AN ABSTRACT OF THE THESIS OF

Donald William Harney for the degree of Master of Science

in Computer Science presented on August 10, 1976

Title: An Implementation of a Parallel Processing Package in LISP

Abstract approved: Redacted for Privacy
William S. Bregar

The concept of a process is often used in connection with operations of parts of a computer system. This thesis discusses processes in terms of their use as representations of a physical object or system. Five primitives are introduced as operators for allowing processes to be run in a piecemeal fashion. A function package is presented which supplements UCI LISP and allows the user to define LISP functions as processes which (conceptually) run in parallel. The process functions are presented as a package which supplements the LISP system. The functions SUSPEND, RESUME and TERMINATE allow the user to temporarily stop the execution of an instance, continue the execution of an instance, or destroy an instance of a process, respectively. The ACCESS function allows one instance to look up variable values local to another instance. The START function is used as a process controller and begins the execution of the processes. Two illustrations of the function package are given. The first example, a growth model of a stand of grand fir trees, has been implemented and illustrates

parallel processing in a numerical simulation environment. The second example is the Lee algorithm for finding the shortest path through a directed weighted graph. This example has not been implemented, but is discussed as an example of parallel processing in a symbolic computational environment.

An Implementation of a Parallel Processing Package in LISP

by

Donald William Harney

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed August 10, 1976

Commencement June 1977

APPROVED:

Redacted for Privacy

Assistant Professor of Computer Science

Redacted for Privacy

Chairmán, Department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis is presented August 10, 1976

Typed by Jerri Harney for Donald William Harney

ACKNOWLEDGEMENT

I would like to thank my major professor, Bill Bregar, for the tremendous amount of assistance he gave me during the preparation of this document. Thanks also go to Ves Marinov and Kevin McCoy whose knowledge of LISP and parallel processing contributed significantly to my own. I am also appreciative of the help of Scott Overton and Jim Colbert for their assistance with the grand fir tree model, and last but not least, thanks to my loving wife Jerri for her support during my education, and for typing the final copy of this thesis.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

An Implementation of a Parallel Processing Package in LISP

I. SCOPE AND PURPOSE OF RESEARCH

The function package which is presented here is intended as a beginning at the development of a general purpose parallel processing package for LISP. A package called PLISP is being implemented at Stanford as of this writing, but work is still being done on it. The intent of PLISP is to implement parallel processing in a semantic analysis environment.[8] Many of the concepts which are being presented here are adaptations of those ideas used in SIMULA. Many more ideas are contained in SIMULA which are not implemented in the package, but these can easily be added by the programmer who writes the code for the processes. Although the functions and programming suggestions to be presented are not elegant and complex, they do represent a starting point from which a parallel processing simulator can be developed in LISP. The functions given in Appendix A are a workable set of functions and do much of the work necessary for a parallel processing simulator.

II. INTRODUCTION

In physical systems which have observable characteristics, component parts of the system often operate in parallel. Alternatively, the different parts of a system operate simultaneously to produce some effect. Breaking the system down into its component parts, these parts can be thought of as individual processes, each interacting with other processes and producing some result, either directly observable or not directly observable. It is these processes which operate in parallel which will be dealt with.

In implementing a parallel processing package, it is important to note that strict parallelism is not possible on a single processor machine. Thus, a simulator is necessary. It is worth noting that in the early days of the computer, programming was done in machine language and was slow and tedious. Programs which were written were usually mathematical in nature and were highly machine dependent. With the advent of assemblers, the programmer was able to get away from using machine addresses for variables and at the same time use a mnemonic code for the operation code of the instruction. Although this simplified matters slightly, programming was still slow, and one program instruction corresponded to one machine instruction. When compilers appeared, the programmer was freed from having to do much of the bookwork in programming. For the first time, programs became more problem oriented than machine oriented, and compilers generated many machine instructions for each program instruction. However, even with compilers taking away much of the drudgery from

writing programs, many physical systems were simulated in terms of mathematical models. That is, the relationships between the parts of a system could be expressed mathematically and these expressions could be evaluated with respect to a discrete increasing time variable to find the state of a system at any particular time. With parallel processing, the program becomes oriented more toward the system. Blocks of code are written which represent all of the desired characteristics of the particular part of the system, and these blocks of code are executed in parallel with other blocks of code which represent other parts of the system. Thus, parallel processing brings the program one step further away from machine dependence and one step closer to the structure of the physical system itself.

The function package which will be presented here has been implemented on the PDP-10 computer of the University of Oregon using UCI LISP.

Aspects of Parallel Processing: Processes

A process as considered in this paper will consist of a formalization of an independent computation. It is a set of instructions which is carried out by a machine without regard to any other concurrently executing set of instructions. Formally, a process consists of a state variable set X, a state, a state space, a computation, an action, and an action function. The state variable set X is an indexed set and is possibly infinite. The state is an assignment of values to all elements of the state variable set. The state space, then consists of the set of values which each state can assume. A computation is a sequence of states from the state space. The action is the assignment of values to the state variables, and the action function maps states into actions. If the action function is given an initial state, a computation can be generated.[1,9]

In the context of a computer, the memory itself can be thought of as being the state variable set. The state space would then be the set of all values which each word in memory could contain. The computation consists of a sequence of states from the state space, where each state is the contents of memory. The action function is the program which is being executed, and the action is the execution of the program itself. The process P is defined as an ordered triplet;

$$P = (S, A, I)$$

where S is the state space, A is the action function, and I is the initial state.

Although the single process has a great many applications, the

use of this kind of programming structure forces the programmer to consider the interactions between parts of a system rather than the individual natures of the parts of the system themselves. It is perhaps more useful to consider a programming structure which allows the user to construct a program in terms of the characteristics of the system's parts rather than strict interaction between parts. Using this structure, the programmer must concentrate on defining the system components as acting entities in themselves rather than as relationships between system parts. Defined as an individual entity, a component has more of a capacity to act in the model of the system. As a model of either a physical or conceptual system, it should reflect that other parts of the system are acting both in concert and simultaneously. Continuing this analogy between the system and its representation, it follows that the execution of these processes which represent component parts of the system should be simultaneous. This leads to the idea of parallel processing.

Aspects of Parallel Processing: Parallel Processes

The simultaneous execution of two or more processes will be called parallel processing. In many situations, it is desirable to have interaction between processes to direct the execution of the processes themselves. This interaction may occur at specific points during execution, but the process will perform much of its work in isolation. Interaction between processes is achieved by allowing the executing process to access specific memory cells local to other processes for the retrieval of information. These memory cells are not global in the sense that they are immediately accessable to each process; they must be sought out by the executing process. Once needed information is retrieved, execution in isolation is resumed.

In examining a situation in which more than one object acts, it is often useful to consider each object individually. Each object has characteristics which are of interest to the programmer, and can be represented in a computer program. For example, in a simple predator/prey system, both of the species are living animals, and the two species interact. The predator eats the prey as its primary food. The predator's population grows, and the prey's number diminishes. When the number of prey is no longer sufficient to sustain life, the population of predators diminishes. When the predator's population diminishes, the prey can increase in number and the cycle begins again.

Each of the species exists at the same time and lives in the same physical space. In that respect, they resemble interacting pro-

cesses which execute simultaneously. Thus, each of the species, predator and prey, can be simulated with processes - both executing simultaneously and interacting with each other. Hence, this simple system lends itself well to parallel processing.

Another kind of parallel processing application is found in a computer operating system. Large computer systems often use data channels which function as minicomputers to perform the functions of input and output. Although the data channel gives information to and receives information from the central processing unit (CPU), it performs its functions independent of the CPU. Thus, the CPU may be executing a user program while the data channel is executing an input/output operation. Although this system can be simulated through parallel processing, it is important to note that two processors are functioning, and are, therefore, actually operating in parallel. Two distinct processors perform their functions simultaneously rather than being simulated by a body of code running on a single processor. In analyzing a computer system, the two processors are represented as distinct processes; their operation is often used as a classical example of simultaneous execution.[1]

A third example is that of the Lee algorithm for finding the shortest path through a directed network.[2] The algorithm starts by sending a simulated pulse through each directed edge of the graph from the start node (source node). The pulse moves some distance at each simulated time step. As each pulse reaches a new node, new pulses are initiated from the new node along each directed edge adjacent to it. Eventually, some pulse will reach the end node, and at that point,

the path taken to reach the end node (sink node) can be traced. Using a single process, only one pulse can be moved at a time step, but if parallel processing is used, each pulse in the graph is moved the same distance before the next time step is reached.

Simulation of Parallel Processes

For two or more processes to run in parallel, it is necessary for these processes to be executing on two different processors. In practice, however, it is extremely rare that this happens, particularly in a modelling situation. Processes which are conceptually executed in parallel must generally be implemented on a single processor. When this happens, the result is not parallel processing, but quasi-parallel processing, and requires a simulation of its own. The idea of quasi-parallel processing will be referred to as parallel processing and is discussed in the next section.

A simulation is an imitative representation and in many systems utilizes a discretely increasing time variable. In many instances, the time variable, which is an integral part of the system, is held constant while all necessary computations are carried out. In a heirarchical program structure containing a calling program and sub-programs, conceptual parallelism may be easily achieved in this way. The calling program holds the value of time constant while all needed subprograms are executed.

In a parallel processing environment, the same idea of holding the value of time constant during computations of subprograms is the same, but there is no heirarchical structure to the program. For example, in one situation, process A may be suspended while process B executes to completion. In another situation, this arrangement may be reversed and the one-time "calling process" becomes the "called process." The structure of the execution of processes may not be predictable, but

the idea of completing necessary computations within a simulated time interval still holds.

