illegible]HED?)

See above.

FETCH-UNCACHED (SEQ SAMPLE)

Same as FETCH but uncached.

FETCH-UNCACHED-IMAGINARY (SEQ SAMPLE)

Same as FETCH-IMAGINARY but uncached.

FETCH-UNCACHED-COMPLEX (SEQ SAMPLE)

Same as FETCH-COMPLEX but uncached.

3.2.2 UTILITY FUNCTIONS FOR DEALING WITH SEQUENCES

These functions perform certain miscellaneous operations on sequences, such as naming, unnaming or finding the name of a sequence, showing the computations that led to a sequence and flushing sequences. For more explanation of buffering (caching) of sequences the reader is referred to the chapter on the underlying implementation mechanisms.

SEQ-SETQ (&QUOTE &REST ARGS)

The version of the Lisp form "setq" that must be used for naming sequences. An even number of arguments is needed. The second, fourth, ... are sequences (objects) while the first, third, ... are the corresponding names. The keyword &QUOTE means that the arguments are not evaluated (so it works like the Lisp SETQ).

UNNAME (SEQ)

Unnames SEQ.

SEQ-NAME (OBJECT)

Prints out the name of the OBJECT.

DOMAIN (SEQ)

Returns the domain of SEQ.

PERIOD (SEQ)

Returns the period of SEQ.

COMPUTE-DOMAIN (SEQ)

Returns the default compute-domain of SEQ.

SEQUENCEP (OBJ)

Predicate which tests whether OBJ is a sequence.

$DOMAIN (OBJECT)

Returns the domain of OBJECT, if it has one, else NIL.

SHOW (OBJECT)

Prints the last computation that led to OBJECT.

SHOWR (OBJECT)

Prints all computations that led to OBJECT.

ATOMIC-TYPE (SEQ &OPTIONAL TYPE)

Returns the type of the elements of SEQ, if TYPE is not provided. Otherwise, it acts as a predicate, i.e. tests whether the type of SEQ is TYPE.

STRUCTURE (SEQ)

Returns the structure of the sequence. For a numeric sequence, the answer is that it is a sequence of atoms.

SEQ-TYPEP (SEQ &OPTIONAL TYPE)

Recursive checking for TYPE. Without a second argument, it returns the type of SEQ.

SEQ-GET (SEQ INDICATOR)

The property list GET operation for sequences.

KBSP-APROPOS (STRING)

Find any system whose name includes the given string. Same as the APROPOS function of the Lisp Machine, but much faster because it confines search only to the KBSP-defined systems.

The following functions give the user some control over the buffering and unbuffering of sequences. The reader may want to read their deacription after reading section 4.4 on array memory management. It should be mentioned here that each system remembers the sequence instances that were created by its application.

SEQ-FLUSH (&REST SEQ-LIST)

Flush the sequences in the argument list from their system's memory and flush their buffers, that hold their numerical values.

SEQ-UNBUFFER (&REST SEQ-LIST)

Unchaches a list of sequences.

SYS-FLUSH (&QUOTE SYSTEM)

Remove all sequences from SYSTEM's memory and uncache them.

3.3 SIMPLE SEQUENCE OPERATIONS

Operations on a single sequence.

SEQ-SHIFT (SEQUENCE SHIFT) system

Return the sequence obtained by shifting SEQUENCE to the left by SHIFT so that index SHIFT of the input sequence corresponds to index 0 of the output sequence, i.e. if the input sequence is x[n], the output sequence is x[n-SHIFT].

SEQ-SCALE (SEQUENCE SCALE &OPTIONAL REAL-OFFSET IMAG-OFFSET) system

Return the sequence obtained by scaling SEQUENCE by a scale factor SCALE. Subtract the offset first if it is given. The default offsets are zero.

SEQ-RECIPROCAL (SEQUENCE) system alias

Return the sequence obtained by taking the point by point reciprocal of SEQUENCE. It calls SEQ-REAL-RECIPROCAL or SEQ-COMPLEX-RECIPROCAL according to the type of the input sequence.

SEQ-NEGATE (SEQUENCE) system

Return the sequence obtained by taking the point by point negation of SEQUENCE.

SEQ-CONJUGATE (SEQUENCE) system

Return the sequence obtained by taking the point by point complex conjugate of a sequence.

Point-by-point operations on more than one sequence.

SEQ-ADD (&REST SEQUENCES) system

Return the point by point sum of the arguments.

SEQ-SUBTRACT (&REST SEQUENCES) system

Return the sequence obtained by subtracting all sequences except the first one from the first sequence.

SEQ-MULTIPLY (&REST SEQUENCES) system alias

Return the point by point product of a set of sequences. This is a generic operation, i.e. it accepts both real and complex sequences, by invoking SEQ-REAL-MULTIPLY or SEQ-COMPLEX-MULTIPLY respectively.

SEQ-* (&REST SEQUENCES) system-alias

An alias for SEQ-MULTIPLY.

SEQ-DIVIDE (&REST SEQUENCES) system alias

Return the sequence obtained by dividing a set of sequences. Output is first sequence divided by the rest. This is a generic operation and calls SEQ-REAL-DIVIDE or SEQ-COMPLEX-DIVIDE.

Generate a sequence over a domain.

SEQ-CONSTANT (REAL-VALUE &OPTIONAL IMAG-VALUE) system

Return a constant sequence over some domain. Default of IMAG-VALUE is 0.

SEQ-FUNCTION (FUNCTION DOMAIN &REST OTHER-ARGS) system

Returns the sequence computed from the function FUNCTION over DOMAIN. The argument FUNCTION must evaluate to a lisp function spec (if the name of the function is used, it must be quoted). OTHER-ARGS are passed to the FUNCTION as they are, so they can serve as parameters of the output sequence. Example: (SEQ-FUNCTION 'FOO [-10. +10.] 3.), with FOO being (DEFUN FOO (X A) (* X A)), computes the values of function 3x.

SEQ-COMPLEX-FUNCTION (FUNCTION DOMAIN &REST OTHER-ARGS) system

Returns the complex sequence computed from the function FUNCTION. FUNCTION must be a complex-valued function of a single real argument. OTHER-ARGS are passed to FUNCTION.

SEQ-FROM-ARRAY (ARRAY) system

Returns a sequence from an array. The domain of the returned sequence starts at 0 and has length equal to the array length.

Application of a given function to each point of a sequence.

SEQ-APPLY (FUNCTION SEQUENCE &REST OTHER-ARGS) system alias

Return the sequence whose values are obtained by applying a function to each point of a sequence. If SEQUENCE is real, then SEQ-REAL-APPLY is used and FUNCTION should take one argument. Otherwise, SEQ-COMPLEX-APPLY is used. In the latter case, FUNCTION should take two arguments, the real part and the imaginary part. As an example,

```
SEQ-APPLY ('+  SEQ  15)
```

returns a sequence obtained from SEQ by adding 15 to each one of its points.

SEQ-MAP (FUNCTION &REST SEQUENCES) system

Returns the sequence whose values are obtained by applying FUNCTION to each point of the N-dimensional sequence obtained as the Cartesian product of the N SEQUENCES. FUNCTION should take N arguments. As an example,

```
SEQ-MAP ('+ SEQ1 SEQ2 SEQ3)
```

returns the sum of the three sequences.

Operations on complex sequences.

SEQ-REAL-PART (SEQUENCE) system

Returns the real part of a sequence.

SEQ-IMAG-PART (SEQUENCE) system

Returns the imaginary part of a sequence.

SEQ-COMPLEX (REAL-PART-SEQ IMAG-PART-SEQ) system

Returns a complex sequence built from two real sequences.

SEQ-POLAR (SEQUENCE) system alias

Returns the polar version of SEQUENCE. The "real part" of the output sequence is the magnitude of SEQUENCE and the "imaginary part" is the phase.

SEQ-RECTANGULAR (SEQUENCE) system alias

Returns the sequence obtained if SEQUENCE is converted from polar to rectangular.

SEQ-MAG (SEQUENCE) system

Returns the point by point magnitude of a sequence.

SEQ-MAG-SQUARE (SEQUENCE) system

Returns the point by point magnitude squared of a sequence.

