BROWN UNIVERSITY
Department of Computer Science
Master's Project
CS-89-M3

"Temporal Reasoning with Infinitesimals"

by
Louis P. DiPalma

Temporal Reasoning with Infinitesimals

submitted by

Louis P. DiPalma

in partial fulfillment of the requirements for the
Master of Science Degree
in Computer Science at Brown University

May, 1989

Thomas Dean

Thomas Dean, Advisor

TABLE OF CONTENTS

LIST OF TABLES

1. INTRODUCTION

Early Temporal Reasoning based Planning Systems had not represented nor dealt with time effectively. Only recently have Temporal Reasoning based Planning Systems attempted to address these limitations. These systems typically refer to time as a single-valued entity (scalar). Typically this scalar is represented and implemented as an integer.

These systems utilize time to demarcate the beginning and ending point (interval) for a particular action. During plan creation plan intervals are designated. These intervals are denoted by 2 scalars of the form **<begin,end>**. These intervals denote a time period in which the state of a particular fact/condition/event is known. This known status can either represent the existent (true) or non-existent (false) of a fact/condition/event. Assume that if the status of a particular fact/condition/event is not present it is assumed to be false.

In the process of developing a complete plan for the original goal (top-level), these planning systems create sub-plans. These sub-plans eventually make up the complete plan. During plan ordering or reordering, reasoning about time using the representation described above poses severe limitations for potential ordering/reordering. This limitation can also pose restrictions in the situation where replanning is required during the execution of a plan. These limitations arise from the **limited expressibility** of the representation of time as a single scalar.

Overcoming this problem of limited expressibility is a prerequisite of advanced Temporal Reasoning based Planning Systems.

2. PURPOSE

The purpose of this project was to investigate the expressibility problem of representing time as a scalar. The first part of the project was to determine the approximate limitations. Based on these limitations, a method which would alleviate these limitations was to be developed and implemented.

3. TECHNICAL APPROACH

The technical approach developed and followed in undertaking this task consisted of five steps. Results of each step were utilized in the succeeding step. The technical approach developed was as follows:

1. Analyze problem of limited expressibility,
2. Develop representation and associated operations,
3. Implement representation scheme,
4. Perform comparative analysis, and
5. Propose possible future enhancements.

4. PROBLEM DEFINITION

4.1 Background

The beginning of classical planning systems was marked at the onset by Sacerdotti's NOAH and Fikes and Nilsson's STRIPS systems. Tate's NONLIN and Vere's DEVISER further advanced the field. These early systems either totally disregarded the concept of time or merely scratched the surface. Reason being, extensive temporal reasoning for some of these early systems was not required. It was not until Vere's DEVISER did planning systems incorporate a temporal reasoning facility, reasoning about the effect of time.

These early planning systems assumed that action in the "world" could be represented as a series of states. These states can be viewed as a snapshot [9]. In light of this, Fikes and Nilsson's STRIPS system solved reasoning planning problems via linear planning. Sacerdotti's NOAH system ushered in the era of hierarchical non-linearing planning. Tate's NONLIN system expanded Sacerdotti's notion of hierarchical non-linear planning by introducing backtracking [9]. It was not until the introduction of Vere's DEVISER system did temporal reasoning come upon the scene.

As was the case with STRIPS, its power being limited by its state representation, so too was Vere's DEVISER limited. DEVISER's limitation stemmed from its representation of time. Many of the planning systems

since DEVISER have also used a scalar representation of time. Some systems though have been developed (eg: FORBIN) which included a more robust representation of time. Miller, Firby, and Dean's FORBIN system extended the scalar representation of time, and, by doing so, were able to provide a more advanced temporal reasoning facility.

4.2 Limited Expressibility Analysis

The aforementioned early planning systems that performed temporal reasoning represented time as a scalar quantity, a point on the real number line. Also, as stated previously, many of these early systems did not require any more in the representation of time. One might ask then where in does the limited expressibility lie. In was not until the introduction of the hierarchical non-linear planning systems that the requirement for a more advanced representation of time arise.

These hierarchical non-linear planning systems attempt to build a single overall plan. This single overall plan is composed of many smaller sub-plans. These sub-plans may in turn be decomposed into further sub-plans until ultimately the sub-plan actually represents a primitive task. A primitive task is defined here to be the lowest level sub-plan, the level at which action actually takes place.

It is these sub-plans that rely on each other's (other sub-plans) actions as well as preconditions. Associated with each sub-plan is its point of

inception (beginning) and point of termination (ending). In the development of these sub-plans certain constraints are uncovered which must be satisfied. As one would expect, some of these sub-plans are not totally independent of all other sub-plans (**sub-plan dependency**). Likewise, there are some sub-plans which may be carried out in parallel with other sub-plans (**sub-plan independence**). Primarily, it is this **sub-plan dependency** that is of concern. Not only does the problem arise in developing the initial plan, but also in the event of an occurrence of an unexpected circumstance that arises which adversely affects the original plan. If the original plan becomes invalidated due to these unexpected circumstances, replanning is required.

5. REPRESENTATION AND ASSOCIATED OPERATIONS

5.1 Representation

In order to circumvent this expressibility limitation and provide some flexibility in light of the possible replanning that might be required, a new representation for time is proposed. This represenation is not entirely unique, but it attempts to expand on the notion and representation of time outlined and defined in FORBIN.

The representation realized here provides the means by which a time **window** can be associated with a particular entity, no matter what the entity. The entity can include events, conditions, activities, and/or states in the world. This representation allows the specification of the minimum possible start/stop time and maximum possible start/stop time. What this representation provides is the capability for uncertain/imprecise time quantities to be specified. There is no restriction that these minimum and maximum times be distinct. The minimum and maximum time may in fact be identical. If these values were identical, it would imply that the data was precise.

The time **window** can be viewed as representing the interval in which an entity must begin or end. This **window** can also be viewed as representing a duration for a particular entity. This representation provides a considerable amount of flexibility.

The representation chosen to express time is a follows:

$$(< t_1 , t_{1epsilon} > , < t_2 , t_{2epsilon} >)$$

where:

$t_1 \equiv$ **integer; min_integer** $< t_1 <$ **max_integer**

$t_{1epsilon} \equiv$ **integer; min_integer** $< t_{1epsilon} <$ **max_integer**

$t_2 \equiv$ **integer;** $0 < t_2 <$ **max_integer**

$t_{2epsilon} \equiv$ **integer;** $0 < t_{2epsilon} <$ **max_integer**

Default values have been assigned for each of the items as follow:

$t_1 =$ **0**

$t_{1epsilon} =$ **0**

$t_2 =$ **most–positive–fixnum**

$t_{2epsilon} =$ **most–positive–fixnum**

The $< t_1 , t_{1epsilon} >$ notation denotes a point on the real number line. The $(< t_1 , t_{1epsilon} > , < t_2 , t_{2epsilon} >)$ notation denotes a fuzzy interval on the real number line or bounds on a time point. The value **most-positive-fixnum** is defined to be the largest integer representable and subsequently is machine dependent. As can be seen from the definition, the only permissible values for each of the items is an integer between the smallest and the largest possible integers representable. Each item is

defined to be of type integer. Defining the items to be of type floating point does provide some additional expressiveness but at a cost. The cost here being time, both in the computational sense, that which is required by a computer's CPU/ALU to perform floating point operations, but also in the available time in which the planning must be performed. Probably the greatest effect would be in the time available for replanning where time is definitely at a premium. Based on this, the added expressiveness of floating point types does not warrant their use. The Comparative Analysis section further addresses this and other issues associated with the various representations and their associated operations.

In one light, the representation chosen here can be viewed as an extension of fuzzy numbers and their associated arithmetic. Theorists of fuzzy numbers would view the $< t_1 , t_2 >$ representation as an **Interval of Confidence** [7]. This representation has been extended to accommodate a more imprecise notion of time. This extended representation provides a greater degree of flexibility over the dual value Interval of Confidence. In general, with respect to intervals, this extended representation also accommodates the notion of inclusion and exclusion. For instance, if it were desirable to express the interval between 1 and 4 inclusively, the representation would be as follows: **(<1,0> , <4,0>).** If it were the interval between 1 and 4 excluding 1 the representation would be **(<1,1> , <4,0>)** and likewise for the other side of the interval with 4.

The epsilon part of the value can be viewed as an **infinitesimal**. An infinitesimal is defined as the smallest unit to which all values can be decomposed. Infinitesimals, and in particular the Infinitesimal Calculus, were developed by Leibniz [6]. The infinitesimal forms the foundation of our modern day Integral Calculus. So, in essence, this representation can be likened to a floating point number with its integer and fractional part or complex number with its real and imaginary part. Giving credit to Leibniz, the time representation is denoted by **liebnum** after Leibniz.

5.2 Associated Operations

Given this representation of time, it is desirable to be able to perform various operations. These operations are performed via a set of associated functions. These functions include the capability to determine if one liebnum is less than another liebnum.

lieb1< lieb2

```
;function to determine if one lieb-num is less than another lieb-num
;type check performed to verify correct parameters
(defun lieb1<lieb2 (lieb1 lieb2)
  (declare (optimize (speed 3) (space 3))
        (inline lieb-num-initial-fixnum lieb-num-initial-epsilon
              lieb-num-terminal-fixnum lieb-num-terminal-epsilon))


 (cond
  ((and (typep lieb1 'lieb-num) (typep lieb2 'lieb-num))
   (let ((lieb2.fx1 (lieb-num-initial-fixnum lieb2)))
     (or
       (and
```

```
        (< (lieb-num-initial-fixnum lieb1) lieb2.fx1)
        (< (lieb-num-terminal-fixnum lieb1) lieb2.fx1)
        (< (lieb-num-terminal-epsilon lieb1)
           (lieb-num-initial-epsilon lieb2))
        t)
       nil)))


;return error on incorrect parameter
(t (error "LIEB:LIEB1<LIEB2 Incorrect parameter type"))))
```

Another function determines whether one liebnum subsumes another liebnum.

lieb1subsumeslieb2

```
;function to determine if one lieb-num subsumes another lieb-num
;type check performed to verify correct parameters
(defun lieb1subsumeslieb2 (lieb1 lieb2)
  (declare (optimize (speed 3) (space 3))
         (inline lieb-num-initial-fixnum lieb-num-initial-epsilon
                 lieb-num-terminal-fixnum lieb-num-terminal-epsilon))


 (cond
  ((and (typep lieb1 'lieb-num) (typep lieb2 'lieb-num))

   (and
    (or
     (< (lieb-num-initial-fixnum lieb1)
        (lieb-num-initial-fixnum lieb2))

     (and

      (<= (lieb-num-initial-fixnum lieb1)
          (lieb-num-initial-fixnum lieb2))
      (<= (lieb-num-initial-epsilon lieb1)
          (lieb-num-initial-epsilon lieb2))))

    (or
```

```
        (> (lieb-num-terminal-fixnum lieb1)
          (lieb-num-terminal-fixnum lieb2))


        (and
          (>= (lieb-num-terminal-fixnum lieb1)
             (lieb-num-terminal-fixnum lieb2))

          (>= (lieb-num-terminal-epsilon lieb1)
             (lieb-num-terminal-epsilon lieb2))))))



      ;return error on incorrect parameter
      (t (error "LIEB:LIEB1SUBSUMESLIEB2 Incorrect parameter type"))))
```

Another set of functions provide the capability to add one liebnum to another liebnum, subtract one liebnum from another liebnum, as well as add/subtract an integer (fixnum) to/from a liebnum.