If a system of gears is being examined, the system is represented physically as a series of objects which are interconnected. The rotation of the driving gear x degrees causes the rotation of an attached gear y degrees, and so on. In the physical system, the rotations occur simultaneously. This is simulated by defining processes which represent gears and executing them in parallel. In actually running the processes, the driving gear is rotated x degrees independent of any of the other gears. The attached gear is rotated y degrees where y is a function of x, and all of the other gears in the system are rotated similarly. This is done before any time variable in the system is updated.(3)

Parallel processing also has applications other than simulation involving a time variable. An example of a non-simulation application would be an English sentence parser. A process could be programmed which would find all possible parse trees for the subject of a sentence, and another process could be programmed for finding all parse trees for the predicate of the same sentence. These two processes would execute in parallel to produce all possible parsings of a sentence. In this application, a time variable is not considered since the processes are designed to perform a task rather than simulate the characteristics of an object.

An example of this kind of process is seen in the parsing of the sentence "The ball and the box are both blue." The subject parser will receive the phrase "The ball and the box" while the predicate parser

will receive the words "both blue." The subject parser will begin by constructing a tree for the words "The ball." While this is happening, the predicate parser will recognize "blue" as an adjective. Upon recognizing the word "both", it must make information available to the subject parser to construct a tree for a compound subject.

Three of the most widely used simulation languages, GPSS, SIMSCRIPT, and SIMULA use the idea of representing objects in a system as subroutines or blocks of code. Although all three do not contain parallel processing simulators, they each represent a simulated system as a collection of interacting objects.[15]

SIMULA is a simulation language which contains a function package for the simulation of parallel processing. The language has a block structure and is quite similar to ALGOL in appearance. The blocks which are defined in a SIMULA program are executed as processes in a manner which is discussed in the next section. The language itself contains many features which strongly influenced the development of the function package which will be presented. Most notably, the idea of breaking a process up into small pieces is used in the LISP function package. The idea of centralizing the storage areas and defining one process to represent a class of objects are ideas which also come from SIMULA.

In SIMULA the concept of an event notice is used to bind together blocks of code with variables which refer to specific instances of that object which is represented by the code.[3,6] The event notice consists of an entry in the system process scheduler which contains a pointer to a block of code for the process, a pointer to a data struc-

ture containing variable values local to the process code, and a unique name. The name is used to refer to the instance of the process.

SIMULA allows the user to define a block of code to be used for the representation of a generalized object. This same block of code is used to represent every specific object within the same class. Unique instances of that object are distinguished using unique names called reference variables, and the local variable values. For example, consider a process VEHICLE containing the local variable NUMBEROFSEATS. One instance of VEHICLE might have "BUS" as the value of its reference variable and the corresponding value of NUMBEROFSEATS equal to 40. Another instance of VEHICLE could have "SPORTSCAR" as the value of its reference value with NUMBEROFSEATS having the value two. In this way, one block of code can represent a variety of objects. This idea has been exploited in the function package to be presented here.

Each process body may have a number of reference variables associated with it. By convention, the reference variable in SIMULA is used to access the particular instance of a process. When a process is referenced through the reference variable, it has the appearance of defining a specific, unique object rather than a class of objects. This convention also eliminates the problem of referencing a process and finding that unwanted variable values are bound to the local variables. Referencing a process by starting or resuming its execution causes local variable values associated with the reference variable to be bound to the local variables of the process.

SIMULA also allows the accessing of variable values which are local to process instances other than the one currently executing.

Associating the local variable values with a reference variable simplifies the problem of finding the proper instance of the process and the proper value of the desired variable.

Each instance of a process may be in one of four possible states - active, passive, suspended, and terminated. If the process is currently executing, it is in the active state. This means that an event notice has been constructed for it, and its execution has been initiated. An instance which is in the passive state has had an event notice constructed for it, but its execution has not yet begun. An instance which is in the suspended state has an event notice associated with it, and is "partially executed." The execution of the instance has been started, and has been stopped before the instance has run to completion. At the time of suspension, the variable values local to the instance are stored to facilitate the resumption of the instance's execution. An instance which is in the terminated state is one which has been run to completion. The termination of an instance causes the removal of the event notice from the scheduler. Following termination, the instance cannot be resumed, and variable values which were local to it cannot be accessed any longer.

These four states define the possible phases of execution of an instance of a process. Instances which are in the active, passive or suspended state have variable values associated with them which may be accessed by the active instance. These states of execution arise from the structure of the parallel processing simulator as presented in the next section.

III. IMPLEMENTATION

Extension of Coroutine Structure in a Parallel Processing Simulator

Since true parallel processing cannot occur on a single CPU machine, a method of processing must be set up which simulates parallel processing on a single processor. To do this, the set of processes to be run in parallel are defined as under a control structure similar to that of coroutines. Coroutines are executed in a piecemeal fashion, shifting control back and forth between each other. For example, given two coroutines A and B, A will begin its run first (see Fig. 1) and execute until point A1 is reached. Then it will jump to B, and B will execute until point B1. When B reaches B1, it will jump back to routine A, and resume execution from where A left off; point A2 which immediately follows A1. This process of trading control between the two routines during execution is a fundamental idea in the implementation of parallel processing. A process P is executed to a certain point P1, at which point execution is temporarily suspended. Depending upon the programming situation, a new process may be started, an already suspended process may resume execution, or a process may be terminated. Thus, four primitives of parallel processing emerge: START, SUSPEND, RESUME, and TERMINATE. In modeling a real-world situation, the parts of the system interact, so in a simulation situation, processes interact. Therefore, each process must have the capability of accessing any variable which is local to another process instance. This gives rise to a fifth primitive, ACCESS which allows variable

values to be retrieved from other instances.

Extending this same idea to include more than one routine, consider four processes which are to execute in parallel. One of the routines will begin its execution first, say A (see Fig. 2). A will execute to some point A1, at which time the SUSPEND primitive will be used to save the local environment. Execution will begin for some other process, C, which will execute to point C1. SUSPEND will then be needed again for saving the environment local to process C. The control will similarly pass to process D, which will execute in the same manner. Once point D1 is reached, however, control may pass back to process A which will continue its execution at point A2, which immediately follows point A1 in the process. Process A runs to completion, and terminates. Control then passes from the terminated A process. Let B begin its execution. B executes to some point B1, and is suspended. B will then pass control to D, which executes to termination. C is then resumed from point C2 and executed to termination. The only process which has not been run to execution is B, which is then resumed and executed to conclude the run. It should be noted that the processes in this example may in fact be four distinct processes, four instances of the same process, or any combination of distinct processes and instances of the same process.

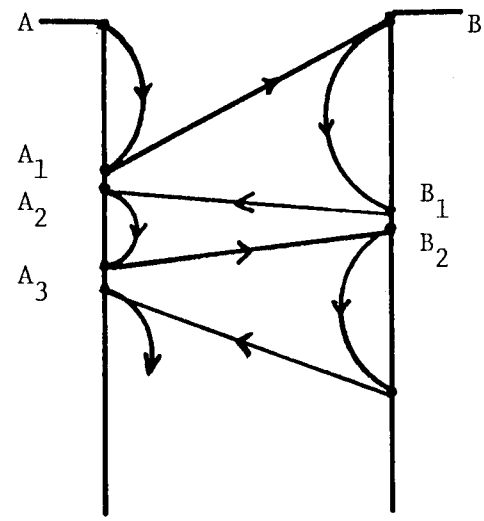


Figure 1. Flow of control of two coroutines

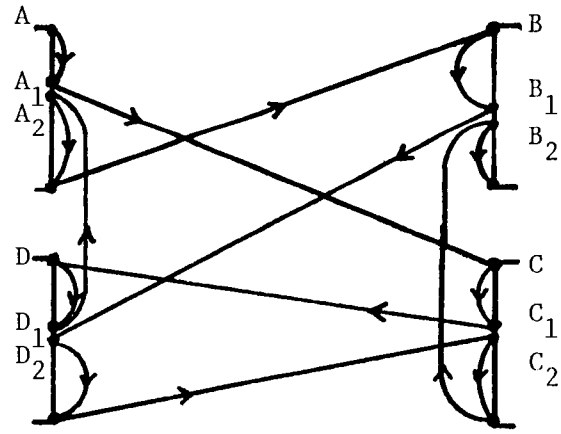


Figure 2. Hypothetical flow of control among four processes

The Function Package

The concept which is perhaps most basic to the implementation of the function package is the breaking up of control among the executing instances. Each process will begin execution, execute to some point, and relinquish control to another process, only to be resumed at a later time.

Functions are therefore needed for performing the necessary operations of beginning execution at some arbitrary point in the process. The START and RESUME functions perform these operations. SUSPEND is needed for stopping the execution of a process and storing the execution environment before it has run to completion. A place is needed in which these environments can be stored - the REFLIST. Once an instance has executed to completion, it is no longer needed. The TERMINATE function removes the instance from the run. In many cases, it is desirable to allow instances to interact. This is done by allowing the currently executing process to access variables local to another instance. The ACCESS function is used to look-up variable values associated with other instances. These functions, START, RESUME, SUSPEND, TERMINATE, and ACCESS are the five primitive functions of the function package, and perform operations analogous to the primitives given in the previous section. The name REFLIST refers to a data structure which contains information from all instances in the run.