SEQ-PHASE (SEQUENCE) system

Returns the point by point phase of a sequence.

SEQ-LOG-POLAR (SEQUENCE) system alias

Returns the sequence obtained by converting SEQUENCE to polar but give mag in dbs. "Real part" is 20log(mag) and "imaginary part" is phase.

SEQ-LOG-MAG (SEQUENCE) system

Return the point by point log magnitude of SEQUENCE in db.

LOG-MAG (SEQUENCE) system alias

Return the point by point log magnitude of SEQUENCE over the first half of its domain. This system is especially suited to plotting the log-magnitude of the Fourier transform of a real sequence, which is an even function of frequency, and thus only the positive half needs to be plotted.

Useful variables:

CLIP-OFFSET-IN-DBS variable

Maximum range in db. Default value is 200.

CLIP-OFFSET variable

Maximum range. Default value is (LOG (EXPT 10 (// *CLIP-OFFSET-IN-DBS* 20))).

Utility functions:

LOG-10 (X) function

Log to the base 10 of X.

COMPLEX-MULTIPLY (REAL-Z-1 IMAG-Z-1 REAL-Z-2 IMAG-Z-2) function

Multiply two complex numbers z-1 and z-2.

COMPLEX-DIVIDE (REAL-Z-1 IMAG-Z-1 REAL-Z-2 IMAG-Z-2) function

Divide z-1 by z-2.

COMPLEX-RECIPROCAL (REAL-Z IMAG-Z) function

Take the reciprocal of z.

PHASE (Y X) function

Returns the angle, in radians, whose tangent is y/x. The returned value is always a number between $-\pi$ and π. If x=y=0, the returned value is 0. This function is a smart version of ATAN2 of Zeta Lisp.

3.4 CONVOLUTION AND RELATED OPERATIONS

This section describes systems and functions useful in performing convolution of sequences and related operations. From the user's viewpoint, a system alias is no different from a system. From the implementation viewpoint, a system alias is defined by composing systems as mappings, whereas a system is defined by explicitly operating on the concrete repreaentation of the input and output sequence. A system alias is usually less space efficient than a corresponding system for the same operation, because a system alias by default buffers all intermediate sequences. For more explanation of the differences between a system and a system alias, the reader is referred to section 4.3.

SEQ-REVERSE (SEQUENCE) system alias

Time reverse a sequence. If the input sequence is x[n], the output sequence is x[-n].

SEQ-FFT-CONVOLVE (X H) system alias

Perform linear convolution of two sequences X and H using the FFT.

OVCONV (SEQA SEQB) system

Overlap-add convolution of SEQ by FLTR. FLTR should be substantially shorter compared to SEQ. The system decides which one of SEQA and SEQB plays the role of FLTR based on their length.

SEQ-CONVOLVE (X H) system alias

Convolve sequences X and H. This is a generic operation (i.e. deals with both real and complex sequences) built upon SEQ-REAL-CONVOLVE and SEQUENCE-COMPLEX-CONVOLVE.

SEQ-CORRELATE (X H) system alias

Find the correlation between sequences X and H, by convolving X with the time-reversed version of H.

SEQ-AUTOCOR (X) system alias

Find the autocorrelation of sequence X, by convolving X with itself.

SEQ-ENERGY (SEQ &OPTIONAL WINDOW OFFSET-BETWEEN-SAMPLES) system

The short-time energy in a sequence. If the input sequence has length L, the output sequence has length L/OFFSET-BETWEEN-SAMPLES. To compute the output sequence, the input sequence is split into (possibly overlapping) blocks of length the same as the length of

the WINDOW, each block is individually windowed, the sum of the squares of its samples is computed and the result becomes a single sample of the output sequence. The default values for the optional parameters are: WINDOW is a hamming window of length 256 and OFFSET-BETWEEN-SAMPLES is 100.

Related utility functions:

INTERVAL-REVERSE (INTERVAL) function

Get the time reversed interval. If the input interval is [a, b], then the output interval is [-b+1, -a+1]. Remember that the convention about intervals is that the first point is included int the interval, but the last point is not.

CONVOLUTION-SIZE (X H) function

Determine the appropriate FFT length to use for convolution, equal to the sum of the length of the sequences X and H minus 1.

CONVOLUTION-DOMAIN (X H) function

Determine the domain over which the convolution of sequences X and H will be non-zero.

3.5 DISCRETE FOURIER TRANSFORM COMPUTATIONS

A number of systems and functions is provided for DFT and FFT computations. Their main characteristics are the following:

Sine and cosine tables are used (they are implemented with the "resource" mechanism of ZetaLisp). This ensures efficiency in time (sines and cosines are computed once when needed and then looked up, if they are needed again), and in Lisp Machine storage.

The FFT and DFT operations are "generic". This means that they accomodate real and complex sequences and they branch using the specialized subordinate functions as needed.

The top-level systems/functions that a user would normally use are the following:

FFT (SEQ &OPTIONAL LENGTH) system

Return the (complex) FFT of a sequence. The input sequence can be real or complex. The output sequence is always complex. The default value of LENGTH is the smallest power of 2 which is greater than or equal to the length of the sequence. In case the length of SEQ is longer than LENGTH, no truncation takes place, but instead "aliasing" occurs, i.e. all the elements of SEQ are taken into account, while the sequence of exponential coefficients repeats itself periodically. This implies that taking the inverse fft of the fft of a sequence may not return the original sequence or a portion of it.

IFFT (SEQUENCE &OPTIONAL LENGTH) system

Return the complex inverse FFT of a sequence. If the input sequence is real, it is treated as complex with zero imaginary part. The default value of LENGTH is the smallest power of 2 which is greater than or equal to the length of the sequence.

IFFT-REAL (SEQUENCE &OPTIONAL LENGTH) system

Return the inverse fft in the form of a real sequence. SEQUENCE should be a complex sequence corresponding to the fft of a real sequence.

DFT (SEQUENCE &OPTIONAL LENGTH) system

Return the DFT of a sequence. The default LENGTH is equal to the length of the sequence. Notice that it doesn't have to be a power of 2.

IDFT (SEQUENCE &OPTIONAL LENGTH) system

Return the inverse DFT of a sequence. The default LENGTH is equal to the length of the sequence. Notice that it doesn't have to be a power of 2.

Utility functions/systems:

SEQ-COS-SINGLE (PERIOD) system

Returns a cosine with the specified period. The compute-domain of the cosine is equal to a single cycle.

SEQ-SIN-SINGLE (PERIOD) system

Returns a sine with the specified period. The compute-domain of the sine is equal to a single cycle.

SEQ-COMPLEX-EXP-SINGLE (PERIOD) system alias

Returns a complex exponential with the specified period. The compute-domain of the signal is equal to a single cycle.

SEQ-ROTATE (SEQUENCE AMOUNT) system alias

Rotate a sequence to the left by specified amount. Equivalently, shift the sequence to the left as if it were periodic with period equal to its domain and then grab one period. The output sequence has the same domain as that of SEQUENCE.

POWER-OF-2-P (NUM) function

Test if NUM is a power of 2

NEXT-POWER-OF-2 (NUM) function

Returns the next power of 2 >= NUM.

SEQ-ALIAS (SEQUENCE DOMAIN &OPTIONAL REPETITION-LENGTH) system

Return the aliased version of a sequence into a specified domain. (This function needs fixing to check for invalid inputs. It is not likely to be useful to a ordinary user of the KBSP sys-

tem).

3.6 FILE INPUT/OUTPUT

The following facility has been designed to read from a variety of data file formats that have existed (and still exist) on the Digital Signal Processing Group computers.

The old dat format consists of a UNIX binary file with an ASCII header, which is a block of 512 bytes with type and size information, as follows:

s02 (for 2-byte integers or r04 for 4-byte reals or c08 for 8-byte complex)
dim
12456 (or whatever the number of samples in the file is).

The new DSPG dat format uses a directory containing two files: one file contains the data in binary form and the other is the descriptor file, containing type, size and miscallaneous information in ASCII form (so that it can be viewed easily on the terminal). The 3600 dat format is very similar.