lieb+lieb

```
;function to add 2 lieb-nums
;type check performed to verify correct parameters
(defun lieb+lieb (lieb1 lieb2)
 (declare (optimize (speed 3) (space 3))
        (inline lieb-num-initial-fixnum lieb-num-initial-epsilon
              lieb-num-terminal-fixnum lieb-num-terminal-epsilon
              fixnum+ epsilon+ liebcreatesyms))

 (cond
  ((and (typep lieb1 'lieb-num) (typep lieb2 'lieb-num))
   (let ((temp-liebnum nil))
    (and (not *listof-symbols*) (liebcreatesyms *default-numof-symstocreate*))
    (setq temp-liebnum (car *listof-symbols*))
    (setq *listof-symbols* (cdr *listof-symbols*))


    (setf (lieb-num-initial-fixnum  temp-liebnum))
     (fixnum+
       (lieb-num-initial-fixnum lieb1)
```

```
      (lieb-num-initial-fixnum lieb2)))

    (setf (lieb-num-initial-epsilon temp-liebnum)
     (epsilon+
       (lieb-num-initial-epsilon lieb1)
       (lieb-num-initial-epsilon lieb2)))

    (setf (lieb-num-terminal-fixnum temp-liebnum)
     (fixnum+
       (lieb-num-terminal-fixnum lieb1)
       (lieb-num-terminal-fixnum lieb2)))

    (setf (lieb-num-terminal-epsilon temp-liebnum)
     (epsilon+
       (lieb-num-terminal-epsilon lieb1)
       (lieb-num-terminal-epsilon lieb2)))

     temp-liebnum))


   ;return error on incorrect parameter
   (t (error "LIEB:LIEB+LIEB Incorrect parameter type"))))
```

lieb-lieb

```
;function to subtract 2 lieb-nums
;type check performed to verify correct parameters
(defun lieb-lieb (lieb1 lieb2)
  (declare (optimize (speed 3) (space 3))
         (inline lieb-num-initial-fixnum lieb-num-initial-epsilon
                 lieb-num-terminal-fixnum lieb-num-terminal-epsilon
                 fixnum- epsilon- liebcreatesyms))

 (cond
  ((and (typep lieb1 'lieb-num) (typep lieb2 'lieb-num))
  (let ((temp-liebnum nil))
    (and (not *listof-symbols*) (liebcreatesyms *default-numof-symstocreate*))
    (setq temp-liebnum (car *listof-symbols*))
    (setq *listof-symbols* (cdr *listof-symbols*))


    (setf (lieb-num-initial-fixnum  temp-liebnum)
     (fixnum-
       (lieb-num-initial-fixnum lieb1)
       (lieb-num-initial-fixnum lieb2)))
```

```
(setf (lieb-num-initial-epsilon  temp-liebnum)
 (epsilon-
   (lieb-num-initial-epsilon lieb1)
   (lieb-num-initial-epsilon lieb2)))

(setf (lieb-num-terminal-fixnum  temp-liebnum)
 (fixnum-
   (lieb-num-terminal-fixnum lieb1)
   (lieb-num-terminal-fixnum lieb2)))

(setf (lieb-num-terminal-epsilon  temp-liebnum)
 (epsilon-
   (lieb-num-terminal-epsilon lieb1)
   (lieb-num-terminal-epsilon lieb2)))


 temp-liebnum))


;return error on incorrect parameter
(t (error "LIEB:LIEB-LIEB Incorrect parameter type"))))
```

lieb+fixnum

```
;function to add a lieb-num and a fixnum
;type check performed to verify correct parameters
(defun lieb+fixnum (lieb1 fx1)
 (declare (optimize (speed 3) (space 3))
        (inline lieb-num-initial-fixnum lieb-num-initial-epsilon
                lieb-num-terminal-fixnum lieb-num-terminal-epsilon
                fixnum+ liebcreatesyms))

 (cond
  ((and (typep lieb1 'lieb-num) (typep fx1 'integer))
  (let ((temp-liebnum nil))
   (and (not *listof-symbols*) (liebcreatesyms *default-numof-symstocreate*))
   (setq temp-liebnum (car *listof-symbols*))
   (setq *listof-symbols* (cdr *listof-symbols*))


   (setf (lieb-num-initial-fixnum temp-liebnum)
    (fixnum+
      (lieb-num-initial-fixnum lieb1) fx1))

   (setf (lieb-num-initial-epsilon  temp-liebnum)
```

```
    (lieb-num-initial-epsilon lieb1))

   (setf (lieb-num-terminal-fixnum  temp-liebnum)
    (fixnum+ (lieb-num-terminal-fixnum lieb1) fx1))

   (setf (lieb-num-terminal-epsilon  temp-liebnum)
    (lieb-num-terminal-fixnum lieb1))

   temp-liebnum))



  ;return error on incorrect parameter
  (t (error "LIEB:LIEB+FIXNUM Incorrect parameter type"))))
```

lieb-fixnum

```
;function to subtract a lieb-num and a fixnum
;type check performed to verify correct parameters
(defun lieb-fixnum (lieb1 fx1)
 (declare (optimize (speed 3) (space 3))
        (inline lieb-num-initial-fixnum lieb-num-initial-epsilon
              lieb-num-terminal-fixnum lieb-num-terminal-epsilon
              fixnum- liebcreatesyms))

 (cond

  ((and (typep lieb1 'lieb-num) (typep fx1 'integer))
  (let ((temp-liebnum nil))
   (and (not *listof-symbols*) (liebcreatesyms *default-numof-symstocreate*))
   (setq temp-liebnum (car *listof-symbols*))
   (setq *listof-symbols* (cdr *listof-symbols*))


   (setf (lieb-num-initial-fixnum  temp-liebnum)
    (fixnum-
      (lieb-num-initial-fixnum lieb1) fx1))

   (setf (lieb-num-initial-epsilon  temp-liebnum)
    (lieb-num-initial-epsilon lieb1))

   (setf (lieb-num-terminal-fixnum  temp-liebnum)
    (fixnum-
      (lieb-num-terminal-fixnum lieb1) fx1))
```

```
(setf (lieb-num-terminal-epsilon  temp-liebnum)
  (lieb-num-terminal-epsilon lieb1))

 temp-liebnum))


;return error on incorrect parameter
(t (error "LIEB:LIEB-FIXNUM Incorrect parameter type"))))
```

A function is also provided to determine the real interval of a particular liebnum to get an approximate span of the interval.

liebrealinterval

```
;function to determine approximate span of interval, epsilons disregarded
;type checks performed to verify correct parameter type
(defun liebrealinterval (lieb)
 (declare (optimize (speed 3) (space 3))
        (inline lieb-num-initial-fixnum lieb-num-terminal-fixnum))

 (cond
  ((typep lieb 'lieb-num)


   (fixnum- (lieb-num-terminal-fixnum lieb)
         (lieb-num-initial-fixnum lieb)))


   ;return error on incorrect parameter
   (t (error "LIEB:LIEBREALINTERVAL Incorrect parameter type"))))
```

For each of these operations, limit checks are performed, even though some should never occur. As was the case with the representation name **liebnum**, so too does each operation begin with the prefix **lieb**. The

functions presented here are samples from the LIEBNUM package which contains the representation and associated operations. The package was implemented in Common Lisp (Allegro Common lisp) on a Macintosh Plus (Apple) computer.

6. COMPARATIVE ANALYSIS

In order to asses the total impact of the **LIEBNUMS** representation and associated operations, a comparative analysis was performed. Three additional representations and associated operations were developed and implemented. One of those was the representation which was utilized in the **zztop** implementation. Minor modifications were required to make the associated operations consistent with the rest. In the **zztop** implementation, a time interval was represented as follows: $< s_1 , s_2 >$, where each **s** value was a symbolic number consisting of either an integer or a symbolic number from the set {pos-tiny, neg-tiny, pos-inf, neg-inf}. This representation was labeled **ZZNUMS** after **zztop**.

The second representation $< i_1 , i_2 >$ and associated operations were similar to the **ZZNUMS**, with one big exception, the **i** values were restricted to integers only. No symbolic numbers were permitted. This representation was labeled **FORBNUMS** after the **FORBIN** system.

The third and final representation $< r_1 , r_2 >$ and associated operations to which the **LIEBNUMS** were compared was a representation closely related to the **FORBNUMS**, but in this case, the **r** values were restricted to floating point numbers. Likewise, as with the **FORBNUMS**, no symbolic numbers were permitted. This representation was labeled **FLOFORBNUMS**.

The four representations are summarized as follows:

LIEBNUMS: $(< i_1 , i_{1epsilon} > , < i_2 , i_{2epsilon} >)$

ZZNUMS: $< s_1 , s_2 >$

FORBNUMS: $< i_1 , i_2 >$

FLOFORBNUMS: $< r_1 , r_2 >$

where i = integer, r = float, and s = symbolic number. Each of these time representations denotes an interval on the real number line or bounds on a time point.

The comparative analysis performed was divided into two phases. The first phase consisted of performing a qualitative analysis of each representation. The qualitative analysis compared the salient features of the representation with respect to the others. The second phase of the comparative analysis consisted of performing a quantitative analysis of each representation. The quantitative analysis consisted of measuring the computational resource consumption (CPU time) for each representation.

6.1 Qualitative Analysis

With respect to the expressibility of each representation, the **FLOFORBNUMS** representation provided the greatest flexibility. This representation was followed closely by the **LIEBNUMS** representation.

The **ZZNUMS** and **FORBNUMS** followed the **LIEBNUMS** respectively. The one advantage the **ZZNUMS** had over the **FORBNUMS** was the permission of symbolic numbers. As will be evident later, this proved to be a hindrance in the quantitative analysis.

A problem which, as one would expect, reared its ugly head was the round-off error associated with floating point arithmetic. This only occurred with the **FLOFORBNUMS** since this was the only representation was permitted floating point numbers. An error which actually occurred during testing happened when 1.2 was subtracted from 5.6. One would expect this to result in 4.4, but the actual result was 4.399999999. Another problem which also only occurred with the **FLOFORBNUMS** was the problem of overflow.

6.2 Quantitative Analysis

As mentioned earlier, CPU resource consumption was to be the measure for the quantitative analysis. Two sets of benchmark programs were developed to evaluate the computational efficiency of each representation and its associated operations. As a note, all functions were optimized by directing the compiler to optimize speed and space as well as using the open coding technique of declaring functions to be inline. The difference between the 2 sets is that one set of benchmark programs attempts to manage its own memory by obtaining time symbols

from the list of available symbols and returning the symbol when it was no longer needed. The other set continually gets new symbols but never returns them when there use is no longer needed. It is intended that these functions which allocate, reallocate, and deallocate symbols would be used by an actual planning system.

Each of the benchmark programs attempts to simulate the operation of an actual planning application. Each program accepts as input the number of iterations to perform on the set of operations. A temporal database is setup to operate on. The number of operations for each program is identical. These operations consist of a mixture which spans all of the associated operations. Not only does each program consist of the same number of operations, but they are also of the same type and in the same order for each representation.

Table 1 contains the results of the benchmarks which did not utilize memory management. Table 2 contains results of the benchmarks which did in fact utilize memory management. As was mentioned earlier, the permission of symbolic numbers proved to be a hindrance, as is evident from the benchmarks. Also, the permission of floating point numbers resulted in increased times. It is not completely fair to the floating point representation since the machine (MAC Plus) which the package was developed and tested on did not have a floating point coprocessor. Based on the Qualitative and Quantitative Analysis performed, it is evident that the combination of increased expressibility and CPU

efficiency, make the **LIEBNUMS** representation the choice of the four.