The environment of two coroutines is one in which all variables used by either routine are global. The coroutines are structured to execute in a completely predictable way. Since this is not necessarily

the case for a set of parallel processes. The execution environment of the processes is quite different. The same variable may have different values depending upon what instance of a process it is associated with. Each "variable" may then have many values, one for each process instance in which it occurs. Each of these values is local to the process, although it may be accessed externally if desired.

Control of the processes may be unpredictable. The scheduling of the processes may be set up initially, but the processes may reschedule each other. Instances may even be terminated, and their execution deleted. Thus, although coroutines and processes use a similar mechanism for execution, they have basic differences which preclude defining processes as an extension of coroutines.

Considerations in Implementation

There are space limitations which must be considered in implementing a package. In using a program to describe a physical system, each object in the system could have its own process definition. However, when a substantial number of objects is to be considered, there seems to be no need for storing many copies of the same code. Rather than keep a copy of the process code for each object, each object is given a name, and a process is associated with that name. This name is called a reference variable and refers to a specific instance of the process. This reduces the storage requirement within the machine by representing similar or identical objects by the same code.

One reason for keeping a copy of the process code is the convenience of storing the local variables as seen from the programmer's point of view. Each process has its own local storage area where variable values are kept. However, it is not necessary to keep many copies of the process code when only the variable values distinguish between instances. The variable names and values may be kept in a list and associated with a reference variable. This makes each instance of a process unique although many instances may share the same code. It also necessitates writing code for saving the environment of the instance, but each process function is written only once. Thus, with a minimum of programming effort, the duplication of code and waste of storage space can be eliminated.

Under the LISP system, there are two classes of variables; global variables and local variables. Global variables are accessible

to all functions; local variables are restricted to the confines of an executing function. With the storage of variable names and values in the reference list, a third type of variable is generated which can be examined using the package functions. The variables in the association list are neither global nor local since local variables are lost when a function finishes execution. The variables in the association list will be called run variables since they are in existence only during the parallel processing run.

Once a name has been associated with each process, the next step is to associate a list of variable values with the reference variable/process body pair. By keeping a list of variable values separate from the process body, the problem of one instance of a process using the same local variable values as another instance of the same process is avoided. This list of local variable values needs only to be substituted into the process body when execution begins or resumes, and saved when the execution of the instance of the process is suspended.[4,5] Saving the process environment will insure that even though interaction with other processes is needed, each instance of a process remains distinct. All of this information is saved in REFLIST.

Each process is represented as a LISP function, and may be defined using the DE macro. For reasons described later, the process must be defined using the LISP PROG feature. This is necessary to allow the process function to jump to a label which is the return point for the partially executed instance. In this implementation, the process definition may not be recursive, but recursive sub-functions may be used. After saving the execution environment, the process must execute

a RETURN function to remove unnecessary information from LISP's stacks. This RETURN function is another factor in the requirement of the PROG feature in that it can only be executed from a PROG.

In the implementation of this parallel processing simulator, a list has been set up which contains the information necessary for using one set of code for many processes and holding information local to that instance of the process. It is called the reference list and is bound to the LISP atom REFLIST in the function package (see Fig. 3). REFLIST is a list of lists, each sublist having the same structure as every other sublist. The order of the sublists within the reference list initially defines the order in which the processes will be executed.

The first element of a reference list sublist is the reference variable, or the name through which the unique instance of the process is accessed. The second element of the sublist is the name of the process which is to be associated with the reference variable. The third element of the sublist is a label which marks the point at which execution will resume when that instance of the process is to continue. The fourth and final element of the sublist is an association list of variable names and their values at the time that the execution of that particular instance of the process was suspended. In the event that the process execution has not begun, this list will contain the variable names and their initial values. The association list consists of dotted pairs of variable names and corresponding values. The sublist written in LISP takes the form:

(REFVAR PROCNAME LABEL ((VAR1 . VALUE1)(VAR2 . VALUE2) . . .)).

The reference list is a list of the following sublists:

```
REFLIST = ((REFVAR1 PROCNAME1 LABEL1 ((V11 . VAL11)(V12 . VAL12) . . .))
           (REFVAL2 PROCNAME2 LABEL2 ((V21 . VAL21)(V22 . VAL22) . . .))
           .
           .
           .
           (REFVARN PROCNAMEN LABELN ((VN1 . VALN1)(VN2 . VALN2) . . .)))
```

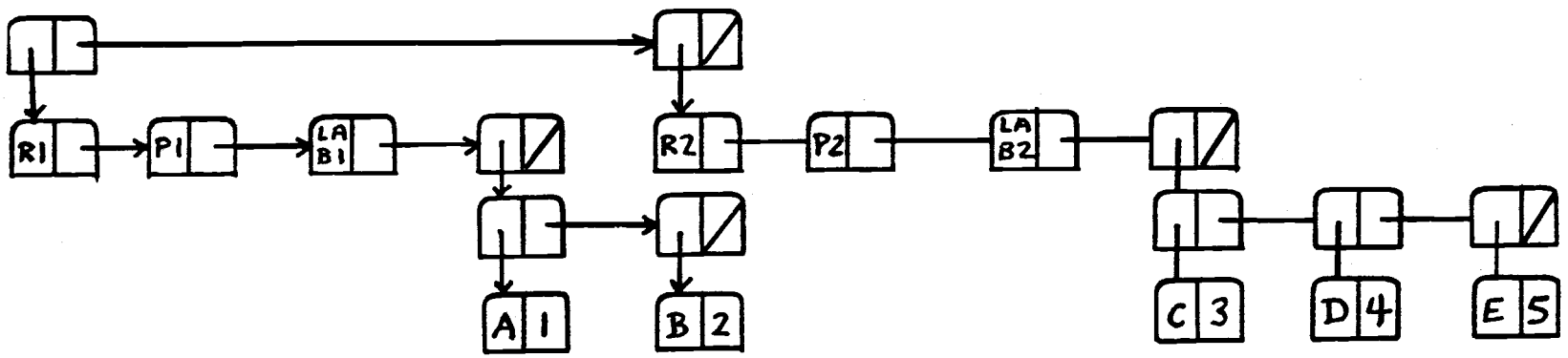


Figure 3. Internal structure of the reference list
((R1 P1 LAB1 ((A . 1)(B . 2)))(R2 P2 LAB2 ((C . 3)(D . 4)(E . 5))))

Primitive Functions

When an instance of a process is to be resumed, the process must first retrieve its environment, or variable bindings, from the proper sublist within the reference list. This is easily accomplished as the sublist referring to the desired instance of the process is placed at the beginning of the reference list by the START function.

The START function is used to begin the execution of the processes. Its argument is the name of the reference variable which is to be handled first. START contains a loop which evaluates processes based on the variable (atom) NEXT. If the atom NEXT is bound to the characters "NEXT", then the next process in the reference list is initiated. If the atom NEXT is bound to the name of a reference variable, then the process associated with that reference variable begins or resumes execution. Thus, START becomes a controller for process execution. In the event that the reference list becomes empty, the run is concluded. An empty reference list signifies that all of the processes have been run to conclusion, and have been terminated. An example of a call to START is:

```
(START  (QUOTE CONTROL)).
```

This would start a simulation run with the execution of a process called CONTROL.

Before another process is resumed from a currently running one, the run variable names and their values must be put into the proper sublist in the reference list. This is done by the function SUSPEND. SUSPEND takes as its two arguments the list of dotted pairs of variable

names and values and the label which constitutes the return point for that instance of the process. This function updates the reference list entry, and depends solely on the programmer to supply it with all of the appropriate information for updating the reference list sublist. The SUSPEND function is called by:

(SUSPEND (LIST (CONS (QUOTE B) B))(QUOTE LABEL2)).

The function (LIST (CONS (QUOTE B) B)) generates the list of the dotted pair of the name B and the value of B.

The first attempt at implementing the SUSPEND function was to have used an approach which was different from the current running version. The variable names and values which formed the process environment were to have been retrieved from LISP's context stack and special push-down list. The only argument to have been used was the literal name of the return point label. After several unsuccessful attempts at implementing this idea, it was discovered that the context stack contains entries which are used by the LISP interpreter. The presence of these items caused unpredictable context stack references and made the approach infeasible. As a result, the SUSPEND function was changed, and the responsibility for storing the variables was placed on the user. All of the variables which the user decides should be saved must appear in the list which constitutes the first argument of SUSPEND. The SUSPEND function replaces the association list in the reference list sublist with the list given as its first parameter. Thus, any variable name which does not appear in the first argument to SUSPEND will not be in the reference list entry after SUSPEND is executed whether or not it was ever there. If a variable is deemed temporary and its value is

not important to the execution of the process, it need not be saved at suspension time.

Following the storing of run variables by the SUSPEND routine, the process must perform a normal exit, which is accomplished by the execution of a (RETURN T) function. This is necessary to remove any variable bindings which may accumulate on LISP's special push-down list and context stack.

The RESUME function allows the programmer to designate the instance which will execute after the suspension of the currently executing process. The argument of the function is a reference variable. An example of the function call is:

(RESUME (QUOTE WALK))

This designates an instance whose reference variable is WALK to continue execution after the currently running instance is exited. If designation of the next running instance is unnecessary, the function may be called:

(RESUME (QUOTE NEXT))

If the latter form of RESUME is used, the reference list becomes a scheduler for the run. The instances execute in the order in which they appear in the reference list. The first form of RESUME given above allows the instances to reschedule themselves and each other. When this form is used, the reference list is no longer used as a scheduler, and is used only in the capacity of a data storage area.