FILE (FILENAME) system

This system creates and returns a sequence from a file. The file must be one of the following kinds: DSPG old dat format, DSPG new dat format, 3600 dat format or ASCII. FILENAME must be a host/path specification conforming with the Lisp Machine's conventions for pathnames for different operating systems. See [Weinreb] (section on Naming of Files).

SEQ-DUMP-TO-FILE (SEQ PATHNAME) function

Dump SEQ into a file specified by PATHNAME. Only works for real or complex sequences. The format of the resulting file is the 3600 dat file format.

3.7 FILTERING

Functions/systems supporting filtering of signals are provided. Note that the filter coefficients must be provided by the user (i.e. no filter design is provided). Filter structures currently supported are FIR and IIR filters. More general filters can be created by using the SYS-ALIAS fa-

cility and the provided systems as building blocks.

FIR (&REST COEFFICIENTS) system

Return a sequence whose value is the impulse response of an FIR filter. For example, if COEFFICIENTS are 1 and 0.5, the output of FIR is a sequence with value 1 at time 0 and 0.5 at time 1.

IIR (&REST COEFFICIENTS) system

Return a sequence whose value is the impulse response of an IIR filter. For example: (IIR .5) has an impulse response of 1.0, -.5, .25, -.125, ... In general, (IIR c1 c2 ... cn) implements the impulse response of the filter

$$\frac{1}{1+c_1z^{-1}+c_2z^{-2}+\cdots c_nz^{-n}}$$

FIR-FILTER (INPUT FIR-SEQUENCE) system alias

Return the output of the specified FIR filter if INPUT is the input sequence to the filter. The specified filter must be a sequence constructed using FIR.

IIR-FILTER (INPUT IIR-FILTER &OPTIONAL GAIN INITIAL-STATE-ARRAY) system

Return the output of the specified IIR filter (scaled by GAIN) if INPUT is the input sequence to the filter. The default GAIN is 1. The INITIAL-STATE-ARRAY, if provided, becomes the initial condition for the IIR filter. Otherwise, the initial conditions are zero.

STATE-SHIFT (ARRAY NEW-FIRST-ELEMENT) function

Shift the array elements to right by one and put in a new first element in the first position. This function works by side-effect, i.e. it actually changes its argument ARRAY, and should be used with caution, if at all.

IIR-FILTER-FROM-ARRAY
(INPUT ARRAY &OPTIONAL GAIN INITIAL-STATE-ARRAY) system alias

Filter an input sequence with coefficients from the given array. In order to implement the IIR filter described in the documentation of IIR, the array must be [c1, c2, ...,cn].

3.8 FUNCTIONS OPERATING ON SEQUENCES

MEAN-OF-SEQ (SEQUENCE &OPTIONAL INTERVAL)

Return the mean value of a sequence over some interval. The default value of INTERVAL is the domain of the sequence. The function returns two values, the mean of the real and the imaginary part of the sequence, using the Zeta Lisp multiple value mechanism.

SUM-OF-SEQ (SEQUENCE &OPTIONAL INTERVAL)

Return the sum of the real and imaginary values of a sequence over some interval.

MEAN-SQUARE-OF-SEQ (SEQUENCE &OPTIONAL INTERVAL)

Return the mean magnitude squared value of a sequence over some interval, i.e. the sum of the squares of the values over the interval divided by the length of the interval. The default value of INTERVAL is the domain of SEQUENCE.

VARIANCE-OF-SEQ (SEQUENCE &OPTIONAL INTERVAL)

Return the variance $(E(|x|^2) - |E(x)|^2)$ of a sequence over some interval. The default value of INTERVAL is the domain of the sequence.

INNER-PRODUCT (X Y)

Return the inner product of the vectors X and Y, represented by arrays.

3.9 NORMALIZING A SEQUENCE

Several systems are provided for normalizing a given sequence in different ways. All of these systems return a normalized version of the input sequence.

SEQ-NORMALIZE (SEQUENCE) system

Return the normalized version of SEQUENCE with zero mean and unit variance.

PREEMPHASIZE (SEQ &OPTIONAL PREEMPHASIS-FILTER) system alias

Return the preemphasized version of SEQ. The default preemphasis-filter is the one specified by variable *DEFAULT-PREEMPHASIS-FILTER*.

DEEMPHASIZE (SEQ &OPTIONAL DEEMPHASIS-FILTER) system alias

Return the deemphasized version of SEQ. The default deemphasis-filter is the one specified by variable *DEFAULT-DEEMPHASIS-FILTER*.

SEQ-UNIT-ENERGY (SEQUENCE) system alias

Return the normalized version of SEQUENCE with sum of squares equal to 1.

SEQ-UNIT-AREA (SEQUENCE) system alias

Return the normalized version of SEQUENCE with unit area, i.e. the sum of values being equal to 1.

Useful variables:

DEFAULT-PREEMPHASIS-FILTER variable

Default preemphasis filter. Its default value is (FIR 1.0 -.95). If a different default preemphasis filter is desired, this variable must be set to the desired filter obtained with FIR.

DEFAULT-DEEMPHASIS-FILTER variable

Default deemphasis filter. Its default value is (IIR -.95).

3.10 WINDOW OPERATIONS

Windowing operations on sequences:

SEQ-WINDOW (SEQ WINDOW OFFSET) system alias

Grab portion of sequence overlapping with WINDOW when the zero index of the WINDOW is aligned with index OFFSET of SEQ and window the portion using the specified window. The domain of the resulting sequence is identical to the domain of the WINDOW.

SEQ-GATE (SEQUENCE DOMAIN) system

Grab a portion of a sequence. The domain of the resulting sequence is DOMAIN.

SEQ-SECTION (SEQUENCE SECTION-INTERVAL) system alias

Grab the portion of SEQUENCE specified by SECTION-INTERVAL and shift it to start at the origin.

SECTION (SEQUENCE SECTION-INTERVAL) system alias

Alias for SEQ-SECTION.

Window generators:

RECTANGULAR (LENGTH &OPTIONAL CENTERED) system

A Rectangular window of the specified length. If CENTERED is T (the default), then the window is centered around 0.

HAMMING (LENGTH &OPTIONAL CENTERED) system

A Hamming window of the specified length. If CENTERED is T, (the default), the Hamming window is centered around zero.

IMPULSE () system alias

An impulse at 0.

UNIT-STEP () system

A step.

4. THE KBSP IMPLEMENTATION.

4.1 INTRODUCTION

The KBSP package introduced an extension to the Lisp language in order to represent systems and signals.

Systems are generalized Lisp functions with arguments almost anything, but most typically signals or parameters of signals, and outputs other signals.

Signals are represented by "sequences". Sequences are abstract data types (flavors) including various pieces of information about the signal they represent, for example its name, where it came from, how to compute its values and its domain.

The section on sequences as abstract data types explains the implementation of sequences as flavor types. The components of a sequence are explained and a summary explanation of the main methods that all sequences are equipped with is given. It will be helpful to the reader if he is familiar with the basic philosophy of flavors as described in the corresponding chapter of the Lisp machine manual.

The section on system implementation explains the facilities that are provided so that the reader can define his own systems. Many examples are provided and explained in detail in order for the reader to be able to read the definitions of the current systems and write his own. Some familiarity with Lisp macros and flavors will be helpful.

The section on array memory management explains the techniques that enable KBSP to make efficient use of the Lisp machine memory. A list of forms that allow the user to have control over the amount of array memory his program consumes is given. Finally, an example of changing the definition of a system to make it more memory efficient without sacrificing the clarity of code is given. Some familiarity with the concepts of memory management in Lisp, especially garbage collection, and with the concept of hashing will be helpful.

4.2 SEQUENCES AS ABSTRACT DATA TYPES

The KBSP package provides facilities for the definition of sequences of arbitrary objects, including sequences of sequences. The ability to *mix flavors* (see [Weinreb]) is used to implement abstract data types in a modular fashion. KBSP defines several flavor types for sequences, which can be combined in different ways to implement types such as numeric sequences and sequences of sequences. BASIC-SEQUENCE implements a sequence with a domain and a buffer for holding its values. These values can be any Lisp objects. SEQUENCE is built on top of BASIC-SEQUENCE and adds to it the ability to plot itself and the ability to have its own property list. BASIC-NUMERIC-SEQUENCE is built on top of BASIC-SEQUENCE and includes extra slots to accomodate complex sequences. NUMERIC-SEQUENCE is built on top of BASIC-NUMERIC-SEQUENCE and SEQUENCE and includes operations appropriate to numerical sequences (e.g. range).