TABLE 1. BENCHMARK RESULTS (No Memory Management)

REPRESENTATION	ITERATIONS	TIME TO RUN(SEC.)
LIEBNUMS	50	124.767
FORBNUMS	50	114.533
FLOFORBNUMS	50	181.517
ZZNUMS	50	599.783

TABLE 2. BENCHMARK RESULTS (With Memory Management)

REPRESENTATION	ITERATIONS	TIME TO RUN(SEC.)
LIEBNUMS	50	67.383
FORBNUMS	50	54.000
FLOFORBNUMS	50	121.250
ZZNUMS	50	537.817

7. POSSIBLE FUTURE ENHANCEMENTS

It would be idealistic to assume that the capability provided by this package of representation and associated operations, which support temporal reasoning, would suit all needs all the time, forever. For instance, the need for additional operations might be desirable. If so, the associated functions would need to be investigated, developed, implemented, and tested. As Temporal Reasoning Systems mature, the representation developed here might not be adequate. Possibly a third item of the same form would be added. This additional item could denote the most probable value for the given **window** specified. This enhanced representation might possibly be defined as follows:

$$(< t_1 , t_{1epsilon} > , < t , t_{\epsilon} > , < t_2 , t_{2epsilon} >).$$

While on the subject of probabilities, the possibility exists whereby it might be desirable to associate a probability value with each item in the representation. Default probabilities could also be assigned to each item. These probability values, certainity index, could give some degree of confidence for the associated item.

8. REFERENCES

[1] Boddy, Mark and Dean, Thomas, *Solving Time-Dependent Planning Problems*, Technical Report No. CS-89-03, Brown University Department of Computer Science, February, 1989.

[2] Charniak, Eugene and McDermott, Drew, *Introduction to Artificial Intelligence*, Addison-Wesley, Reading, MA, 1985.

[3] Dean, Thomas, *Planning Paradigms*, Brown University Department of Computer Science, October, 1987.

[4] Dean, Thomas, *Reasoning About the Effects of Actions in Automated Planning Systems*, Brown University Department of Computer Science.

[5] Hurd, A. and Loeb, P. A., *An Introduction to Nonstandard Real Analysis*, Academic Press, San Diego, CA, 1985.

[6] Jourdain, Philip C., *Contributions to the Founding of the Theory of Transfinite Numbers*, Dover, New York, NY, 1981.

[7] Kaufman, Arnold and Gupta, Madan, *Introduction to Fuzzy Mathematics*, Van Nostrand Reinhold, NY, New York, 1988.

[8] Kautz, Henry A. and Pednault, Edwin P. D., *Planning and Plan Recognition*, AT&T Technical Journal, Volume 67, Issue 1, January/February, 1988.

[9] Wilkins, David E., *Practical Planning*, Morgan Kaufman, San Mateo, CA, 1988.