The TERMINATE function does exactly as the name implies. When an instance of a process is run to completion, or when some other condition occurs which necessitates the termination of an instance of

a process, the TERMINATE function will cause the removal of that instance. The argument for TERMINATE is the name of a reference variable. The function must be specified in the process body by the programmer and should be the last function executed by a process. If the TERMINATE function is not called at the conclusion of a process, the START function will enter an infinite loop; always returning to resume a process which will not terminate.

Once an instance of a process has been terminated, there is no way of accessing any run variables which were local to it. The function takes the storage space which held a sublist of the reference list, and returns that space to free storage. In the event that the process tries to terminate an instance which has already been terminated, no error is returned. An example of the function call is:

(TERMINATE (QUOTE SPECIES1))

When it becomes desirable to access information (i.e. a value of a variable local to an instance which is not currently executing), the ACCESS function must be used. The argument of the function is a dotted pair consisting of first the reference variable as the first atom and the local variable name as the second atom. The function looks for the appropriate sublist of the reference list and finds the association list given in the sublist. If the variable name is found, the function returns the value associated with that instance of the variable. In the event that the variable name is not found after the correct instance of the process has been found, the function will return useless information.

The ACCESS function should be used with caution. In the event

that the named reference variable does not exist within the reference list, an error is returned. An example of a call to the ACCESS function is:

```
(ACCESS (QUOTE (SPECIES . GENUS)))
```

The user is cautioned that after a call to ACCESS which returns a valid variable value, the reference list have been rotated and left in a state which will probably not be the same as before the call. The function (CHECK (QUOTE CURRENTNAME)) will cause the REFLIST to be rotated to its position prior to the call to ACCESS.

In the example of grand fir growth given in the next section, the ACCESS function is used to allow the tree instances to look-up the value of time contained in the system clocking process, CLOCK. This value of time is used to determine whether or not the processes should terminate.

Utility Functions

Five utility functions are presented in this section - ROTATE, PUTVAL, PUTLAB, CHECK, and CADDDAR. The functions presented here support the primitive functions given previously. These utility functions are used in manipulating the processes and associated data structures. In most cases, these functions are called by the primitive functions to perform specific tasks. An additional function, COMMENT is provided to allow documentation by the user. This section also contains descriptions of three atoms: REFLIST, which must be used in a parallel processing run; CURRENTNAME, which may be used as a convenience to the user; and NEXT, which is internal to the function package.

The CHECK and ROTATE functions are used for storing data into or retrieving data from the reference list and work with the first sublist of the reference list. The CHECK functions, for example, may be used to check for a reference variable's presence in the reference list and to rotate the list for the purpose of leaving the sublist containing the reference variable in the first position. The CHECK function can then be used in scheduling and is simpler than maintaining a stack of pointers to the reference list. It is easy to implement and easy to comprehend.

The ROTATE function is used to produce a rotation of the reference list. It rotates to the left with the first sublist of the reference list becoming the last sublist. It is used by the CHECK function during the search for a given reference variable, and by the START function to rotate the reference list prior to executing the "next"

process. ROTATE has no parameters. An example of its call is:

(ROTATE)

The PUTVAL function supports the SUSPEND function by performing the first half of the operation of saving the execution environment; storing the association list of variable names and values into the first sublist of the reference list. It does not check to see whether the reference list is in the proper position before any stores are made. The programmer is expected to execute (CHECK (QUOTE CURRENTNAME)) before SUSPEND is called to be certain that the first sublist in REFLIST is the sublist associated with the currently running process. An example of its call is:

(PUTVAL (QUOTE ((A . 1)(B . 2)(C . 3)))).

The PUTLAB function performs the second half of the operation of saving the environment of the executing process. Its argument is the label which marks the return point for the executing process. The function stores the label into the first sublist of the reference list without checking to be certain that it corresponds to the currently executing process. An example of a call to the function is:

(PUTLAB (QUOTE LABEL2)).

It is called by the SUSPEND function, although it may be used separately by the user.

The COMMENT function takes a list of atoms as an argument and returns immediately after binding these atoms to a list which is lost after the function is exited. The net effect of this is that the COMMENT function allows the user to insert remarks into the body of the process and function definitions. The COMMENT function is actually

executed by the LISP interpreter, but nothing is evaluated during execution and its execution time is negligible. An example of its call is:

(COMMENT TEST OF PLUME CASE NUMBER TWO).

The CHECK function is used to search the reference list to determine if the reference list contains a sublist which has as its first element the reference variable given in the call to the function. If the CHECK function finds the sublist containing the given reference variable, the function exists and the reference list is left rotated with the sublist containing the given reference variable as the first sublist. In this case, the function returns the value T, or "true." In the event that the reference variable is not found in any of the sublists of the reference list, the reference list is left as it was before the function was called, and the value NIL, or "false" is returned. CHECK operates by rotating the reference list and checking the first sublist for the given reference variable, which is done in a loop. An example of a call to the function is:

(CHECK (QUOTE RABBIT)).

The CADDDAR function was written as a simple function to access the association list in the first sublist of the reference list. In most instances, the first sublist of the reference list will contain the sublist from which information is desired. In LISP code, this is the (CAR REFLIST). The first element of the sublist is the reference variable, and is accessed through (CAAR REFLIST). The second element is the process name, and is accessed through (CADAR REFLIST). The third element is the return point label, and is accessed through

(CADDAR REFLIST). These four functions, CAR, CAAR, CADAR, and CADDAR are built into LISP while CADDDAR had to be written and implemented in the function package. An example of its call is:

(CADDDAR REFLIST).

This function returns the list of dotted pairs of variable names and values.

The package contains several useful atoms, one of which must be used. The name REFLIST in the package is set as an atom, and it is up to the user to bind it to the proper list. No other name may be used for the reference list since all of the functions use the name. The reference list may be bound to REFLIST through the function:

(SETQ REFLIST (QUOTE (*****)))

The ***** in the function call represents the contents of the reference list.

The atom CURRENTNAME is in the package as a convenience for the programmer. Upon entering the process, the atom may be set to the name of the reference variable of the instance just entered using the function:

(SETQ CURRENTNAME (CAAR REFLIST)).

Having the reference variable bound to CURRENTNAME is useful for returning the reference list to the position it was in when the process instance was entered. For example, during the execution of the process, it may be necessary to use the ACCESS function to look-up the value of a variable which is local to another instance. The use of ACCESS will change the position of the reference list. If the reference list is not rotated back to its original position before

the SUSPEND function is used, the association list and return point label will be placed into the wrong sublist. This happens because SUSPEND stores into the reference list's first sublist without checking for position. This situation is remedied by using the function:

(CHECK CURRENTNAME)

This rotates the reference list to the proper position before SUSPEND stores into it.

Setting Up The Process

Setting up the process to run, the programmer must first use the name LABEL as the only parameter to the process function. The PROG feature must be used. As a convenience, the user may set CURRENTNAME using the function (SETQ CURRENTNAME (QUOTE *****)) where ***** is the literal name of the process. The process variables must then be moved from the reference list and bound to the variable names internal to the process body. The function (SETQ B (ACCESS (CONS (CAAR REFLIST)(QUOTE B)))) is used to accomplish this. The variable name is given here as "B", and it is assumed that the reference list is in the proper position (i.e. (CAAR REFLIST) is the name of the reference variable of the process which is currently executing). The next function which is called is the function which causes the jump to the proper label inside the process. This function is (EVAL (CONS (QUOTE GO)(LIST LABEL))). Following this function call, the label of the first "statement" of the process will appear.

When a process is to be suspended, the SUSPEND function is used to save the execution environment. The following will save one variable value and the return point label:

(SUSPEND (LIST (CONS (QUOTE B) B)) (QUOTE LABEL2)).

In the event that the process contains a local temporary variable its value need not be saved. The SUSPEND function does not update the association list under the reference variable, it stores a new list in the proper place in the sublist. Therefore, if a variable value is to be accessed by the ACCESS function, it must appear in the list

given as an argument to the SUSPEND function.

If a specific instance of a process is to be resumed, the RESUME function is called to set the proper information. The function call (RESUME (QUOTE TRAVEL)) will cause the resumption of an instance of a process referred to by the literal TRAVEL. In the event that the specification of a next process is not necessary, the function may be called (RESUME (QUOTE NEXT)). This may be desirable for purposes of clarity in reading the program. If (RESUME (QUOTE NEXT)) is omitted, there will be no ill effects. The START function will simply process the next instance in the REFLIST. To exit a process, the function (RETURN T) is necessary.

The TERMINATE function may be called at any time, and may terminate any given instance, including the currently running one. If the TERMINATE function is called on a currently executing process, the sublist referring to it is removed from the reference list, and the process will continue executing until a RETURN function is executed.

The use of the (RETURN T) function forces the process to be non-recursive. The reason for this is based upon the implementation of the function package. There is no conceptual reason why a process could not be recursive. The idea which is basic to parallel processing is the idea of simultaneous execution, and this concept is completely independent from the structuring of the individual processes. Recursive as well as iterative processes should be allowed to execute. The problem comes in when the LISP interpreter is considered. In implementing a recursive function, it is necessary to mark the point at which execution of the function resumes when the function ends its recursive calls.

The interpreter does keep track of this return point, but it is inaccessible to the user. This forces the parallel processing package functions, which have the appearance of being user functions (as viewed by the interpreter) to mark their own return points using labels, and jump to these labels when execution is resumed. A LISP (GO LABEL) expression is used for this. These two functions, (RETURN T) and (GO LABEL), each require the PROG feature in LISP to be used. Under UCI LISP on the PDP-10 Computer at the University of Oregon, the PROG feature does not allow recursion.