The basic flavor out of which sequences are built is the BASIC-SEQUENCE flavor. Its *instance variables* (i.e. the components of the corresponding abstract data type) are the following:

BUFFER-DOMAIN: This is the interval over which the sequence has previously been computed. The corresponding values are stored in BUFFER.

BUFFER: This is normally an array containing the values of a part of the sequence (the part that has previously been computed).

DOMAIN: The interval over which the sequence is nonzero (or non-constant).

PERIOD: The period of the sequence. It is equal to INF for a non-periodic sequence.

CACHED?: When this is T, the computed values of the sequence are cached, i.e. saved in BUFFER so that they will not be recomputed when needed again in the future.

The SEQUENCE flavor is built from BASIC-SEQUENCE, OBJECT-PLOT-MIXIN (a mixin that enables a sequence to plot itself) and SI:PROPERTY-LIST-MIXIN (a mixin flavor that implements the basic property list operations for the individual sequence instances). The reader is re-

ferred to chapter 2 for an explanation of the concept of building new abstract data types by combining existing ones. In the Zeta Lisp terminology this is called *mixing of flavors* and it is fully explained in [Weinreb].

The BASIC-NUMERIC-SEQUENCE flavor is built from BASIC-SEQUENCE. It has additional slots IMAGINARY-BUFFER-DOMAIN and IMAGINARY-BUFFER, to accommodate complex-valued sequences.

The NUMERIC-SEQUENCE simply consists of BASIC-NUMERIC-SEQUENCE and SEQUENCE.

Sequences can be examined using the Lisp "describe" facility. The Lisp form (DESCRIBE OBJ), where OBJ is the name of an object (or a sequence), prints a list of all instance variables of OBJ together with their values. The reader may find it helpful to apply this facility at various points of the example session and verify for himself the caching and uncaching of sequences. SEQ-NAME and SHOWR are other ways to examine sequences.

The above flavors understand a wide variety of messages via methods that are defined for them: predicates for testing values, "fetch" messages that return portions of the data or domains, "set" messages that change the values. When a system is defined, a new flavor type is defined on top of a sequence flavor type (using the mixin facility) with new methods that are specialized for the system. The user has the ability to define many methods, but DOMAIN and FETCH methods are the minimal methods that must be provided so that the new flavor is compatible with the rest of the KBSP package. For example, many KBSP functions and systems expect an object to be able to fetch its values. A sequence is able to fetch its values only if some form of FETCH method has been defined for it.

As explained in the next section, DEFINE-SYS and SYS-ALIAS are extensions to Lisp that allow convenient definition of systems with their associated methods. DEFINE-SYS provides a convenient syntax for defining a new sequence type by explicitly providing its basic methods. SYS-ALIAS allows definition of a new sequence by abstractly combining existing systems, namely

without explicit definition of the new sequence's methods, and by using a Lisp-like syntax. SYS-ALIAS is easier to write than a DEFINE-SYS, but it can often be more inefficient in terms of speed and memory consumption.

The following is a list of most methods associated with the sequence flavors. They are provided here for reference purposes only. In terms of the role methods play in the KBSP package, there are three classes of methods:

Methods that depend on the particular sequence being defined, such as FETCH, COMPUTE and DOMAIN. These must be defined separately for each new sequence type. It is possible for COMPUTE (which works over an interval) to rely on FETCH or vice versa. The preferred implementation depends on efficiency considerations and is subject to change. The current conventions are described in the next section.

Methods that are sequence dependent, but are not necessary for the proper use of the sequence, such as PERIOD, STRUCTURE or ATOMIC-TYPE. Default method definitions provide reasonable answers, but not always correct.

Methods that perform some function which is universally useful and does not depend on the particular sequence, such as FIND-COMPUTATION-INTERVALS or FETCH-INTERVAL. These methods use the sequence-specific methods internally and hence they can have a universal definition, which should not be superseded. These methods could have been defined as top-level Lisp functions.

Methods for BASIC-SEQUENCE:

HAS-NO-BUFFERS ()

Predicate for testing whether buffer is NIL.

DECACHE ()

It frees the array used by BUFFER for reuse. Sets BUFFER to NIL and BUFFER-DOMAIN to the zero interval.

Methods for SEQUENCE:

DOMAIN ()

Returns the nonzero domain of the sequence.

COMPUTE (INTERVAL ARRAY)

Computes the values of the sequence over the INTERVAL and places them into ARRAY.

FETCH (INDEX)

Fetches the value of the sequence at INDEX.

FIND-COMPUTATION-INTERVALS
(OLD-BUFFER-DOMAIN DESIRED-BUFFER-DOMAIN)

Returns the intervals over which the values of the sequence must be actually computed to cover the DESIRED-BUFFER-DOMAIN, if the values over OLD-BUFFER-DOMAIN are already known.

FETCH-INTERVAL (INTERVAL &OPTIONAL OUTPUT-ARRAY NEW-CACHED?)

Returns an array containing the values of the sequence over INTERVAL. If OUTPUT-ARRAY is provided, the values are put into it.

PERIOD ()

Returns the period of the sequence.

MODIFY-CACHED? (NEW-VALUE)

Sets the value of instance variable CACHED? to the new value and returns the old value.

ATOMIC-TYPE ()

Returns the type of the elements of the sequence.

STRUCTURE ()

Returns the type of a sequence, for example whether it is a sequence of numeric sequences or a sequence of numbers.

COPY-BUFFER (NEW-BUFFER NEW-BUFFER-DOMAIN)

Copies the contents of instance variable BUFFER into the given NEW-BUFFER over the specified NEW-BUFFER-DOMAIN. Does a reasonable thing if NEW-BUFFER-DOMAIN and the instance variable BUFFER-DOMAIN have a nonempty intersection.

SET-BUFFER (NEW-BUFFER NEW-BUFFER-DOMAIN)

Swaps the contents of instance variables BUFFER and BUFFER-DOMAIN with those of NEW-BUFFER and NEW-BUFFER-DOMAIN respectively.

Methods for BASIC-NUMERIC-SEQUENCE:

HAS-NO-BUFFERS ()

Predicate for testing whether sequence has at least one buffer (real or imaginary)

DECACHE ()

It zeroes both the real and the imaginary part and returns the corresponding arrays to public use.

Methods for NUMERIC-SEQUENCE:

COPY-IMAGINARY-BUFFER (NEW-BUFFER NEW-BUFFER-DOMAIN)

Does the same thing as COPY-BUFFER of SEQUENCE but for the imaginary part of a numeric sequence.

COPY-COMPLEX-BUFFERS
(NEW-REAL-BUFFER NEW-IMAG-BUFFER NEW-BUFFER-DOMAIN)

Does the same thing as COPY-BUFFER of SEQUENCE but for both the real and the imaginary part of a numeric sequence.

SET-IMAGINARY-BUFFER (NEW-BUFFER NEW-BUFFER-DOMAIN)

SET-COMPLEX-BUFFERS
(NEW-REAL-BUFFER NEW-IMAG-BUFFER NEW-BUFFER-DOMAIN)

FETCH-IMAGINARY-INTERVAL (INTERVAL &OPTIONAL OUTPUT-ARRAY NEW-CACHED?)

FETCH-COMPLEX-INTERVAL (INTERVAL &OPTIONAL OUTPUT-ARRAY NEW-CACHED?)

RANGE (&OPTIONAL INTERVAL)

Returns the range of the sequence over the specified interval. If no interval is specified, a default domain is used. For complex sequences, the limits of a square that represents the range in the complex plane is returned.

IMAGINARY-RANGE (&OPTIONAL INTERVAL)

Returns the range of the imaginary part of the sequence over the specified interval. If no interval is specified, a default domain is used.

COMPUTE-IMAGINARY (INTERVAL ARRAY)

The default compute method for the imaginary part of numeric sequences.