9. LIEBNUM PACKAGE

```
;;
;;
;;
;;*****************************************************************************
;;
;;
;;    Package Name: LIEBNUM_Package
;;    Package Abbreviation: LIEB
;;    Description: Representation and Functions for LIEBNUMS
;;    Last Modified: 9 May 1989
;;
;;*****************************************************************************
;;
;;
;;
;;
;;
;;
;;    Setup package specification
(in-package 'LIEBNUM_Package :nicknames '(LIEB))

;defines largest value
(defparameter *most-positive-fixnum* most-positive-fixnum)

;defines smallest value, defined for future use
(defparameter *most-negative-fixnum* most-negative-fixnum)




;defines list for holding the available liebnums

(defparameter *listof-symbols* '())

(defparameter *listof-syms* '())

(defparameter *default-numof-symstocreate* 10)




;structure definition for LIEBNUM, utilizes defstruct definition from
;the FORBIN system
(defstruct (lieb-num
```

```
        (:print "[< a,  a> , < a,  a>]"
          (print-liebnum (lieb-num-initial-fixnum lieb-num))
          (print-liebnum (lieb-num-initial-epsilon lieb-num))
          (print-liebnum (lieb-num-terminal-fixnum lieb-num))
          (print-liebnum (lieb-num-terminal-epsilon lieb-num)))
                    (:constructor make-lieb-num (initial-fixnum initial-epsilon
                                                 terminal-fixnum
                                                 terminal-epsilon)))

 ;lieb-num item definitions
 (initial-fixnum 0 :type integer)
 (initial-epsilon 0 :type integer)
 (terminal-fixnum *most-positive-fixnum* :type integer)
 (terminal-epsilon *most-positive-fixnum*  :type integer))




;function to print/output lieb-num item
(defun print-liebnum (item)
  (declare (optimize (speed 3) (space 3)))
  item)




;function to add 2 fixnums
;type/limit checks performed to verify parameter correctness/legality
(defun fixnum+ (fx1 fx2)
 (declare (optimize (speed 3) (space 3)))
 (cond

  ((and (typep fx1 'integer) (typep fx2 'integer))
   (let ((result (+ fx1 fx2)))
     (or
       (and (< result *most-positive-fixnum*) result)
       *most-positive-fixnum*)))

  ;return error on incorrect parameter
  (t (error "LIEB:FIXNUM+ Incorrect parameter type"))))
```

```
;function to add 2 epsilons
;type/limit checks performed to verify parameter correctness/legality
(defun epsilon+ (ep1 ep2)
 (declare (optimize (speed 3) (space 3)))
 (cond
  ((and (typep ep1 'integer) (typep ep2 'integer))
   (let ((result (+ ep1 ep2)))
    (or
      (and (< result *most-positive-fixnum*) result)
      *most-positive-fixnum*)))


  ;return error on incorrect parameter
  (t (error "LIEB:EPSILON+ Incorrect parameter type"))))




;function to subtract 2 fixnums
;type/limit checks performed to verify parameter correctness/legality
(defun fixnum- (fx1 fx2)
 (declare (optimize (speed 3) (space 3)))
 (cond
  ((and (typep fx1 'integer) (typep fx2 'integer))
   (let ((result (- fx1 fx2)))
    (or
      (and (plusp result) result)
      0)))


  ;return error on incorrect parameter
  (t (error "LIEB:FIXNUM- Incorrect parameter type"))))




;function to subtract 2 epsilons
;type/limit checks performed to verify parameter correctness/legality
(defun epsilon- (ep1 ep2)
 (declare (optimize (speed 3) (space 3)))
```

```
(cond
 ((and (typep ep1 'integer) (typep ep2 'integer))
  (let ((result (- ep1 ep2)))
   (or
     (and (plusp result) result)
     0)))


 ;return error on incorrect parameter
 (t (error "LIEB:EPSILON- Incorrect parameter type"))))
```

```
;function to add 2 lieb-nums
;type check performed to verify correct parameters

(defun lieb+lieb (lieb1 lieb2)
 (declare (optimize (speed 3) (space 3))
        (inline lieb-num-initial-fixnum lieb-num-initial-epsilon
                lieb-num-terminal-fixnum lieb-num-terminal-epsilon
                fixnum+ epsilon+ liebcreatesyms))

 (cond
  ((and (typep lieb1 'lieb-num) (typep lieb2 'lieb-num))
   (let ((temp-liebnum nil))
    (and (not *listof-symbols*) (liebcreatesyms *default-numof-symstocreate* ))
    (setq temp-liebnum (car *listof-symbols*))
    (setq *listof-symbols* (cdr *listof-symbols*))


    (setf (lieb-num-initial-fixnum  temp-liebnum)
     (fixnum+
       (lieb-num-initial-fixnum lieb1)
       (lieb-num-initial-fixnum lieb2)))

    (setf (lieb-num-initial-epsilon  temp-liebnum)
     (epsilon+
       (lieb-num-initial-epsilon lieb1)
       (lieb-num-initial-epsilon lieb2)))

    (setf (lieb-num-terminal-fixnum  temp-liebnum)
     (fixnum+
```

```
      (lieb-num-terminal-fixnum lieb1)
      (lieb-num-terminal-fixnum lieb2)))

   (setf (lieb-num-terminal-epsilon  temp-liebnum)
    (epsilon+
      (lieb-num-terminal-epsilon lieb1)
      (lieb-num-terminal-epsilon lieb2)))


    temp-liebnum))


 ;return error on incorrect parameter
 (t (error "LIEB:LIEB+LIEB Incorrect parameter type"))))




;function to subtract 2 lieb-nums
;type check performed to verify correct parameters
(defun lieb-lieb (lieb1 lieb2)
  (declare (optimize (speed 3) (space 3))
         (inline lieb-num-initial-fixnum lieb-num-initial-epsilon
               lieb-num-terminal-fixnum lieb-num-terminal-epsilon
               fixnum- epsilon- liebcreatesyms))

 (cond

  ((and (typep lieb1 'lieb-num) (typep lieb2 'lieb-num))
  (let ((temp-liebnum nil))
    (and (not *listof-symbols*) (liebcreatesyms *default-numof-symstocreate*))
    (setq temp-liebnum (car *listof-symbols*))
    (setq *listof-symbols* (cdr *listof-symbols*))


    (setf (lieb-num-initial-fixnum  temp-liebnum)
     (fixnum-
       (lieb-num-initial-fixnum lieb1)
       (lieb-num-initial-fixnum lieb2)))



    (setf (lieb-num-initial-epsilon  temp-liebnum)
```

```
    (epsilon-
      (lieb-num-initial-epsilon lieb1)
      (lieb-num-initial-epsilon lieb2)))


   (setf (lieb-num-terminal-fixnum  temp-liebnum)
    (fixnum-
      (lieb-num-terminal-fixnum lieb1)
      (lieb-num-terminal-fixnum lieb2)))


   (setf (lieb-num-terminal-epsilon   temp-liebnum)
    (epsilon-
      (lieb-num-terminal-epsilon lieb1)
      (lieb-num-terminal-epsilon lieb2)))


    temp-liebnum))


  ;return error on incorrect parameter
  (t (error "LIEB:LIEB-LIEB Incorrect parameter type"))))




;function to add a lieb-num and a fixnum
;type check performed to verify correct parameters
(defun lieb+fixnum (lieb1 fx1)
 (declare (optimize (speed 3) (space 3))
        (inline lieb-num-initial-fixnum lieb-num-initial-epsilon
                lieb-num-terminal-fixnum lieb-num-terminal-epsilon
                fixnum+ liebcreatesyms))


 (cond

  ((and (typep lieb1 'lieb-num) (typep fx1 'integer))
```

```
 (let ((temp-liebnum nil))
  (and (not *listof-symbols*) (liebcreatesyms *default-numof-symstocreate*))
  (setq temp-liebnum (car *listof-symbols*))
  (setq *listof-symbols* (cdr *listof-symbols*))


  (setf (lieb-num-initial-fixnum  temp-liebnum)
   (fixnum+
     (lieb-num-initial-fixnum lieb1) fx1))


  (setf (lieb-num-initial-epsilon  temp-liebnum)
   (lieb-num-initial-epsilon lieb1))


  (setf (lieb-num-terminal-fixnum  temp-liebnum)
   (fixnum+ (lieb-num-terminal-fixnum lieb1) fx1))


  (setf (lieb-num-terminal-epsilon  temp-liebnum)
   (lieb-num-terminal-fixnum lieb1))


   temp-liebnum))



 ;return error on incorrect parameter
 (t (error "LIEB:LIEB+FIXNUM Incorrect parameter type"))))




;function to subtract a lieb-num and a fixnum
;type check performed to verify correct parameters
(defun lieb-fixnum (lieb1 fx1)
 (declare (optimize (speed 3) (space 3))
        (inline lieb-num-initial-fixnum lieb-num-initial-epsilon
              lieb-num-terminal-fixnum lieb-num-terminal-epsilon
              fixnum- liebcreatesyms))
```

```
(cond

 ((and (typep lieb1 'lieb-num) (typep fx1 'integer))
 (let ((temp-liebnum nil))
  (and (not *listof-symbols*) (liebcreatesyms *default-numof-symstocreate*))
  (setq temp-liebnum (car *listof-symbols*))
  (setq *listof-symbols* (cdr *listof-symbols*))


  (setf (lieb-num-initial-fixnum  temp-liebnum)
   (fixnum-
     (lieb-num-initial-fixnum lieb1) fx1))



  (setf (lieb-num-initial-epsilon  temp-liebnum)
   (lieb-num-initial-epsilon lieb1))



  (setf (lieb-num-terminal-fixnum  temp-liebnum)
   (fixnum-
     (lieb-num-terminal-fixnum lieb1) fx1))



  (setf (lieb-num-terminal-epsilon  temp-liebnum)
   (lieb-num-terminal-epsilon lieb1))


   temp-liebnum))



 ;return error on incorrect parameter
 (t (error "LIEB:LIEB+FIXNUM Incorrect parameter type'))))




;function to determine if one lieb-num is less than another lieb-num
;type check performed to verify correct parameters
(defun lieb1<lieb2 (lieb1 lieb2)
```

```
 (declare (optimize (speed 3) (space 3))
        (inline lieb-num-initial-fixnum lieb-num-initial-epsilon
              lieb-num-terminal-fixnum lieb-num-terminal-epsilon))



(cond
 ((and (typep lieb1 'lieb-num) (typep lieb2 'lieb-num))
  (let ((lieb2.fx1 (lieb-num-initial-fixnum lieb2)))
    (or
      (and
        (< (lieb-num-initial-fixnum lieb1) lieb2.fx1)
        (< (lieb-num-terminal-fixnum lieb1) lieb2.fx1)
        (< (lieb-num-terminal-epsilon lieb1)
           (lieb-num-initial-epsilon lieb2))
        t)

      nil)))



 ;return error on incorrect parameter
 (t (error "LIEB:LIEB1<LIEB2 Incorrect parameter type"))))




;function to determine if one lieb-num subsumes another lieb-num
;type check performed to verify correct parameters
(defun lieb1subsumeslieb2 (lieb1 lieb2)
  (declare (optimize (speed 3) (space 3))
        (inline lieb-num-initial-fixnum lieb-num-initial-epsilon
              lieb-num-terminal-fixnum lieb-num-terminal-epsilon))



 (cond
  ((and (typep lieb1 'lieb-num) (typep lieb2 'lieb-num))
   (and
     (or
       (< (lieb-num-initial-fixnum lieb1)
          (lieb-num-initial-fixnum lieb2))
```

```
      (and
        (<= (lieb-num-initial-fixnum lieb1)
            (lieb-num-initial-fixnum lieb2))
        (<= (lieb-num-initial-epsilon lieb1)
            (lieb-num-initial-epsilon lieb2))))
     (or
      (> (lieb-num-terminal-fixnum lieb1)
         (lieb-num-terminal-fixnum lieb2))

      (and
        (>= (lieb-num-terminal-fixnum lieb1)
            (lieb-num-terminal-fixnum lieb2))

        (>= (lieb-num-terminal-epsilon lieb1)
            (lieb-num-terminal-epsilon lieb2))))))


  ;return error on incorrect parameter
  (t (error "LIEB:LIEB1SUBSUMESLIEB2 Incorrect parameter type"))))




;function to determine approximate span of interval, epsilons disregarded
;type checks performed to verify correct parameter type
(defun liebrealinterval (lieb)
 (declare (optimize (speed 3) (space 3))
          (inline lieb-num-initial-fixnum lieb-num-terminal-fixnum))

 (cond
  ((typep lieb 'lieb-num)

   (fixnum- (lieb-num-terminal-fixnum lieb)
            (lieb-num-initial-fixnum lieb)))


  ;return error on incorrect parameter
  (t (error "LIEB:LIEBREALINTERVAL Incorrect parameter type"))))
```

```
;function to create original and additional liebnums as needed
(defun liebcreatesyms (numof-liebs-tocreate)
  (declare (optimize (speed 3) (space 3)))
  (let ((tempsym nil))
  (do ((counter 1 (1+ counter)))

      ((> counter numof-liebs-tocreate) *listof-symbols*)

      ;(setq tempsym (gensym "liebnum"))
      (setq tempsym (make-lieb-num 0 0 0 0))
      (cond (*listof-symbols* (nconc *listof-symbols* (list  tempsym)))

            (t  (setq *listof-symbols* (list  tempsym)))))))




;function to return liebnum of available list
(defun return-liebnum (liebnum)
  (declare (optimize (speed 3) (space 3)))
  (nconc *listof-symbols* (list liebnum)))



;function to get next available liebnum
(defun get-next-liebnum ()
  (declare (optimize (speed 3) (space 3)))