IV. EXAMPLE SIMULATIONS

Grand Fir Growth Model

As an example of the use of the system, a growth model of grand fir trees will be considered. The model will be run for a simulated period of 100 years using decade resolution. The model considers quantities which are measurements of the tree as well as the number of trees of that size which can be supported by a one-acre stand. The quantities modeled are the diameter of the tree at breast height (d.b.h.), the height of the tree, the height of the crown base (HCB), and the stand density.(16)

The tree which is represented here is the "typical" tree in the stand. It is represented by the mean measurements of the trees in the stand (see Fig. 4 for a description of the measurements of the tree). The diameter at breast height is given in inches, the height of the tree and the height of the crown base in feet, and the stand density in number of trees per acre. These quantities are represented in the program by the variables X1, X2, X3, and X4, respectively.

The changes in the above quantities are represented by equations in the program which are designated f functions. The f functions represent the increment accumulations for the modeled quantities over a ten year period. The equation $f_{1,1}$ gives the diameter increment accumulation for ten years growth. The $f_{2,2}$ equation gives the height increment accumulation for ten years growth. The $f_{3,3}$ equation gives the height of the crown base increment accumulation for ten years

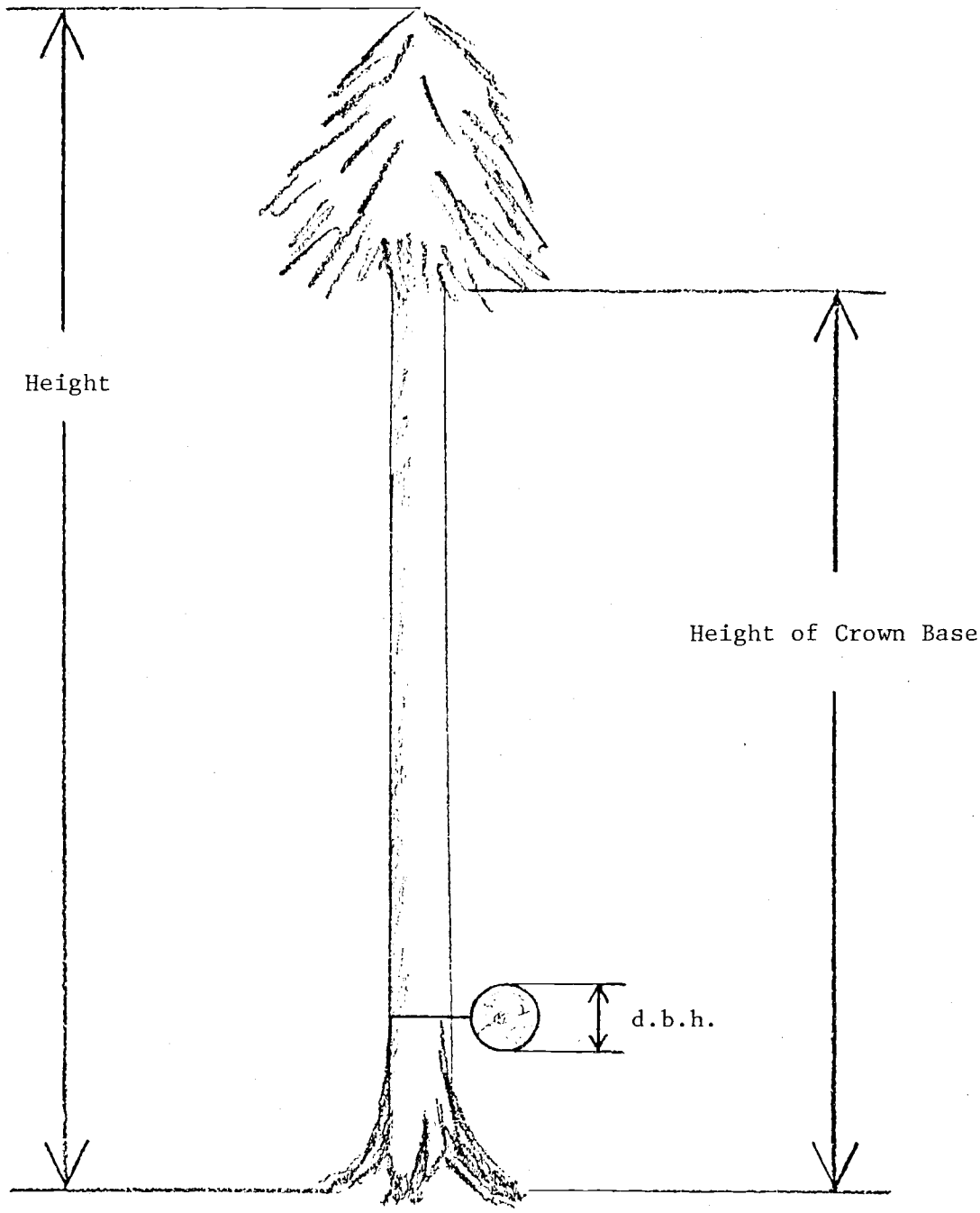


Figure 4. Tree measurements used in the growth model

growth, and $f_{4,4}$ gives the tree mortality accumulation for a ten year period. The convention of using a double subscript for the equation names comes from the FLEX modeling paradigm,[14] from which this example was taken. The f functions refer to the flux of some quantity and its corresponding affect on the state variable. In the FLEX system, flows of a quantity are represented using a function of state variables. The equantion f_{ij} represents a flow into compartment j from compartment i. It is often useful to represent a compartment in terms of the total flow through it. The equation f_{ii} then represents the sum of the flows into the compartment minus the flows out of the compartment. These equations have been modified for use in the example, although their difference from those f functions found in the FLEX representation is minimal. The double subscript has been incorporated into the function names under the LISP system to aid in identifying correspondence between the model in the FLEX paradigm, and the same model implemented here.

The FLEX equations are:

$$f_{1,1}=X_1+\sqrt{X_1^2 + e^{(7.5632+.4776(\ln X_1)-.96661n(X_4(.04+.027X_1+.00405X_1^2)))}}$$

$$f_{2,2}=e^{(3.10707+.46651(\ln(0.5f_{1,1})-.34633(\ln X_1)))}$$

$$f_{3,3}=\text{Max}\ (X_3,\ (-30.142+.61X_2+9.1781n(f_{1,1}))) - X_3$$

$$f_{4,4}= -.113X_4$$

Representations of these same functions under the LISP system are found in Appendix C. The FLEX paradigm uses the f functions for adding variable increments to update the values of the X variables. Updating the variables is done by the following :

$$X_1 = X_1 + f_{1,1}$$

$$X_2 = X_2 + f_{2,2}$$

$$X_3 = X_3 + f_{3,3}$$

$$X_4 = X_4 + f_{4,4}$$

Each of the f functions is evaluated before the X variables are updated. Following the modification of the X variables, these same variables are output, and the variables are re-evaluated for the next simulated ten year period.

The process function GFIR is the process body which represents the two instances of a stand of trees (see Fig. 5). As stated previously, the only parameter needed is the return point label which is passed to the process from the START function. The list of variables given next contains the names of the variables which LISP views as local. The local variable list contains the names of the X variables which will be output for examination as well as the variables which will temporarily hold the values of the f functions during the execution of each instance.

All of the PROG variables are considered local by the LISP system, but in terms of the simulation package, they may be either temporary variables or run variables. In this process, the variables which begin with the letter X are run variables, while the variables which begin with the letter F are temporary variables. The variable CURRENTNAME is set to the value of the reference variable of the currently executing process. This is done to facilitate checking the position of the reference list later in the process.

```
(DEFPROP GFIR
 (LAMBDA(LABEL)
  (PROG (X1 X2 X3 X4 F11VAL F22VAL F33VAL F44VAL)
        (COMMENT CURRENTNAME SET TO REFERENCE VARIABLE)
        (SETQ CURRENTNAME (CAAR REFLIST))
        (COMMENT RESTORE ENVIRONMENT)
        (SETQ X1 (ACCESS (CONS CURRENTNAME (QUOTE X1))))
        (SETQ X2 (ACCESS (CONS CURRENTNAME (QUOTE X2))))
        (SETQ X3 (ACCESS (CONS CURRENTNAME (QUOTE X3))))
        (SETQ X4 (ACCESS (CONS CURRENTNAME (QUOTE X4))))
        (COMMENT CHECK TIME VALUE)
        (COND ((*GREAT (ACCESS (CONS (QUOTE TIME) (QUOTE DECADES)))
                       10)
               (TERMINATE CURRENTNAME)
               (RETURN T))
              (T (EVAL (CONS (QUOTE GO) (LIST LABEL)))))
        (COMMENT COMPUTE FUNCTION VALUES)
  LAB1  (SETQ F11VAL (F11 X1 X4))
        (SETQ F22VAL (F22 X1 X2 F11VAL))
        (SETQ F33VAL (F33 X1 X2 X4))
        (SETQ F44VAL (F44 X4))
        (COMMENT UPDATE VARIABLES)
        (SETQ X1 (*PLUS X1 F11VAL))
        (SETQ X2 (*PLUS X2 F22VAL))
        (SETQ X3 (*PLUS X3 F33VAL))
        (SETQ X4 (*PLUS X4 F44VAL))
        (PRINT
         (LIST CURRENTNAME
               (ACCESS (CONS (QUOTE TIME) (QUATE DECADES)))
               X1
               X2
               X3
               X4))
        (COMMENT CHECK THE REFLIST)
        (CHECK CURRENTNAME)
        (SUSPEND
         (LIST (CONS (QUOTE X1) X1)
               (CONS (QUOTE X2) X2)
               (CONS (QUOTE X3) X3)
               (CONS (QUOTE X4) X4))
         (QUOTE LAB1))
        (RESUME (QUOTE NEXT))
        (RETURN T)))
EXPR)
```

Figure 5. Source listing of GFIR process function

The next group of statements is used to retrieve the execution environment from the reference list. The conditional statement is used to determine whether or not the instance should terminate at that point. The ACCESS function is used to retrieve the DECADES variable value which is a run variable of the TIME instance of the CLOCK process. This illustrates the interaction of two instances through their variable values. In the code, the function (CONS (QUOTE TIME)(QUOTE DECADES)) evaluates to (TIME . DECADES) which is the dotted pair of reference variable and run variable name needed as the argument of ACCESS.