4.3 SYSTEM IMPLEMENTATION

Systems in the KBSP package implement mathematical systems, i.e. entities which generate signals based on some input signals and/or parameters. Systems present a Lisp function interface to the top-level user. They accept the *lambda list* keywords that Lisp functions accept (see [Weinreb] and they generate an output sequence in a side-effect-free manner (i.e. the system arguments are not modified as a result of calling the system).

At the implementation level, a basic feature of systems is *lazy or delayed evaluation*: The effect of calling a system is not computation of values of the output sequence, but just creation of the necessary machinery for doing so, namely generation of Lisp code that can produce the numeric values of the output sequence.

Another basic feature is that a new flavor type is generated for each system. The new flavor type is basically a numeric sequence with extra methods that are particular to the corresponding system like the *compute or domain methods*, which implement the computation of values in a system dependent manner. Calling a system generates Lisp code that defines this new flavor and creates an instance of the flavor type which implements the output sequence according to the arguments of the system call. All sequences generated as a result of the call of a system, i.e. all sequences that are instances of the flavor type associated with a system, are "remembered" by the system in a hashing table in the system's property list. Thus repeating the call (HAMMING 32) twice does not create two objects, but one. Also an unnamed sequence is not inaccessible as an unnamed general object. A property assigned to a sequence using the special PUTPROP version for sequences, called SEQ-PUTPROP, is not attached to the name of the object, but to the actual object itself. The buffers of the output sequence that hold its numerical values are generally empty immediately after its creation. They get filled with some of the sequence's values as a result of a fetch-interval or plot request.

(a) DEFINE-SYS

DEFINE-SYS is a macro provided for the definition of new systems. In Lisp, the macro facility is an orderly way of generating lisp code. DEFINE-SYS provides a mechanism for defining new systems in a concise manner. A DEFINE-SYS expands to full lisp code performing a variety of tasks related to the generation and maintenance of the output sequence.

Let us first examine what the arguments of DEFINE-SYS are and how they expand into lisp code in the context of an example:

DEFINE-SYS (SYSTEM-NAME PARAMETERS FLAVOR-TYPE-LIST FORMS)

The example that will be analyzed here is the definition of a system that generates a Hamming window of specified length. The reader should attempt to understand this definition while reading the explanation that follows.

```
(DEFINE-SYS HAMMING (LENGTH &OPTIONAL (CENTERED T))
    (NUMERIC-SEQUENCE)
 "A Hamming window"
 (COMPUTE (INTERVAL OUTPUT-ARRAY)
      (LOOP FOR SAMPLE-INDEX FROM ($START INTERVAL) BELOW ($END INTERVAL)
          FOR ARRAY-INDEX FROM 0
          WITH OFFSET = (IF CENTERED 0 (// LENGTH 2))
          DO (SETF (AREF OUTPUT-ARRAY ARRAY-INDEX)
               (+ .54 (* .46 (COS (// (* 2.0 PI
                    (- SAMPLE-INDEX OFFSET))
                       (1- LENGTH))))))))
 (DOMAIN ()
      (IF CENTERED (INTERVAL (MINUS (// LENGTH 2))
                       (+ LENGTH (MINUS (// LENGTH 2))))
          (INTERVAL 0 LENGTH))))
```

SYSTEM-NAME is the name of the system. In the context of the example, this corresponds to HAMMING.

PARAMETERS are the arguments to the system, typically signal parameters, if the system generates a signal from its parameters, or one or more sequences, if the system operates on sequences (like adding or multiplying two sequences). In the Hamming example, the arguments are two: LENGTH, which is the length of the window, and an optional argument (note here that the Zeta Lisp function keywords apply to DEFINE-SYS as well) CENTERED, which determines whether the window will be centered around 0 or whether it will start at 0. The default value of

CENTERED is T, i.e. the window is centered around 0.

The output sequence of a system is a flavor of a type particular to the system. The output flavor type is obtained by a flavor definition, which is one of the things included in the expansion of DEFINE-SYS. FLAVOR-TYPE-LIST is a list of flavor types to be combined into the output flavor. In the Hamming example, there is only one flavor in the flavor-type-list, NUMERIC-SEQUENCE. This is the flavor type of the window which the system generates.

FORMS is a list of forms which expand into method definitions for the output flavor. These methods are particular to the defined system and typically specify how the output values should be computed, and how the domain of the output sequence is found. These forms are required for the sequence to be any useful. Compute forms may compute over an interval (COMPUTE) or at a single index (FETCH). COMPUTE forms are faster, because they operate on blocks of data, but they require the user to write a loop of array operations. FETCH forms are much slower, because they perform a function call per index, but are easier to write.

In the current implementation, the COMPUTE form is required, while the FETCH message is reduced to a COMPUTE message over an interval of length one, unless it has been defined otherwise. This may change in future implementations.

The COMPUTE form by convention always takes two arguments, INTERVAL and OUTPUT-ARRAY. INTERVAL is the interval over which the values are to be computed and OUTPUT-ARRAY receives the computed values. LOOP is a Zeta Lisp iteration construct similar to a DO or a FOR loop, but more general and powerful [Weinreb]. The compute form expands into a COMPUTE method definition for the output flavor (refer to the section on sequences for more explanation). The compute message will be sent to the output flavor when its values are requested, typically by PLOT or FETCH-INTERVAL. The senders of the message will provide values for the two arguments of the method. In the example, the compute form computes the values over the interval by applying the mathematical formula for the Hamming window. The domain form always takes no arguments and expands into a DOMAIN method definition. In the

example, the domain computation depends on whether the window is centered or not.

A form called INIT-FORMS with no arguments is *the init form*. The init form expands into an after-init method definition. This means that the corresponding message is automatically sent immediately after the creation of a flavor instance and performs initialization tasks. INIT-FORMS are not required and in the example of Hamming window they have not been used.

In summary, the required compute forms are:

COMPUTE, if the output sequence is real,

COMPUTE and COMPUTE-IMAGINARY or
COMPUTE-COMPLEX, if the output sequence is complex.

DOMAIN, in all cases.

Some of the optional ones with default definitions (refer to the section on sequences for these) are:

RANGE (&OPTIONAL INTERVAL)
IMAGINARY-RANGE (&OPTIONAL INTERVAL)
ATOMIC-TYPE ()
PLOT-RANGE ()
FETCH (INDEX)
FETCH-IMAGINARY (INDEX)
FETCH-COMPLEX (INDEX)
PERIOD ()
INIT-FORMS () (default is nil)

As mentioned before, the format of the *compute* and *domain forms* is not arbitrary but depends on the conventions that other parts of the KBSP system follow. For example, the compute form takes two arguments because this is how the PLOT or FETCH-INTERVAL function will send the corresponding message.

Another simple example is the definition of SEQ-SHIFT:

```
(DEFINE-SYS SEQ-SHIFT (SEQUENCE SHIFT)
      (NUMERIC-SEQUENCE)
 "Shift the input sequence to the left by SHIFT so that index SHIFT
of the input sequence corresponds to index 0 of the output sequence"
 (COMPUTE (INTERVAL OUTPUT-ARRAY)
   (FETCH-INTERVAL SEQUENCE
      (INTERVAL-DELAY INTERVAL SHIFT) OUTPUT-ARRAY CACHED?))
 (COMPUTE-IMAGINARY (INTERVAL OUTPUT-ARRAY)
   (FETCH-IMAGINARY-INTERVAL SEQUENCE
      (INTERVAL-DELAY INTERVAL SHIFT) OUTPUT-ARRAY CACHED?))
```

The second major way of defining new systems is using SYS-ALIAS. This is another macro that allows definition of a new system by combining existing systems in an abstract way, i.e. without the need to access the internal representation of the input arguments.