  (let ((next-symbol nil))
      (cond
                        (*listof-symbols* (setq next-symbol (car *listof-symbols*)))
                        (t (liebcreatesyms *default-numof-symstocreate*) (setq next-symbol (c
      (setq *listof-symbols* (cdr *listof-symbols*))
      next-symbol))


;;*********************************************************************
;
;
;structure definitions and routines to compare FORBNUMS with LIEBNUMS
;
;
;;*********************************************************************
;FORBNUMS
```

```
;
;structure definition for FORBNUM, utilizes defstruct definition from
;FORBIN system

(defstruct (forb-num
          (:print "[ a,  a]"
            (print-forbnum (forb-num-initial-fixnum forb-num))
            (print-forbnum (forb-num-terminal-fixnum forb-num)))
                        (:constructor make-forb-num (initial-fixnum terminal-fixnum)))

  ;forb-num item definitions
  (initial-fixnum 0 :type integer)
  (terminal-fixnum *most-positive-fixnum* :type integer))



;function to print/output forb-num item
(defun print-forbnum (item)
  (declare (optimize (speed 3) (space 3)))
  item)




;function to add 2 forb-nums
;type check performed to verify correct parameters

(defun forb+forb (forb1 forb2)
 (declare (optimize (speed 3) (space 3))
         (inline forb-num-initial-fixnum forb-num-terminal-fixnum
                 fixnum+ forbcreatesyms))

 (cond
  ((and (typep forb1 'forb-num) (typep forb2 'forb-num))
   (let ((temp-forbnum nil))
    (and (not *listof-symbols*) (forbcreatesyms *default-numof-symstocreate*))
    (setq temp-forbnum (car *listof-symbols*))
    (setq *listof-symbols* (cdr *listof-symbols*))


    (setf (forb-num-initial-fixnum  temp-forbnum)
     (fixnum+
       (forb-num-initial-fixnum forb1)
       (forb-num-initial-fixnum forb2)))
```

```
   (setf (forb-num-terminal-fixnum  temp-forbnum)
    (fixnum+
       (forb-num-terminal-fixnum forb1)
       (forb-num-terminal-fixnum forb2)))


    temp-forbnum))


  ;return error on incorrect parameter
  (t (error "FORB:FORB+FORB Incorrect parameter type"))))




;function to subtract 2 forb-nums
;type check performed to verify correct parameters
(defun forb-forb (forb1 forb2)
  (declare (optimize (speed 3) (space 3))
          (inline forb-num-initial-fixnum forb-num-terminal-fixnum
                  fixnum- forbcreatesyms))

 (cond

  ((and (typep forb1 'forb-num) (typep forb2 'forb-num))
  (let ((temp-forbnum nil))
    (and (not *listof-symbols*) (forbcreatesyms *default-numof-symstocreate*))
    (setq temp-forbnum (car *listof-symbols*))
    (setq *listof-symbols* (cdr *listof-symbols*))


    (setf (forb-num-initial-fixnum  temp-forbnum)
     (fixnum-
       (forb-num-initial-fixnum forb1)
       (forb-num-initial-fixnum forb2)))



    (setf (forb-num-terminal-fixnum  temp-forbnum)
     (fixnum-
       (forb-num-terminal-fixnum forb1)
       (forb-num-terminal-fixnum forb2)))
```

```
    temp-forbnum))


  ;return error on incorrect parameter
  (t (error "FORB:FORB-FORB Incorrect parameter type"))))



;function to add a forb-num and a fixnum
;type check performed to verify correct parameters
(defun forb+fixnum (forb1 fx1)
 (declare (optimize (speed 3) (space 3))
        (inline forb-num-initial-fixnum forb-num-terminal-fixnum
              fixnum+ forbcreatesyms))

 (cond

  ((and (typep forb1 'forb-num) (typep fx1 'integer))
  (let ((temp-forbnum nil))
    (and (not *listof-symbols*) (forbcreatesyms *default-numof-symstocreate*))
    (setq temp-forbnum (car *listof-symbols*))
    (setq *listof-symbols* (cdr *listof-symbols*))


    (setf (forb-num-initial-fixnum  temp-forbnum)
      (fixnum+
        (forb-num-initial-fixnum forb1) fx1))


    (setf (forb-num-terminal-fixnum  temp-forbnum)
      (fixnum+ (forb-num-terminal-fixnum forb1) fx1))



    temp-forbnum))


  ;return error on incorrect parameter
  (t (error "FORB:FORB+FIXNUM Incorrect parameter type"))))
```

```
;function to subtract a forb-num and a fixnum
;type check performed to verify correct parameters
(defun forb-fixnum (forb1 fx1)
  (declare (optimize (speed 3) (space 3))
         (inline forb-num-initial-fixnum forb-num-terminal-fixnum
               fixnum- forbcreatesyms))


 (cond

  ((and (typep forb1 'forb-num) (typep fx1 'integer))
  (let ((temp-forbnum nil))
    (and (not *listof-symbols*) (forbcreatesyms *default-numof-symstocreate*))
    (setq temp-forbnum (car *listof-symbols*))
    (setq *listof-symbols* (cdr *listof-symbols*))


    (setf (forb-num-initial-fixnum  temp-forbnum)
      (fixnum-
        (forb-num-initial-fixnum forb1) fx1))


    (setf (forb-num-terminal-fixnum  temp-forbnum)
      (fixnum-
        (forb-num-terminal-fixnum forb1) fx1))



     temp-forbnum))


  ;return error on incorrect parameter
  (t (error "FORB:FORB-FIXNUM Incorrect parameter type"))))


;function to determine if one forb-num is less than another forb-num
;type check performed to verify correct parameters
(defun forb1<forb2 (forb1 forb2)
  (declare (optimize (speed 3) (space 3))
         (inline forb-num-initial-fixnum forb-num-terminal-fixnum))
```

```
(cond
 ((and (typep forb1 'forb-num) (typep forb2 'forb-num))
  (let ((forb2.fx1 (forb-num-initial-fixnum forb2)))
   (or
    (and
     (< (forb-num-initial-fixnum forb1) forb2.fx1)

     (< (forb-num-terminal-fixnum forb1) forb2.fx1)

     t)

    nil)))


 ;return error on incorrect parameter
 (t (error "FORB:FORB1<forb2 Incorrect parameter type"))))



;function to determine if one forb-num subsumes another forb-num
;type check performed to verify correct parameters
(defun forb1subsumesforb2 (forb1 forb2)
 (declare (optimize (speed 3) (space 3))
        (inline forb-num-initial-fixnum forb-num-terminal-fixnum))



(cond
 ((and (typep forb1 'forb-num) (typep forb2 'forb-num))

  (and
      (<= (forb-num-initial-fixnum forb1)
         (forb-num-initial-fixnum forb2))

      (>= (forb-num-terminal-fixnum forb1)
         (forb-num-terminal-fixnum forb2))
      t))


 ;return error on incorrect parameter
 (t (error "FORB:FORB1SUBSUMESFORB2 Incorrect parameter type"))))
```

```
;function to determine approximate span of interval
;type checks performed to verify correct parameter type
(defun forbrealinterval (forb)
 (declare (optimize (speed 3) (space 3))
        (inline forb-num-initial-fixnum forb-num-terminal-fixnum))

 (cond
  ((typep forb 'forb-num)
   (fixnum- (forb-num-terminal-fixnum forb)
         (forb-num-initial-fixnum forb)))

  ;return error on incorrect parameter
  (t (error "FORB:FORBREALINTERVAL Incorrect parameter type"))))



;function to create original and additional forbnums as needed
(defun forbcreatesyms (numof-forbs-tocreate)

  (declare (optimize (speed 3) (space 3)))

  (let ((tempsym nil))
  (do ((counter 1 (1+ counter)))

     ((> counter numof-forbs-tocreate) *listof-symbols*)

     ;(setq tempsym (gensym "forbnum"))
     (setq tempsym (make-forb-num 0 0))
     (cond (*listof-symbols* (nconc *listof-symbols* (list  tempsym)))

          (t  (setq *listof-symbols* (list  tempsym)))))))



;function to return forbnum of available list
(defun return-forbnum (forbnum)
  (declare (optimize (speed 3) (space 3)))
  (nconc *listof-symbols* (list forbnum)))



;function to get next available forbnum
```

```
(defun get-next-forbnum ()
  (declare (optimize (speed 3) (space 3)))
  (let ((next-symbol nil))
      (cond
                        (*listof-symbols* (setq next-symbol (car *listof-symbols*)))
                        (t (forbcreatesyms *default-numof-symstocreate*) (setq next-symbol (
      (setq *listof-symbols* (cdr *listof-symbols*))
      next-symbol))




;;*****************************************************************************
;
;
;structure definitions and routines to compare FLOFORBNUMS with LIEBNUMS
;
;
;;*****************************************************************************
;FLOFORBNUM

;defines largest value
(defparameter *most-positive-floatnum* most-positive-short-float)

;defines smallest value, defined for future use
(defparameter *most-negative-floatnum* most-negative-short-float)



;
;structure definition for FLOFORBNUM, utilizes defstruct definition from
;FORBIN system

(defstruct (floforb-num
          (:print "[ a,  a]"
            (print-floforbnum (floforb-num-initial-flonum floforb-num))
            (print-floforbnum (floforb-num-terminal-flonum floforb-num)))
                        (:constructor make-floforb-num (initial-flonum terminal-flonum)))

  ;floforb-num item definitions
  (initial-flonum 0.0 :type short-float)
  (terminal-flonum *most-positive-short-float* :type short-float))
```

```
;function to print/output floforb-num item
(defun print-floforbnum (item)
  (declare (optimize (speed 3) (space 3)))
  item)



;function to add 2 flonums
;type/limit checks performed to verify parameter correctness/legality
(defun flonum+ (fx1 fx2)
 (declare (optimize (speed 3) (space 3)))
 (cond

  ((and (typep fx1 'short-float) (typep fx2 'short-float))
   (let ((result (+ fx1 fx2)))
     (or
       (and (< result *most-positive-floatnum*) result)
       *most-positive-floatnum*)))

  ;return error on incorrect parameter
  (t (error "FLOFORB:FLONUM+ Incorrect parameter type"))))



;function to subtract 2 flonums
;type/limit checks performed to verify parameter correctness/legality
(defun flonum- (fx1 fx2)
 (declare (optimize (speed 3) (space 3)))
 (cond
  ((and (typep fx1 'short-float) (typep fx2 'short-float))
   (let ((result (- fx1 fx2)))
    (or
       (and (plusp result) result)
       0.0)))


  ;return error on incorrect parameter
  (t (error "FLOFORB:FLONUM- Incorrect parameter type"))))



;function to add 2 floforb-nums
```

```
;type check performed to verify correct parameters

(defun floforb+floforb (floforb1 floforb2)
 (declare (optimize (speed 3) (space 3))
        (inline floforb-num-initial-flonum floforb-num-terminal-flonum
              flonum+ floforbcreatesyms))

 (cond
  ((and (typep floforb1 'floforb-num) (typep floforb2 'floforb-num))
   (let ((temp-floforbnum nil))
    (and (not *listof-symbols*) (floforbcreatesyms *default-numof-symstocreate*))
    (setq temp-floforbnum (car *listof-symbols*))
    (setq *listof-symbols* (cdr *listof-symbols*))


    (setf (floforb-num-initial-flonum  temp-floforbnum)
     (flonum+
       (floforb-num-initial-flonum floforb1)
       (floforb-num-initial-flonum floforb2)))

    (setf (floforb-num-terminal-flonum  temp-floforbnum)
     (flonum+
       (floforb-num-terminal-flonum floforb1)
       (floforb-num-terminal-flonum floforb2)))


     temp-floforbnum))


  ;return error on incorrect parameter
  (t (error "FLOFORB:FLOFORB+FLOFORB Incorrect parameter type"))))




;function to subtract 2 floforb-nums
;type check performed to verify correct parameters
(defun floforb-floforb (floforb1 floforb2)
  (declare (optimize (speed 3) (space 3))
         (inline floforb-num-initial-flonum floforb-num-terminal-flonum
               flonum- floforbcreatesyms))

 (cond

  ((and (typep floforb1 'floforb-num) (typep floforb2 'floforb-num))
  (let ((temp-floforbnum nil))
```

```
  (and (not *listof-symbols*) (floforbcreatesyms *default-numof-symstocreate*))
  (setq temp-floforbnum (car *listof-symbols*))
  (setq *listof-symbols* (cdr *listof-symbols*))


  (setf (floforb-num-initial-flonum  temp-floforbnum)
   (flonum-
     (floforb-num-initial-flonum floforb1)
     (floforb-num-initial-flonum floforb2)))



  (setf (floforb-num-terminal-flonum  temp-floforbnum)
   (flonum-
     (floforb-num-terminal-flonum floforb1)
     (floforb-num-terminal-flonum floforb2)))