If the execution is to continue, the control jumps to the LAB1 label. At this label, the f functions are evaluated and their results stored temporarily. These results are used to update the X variables which are printed along with the process name and the system time in units of decades. The process is then suspended and the environment saved. The CHECK function is used to set the reference list to the proper position so the association list and return point label can be stored in the proper place. This is necessary since the ACCESS function has been called during the execution of the instance. ACCESS did not deal with the variables of the currently running instance, so the reference list has been left in a rotated state. Following the suspension of the currently running process, the RESUME function is used to indicate that the next entry in the reference list is to be resumed.

The CLOCK process is a simple function used for incrementing the value of the time variable DECADES during the run (see Fig. 6).

```
(DEFPROP CLOCK
 (LAMBDA (LABEL)
  (PROG (DECADES)
        (SETQ CURRENTNAME (CAAR REFLIST))
        (SETQ DECADES (ACCESS (CONS CURRENTNAME (QUOTE DECADES)))
        (SETQ DECADES (ADD1 DECADES))
        (COND ((EQ DECADES 11)(SETQ REFLIST NIL))
              (T (PUTVAL (LIST (CONS (QUOTE DECADES) DECADES)))))))
EXPR)
```

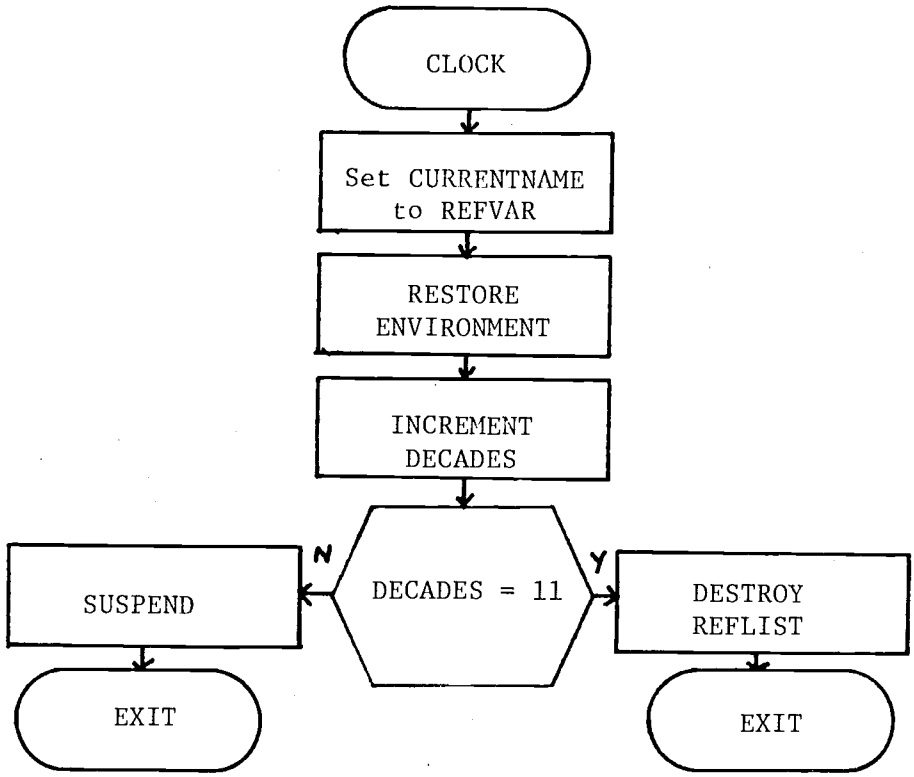


Figure 6. Source listing and flowchart of CLOCK function

Like the GFIR process, it begins by setting CURRENTNAME to its reference variable, although nothing is gained through this action. The next statement retrieves the last value of DECADES from the reference list. The DECADES variable is then incremented by one. The value of DECADES is then tested to determine whether or not the run should be terminated, the CLOCK process sets the reference list to NIL. When this is done, the process is exited, and control is returned to the START function. START checks REFLIST, finds it to be empty, and stops the run. If the run is to be continued, the CLOCK function saves its environment of one variable using the PUTVAL function. Note that since CLOCK has no label in it, the use of (EVAL (CONS (QUOTE GO) (LIST LABEL))) is unnecessary as is the use of the SUSPEND function. The PUTVAL function stores only an association list.

Figure 7a gives the reference list for the run. The X variables in the reference list entries represent the initial conditions for the run. The DECADES variable in the TIME entry gives the initial value of time as the simulation starts.

Output from the program consists of one line for each executed GFIR process. The first element of the output line (list) is the reference variable for the process (see Fig. 7b). The second element is the time increment in decades. The next four elements are the data items calculated by the run. They are X_1, X_2, X_3 and X_4; the diameter at breast height, the height of the tree, the height of the crown base, and the stand density. The units for each have been given previously. These output lines are paired as each of the two GFIR instances is evaluated during each time step. This is an aid in com-

```
((STAND1 GFIR LAB1 ((X1 . 7.0) (X2 . 60.0) (X3 . 30.0) (X4 . 292.0)))
 (STAND2 GFIR LAB1 ((X1 . 6.0) (X2 . 45.0) (X3 . 20.0) (X4 . 350.0)))
 (TIME CLOCK NIL ((DECADES . 1))))
```

Figure 7a. Reference list for tree simulation

```
(STAND1 1 9.7453287 73.671079 39.750917 259.004001)
(STAND2 1 8.930474 60.275111 30.371440 310.450001)
(STAND1 2 11.539682 83.516432 51.350788 229.73654)
(STAND2 2 10.705576 70.569764 43.651391 275.36915)
(STAND1 3 13.014390 91.911458 58.628967 203.77631)
(STAND2 3 12.141639 79.222085 51.327551 244.25243)
(STAND1 4 14.325335 99.478412 64.392556 180.74959)
(STAND2 4 13.408917 86.965809 57.301621 216.65191)
(STAND1 5 15.539959 106.49804 69.329276 160.32489)
(STAND2 5 14.578079 94.118125 62.375795 192.17024)
(STAND1 6 16.694617 113.12613 73.736114 142.20817)
(STAND2 6 15.686431 100.85115 66.882077 170.455001)
(STAND1 7 17.811638 119.45988 77.772269 126.13865)
(STAND2 7 16.756549 107.27076 70.994837 151.19359)
(END OF RUN)
T
*
```

Figure 7b. Output from tree simulation

paring the progress of the two stands as the output lines from STAND1 is adjacent to the output lines from STAND2, where STAND1 and STAND2 are the reference variables for the two GFIR instances.

In terms of the interaction the GFIR processes do not interact at all with each other while both of these processes interact with the CLOCK process. In this simulation, there is no need for the GFIR processes to interact, since each represents a separate stand of trees which is homogeneous with respect to type and size. The CLOCK process, however, is used to keep track of the time which is simulated; a quantity which influences both the GFIR processes (see Fig. 6). This influence makes interaction between GFIR and CLOCK mandatory. The flow of control in this simulation will start with one GFIR process, move to the other instance of GFIR, and then to the CLOCK process (see Fig. 8). This pattern is repeated until a system time of ten decades is reached. When this happens, the GFIR processes terminate themselves as they are entered, and the CLOCK process sets the reference list to NIL, a measure which is certain to terminate the run. When the reference list becomes NIL, the process evaluating function, START, yields control to the LISP system, and the simulation run is over.

In terms of the actual execution of the processes, this run indicates a simple structure of execution. The RESUME function is used only with (QUOTE NEXT) as its argument, and the result is the interative structure found in the flowchart of figure 9. Had the RESUME function been used with a reference variable name, this iterative structure would have broken down into a linear structure.

This illustration is indicative of the kind of work which is nor-

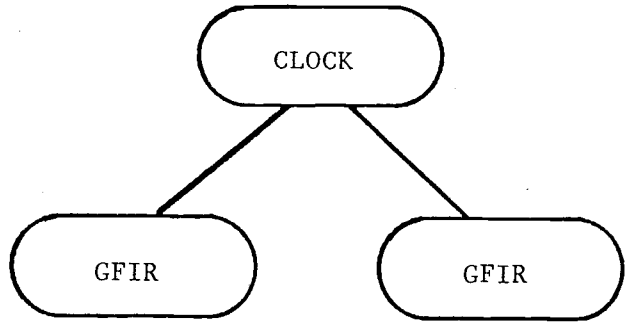


Figure 8. Interaction among the three instances

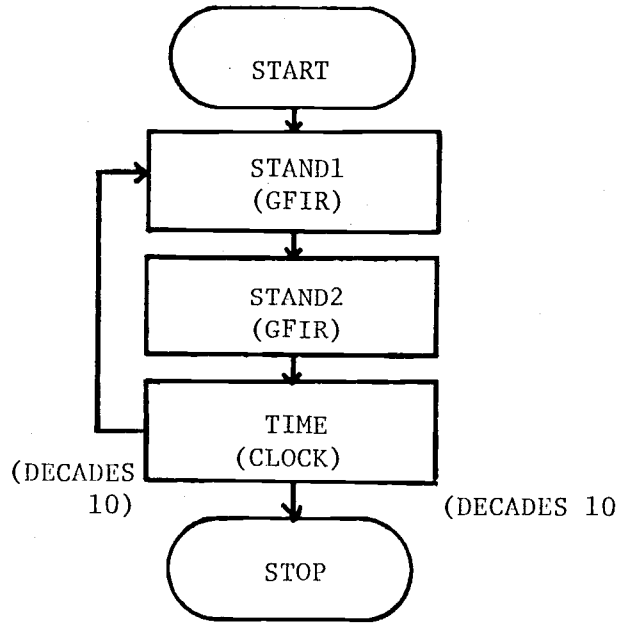


Figure 9. Structure of the simulation

mally done in simulation, but is somewhat out of character for LISP. The power of LISP is derived from its non-numeric data structure orientation and its built-in recursion.