SYS-ALIAS (SYSTEM-NAME PARAMETERS BODY)

BODY must be of the form (EQUIVALENT-EXPRESSION . METHODS). Equivalent-expression is a lisp expression built on existing systems and/or Lisp. A simple example of the use of SYS-ALIAS is the definition of SEQ-SECTION:

```
(SYS-ALIAS SEQ-SECTION (SEQUENCE SECTION-INTERVAL)
  (SEQ-SHIFT
    (SEQ-GATE SEQUENCE SECTION-INTERVAL) ($START SECTION-INTERVAL)))
```

In this definition, METHODS is equal to nil, i.e. no extra methods have been defined for its output sequence on top of the default ones for its type. doesn't exist. The equivalent expression grabs a portion of the sequence corresponding to the section interval and shifts it so that it starts at the origin. Notice that there was no need to think about the values of the input sequence in this definition. The input sequence was treated as an abstract entity and its internal representation was not accessed at all. The definition (ignoring the Lisp syntax) reflects the abstract way of thinking about this operation. SYS-ALIAS is the simplest way of writing new systems. It uses only the abstract notion of a sequence, in contrast with DEFINE-SYS, which in general needs to access the internal sequence representation through array operations. Most systems that the user might ever need can be written (if inefficiently) as system aliases. In many cases, however, system aliases are very inefficient in their usage of space (and time), and it is necessary to rewrite the system alias definition using DEFINE-SYS. Consider another example of SYS-ALIAS, the system that computes the periodogram of a sequence:

```
(SYS-ALIAS PERIODOGRAM (SEQ BLOCK-SIZE STARTING-POINT BLOCK-OFFSET
                        NUMBER-OF-BLOCKS &OPTIONAL (FFT-LENGTH 2048))
 "Get the periodogram of a sequence. Return its log magnitude over
 the positive frequencies  only. Cached version - all intermediate
 seqs are saved"
 (LOOP FOR INDEX FROM 0 TO (- NUMBER-OF-BLOCKS 1)
   COLLECT  (SEQ-MAG-SQUARE
     (FFT
       (SEQ-WINDOW SEQ (HAMMING BLOCK-SIZE)
```

```
        (+ STARTING-POINT (* BLOCK-OFFSET INDEX)))
                (NEXT-POWER-OF-2 FFT-LENGTH)))
INTO SEQ-LIST
FINALLY (RETURN (LOG-MAG (APPLY 'SEQ-ADD SEQ-LIST)
        (INTERVAL 0 (1+ (// (NEXT-POWER-OF-2 FFT-LENGTH) 2)))))))
```

In this definition, each intermediate FFT result, each windowed result and each magnitude squared result is cached. If a large number of periodograms, each with a large number of blocks is taken, then considerable memory may be unnecessarily consumed. In this case it is possible to perform the same computations but uncached. This could mean that after each periodogram, all intermediate results are flushed, so that the arrays they occupy are returned to the pool of free arrays for reuse by the rest periodogram computations. One way to achieve this by systematically converting a SYS-ALIAS into a DEFINE-SYS is shown in the section on array memory management.

4.4 ARRAY MEMORY MANAGEMENT

Signal processing consumes lots of memory space very quickly if the memory space is not reused, especially when the processing involves long waveforms. In conventional computer environments, array storage is allocated by means of array declarations. If the array declarations require more than the available computer (core) memory, then a compiler error is signalled. In this case, the user must take care of the storage problem manually, meaning that he must configure his programs in such a way that only a small part of the data resides in core memory at any given instant, while the rest is on a mass storage medium, and also in such a way that arrays are reusable. In more advanced systems which have virtual memory management, array declarations that exceed the available core memory are not errors because the operating system takes care so that only a small part of the data is in core. If a request for a piece of data not in core is issued, the operating system knows how to find it on mass storage and swaps it with another piece in core. Using a virtual memory system is not without a price: programs that do not access data sequentially will cause the computer to spend most of his time swapping data in and out of core.

Storage in the Lisp language is based on an entirely different concept of memory management. Memory is not allocated explicitly, but implicitly by the Lisp system, and in a less structured way than in conventional computer environments: memory is allocated automatically when needed and it is freed when it is not accessible any more ("not accessible" means that there is no name or pointer for a chunk of memory, in which case there is no way to reach it). For example, performing a side-effect-free operation on a data structure returns a nameless modified copy of the data structure. If a name is assigned upon creation of the copy, then the name provides a way to access the modified version. If the user does not want the old version any more, he will give its name to the modified data structure. Then the storage the old version occupies is no longer accessible because it became unnamed and it will be garbage-collected sooner or later.

In fact, when one considers arrays on the Lisp machine, the scene becomes slightly more complicated. The Lisp machine memory has distinct regions of prespecified sizes for storing arrays

and lists. These two regions are garbage collected separately.

The KBSP system cannot rely on the Lisp garbage collection, because arrays are consumed and freed at a rate much higher than what the garbage collector could accomodate. Thus the need for some array management scheme was needed, which would provide a simple and easy-to-use mechanism for alleviating the memory management problem on one hand and avoiding repeating the same computation on the other.

The solution chosen is the following:

Sequences are going to be cached by default. This means that if a sequence is created and at some point its values over some interval are needed and computed, these values will be saved in a buffer (array) so that they are readily available if they are needed again in the future. Moreover, if the values over another interval with non-empty intersection with the first interval are needed, the KBSP system will compute only the unknown values. Buffering is contiguous, i.e. asking for the values over an interval which does not intersect with the interval already cached will cause computation and buffering of all values between the two intervals as well. This design decision may change in the future. Buffering solves the problem of repeating the same computation if the values are necessary. Since buffering happens by itself, the user does not need to explicitly save away computed values that he may need in the future.

However, uncontrolled buffering entails a danger, in cases when lots of intermediate sequences are generated. That this is not an exotic case can be seen by considering a periodogram computation. Assume that a single periodogram is to be computed, with FFT length equal to 2048 points (1 Hz resolution if the sampling rate is 2048 Hz) and with total number of blocks equal to 128 (a reasonable number if the signal-to-noise ratio is low but the signal fairly stationary). Assume that this computation is going to be repeated 10 times. With complete buffering, every intermediate FFT will be cached, as well as every magnitude-squared operation. This means that the total memory consumed will be of the order of 25 Megabytes.

In DEFINE-SYS, the concept of "local arrays" was applied: the COMPUTE forms provide array(s), whose scope is local to the form and they are used for holding and/or returning results. These arrays are distinct from the arrays which serve as buffers of sequence values and are permanently attached to the sequences themselves. The programmer of a DEFINE-SYS has the choice to perform part of the computation abstractly, i.e. by combining systems, or concretely, i.e. by fetching the values of the sequences into local arrays and then operating on the arrays. In contrast, a SYS-ALIAS allows only abstract combinations of systems.

When systems are combined abstractly, memory usage may be excessive. One solution would be to open all sequences up, put their values into local arrays and operate on them. This would violate the abstraction principle and all the convenience, ease and safety of programming that it offers.

Another solution that is offered by the KBSP package is to to allowing the user to control the *uncaching* of buffer arrays. Freed buffer arrays do not go back to the Lisp region of memory for arrays but they become permanent property of the KBSP system. The KBSP system saves all freed arrays in a hashing table, in which free arrays are keyed by size. When a new buffer array is requested, the KBSP system first looks at the hashing table to see if there are any free arrays of this size. If there are, it grabs one of them and uses it. If not, a new array of the required size is made, which means that this memory can never go back to the Lisp region for arrays again. If it becomes part of a cached sequence, it will store its data, if it is freed, it goes into the hashing table for reuse.

This storage discipline is fairly simple to implement and it can offer solution of the storage management problems in most cases. In the periodogram example, if the computations inside a single periodogram are cached, but after a periodogram is completed, all newly cached sequences are flushed, the total storage consumed will be only one tenth of what it was before, and it will have been reused nine times. After all 10 periodograms are completed, this memory will be part of the hashing table of free arrays for further reuse. Cases in which this scheme would be inadequate

might be: (a). if the intermediate cached computations chew up all the available space or (b). if the user allocates many large arrays of many odd sizes, in which case all array space will be partitioned into many odd sizes and when a new odd size is needed, it will not be possible to allocate it. Given that the array space is limited, but large, and given that in signal processing, the case of many odd sizes is unusual, both of the above cases tend to be rather rare.

FORMS FOR ARRAY MEMORY MANAGEMENT

A number of facilities has been provided to give the user control over the way his sequences are cached and uncached. User-level facilities consist of LET-like macro forms that have certain effects on the buffering properties of the code they enclose (see [Weinreb] for the definition of LET).