   temp-floforbnum))


 ;return error on incorrect parameter
 (t (error "FLOFORB:FLORORB-FLOFORB Incorrect parameter type"))))




;function to add a floforb-num and a flonum
;type check performed to verify correct parameters
(defun floforb+flonum (floforb1 fx1)

 (declare (optimize (speed 3) (space 3))

        (inline floforb-num-initial-flonum floforb-num-terminal-flonum
              flonum+ floforbcreatesyms))



 (cond

  ((and (typep floforb1 'floforb-num) (typep fx1 'short-float))
  (let ((temp-floforbnum nil))
    (and (not *listof-symbols*) (floforbcreatesyms *default-numof-symstocreate*))
```

```
   (setq temp-floforbnum (car *listof-symbols*))
   (setq *listof-symbols* (cdr *listof-symbols*))


   (setf (floforb-num-initial-flonum  temp-floforbnum)
     (flonum+
       (floforb-num-initial-flonum floforb1) fx1))


   (setf (floforb-num-terminal-flonum  temp-floforbnum)
     (flonum+ (floforb-num-terminal-flonum floforb1) fx1))



    temp-floforbnum))


  ;return error on incorrect parameter
  (t (error "FLOFORB:FLOFORB+FLONUM Incorrect parameter type"))))



;function to subtract a floforb-num and flonum
;type check performed to verify correct parameters
(defun floforb-flonum (floforb1 fx1)
  (declare (optimize (speed 3) (space 3))
         (inline floforb-num-initial-flonum floforb-num-terminal-flonum
                 flonum- floforbcreatesyms))



 (cond

  ((and (typep floforb1 'floforb-num) (typep fx1 'short-float))
  (let ((temp-floforbnum nil))
    (and (not *listof-symbols*) (floforbcreatesyms *default-numof-symstocreate*))
    (setq temp-floforbnum (car *listof-symbols*))
    (setq *listof-symbols* (cdr *listof-symbols*))


    (setf (floforb-num-initial-flonum  temp-floforbnum)
      (flonum-
        (floforb-num-initial-flonum floforb1) fx1))
```

```
  (setf (floforb-num-terminal-flonum  temp-floforbnum)
    (flonum-
      (floforb-num-terminal-flonum floforb1) fx1))



   temp-floforbnum))


 ;return error on incorrect parameter
 (t (error "FLOFORB:FLOFORB-FLONUM Incorrect parameter type"))))


;function to determine if one floforb-num is less than another floforb-num
;type check performed to verify correct parameters
(defun floforb1<floforb2 (floforb1 floforb2)
  (declare (optimize (speed 3) (space 3))
           (inline floforb-num-initial-flonum floforb-num-terminal-flonum))


 (cond
  ((and (typep floforb1 'floforb-num) (typep floforb2 'floforb-num))
    (let ((floforb2.fx1 (floforb-num-initial-flonum floforb2)))
      (or
        (and
          (< (floforb-num-initial-flonum floforb1) floforb2.fx1)

          (< (floforb-num-terminal-flonum floforb1) floforb2.fx1)
          t)
        nil)))

  ;return error on incorrect parameter
  (t (error "FLOFORB:FLOFORB1<FLOFORB2 Incorrect parameter type"))))



;function to determine if one floforb-num subsumes another floforb-num
;type check performed to verify correct parameters
(defun floforb1subsumesfloforb2 (floforb1 floforb2)
```

```
 (declare (optimize (speed 3) (space 3))
        (inline floforb-num-initial-flonum floforb-num-terminal-flonum))


(cond
 ((and (typep floforb1 'floforb-num) (typep floforb2 'floforb-num))

  (and
       (<= (floforb-num-initial-flonum floforb1)
          (floforb-num-initial-flonum floforb2))

       (>= (floforb-num-terminal-flonum floforb1)
          (floforb-num-terminal-flonum floforb2))
       t))


  ;return error on incorrect parameter
  (t (error "FLOFORB:FLOFORB1SUBSUMESFLOFORB2 Incorrect parameter type"))))



;function to determine approximate span of interval
;type checks performed to verify correct parameter type
(defun floforbrealinterval (floforb)
 (declare (optimize (speed 3) (space 3))
        (inline floforb-num-initial-flonum floforb-num-terminal-flonum))

 (cond
  ((typep floforb 'floforb-num)
   (flonum- (floforb-num-terminal-flonum floforb)
          (floforb-num-initial-flonum floforb)))

  ;return error on incorrect parameter
  (t (error "FLOFORB:FLOFORBREALINTERVAL Incorrect parameter type"))))



;function to create original and additional floforbnums as needed
(defun floforbcreatesyms (numof-floforbs-tocreate)

  (declare (optimize (speed 3) (space 3)))

  (let ((tempsym nil))
```

```
  (do ((counter 1 (1+ counter)))

      ((> counter numof-floforbs-tocreate) *listof-symbols*)

      ;(setq tempsym (gensym "floforbnum"))
      (setq tempsym (make-floforb-num 0.0 0.0))
      (cond (*listof-symbols* (nconc *listof-symbols* (list  tempsym)))

            (t  (setq *listof-symbols* (list  tempsym)))))))




;function to return floforbnum of available list
(defun return-floforbnum (floforbnum)
  (declare (optimize (speed 3) (space 3)))
  (nconc *listof-symbols* (list floforbnum)))



;function to get next available floforbnum
(defun get-next-floforbnum ()
  (declare (optimize (speed 3) (space 3)))
  (let ((next-symbol nil))
       (cond
                          (*listof-symbols* (setq next-symbol (car *listof-symbols*)))
                          (t (floforbcreatesyms *default-numof-symstocreate*) (setq next-symbo
       (setq *listof-symbols* (cdr *listof-symbols*))
       next-symbol))




;;*****************************************************************************
;;
;
;
;structure definitions and routines to compare ZZNUMS with LIEBNUMS
;
;
;;*****************************************************************************
```

```
;ZZNUM

(defparameter *pos-inf*  '*pos-inf*  "positive infinity")
(defparameter *neg-inf*  '*neg-inf*  "negative infinity")
(defparameter *pos-tiny* '*pos-tiny* "an infinitesimal positive quantity")
(defparameter *neg-tiny* '*neg-tiny* "an infinitesimal negative quantity")
(defparameter *pos-inf-fixnum-approximation* most-positive-fixnum "a very big positive nun
(defparameter *neg-inf-fixnum-approximation* most-negative-fixnum "a very big negative nu

;;;
;;; The SYM type
;;;


(deftype sym () '(or integer (member *pos-inf* *neg-inf* *pos-tiny* *neg-tiny*)))

(defun sym-p (x) (typep x 'sym))


;;;
;;; the ZZ type.
;;;

(defstruct (zz (:print "[ a,  a]"
               (print-zznum (zz-low zz))
               (print-zznum (zz-high zz)))
                              (:constructor make-zz (low high)))
  "A fuzzy number, i.e. a closed interval"
  (low 0 :type sym)
  (high 0 :type sym))


;function to print/output zz-num item
(defun print-zznum (item)
  (declare (optimize (speed 3) (space 3)))
  item)



(defun zzrealinterval (z)
 (declare (optimize (speed 3) (space 3))
        (inline zz-low zz-high sym-))
(cond

  ((and (typep z 'zz) (typep z 'zz))
```

```
  (sym- (zz-high z) (zz-low z)))


 ;return error on incorrect parameter
 (t (error "ZZ:ZZREALINTERVAL Incorrect parameter type"))))



;;;
;;; Here are some constant fuzzy numbers
;;;
;;;    See the comments above regarding the use of defparameter.
;;;

(defparameter *zz-zero* (make-zz 0 0) "the interval [0, 0]")
(defparameter *zz-unconstrained* (make-zz *neg-inf* *pos-inf*) "any number at all")
(defparameter *zz-before* (make-zz *pos-tiny* *pos-inf*) "any positive number")
(defparameter *zz-not-after* (make-zz 0 *pos-inf*) "any nonnegative number")



(defun zz+ (z1 z2)
 (declare (optimize (speed 3) (space 3))
        (inline zz-low zz-high sym+ zzcreatesyms))
 (cond

  ((and (typep z1 'zz) (typep z2 'zz))
    (let ((temp-zznum nil))
    (and (not *listof-symbols*) (zzcreatesyms *default-numof-symstocreate*))
    (setq temp-zznum (car *listof-symbols*))
    (setq *listof-symbols* (cdr *listof-symbols*))


    (setf (zz-low  temp-zznum) (sym+ (zz-low z1) (zz-low z2)))
    (setf (zz-high  temp-zznum) (sym+ (zz-high z1) (zz-high z2)))

    temp-zznum))

 ;return error on incorrect parameter
 (t (error "ZZ:ZZ+ Incorrect parameter type"))))
```

```
(defun zz- (z1 z2)
 (declare (optimize (speed 3) (space 3))
        (inline zz-low zz-high sym- zzcreatesyms))
 (cond
  ((and (typep z1 'zz) (typep z2 'zz))
   (let ((temp-zznum nil))
   (and (not *listof-symbols*) (zzcreatesyms *default-numof-symstocreate*))
   (setq temp-zznum (car *listof-symbols*))
   (setq *listof-symbols* (cdr *listof-symbols*))

   (setf (zz-low  temp-zznum) (sym- (zz-low z1) (zz-low z2)))
   (setf (zz-high  temp-zznum) (sym- (zz-high z1) (zz-high z2)))

   temp-zznum))

  ;return error on incorrect parameter
  (t (error "ZZ:ZZ- Incorrect parameter type"))))



(defun zz+sym (z1 s1)
 (declare (optimize (speed 3) (space 3))
        (inline zz-low zz-high sym+ zzcreatesyms))
 (cond

  ((and (typep z1 'zz) (typep s1 'sym))
    (let ((temp-zznum nil))
    (and (not *listof-symbols*) (zzcreatesyms *default-numof-symstocreate*))
    (setq temp-zznum (car *listof-symbols*))
    (setq *listof-symbols* (cdr *listof-symbols*))


    (setf (zz-low  temp-zznum) (sym+ (zz-low z1) s1))
    (setf (zz-high  temp-zznum) (sym+ (zz-high z1) s1))

    temp-zznum))

  ;return error on incorrect parameter
  (t (error "ZZ:ZZ+SYM Incorrect parameter type"))))



(defun zz-sym (z1 s1)
```

```
(declare (optimize (speed 3) (space 3))
        (inline zz-low zz-high sym- zzcreatesyms))
(cond
 ((and (typep z1 'zz) (typep s1 'sym))
  (let ((temp-zznum nil))
  (and (not *listof-symbols*) (zzcreatesyms *default-numof-symstocreate*))
  (setq temp-zznum (car *listof-symbols*))
  (setq *listof-symbols* (cdr *listof-symbols*))

  (setf (zz-low  temp-zznum) (sym- (zz-low z1) s1))
  (setf (zz-high  temp-zznum) (sym- (zz-high z1) s1))

  temp-zznum))

 ;return error on incorrect parameter
 (t (error "ZZ:ZZ-SYM Incorrect parameter type"))))
```

```
(defun zz-neg (z1)
 (declare (optimize (speed 3) (space 3))
        (inline zz-low zz-high sym-neg zzcreatesyms))
 (cond

  ((typep z1 'zz)
   (let ((temp-zznum nil))
    (and (not *listof-symbols*) (zzcreatesyms *default-numof-symstocreate*))
    (setq temp-zznum (car *listof-symbols*))
    (setq *listof-symbols* (cdr *listof-symbols*))


   (setf (zz-low  temp-zznum) (sym-neg (zz-low z1)))
   (setf (zz-high  temp-zznum) (sym-neg (zz-high z1)))

   temp-zznum))

 ;return error on incorrect parameter
 (t (error "ZZ:ZZ-NEG Incorrect parameter type"))))
```

```
(defun z1subsumesz2 (z1 z2)
 (declare (optimize (speed 3) (space 3))
        (inline zz-low zz-high sym=<))


  (cond
   ((and (typep z1 'zz) (typep z2 'zz))
      (sym=< (zz-low z1) (zz-low z2))
      (sym=< (zz-high z2) (zz-high z1)))



 ;return error on incorrect parameter
  (t (error "ZZ:Z1SUBSUMESZ2 Incorrect parameter type"))))


(defun z1<z2 (z1 z2)
 (declare (optimize (speed 3) (space 3))
        (inline zz-low zz-high sym<))


  (cond
   ((and (typep z1 'zz) (typep z2 'zz))
      (sym< (zz-low z1) (zz-low z2))
      (sym=< (zz-high z1) (zz-low z2)))

 ;return error on incorrect parameter
  (t (error "ZZ:Z1<Z2 Incorrect parameter type"))))







(defun sym+ (x y)
 (declare (optimize (speed 3) (space 3)))
;(prin1 "x  ")
;(prin1 x)
;(prin1 "y   ")
;(prin1 y)
;(terpri)

 (cond ((and (typep x 'sym) (typep y 'sym))
       (cond ((and (integerp x) (integerp y)) (+ x y))
```

```
                        ((or (and (eql x *pos-inf*) (eql y *neg-inf*))
                             (and (eql y *pos-inf*) (eql x *neg-inf*)))
                             0)
                        ((or (eql x *pos-inf*) (eql y *pos-inf*)) *pos-inf*)
                        ((eql x 0) y)
                        ((eql y 0) x)
                        ((eql x *neg-inf*) *neg-inf*)
                        ((eql y *neg-inf*) *neg-inf*)
                        ((eql x *pos-tiny*)
                         (cond ((eql y *pos-tiny*) *pos-tiny*) ((eql y *neg-tiny*) 0) (t y)))
                        ((eql y *pos-tiny*) (cond ((eql x *neg-tiny*) 0) (t x)))