In the implementation of this example, functions had to be written for the extraction of square roots (SQRT), as well as exponential (EXP) and logarithmic (LOG) functions. The basic arithmetic operations are available in LISP, but the user must provide functions for anything other than these. In spite of this obstacle, this kind of simulation is still feasible.

The processes are iterative in nature, a feature which is available through the LISP PROG function, but some of the support functions are written in pure LISP (i.e. written without the PROG feature). In this particular example, recursion is not needed, and a variation of iterations is implied in the way the simulation runs. The GFIR process is called repeatedly and alternates between two running environments. Each time the instance is resumed, it restarts execution at the same label. In effect, this creates a loop with a time delay built into it. The process is suspended at the bottom and resumed in the middle. This illustrates vividly the point that the return point label may be located anywhere in the process body, depending upon the needs of the programmer.

The five primitive functions all come into play in this process. The ACCESS function is used for restoring the process environment. The TERMINATE function is called when the time reaches ten decades. The SUSPEND function saves the process environment prior to the call to the RESUME function. The jump to the return point label is illus-

trated along with the CHECK and COMMENT utility functions.

The Lee Algorithm

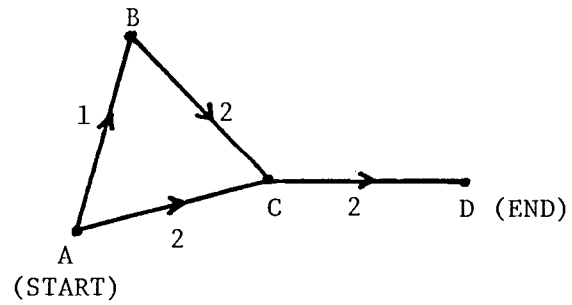
This section presents a discussion of the Lee Algorithm which is used to find the shortest distance through a weighted directed graph. This algorithm has not been implemented, but is included as an illustration of non-numerical computation. As the reader will recall, the Lee algorithm works by sending a simulated pulse through each edge of a directed graph.[2] This pulse travels through the graph and initiates new pulses when it reaches a new node. The algorithm is initiated by sending a pulse through each edge adjacent to the start node. The algorithm terminates when a pulse reaches the end node.

Implementing this algorithm would require an additional data structure. This new data structure, called INFO will be used to hold information about the graph itself. It is basically a list of edges which includes the weight of each edge. This information is used by the instances of the PULSE process to direct the pulses through the graph. Figure 10 contains an example graph and its associated INFO structure. Each edge in the graph has an entry in INFO. For example, there exists an edge between node A and node B in the graph, and the edge has weight one. This edge is represented by the entry (A B 1) in INFO. At each time step, the pulses travel a distance of one along an edge. This distance is arbitrary, but is satisfactory for processing a graph of this size.

The reference list for this run will consist of three entries initially; one entry for each pulse which is to travel through the edges adjacent to the start node A, and one entry for the system

clocking process. For simplicity, the clocking process will not be discussed in detail. It suffices to mention that the clocking process is the last process to be executed during each time step.

The reference variables will initially reflect the names of the edges on which the pulse is moving; for example AB or AC. The process name will be PULSE for each process. The return point label will be the first label in the process. The association list will take on the same form as in the previous example, but in this instance, the variable value slot will be filled by a data structure. The association list will contain only one entry since each instance represents only one pulse, and the appropriate information is easy to represent in a single data structure. The variable name will be the same as the reference variable to facilitate retrieving the associated data structure.



INFO = ((A B 1)(A C 2)(B C 2)(C D 2))

Figure 10. Example graph and its associated INFO list

The first element in the data structure is the name of the node toward which the pulse is traveling. The second element is the length of the edge along which the pulse is traveling. The third element is the distance the pulse has traveled along the edge, and the last element is the list of nodes which the pulse has passed through. The reference list entry will look like:

(AB PULSE LABEL1 ((AB . (B 1 1 (A)))))

As the processes run, the size of the reference list will not decrease until the end node is reached. As a pulse passes a node, new pulses are generated, one for each edge leaving the node. These new pulses will have new reference variable names, new run variable names, and a list of visited nodes which will include all previous nodes (from the pulse entering the node just visited) as well as the last node visited. The name of the next node to be visited is taken from the INFO list along with the distance to that node. The distance traveled along the new edge is (n * d - 1), where 1 is the length of the last edge is taken from the instance of the pulse which visited the last node, d is the distance traveled by a pulse during one time step, and n is an integer greater than zero such that n * d is greater than or equal to the length of the last edge.

The algorithm for the process is then:

1) for each entry in REFLIST do statements 2 through 5

2) is entry referring to CLOCK? If yes, go to 5; If no, go to 6

3) is pulse at or past end node?

   If yes, search REFLIST for any other pulses which are at or past the end node. Output the nodes visited for each

pulse found and stop.

4) is pulse at or past a new node?

If yes, initiate new pulse instances using:

a) look up next-node and distance to next-node in INFO

b) add old next-node to old nodes-visited giving new-nodes visited.

c) distance traveled = n*d - old distance-to-node.

d) generate new atom from first two elements of INFO entry. Use this for reference variable and run variable names.

e) use same process name and label.

Append these new instances to REFLIST, delete the instance representing the pulse before it past the most recent node, and go to 2.

If no, increment distance traveled along edge, rotate REFLIST go to 3.

5) increment time value

6) rotate REFLIST; go to 2

This algorithm is easily applied to the graph in figure 10. The reference list is set up with three entries, one for each of the two pulses initiated in edges AB and AC, and one for the system clock. REFLIST takes the form:

```
((AC PULSE LABEL ((AC . (C 2 0 (A)))))
 (AB PULSE LABEL ((AB . (B 1 0 (A)))))
 (TIME CLOCK LABEL1 ((SECONDS . 0))))
```

After the first time increment, the pulse in edge AB reaches B, and the pulse in AC is halfway to C. The reference list becomes:

```
((BC PULSE LABEL ((BC . (C 2 0 (A B)))))
 (AC PULSE LABEL ((AC . (C 2 1 (A)))))
 (TIME CLOCK LABEL1 ((SECONDS . 1))))
```

The pulse in AC reaches C during the next time step, and the pulse in BC is halfway between B and C. The reference list becomes:

```
((BC PULSE LABEL ((BC . (C 2 1 (A B)))))
 (CD PULSE LABEL ((CD . (D 2 0 (A C)))))
 (TIME CLOCK LABEL1 ((SECONDS . 2))))
```

The next time step finds edge CD with pulses in it, separated by one unit of distance. The reference list becomes:

```
((CD PULSE LABEL ((CD . (D 2 0 (A B C)))))
 (CD PULSE LABEL ((CD . (D 2 1 (A C)))))
 (TIME CLOCK LABEL1 ((SECONDS . 3))))
```

The last time step is needed for the leading pulse in CD to reach D. At this point, the node D is added to the list of nodes visited, and the run stops after printing the list of nodes (A C D).

This example shows that the function package can be used for a very different problem. Rather than using the function package for a numerically-oriented simulation model, the same functions may be used for finding the shortest path through a directed graph. The process function PULSE itself will have very much the same form as the GFIR process function. The process will be executed straight through rather than being suspended in mid-execution and resumed at a later time. The SUSPEND function will be used to update the reference list entry of a pulse which is between nodes, but the process must have the capability of generating a new pulse instance and appending the information for that pulse onto the reference list. The RESUME function will again be used with the function (QUOTE NEXT) as its argument since the structure of execution of the processes will

be iterative. This time, however, the number of sublists in the reference list will vary during the run. The TERMINATE function will not be needed if only one shortest path is desired. The run could also be structured to output the paths through the graph in order of their length. In this case, the TERMINATE function could be used to destroy the pulses as they pass the end node. If only one shortest path is needed, the PULSE process could destroy the reference list when the shortest path is found. In accessing the environment of the PULSE instance, simplicity is gained by using the same name for the reference variable and the run variable. The function:

```
(SETQ CURRENTNAME (CAAR REFLIST))
```

will be executed first upon entering the process. To retrieve the environment, the ACCESS function is called using CURRENTNAME twice:

```
(ACCESS (CONS CURRENTNAME CURRENTMANE))
```

Again, the START function is used to begin the execution of the simulation run.

LISP is oriented toward list processing rather than numerical computation. Since the function package is written in LISP, it naturally assumes many of the characteristics of LISP. Although this second example is more in the character of LISP, the grand fir tree model shows that LISP, and the function package can handle numerical computations as well as the list processing for which it was intended.