WITH-UNCACHING (BODY) macro

This form executes the body and before it terminates, it flushes all the sequences newly cached inside the body. Flushing a sequence does not mean destroying it but simply freeing the buffer containing its data. This of course means that if this data is needed again, it must be recomputed. A minor disadvantage of using the WITH-UNCACHING form is that the result must be passed out of it via an array. Passing the result out with a sequence is not possible, because all sequences are flushed before exiting the body. This implies that WITH-UNCACHING cannot be combined with a SYS-ALIAS definition. However, a SYS-ALIAS can be trivially converted to a DEFINE-SYS with uncached intermediate results as it is explained in section 4.3.

SHOW-SEQUENCE-FLUSH variable

If T, a message is printed every time a sequence is flushed. Defaults to NIL.

SHOW-FREE-USER-ARRAYS variable

If T, a message is printed every time an array is freed and added to the hash table of free arrays. Defaults to NIL.

FLUSH-NEWLY-CACHED-SEQUENCES () function

Flush all newly cached sequences in the current context. It flushes all sequences in the list *NEWLY-CACHED-SEQUENCES*. This function is used inside other macro forms, such

as WITH-UNCACHING, which take care so that *NEWLY-CACHED-SEQUENCES* contains the right sequences.

USER-ARRAYS variable

The list of user arrays.

ALLOC-USER-ARRAY (LENGTH) function

Allocate an array and record it on *USER-ARRAYS*.

DEALLOC-USER-ARRAYS () function

Free all user arrays, i.e. these that are members of the *USER-ARRAYS* list.

LET-SEQ-ARRAY (LET-FORMS BODY) macro

Executes body with LET-FORMS bound as in a normal LET. Flush all user arrays after termination of BODY.

WITH-SEQ-ARRAY ((ARRAY-NAME SEQ INTERVAL) BODY) macro

Execute BODY with ARRAY-NAME bound to (FETCH-INTERVAL SEQ INTERVAL). Flush all user arrays at the end.

WITH-IMAGINARY-SEQ-ARRAY ((ARRAY-NAME SEQ INTERVAL) BODY) macro

Execute BODY with ARRAY-NAME bound to (FETCH-IMAGINARY-INTERVAL SEQ INTERVAL). Flush all user arrays in the end.

WITH-COMPLEX-SEQ-ARRAY
((REAL-ARRAY-NAME IMAG-ARRAY-NAME SEQ INTERVAL) BODY) macro

Execute BODY with REAL-ARRAY-NAME bound to (FETCH-INTERVAL SEQ INTERVAL) and IMAG-ARRAY-NAME bound to (FETCH-IMAGINARY-INTERVAL SEQ INTERVAL). Free all user (local) arrays in the end.

WITH-UNCACHED-SEQ-ARRAY ((ARRAY-NAME SEQ INTERVAL) BODY) macro

Execute BODY with ARRAY-NAME bound to (FETCH-UNCACHED-INTERVAL SEQ INTERVAL). Free all user (local) arrays in the end.

WITH-UNCACHED-IMAGINARY-SEQ-ARRAY
((ARRAY-NAME SEQ INTERVAL) BODY) macro

Execute BODY with ARRAY-NAME bound to (FETCH-UNCACHED-IMAGINARY-INTERVAL SEQ INTERVAL). Free all user arrays in the end.

WITH-UNCACHED-COMPLEX-SEQ-ARRAY macro
((REAL-ARRAY-NAME IMAG-ARRAY-NAME SEQ INTERVAL) BODY)

Execute BODY with REAL-ARRAY-NAME bound to (FETCH-UNCACHED-INTERVAL SEQ INTERVAL) and with IMAG-ARRAY-NAME bound to (FETCH-

UNCACHED-IMAGINARY-INTERVAL SEQ INTERVAL). Free all user arrays in the end.

NEWLY-CACHED-SEQUENCES variable

List of newly cached sequences.

SHOW-ALLOC-ARRAY variable

If T, prints a message every time an array is allocated. Defaults to NIL.

SHOW-DEALLOC-ARRAY variable

If T, prints a message every time an array is freed. Defaults to NIL.

ARRAY-HASH-TABLE constant

The hash table for storing and retrieving all free arrays.

MAKE-INDIRECT-ARRAY (ARRAY ARRAY-INTERVAL DESIRED-INTERVAL) function

Returns an indirect array which overlays the DESIRED-INTERVAL in ARRAY.

ALLOC-ARRAY (SIZE) function

Get an array of SIZE from the internal pool of free arrays, if there exists one, otherwise make a new one.

DEALLOC-ARRAY (ARRAY) function

Put an array into the array hash table, the internal pool of free arrays, for reuse. If array is indirect, put back the thing pointed to. (refer to the Lisp machine manual section on indirect arrays).

BFR (SEQ) system

This form causes all fetch requests to SEQ to be cached and passes them on. The intended use of this form is with sequences which are results of expensive computations inside a form which causes computations to be uncached, like WITH-UNCACHING. If the results of these expensive computations are passed through BFR, they become cached.

As mentioned in the section on system implementation, a potential problem with SYS-ALIAS is that its use may cause memory problems. The possibility of performing uncached computations was suggested. Using the forms described in this section, there are many ways to cause this to happen. One systematic way, which converts a SYS-ALIAS into a DEFINE-SYS with minimal changes is shown here by an example: the periodogram definition presented in the section on system implementation is converted into one that flushes all intermediate results after completion of the computation of the periodogram.

```
(DEFINE-SYS PERIODOGRAM-MAG (SEQ BLOCK-SIZE STARTING-POINT BLOCK-OFFSET
          NUMBER-OF-BLOCKS &OPTIONAL (FFT-LENGTH 2048))
             (NUMERIC-SEQUENCE)

  "Get the periodogram of a sequence. Return its magnitude over
  the positive frequencies only. Uncached version - all intermediate seqs are flushed"

(COMPUTE (INTERVAL OUTPUT-ARRAY)
  (WITH-UNCACHING
    (LOOP FOR INDEX FROM 0 TO (- NUMBER-OF-BLOCKS 1)
      COLLECT  (SEQ-MAG-SQUARE
                 (FFT
                   (SEQ-WINDOW SEQ (HAMMING BLOCK-SIZE)
                     (+ STARTING-POINT (* BLOCK-OFFSET INDEX)))
                       (NEXT-POWER-OF-2 FFT-LENGTH)))
      INTO SEQ-LIST
      FINALLY (FETCH-INTERVAL (SEQ-GATE (APPLY 'SEQ-ADD SEQ-LIST)
                (INTERVAL 0 (// (NEXT-POWER-OF-2 FFT-LENGTH) 2)))
                  INTERVAL OUTPUT-ARRAY))))
(DOMAIN ()
  (INTERVAL 0 (// (NEXT-POWER-OF-2 FFT-LENGTH) 2))))
```

The changes made to the SYS-ALIAS periodogram definition are the following:

First, the whole computation was changed to a compute form and was enclosed within a WITH-UNCACHING. This causes all sequences that are cached inside this form to be flushed in the end.

Second, since there is no way to pass a result out of the WITH-UNCACHING using a sequence (i.e. the abstract notion), the result must be passed out by placing it into the OUTPUT-ARRAY. So the RETURN form of the loop expression is changed into a FETCH-INTERVAL, which places the numerical result into OUTPUT-ARRAY.

Third, the domain of the output sequence must be specified using the domain form and an explicit calculation.

5. THE KBSP GRAPHICS FACILITIES

The KBSP graphics system is based on the Lisp machine window system. According to the philosophy of the window system, windows are just flavor instances. By mixing together a set of basic window flavor definitions a wide range of capabilities can be obtained. Usually, the existing basic windows are sufficient for most applications. The KBSP graphics window is one of the basic windows and is defined by the KBSP package to have properties that are suitable for plotting waveforms. The KBSP graphics window is *mouse sensitive*. This means that clicking the mouse on it can have interesting and useful effects.