                        ((eql x *neg-tiny*) (cond ((eql y *neg-tiny*) *neg-tiny*) (t y)))
                        (t x)))

  ;return error on incorrect parameter
  (t (error "ZZ:SYM+ Incorrect parameter type"))))



(defun sym- (x y)
 (declare (optimize (speed 3) (space 3))
          (inline sym+ sym-neg))

 (cond ((and (typep x 'sym) (typep y 'sym))
        (sym+ x (sym-neg y)))


  ;return error on incorrect parameter
  (t (error "ZZ:SYM- Incorrect parameter type"))))




(defun sym-neg (x)
 (declare (optimize (speed 3) (space 3)))

 (cond ( (typep x 'sym)
        (cond ((integerp x) (- 0 x))
                        ((eql x *pos-inf*) *neg-inf*)
                        ((eql x *neg-inf*) *pos-inf*)
                        ((eql x *pos-tiny*) *neg-tiny*)
```

```
                        ((eql x *neg-tiny*) *pos-tiny*)
                        (t (error "Expecting symbolic integer:  s" x))))

 ;return error on incorrect parameter
 (t (error "ZZ:SYM-NEG Incorrect parameter type"))))
```

```
(defun sym< (x y)
 (declare (optimize (speed 3) (space 3)))

 (cond ((and (typep x 'sym) (typep y 'sym))
        (cond ((and (integerp x) (integerp y)) (< x y))
                        ((eql x y) nil)
                        ((eql y *pos-inf*) t)
                        ((eql x *neg-inf*) t)
                        ((eql y *neg-inf*) nil)
                        ((eql x *pos-inf*) nil)
                        ((eql x *pos-tiny*) (and (integerp y) (> y 0)))
                        ((eql y *pos-tiny*) (or (eql x *neg-tiny*) (<= x 0)))
                        ((eql x *neg-tiny*) (and (integerp y) (>= y 0)))
                        ((eql y *neg-tiny*) (< x 0))))

 ;return error on incorrect parameter
 (t (error "ZZ:SYM< Incorrect parameter type"))))
```

```
(defun sym= (x y)
 (declare (optimize (speed 3) (space 3)))

 (cond ((and (typep x 'sym) (typep y 'sym))
        (eql x y))

 ;return error on incorrect parameter
 (t (error "ZZ:SYM= Incorrect parameter type"))))
```

```
(defun sym>= (x y)
 (declare (optimize (speed 3) (space 3))
        (inline sym<))

 (cond ((and (typep x 'sym) (typep y 'sym))
       (or (eql x y) (sym< y x)))

 ;return error on incorrect parameter
 (t (error "ZZ:SYM>= Incorrect parameter type"))))
```

```
(defun sym=< (x y)
 (declare (optimize (speed 3) (space 3))
        (inline sym<))

 (cond ((and (typep x 'sym) (typep y 'sym))
       (or (eql x y) (sym< x y)))

 ;return error on incorrect parameter
 (t (error "ZZ:SYM=< Incorrect parameter type"))))
```

```
(defun sym> (x y)
 (declare (optimize (speed 3) (space 3))
        (inline sym<))

 (cond ((and (typep x 'sym) (typep y 'sym))
       (sym< y x))

 ;return error on incorrect parameter
 (t (error "ZZ:SYM> Incorrect parameter type"))))
```

```
(defun sym-max (x y)
 (declare (optimize (speed 3) (space 3))
        (inline sym<))
```

```
(cond ((and (typep x 'sym) (typep y 'sym))
       (if (sym< x y) y x))

 ;return error on incorrect parameter
 (t (error "ZZ:SYM-MAX Incorrect parameter type"))))
```

```
(defun sym-min (x y)
 (declare (optimize (speed 3) (space 3)))

 (cond ((and (typep x 'sym) (typep y 'sym))
       (if (sym< x y) x y))

 ;return error on incorrect parameter
 (t (error "ZZ:SYM-MIN Incorrect parameter type"))))
```

```
(defun symnum->fixnum (x)
 (declare (optimize (speed 3) (space 3)))

 (cond ((typep x 'sym)
       (cond ((integerp x) x)
                          ((eql x *pos-inf*) *pos-inf-fixnum-approximation*)
                          ((eql x *neg-inf*) *neg-inf-fixnum-approximation*)
                          ((or (eql x *neg-tiny*) (eql x *pos-tiny*)) 0)
                          (t (error "Expecting symbolic FIXNUM:  s" x))))

 ;return error on incorrect parameter
 (t (error "ZZ:SYMNUM->FIXNUM Incorrect parameter type"))))
```

```
;function to create original and additional zznums as needed
(defun zzcreatesyms (numof-zzs-tocreate)

  (declare (optimize (speed 3) (space 3)))

  (let ((tempsym nil))
  (do ((counter 1 (1+ counter)))
```

```
    ((> counter numof-zzs-tocreate) *listof-symbols*)

    ;(setq tempsym (gensym "zznum"))
    (setq tempsym (make-zz 0 0))
    (cond (*listof-symbols* (nconc *listof-symbols* (list  tempsym)))

        (t  (setq *listof-symbols* (list  tempsym)))))))


;function to return zznum of available list
(defun return-zznum (zznum)
  (declare (optimize (speed 3) (space 3)))
  (nconc *listof-symbols* (list zznum)))


;function to get next available zznum
(defun get-next-zznum ()
  (declare (optimize (speed 3) (space 3)))
  (let ((next-symbol nil))
      (cond
                    (*listof-symbols* (setq next-symbol (car *listof-symbols*)))
                    (t (zzcreatesyms *default-numof-symstocreate*) (setq next-symbol (ca
      (setq *listof-symbols* (cdr *listof-symbols*))
      next-symbol))
```

```
;*****************************************************************************
;
;
;function to test LIEBNUMS
;
;
;
;*****************************************************************************
;
;
;
```

```
;
(defun testliebnums (numof-iterations)


;test case for evaluating representation
;
;
;setup temporal database
(let* ((light17-off (make-lieb-num 4 0 5 0))
     (light17-on (make-lieb-num 3 0 4 0))
     (fuse42-intact (make-lieb-num 2 0 *most-positive-fixnum* *most-positive-fixnum*))
     (fuse42-open (make-lieb-num 1 0 2 0))
     (switch-light17 (make-lieb-num 5 0 6 0))
     (light17-onagain (make-lieb-num 6 0 *most-positive-fixnum* *most-positive-fixnum*))

;
;build list of temporal events
;
     (eventslist '(,light17-off ,light17-on ,fuse42-intact
                  ,fuse42-open  ,switch-light17 ,light17-onagain))



;
;determine number of temporal events
;

     (numof-events (list-length eventslist)))

;
;
;define benchmark to test representation and associated functions


(do ((iterations 1 (1+ iterations)))
   ((> iterations numof-iterations) t)
   (do ((counter1 1 (1+ counter1)))
      ((> counter1 numof-events) t)
      (let ((temp1 nil) (temp2 nil) (factor 5))
         (setq temp1  (nth (1- counter1) eventslist))
         (do ((counter2 1 (1+ counter2)))
            ((> counter2 numof-events) t)
            (setq temp2 (nth (1- counter2) eventslist))

            (set (get-next-symbol) (lieb+lieb temp1 temp2))
```

```
(set (get-next-symbol) (lieb-lieb temp1 temp2))
(set (get-next-symbol) (lieb-lieb temp2 temp1))
(set (get-next-symbol) (lieb+fixnum temp1 factor))
(set (get-next-symbol) (lieb+fixnum temp2 factor))
(set (get-next-symbol) (lieb-fixnum temp1 factor))
(set (get-next-symbol) (lieb-fixnum temp2 factor))
(lieb1<lieb2 temp1 temp2)
(lieb1<lieb2 temp2 temp1)
(lieb1subsumeslieb2 temp1 temp2)
(lieb1subsumeslieb2 temp2 temp1)
(liebrealinterval temp1)
(liebrealinterval temp2)))))))
```

```
;*****************************************************************************
;
;
;function to test FORBNUMS
;
;
;
;*****************************************************************************
;
;
;
;
(defun testforbnums (numof-iterations)


;test case for evaluating representation
;
;
;setup temporal database
(let* ((light17-off (make-forb-num 4  5 ))
       (light17-on (make-forb-num 3  4 ))
       (fuse42-intact (make-forb-num 2  *most-positive-fixnum* ))
       (fuse42-open (make-forb-num 1  2 ))
       (switch-light17 (make-forb-num 5  6 ))
       (light17-onagain (make-forb-num 6  *most-positive-fixnum* ))

;
;build list of temporal events
```

```
;
     (eventslist '(,light17-off ,light17-on ,fuse42-intact
                   ,fuse42-open  ,switch-light17 ,light17-onagain))



;
;determine number of temporal events
;
     (numof-events (list-length eventslist)))


;
;define benchmark to test representation and associated functions
;
(do ((iterations 1 (1+ iterations)))
   ((> iterations numof-iterations) t)
   (do ((counter1 1 (1+ counter1)))
      ((> counter1 numof-events) t)
      (let ((temp1 nil) (temp2 nil) (factor 5))
         (setq temp1 (nth (1- counter1) eventslist))
         (do ((counter2 1 (1+ counter2)))
            ((> counter2 numof-events) t)
            (setq temp2 (nth (1- counter2) eventslist))

            (set (get-next-symbol) (forb+forb temp1 temp2))
            (set (get-next-symbol) (forb-forb temp1 temp2))
            (set (get-next-symbol) (forb-forb temp2 temp1))
            (set (get-next-symbol) (forb+fixnum temp1 factor))
            (set (get-next-symbol) (forb+fixnum temp2 factor))
            (set (get-next-symbol) (forb-fixnum temp1 factor))
            (set (get-next-symbol) (forb-fixnum temp2 factor))
            (forb1<forb2 temp1 temp2)
            (forb1<forb2 temp2 temp1)
            (forb1subsumesforb2 temp1 temp2)
            (forb1subsumesforb2 temp2 temp1)
            (forbrealinterval temp1)
            (forbrealinterval temp2)))))))



;*****************************************************************************
;
;
;
```

```
;function to test FLOFORBNUMS
;
;
;
;*****************************************************************************
;
;
;
;
(defun testfloforbnums (numof-iterations)



;test case for evaluating representation
;
;
;setup temporal database
(let* ((light17-off (make-floforb-num 4.0  5.0 ))
     (light17-on (make-floforb-num 3.0  4.0 ))
     (fuse42-intact (make-floforb-num 2.0  99999999999.0 ))
     (fuse42-open (make-floforb-num 1.0  2.0 ))
     (switch-light17 (make-floforb-num 5.0  6.0 ))
     (light17-onagain (make-floforb-num 6.0  99999999999.0 ))

;
;build list of temporal events
;
     (eventslist '(,light17-off ,light17-on ,fuse42-intact
                   ,fuse42-open  ,switch-light17 ,light17-onagain))



;
;determine number of temporal events
;
     (numof-events (list-length eventslist)))


;
;define benchmark to test representation and associated functions
;
(do ((iterations 1 (1+ iterations)))
   ((> iterations numof-iterations) t)
   (do ((counter1 1 (1+ counter1)))
      ((> counter1 numof-events) t)
      (let ((temp1 nil) (temp2 nil) (factor 5.0))
         (setq temp1  (nth (1- counter1) eventslist))
```

```
        (do ((counter2 1 (1+ counter2)))
            ((> counter2 numof-events) t)
            (setq temp2  (nth (1- counter2) eventslist))

            (set (get-next-symbol) (floforb+floforb temp1 temp2))
            (set (get-next-symbol) (floforb-floforb temp1 temp2))
            (set (get-next-symbol) (floforb-floforb temp2 temp1))
            (set (get-next-symbol) (floforb+flonum temp1 factor))
            (set (get-next-symbol) (floforb+flonum temp2 factor))
            (set (get-next-symbol) (floforb-flonum temp1 factor))
            (set (get-next-symbol) (floforb-flonum temp2 factor))
            (floforb1<floforb2 temp1 temp2)
            (floforb1<floforb2 temp2 temp1)
            (floforb1subsumesfloforb2 temp1 temp2)
            (floforb1subsumesfloforb2 temp2 temp1)
            (floforbrealinterval temp1)
            (floforbrealinterval temp2)))))))



;*******************************************************************************
;
;
;
;function to test ZZNUMS
;
;
;
;
;*******************************************************************************
;
;
;
;
;
(defun testzznums (numof-iterations)



;test case for evaluating representation
;
;
;setup temporal database
(let* ((light17-off (make-zz 4  5 ))
     (light17-on (make-zz 3  4 ))
     (fuse42-intact (make-zz 2  *pos-inf* ))
     (fuse42-open (make-zz 1  2 ))
     (switch-light17 (make-zz 5  6 ))
     (light17-onagain (make-zz 6  *pos-inf* ))
```

```
;
;build list of temporal events
;
     (eventslist '(,light17-off ,light17-on ,fuse42-intact
                   ,fuse42-open  ,switch-light17 ,light17-onagain))



;
;determine number of temporal events
;
     (numof-events (list-length eventslist)))


;
;define benchmark to test representation and associated functions
;
(do ((iterations 1 (1+ iterations)))
   ((> iterations numof-iterations) t)
   (do ((counter1 1 (1+ counter1)))
      ((> counter1 numof-events) t)
      (let ((temp1 nil) (temp2 nil) (factor 5))
         (setq temp1 (nth (1- counter1) eventslist))
         (do ((counter2 1 (1+ counter2)))
            ((> counter2 numof-events) t)
            (setq temp2 (nth (1- counter2) eventslist))

            (set (get-next-symbol) (zz+ temp1 temp2))
            (set (get-next-symbol) (zz- temp1 temp2))
            (set (get-next-symbol) (zz- temp2 temp1))
            (set (get-next-symbol) (zz+sym temp1 factor))
            (set (get-next-symbol) (zz+sym temp1 factor))
            (set (get-next-symbol) (zz-sym temp1 factor))
            (set (get-next-symbol) (zz-sym temp2 factor))
            (z1<z2 temp1 temp2)
            (z1<z2 temp2 temp1)
            (z1subsumesz2 temp1 temp2)
            (z1subsumesz2 temp2 temp1)
            (zzrealinterval temp1)
            (zzrealinterval temp2)))))))



;
```

```
;
;
;*****************************************************************************
;
;
;The following functions test representations and functions by utilizing
;variables and returning them to the available symbol list when their
;use is no longer needed.
;
;
;*****************************************************************************
;
;
;
;
;*****************************************************************************
;
;
;
;function to test LIEBNUMS by managing memory
;
;
;
;*****************************************************************************
;
;
;
;
(defun testliebnums-mm (numof-iterations)
  (declare (optimize (speed 3) (space 3))
           (inline get-next-symbol))


;test case for evaluating representation
;
;
```

```
;setup temporal database
(let* ((light17-off (make-lieb-num 4 0 5 0))
     (light17-on (make-lieb-num 3 0 4 0))
     (fuse42-intact (make-lieb-num 2 0 *most-positive-fixnum* *most-positive-fixnum*))
     (fuse42-open (make-lieb-num 1 0 2 0))
     (switch-light17 (make-lieb-num 5 0 6 0))
     (light17-onagain (make-lieb-num 6 0 *most-positive-fixnum* *most-positive-fixnum*))

;
;build list of temporal events
;
     (eventslist '(,light17-off ,light17-on ,fuse42-intact
                  ,fuse42-open  ,switch-light17 ,light17-onagain))



;
;determine number of temporal events
;

     (numof-events (list-length eventslist)))

;
;
;define benchmark to test representation and associated functions


(do ((iterations 1 (1+ iterations)))
   ((> iterations numof-iterations) t)
   (do ((counter1 1 (1+ counter1)))
      ((> counter1 numof-events) t)
      (let ((temp1 nil) (temp2 nil) (factor 5) (temp-sym nil))
          (setq temp1  (nth (1- counter1) eventslist))
          (do ((counter2 1 (1+ counter2)))
             ((> counter2 numof-events) t)
             (setq temp2 (nth (1- counter2) eventslist))
             (setq temp-sym (get-next-symbol))

             (set temp-sym (lieb+lieb temp1 temp2))
             (set temp-sym (lieb-lieb temp1 temp2))
             (set temp-sym (lieb-lieb temp2 temp1))
             (set temp-sym (lieb+fixnum temp1 factor))
             (set temp-sym (lieb+fixnum temp2 factor))
             (set temp-sym (lieb-fixnum temp1 factor))
             (set temp-sym (lieb-fixnum temp2 factor))
             (lieb1<lieb2 temp1 temp2)
```

```
            (lieb1<lieb2 temp2 temp1)
            (lieb1subsumeslieb2 temp1 temp2)
            (lieb1subsumeslieb2 temp2 temp1)
            (liebrealinterval temp1)
            (liebrealinterval temp2)
            (nconc *listof-syms* (list temp-sym))))))))



;*******************************************************************************
;
;
;function to test FORBNUMS by managing memory
;
;
;
;*******************************************************************************
;
;
;
;
(defun testforbnums-mm (numof-iterations)
  (declare (optimize (speed 3) (space 3))
           (inline get-next-symbol))



;test case for evaluating representation
;
;
;setup temporal database
(let* ((light17-off (make-forb-num 4  5 ))
       (light17-on (make-forb-num 3  4 ))
       (fuse42-intact (make-forb-num 2  *most-positive-fixnum* ))
       (fuse42-open (make-forb-num 1  2 ))
       (switch-light17 (make-forb-num 5  6 ))
       (light17-onagain (make-forb-num 6  *most-positive-fixnum* ))

;
;build list of temporal events
;
       (eventslist '(,light17-off ,light17-on ,fuse42-intact
                     ,fuse42-open  ,switch-light17 ,light17-onagain))
```

```
;
;determine number of temporal events
;
     (numof-events (list-length eventslist)))


;
;define benchmark to test representation and associated functions
;
(do ((iterations 1 (1+ iterations)))
   ((> iterations numof-iterations) t)
   (do ((counter1 1 (1+ counter1)))
      ((> counter1 numof-events) t)
      (let ((temp1 nil) (temp2 nil) (factor 5) (temp-sym nil))
         (setq temp1 (nth (1- counter1) eventslist))
         (do ((counter2 1 (1+ counter2)))
            ((> counter2 numof-events) t)
            (setq temp2 (nth (1- counter2) eventslist))
            (setq temp-sym (get-next-symbol))

            (set temp-sym (forb+forb temp1 temp2))
            (set temp-sym (forb-forb temp1 temp2))
            (set temp-sym (forb-forb temp2 temp1))
            (set temp-sym (forb+fixnum temp1 factor))
            (set temp-sym (forb+fixnum temp2 factor))
            (set temp-sym (forb-fixnum temp1 factor))
            (set temp-sym (forb-fixnum temp2 factor))
            (forb1<forb2 temp1 temp2)
            (forb1<forb2 temp2 temp1)
            (forb1subsumesforb2 temp1 temp2)
            (forb1subsumesforb2 temp2 temp1)
            (forbrealinterval temp1)
            (forbrealinterval temp2)
            (nconc *listof-syms* (list temp-sym))))))))



;*****************************************************************************
;
;
;function to test FLOFORBNUMS with memory management
;
;
;
```

```
;*****************************************************************************
;
;
;
;
(defun testfloforbnums-mm (numof-iterations)
  (declare (optimize (speed 3) (space 3))
           (inline get-next-symbol))



;test case for evaluating representation
;
;
;setup temporal database
(let* ((light17-off (make-floforb-num 4.0  5.0 ))
       (light17-on (make-floforb-num 3.0  4.0 ))
       (fuse42-intact (make-floforb-num 2.0  999999999.0 ))
       (fuse42-open (make-floforb-num 1.0  2.0 ))
       (switch-light17 (make-floforb-num 5.0  6.0 ))
       (light17-onagain (make-floforb-num 6.0  999999999.0 ))

;
;build list of temporal events
;
       (eventslist '(,light17-off ,light17-on ,fuse42-intact
                     ,fuse42-open  ,switch-light17 ,light17-onagain))



;
;determine number of temporal events
;
       (numof-events (list-length eventslist)))


;
;define benchmark to test representation and associated functions
;
(do ((iterations 1 (1+ iterations)))
    ((> iterations numof-iterations) t)
    (do ((counter1 1 (1+ counter1)))
        ((> counter1 numof-events) t)
        (let ((temp1 nil) (temp2 nil) (factor 5.0) (temp-sym nil))
             (setq temp1  (nth (1- counter1) eventslist))
             (do ((counter2 1 (1+ counter2)))
                 ((> counter2 numof-events) t)
```

```
            (setq temp2  (nth (1- counter2) eventslist))
            (setq temp-sym (get-next-symbol))

            (set temp-sym (floforb+floforb temp1 temp2))
            (set temp-sym (floforb-floforb temp1 temp2))
            (set temp-sym (floforb-floforb temp2 temp1))
            (set temp-sym (floforb+flonum temp1 factor))
            (set temp-sym (floforb+flonum temp2 factor))
            (set temp-sym (floforb-flonum temp1 factor))
            (set temp-sym (floforb-flonum temp2 factor))
            (floforb1<floforb2 temp1 temp2)
            (floforb1<floforb2 temp2 temp1)
            (floforb1subsumesfloforb2 temp1 temp2)
            (floforb1subsumesfloforb2 temp2 temp1)
            (floforbrealinterval temp1)
            (floforbrealinterval temp2)
            (nconc *listof-syms* (list temp-sym))))))))


;*****************************************************************************
;
;
;function to test ZZNUMS with memory management
;
;
;
;*****************************************************************************
;
;
;
;
(defun testzznums-mm (numof-iterations)
  (declare (optimize (speed 3) (space 3))
           (inline get-next-symbol))


;test case for evaluating representation
;
;
;setup temporal database
(let* ((light17-off (make-zz 4  5 ))
       (light17-on (make-zz 3  4 ))
       (fuse42-intact (make-zz 2  *pos-inf* ))
       (fuse42-open (make-zz 1  2 ))
       (switch-light17 (make-zz 5  6 ))
```

```
    (light17-onagain (make-zz 6  *pos-inf* ))

;
;build list of temporal events
;
    (eventslist '(,light17-off ,light17-on ,fuse42-intact
                  ,fuse42-open  ,switch-light17 ,light17-onagain))



;
;determine number of temporal events
;
    (numof-events (list-length eventslist)))


;
;define benchmark to test representation and associated functions
;
(do ((iterations 1 (1+ iterations)))
   ((> iterations numof-iterations) t)
   (do ((counter1 1 (1+ counter1)))
      ((> counter1 numof-events) t)
      (let ((temp1 nil) (temp2 nil) (factor 5) (temp-sym nil))
         (setq temp1 (nth (1- counter1) eventslist))
         (do ((counter2 1 (1+ counter2)))
            ((> counter2 numof-events) t)
            (setq temp2 (nth (1- counter2) eventslist))
            (setq temp-sym (get-next-symbol))

            (set temp-sym (zz+ temp1 temp2))
            (set temp-sym (zz- temp1 temp2))
            (set temp-sym (zz- temp2 temp1))
            (set temp-sym (zz+sym temp1 factor))
            (set temp-sym (zz+sym temp1 factor))
            (set temp-sym (zz-sym temp1 factor))
            (set temp-sym (zz-sym temp2 factor))
            (z1<z2 temp1 temp2)
            (z1<z2 temp2 temp1)
            (z1subsumesz2 temp1 temp2)
            (z1subsumesz2 temp2 temp1)
            (zzrealinterval temp1)
            (zzrealinterval temp2)
            (nconc *listof-syms* (list temp-sym))))))))
```

```
;*****************************************************************************
;
;
;
;functions to manage symbol memory that a typical application would use
;
;
;
;*****************************************************************************
;
;
;


;function to get next available symbol
(defun get-next-symbol ()
  (declare (optimize (speed 3) (space 3)))
  (let ((next-symbol nil))
      (cond
                        (*listof-syms* (setq next-symbol (car *listof-syms*)))
                        (t (createsyms *default-numof-symstocreate*) (setq next-symbol (car *
      (setq *listof-syms* (cdr *listof-syms*))
      next-symbol))







;function to create original and additional liebnums as needed
(defun createsyms (numof-syms-tocreate)
  (declare (optimize (speed 3) (space 3)))

  (let ((tempsym nil))
  (do ((counter 1 (1+ counter)))
     ((> counter numof-syms-tocreate) *listof-syms*)

     (setq tempsym (gensym "sym"))
     ;(setq tempsym (make-lieb-num 0 0 0 0))
     (cond (*listof-syms* (nconc *listof-syms* (list  tempsym)))

          (t  (setq *listof-syms* (list  tempsym)))))))



;function to return symbol of available list
```

```
(defun return-sym (sym)
  (declare (optimize (speed 3) (space 3)))
  (nconc *listof-syms* (list sym)))
```