At the user's level two Lisp functions enable plotting of objects on KBSP graphics windows:

PLOT (OBJECT &OPTIONAL WINDOW DOMAIN RANGE)

OBJECT is the object to be plotted. WINDOW is the KBSP graphics window that will show the plot. If the argument is not provided, the oldest KBSP exposed window will be used. If WINDOW is nil, the system will prompt the user for a window to use by left clicking the mouse on it. DOMAIN is the domain of the object to be plotted. If not given, a reasonable default value will be used, depending on the kind of the object. RANGE is the vertical range of the plot. If not given, it will be adjusted so that the plot fits in the graphics window with a small margin. If OBJECT is a complex sequence, then the KBSP window is split into two horizontal panes, one for the real and the other for the imaginary part of the complex sequence.

OVERLAY-PLOT (OBJECT &OPTIONAL WINDOW DOMAIN)

This is similar to plot except that it produces an overlay plot (it does not erase the window it is plotting on and it uses the vertical range of the previous plot).

An interesting feature of the KBSP graphics window is its mouse sensitivity. According to where the mouse points, two kinds of facilities are available, and the documentation strip at the bottom of the Lisp machine screen summarizes them.

If the mouse points at a KBSP graphics window label:

Clicking left once (L1) calls the function SHOWR on the plotted object and the result is plotted on the KBSP Lisp Listener.

Clicking left twice (L2) gives a short description of the KBSP graphics window.

Clicking right once (R1) replots the object after an argument is provided (notice that the KBSP Lisp Listener cursor blips much faster that before waiting for the argument to be typed). If the argument is a number, the object is replotted centered around this number. If the argument is an interval, the object is plotted over the interval.

If the mouse points anywhere else inside a KBSP graphics window except at its label:

Clicking left once (L1) marks the current position of the mouse cursor and shows the coordinates of the corresponding plot point in the upper label pane of the corresponding KBSP graphics window. This feature is useful for reading values off a plot.

Clicking left twice (L2) clears the mark.

From the implementation viewpoint, the picture is as follows:

The *PLOT function* first finds an appropriate KBSP graphics window to use, either by asking the user or by grabbing the oldest exposed KBSP window. Then it checks its argument for acceptability. In general, the object to be plotted must have its own plotting method or function, unless it is a sequence. In the first case, the private plotting method or function is called. In the case of a sequence, the KBSP graphics window is configured appropriately (i.e. the previous plot is erased, the labels are drawn including various pieces of information, such as the name of the sequence, its plotted domain and its range of values) and finally the plotting is done and the window is exposed (i.e. shown on the screen. An unexposed window does not appear on the screen). The work of plotting consists of a lot of bookkeeping operations and is shared between the object and the KBSP graphics window. The ability of a sequence to handle these bookeeping operations is derived from a flavor mixin, called OBJECT-PLOT-MIXIN, which is included in the sequence flavor definition.

BIBLIOGRAPHY

Allen, Elizabeth: "YAPS: Yet Another Production System", *AAAI 1983*, pp. 5-7.

Backus, John: "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", *Communications of the ACM*, August 1978, Vol. 21-8.

Barstow, Shrobe and Sandewall: *Interactive Programming*, McGraw-Hill, 1984 (chapters 1, 3 and 25).

Byte Magazine: *Special issue on Smalltalk*, August 1981.

Kopec, Gary: *The Representation of Discrete-time Signals and Systems in Programs*, Ph.D. Thesis, M.I.T., May 1980.

Liskov, Snyder, Atkinson, Schaffert: "Abstraction Mechanisms in CLU", *Communications of the ACM*, August 1977, Vol. 20-8.

McCarthy and Levin: *Lisp 1.5 Programmer's Manual*, MIT Press, 1965.

Oppenheim, A. V. and R. Schafer: *Digital Signal Processing*, Prentice Hall, 1975.

Ritchie and Thompson: "The UNIX Time Sharing System", *Communications of the ACM*, July 1974, Vol. 17-7.

Steele and Sussman: "Design of a LISP-based Microprocessor", *Communications of the ACM*, November 1980, Vol. 23-11.

Weinreb, Moon and Stallman: *Lisp Machine Manual*, Fifth Edition, January 1983, M.I.T. A.I. Lab Publication.

Winston and Horn: *Lisp*, Addison-Wesley 1981 (chapters 18 and 22).

INDEX

VARIABLES

FUNCTIONS

SYSTEMS

MACROS

DISTRIBUTION LIST

	DODAAD Code	
Director Advanced Research Project Agency 1400 Wilson Boulevard Arlington, Virginia 22209 Attn: Program Management	HX1241	(1)
Group Leader Information Sciences Associate Director for Engineering Sciences Office of Naval Research 800 North Quincy Street Arlington, Virginia 22217	N00014	(1)
Administrative Contracting Officer E19-628 Massachusetts Institute of Technology Cambridge, Massachusetts 02139	N66017	(1)
Director Naval Research Laboratory Attn: Code 2627 Washington, D.C. 20375	N00173	(6)
Defense Technical Information Center Bldg. 5, Cameron Station Alexandria, Virginia 22314	S47031	(12)

P/17/02 P/09/02 ——— CO.

VA T" 1/24

@@ @@ A148963 (U) FIELD/GROUP @@@@@@

UNCLASSIFIED TITLE

AN OBJECT-ORIENTED SIGNAL PROCESSING ENVIRONMENT: THE KNOWLEDGE-BASED SIGNAL PROCESSING PACKAGE.

ABSTRACT

(U) A LISP-BASED SIGNAL PROCESSING PACKAGE FOR INTEGRATED NUMERIC AND SYMBOLIC MANIPULATION OF DISCRETE-TIME SIGNALS I S DESCRIBED. THE PACKAGE IS BASED ON THE CONCEPT OF SIGNAL ABSTRACTION IN WHICH A SIGNAL IS DEFINED BY ITS NON-ZERO DOM AIN AND BY A METHOD FOR COMPUTING ITS SAMPLES. MOST COMMON SIGNAL PROCESSING OPERATIONS ARE DEFINED IN THE PACKAGE AND THE PACKAGE PROVIDES SIMPLE METHODS FOR THE DEFINITION OF NEW OPERATORS. THE PACKAGE PROVIDES FACILITIES FOR THE MANIPU LATION OF INFINITE DURATION SIGNALS AND PERIODIC SIGNALS, FOR THE EFFICIENT COMPUTATION OF SIGNALS OVER INTERVALS, AND FOR THE CATCHING OF SIGNAL VALUES. THE PACKAGE IS CURRENTLY BEING EXPANDED TO PROVIDE FOR MANIPULATION OF CONTINUOUS-TI ME SIGNALS AND SYMBOLIC SIGNAL TRANSFORMATIONS, SUCH AS THE FOURIER TRANSFORM, TO FORM THE BASIS OF KNOWLEDGE-BASED SIG NAL PROCESSING SYSTEMS.

POSTING TERMS ASSIGNED

COMMON SIGNAL PROCESSING OPERATIONS
USE SIGNAL PROCESSING) — CO.

FOURIER TRANSFORM
USE FOURIER TRANSFORMATION) — CO.

KNOWLEDGE-BASED SIGNAL PROCESSING SYSTEMS
USE SIGNAL PROCESSING

PERIODIC SIGNALS
USE SIGNALS

SIGNALS OVER INTERVALS
USE INTERVALS
SIGNALS

DISCRETE-TIME SIGNALS
USE DISCRETE DISTRIBUTION
SIGNALS
TIME

INFINITE DURATION SIGNALS
USE SIGNALS
TIME

NEW OPERATORS
USE OPERATORS(PERSONNEL)

SIGNAL VALUES
USE SIGNALS
VALUE

SYMBOLIC SIGNAL TRANSFORMATIONS
USE SIGNAL PROCESSING
SYMBOLS) — CO.

★TIME SIGNALS) — CO.
★COMPUTER APPLICATIONS) — CO.
COMPUTATIONS) — CO.

MANIPULATORS) — CO.
NUMERICAL ANALYSIS) — CO.

PHRASES NOT FOUND DURING LEXICAL DICTIONARY MATCH PROCESS

#25
KNOWLEDGE BASED SYSTEMS) — CO.

LISP PROGRAMMING LANGUAGE) — CO.

END

FILMED

1-85

DTIC