V. CONCLUSIONS AND IMPLICATIONS FOR FURTHER RESEARCH

Using the primitive functions in the package, the user will be equipped with some powerful tools for creating a simulation model, an implementation of a parallel processing parser, or any of a number of other applications. However, it should also be noted that the other utility functions may be used when the full power of the five primitive functions is not needed. For example, if the process is looping at the time of suspension and the proper label is already contained in the reference list entry, the PUTVAL function may be used for storing the association list since storing the return point label will be unnecessary. Using the utility functions in this way can save on some of the bookwork which is necessary in using the functions.

Another interesting application involves the use of the LISP primitive functions on the reference list. The reference list is not protected from modification by any means. A symbol may be generated and placed in a list along with the proper information such as a process name, label, and initial variable values. This list may be appended to the reference list, and in this way, a new instance of a process may be generated at execution time. Using this technique, the reference list becomes a dynamic data structure, shrinking and swelling during the simulation run. This technique was illustrated in the discussion of the Lee algorithm and is useful when the total number of instances needed for a run is not initially known. The user specifies the instances needed to begin the run, and the process instances generate new instances as needed.

This function package was written to provide a basis for parallel processing in LISP. It has been implemented as a group of utility functions and data structures supplementing the LISP system rather than being integrated into it. This has been done because of a problem with the LISP system. LISP contains a context stack and special push-down list for holding information while recursive functions are executing. These structures hold data such as variable values and return point markers; information which the function package could use to great advantage. However, LISP also uses these two data structures for holding temporary pointers, a fact which is invisible to the user. These pointers make examination of the context stack and special push-down list difficult, as the investigator never quite knows what will be in these structures.

An avenue for further investigation is the implementation of these functions as an integrated part of the LISP system. The advantages to the user would be immediate. The execution environments of the processes could be saved using information in LISP's tables and stacks. This would simplify the calling of the SUSPEND function. The processes themselves could be defined as recursive functions and the point of return for continued execution could be stored by the LISP system rather than forcing the user to use a label and (GO LABEL) expression. Similarly, upon entering or resuming a process, the system could restore the environment for that instance, and the user would not have to perform as much bookkeeping. The user could then concentrate on treating the process as being strictly functional as opposed to performing a function and juggling its environment as well.

The function package allows a great deal of flexibility in the scheduling of the events which take place during the run. The RESUME function can be used to schedule the next instance to be executed, or it can specify the next event as being whatever is next in the reference list. This flexibility must be used with caution, however. In the tree model discussed earlier, if the GFIR processes reschedule each other and the CLOCK function is not allowed to execute until the GFIR processes are terminated, the trees simulated will have the appearance of growing to maturity within one ten-year period.

This function package does not force the user to use a rigorous execution structure for the process instances. The processes are not treated as subroutines or functions which are always subservient to a calling routine. Processes execute without returning a value to a calling program, and for that reason, can be executed in any order. The processes may be scheduled by the user, or written in a way which causes them to schedule themselves.

The user should note that the package functions are an aid in simulating parallel processing, and as such, provide a minimum of error checking. There is no provision for detecting a deadlock, and once a deadlock occurs, there is nothing in the package which will aid the user in breaking the deadlock. An infinite loop can form if one instance is in the reference list which is never terminated by any other instance, and does not terminate itself.

The Grand Fir growth model is a program which does not make the best possible use of LISP, and a few remarks on its performance are in order. LISP is a language which is oriented toward manipulating sym-

bols rather than numbers. Some basic numeric functions are available in LISP, but functions such as SQRT, LOG, and EXP had to be written by the programmer and run under the LISP interpreter. The numeric functions, the interpretive LISP system and the time-sharing operating system of the PDP-10 were three factors which caused the run to give very poor response time. The output in figure 7b took nearly an hour to compute and print.

A second disadvantage of the function package in this form is the large amount of core space which is taken up by the functions when they are loaded into the LISP interpreter. The functions themselves take up approximately 20K core locations. This leaves very little room for defining processes without expanding core. The Grand Fir simulation was run during the PDP-10's non-peak traffic hours when 40K locations were available.

The slow response time of the Grand Fir program is due partially to the numerical nature of the computations involved; test cases involving non-numeric computations gave much better response times. However, the problem of limited storage space remains an important unresolved consideration. An additional avenue for further investigation is the minimization of storage space used by the function package.

This function package is presented as a beginning at developing functions and data structures needed for the simulation of parallel processing in LISP. Parallel processing in LISP can be used to great advantage not only with simulations, but also with symbolic computations such as graph traversals and parsing algorithms. Although this function package has been implemented as a group of functions and data

structures, the functions could be better implemented as compiled functions integrated into the LISP system. The functions would execute faster in this form as well as having the advantages already mentioned. However, the functions in their present state show that the approach used is feasible.

VI. BIBLIOGRAPHY

1) Tsichritzis, D., and P. Berstein, Operating Systems, Academic Press, New York, 1974.

2) Dahl, O. J., E. W. Dijkstra, and C. A. R. Hoare, Structured Programming, Academic Press, New York, 1972.

3) Birtwhistle, Dahl, and K. Nygaard, SIMULA Begin, Auerbach, New York, 1969.

4) Moses, J., "The Function of FUNCTION in LISP or Why the FUNARG Problem Should Be Called the Environment Problem", MIT Project MAC, Pub AI-199, MAC-M-428, Cambridge, Massachusetts, 1970.

5) Weizenbaum, J., "The FUNARG Problem Explained", MIT Project MAC, Cambridge, Massachusetts, 1968.

6) Dahl, O. J., and K. Nygaard, "SIMULA - an ALGOL-Based Simulation Language", Communications of ACM, Vol. 9, number 9, 1968, pp. 671-678.

7) Knuth, D. E., The Art of Computer Programming, Vol. 1, Addison-Wesley Pub. Co., Reading, Massachusetts, 1973, pp. 183.

8) Smith, R., and F. Rawson, "A Multiprocessing Model of Natural Language Understanding", Stanford University, 1976.

9) Horning, J. J., and B. Randall, "Processing Structuring", ACM Computing Surveys, Vol. 5, number 1, 1973, pp. 5-30.

10) Maurer, W. D., The Programmer's Introduction to LISP, Macdonald/ American Elsevier, New York, 1972.

11) Colin, A. J. T., Introduction to Operating Systems, Macdonald/ American Elsevier, New York, 1971.

12) Weissman, C., LISP 1,5 Primer, Dickenson Pub. Co., Belmont, California, 1967, pp. 13.

13) Krajicek, J. E., K. A. Brinkman, and S. F. Gingrich, "Crown Competition - A Measure of Density", Forest Science, Vol. 7 number 1, 1961, pp. 35, 42.

14) White, C., and W. S. Overton, "Users Manual for the FLEX2 and FLEX3 Model Processors for the FLEX Modelling Paradigm", Bulletin 15, Forest Research Lab., Oregon State University, Corvallis, Oregon, 1974.

15) Fishman, G. S., Concepts and Methods in Discrete Event Digital Simulation, Wiley-Interscience, New York, 1973.

16) Stage, A., "A Tree-by-Tree Measure of Site Utilization for Grand Fir Related to Stand Density Index", U.S. Forest Service Res. Note Intermountain Forest Range Experiment Station, No. INT-77, 1968.

APPENDICES

Appendix A - Listings & Flowcharts of Package Functions

ACCESS

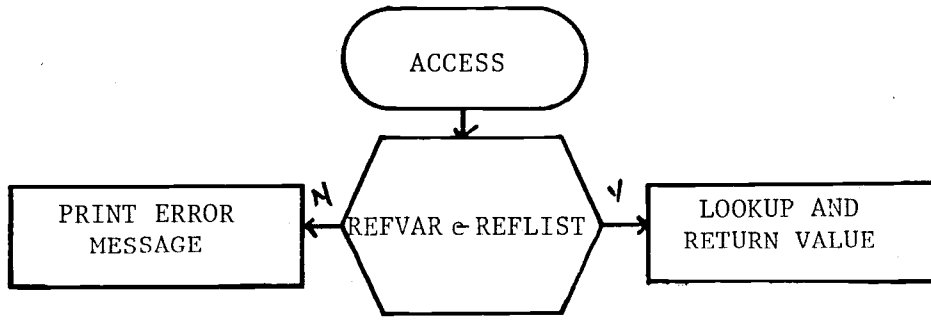
```
(DEFPROP ACCESS
 (LAMBDA(PAIR)
  (COND ((CHECK (CAR PAIR))
         (CDR (ASSOC (CDR PAIR)(CADDDAR REFLIST))))
        (T (PRINT (QUOTE (ERROR IN ACCESS - REFVAR NOT FOUND))))))
EXPR)
```

PURPOSE: used to look up values of variables local to other processes

PARAMETER: dotted pair of reference variable and local variable name

CALLED BY: user program

COMMENTS: returns garbage if local variable name is not found



CADDDAR

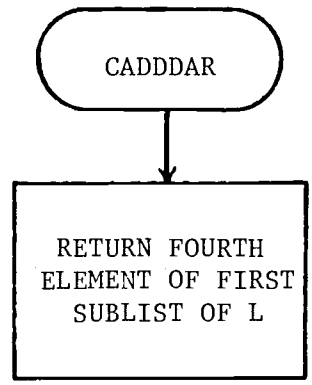
```
(DEFPROP CADDDAR
 (LAMBDA (L)
  (CAR (CDDDAR L)))
EXPR)
```

PURPOSE: utility function

PARAMETER: a list, usually REFLIST in function package

CALLED BY: function package

COMMENTS: easy way of accessing association list in CAR REFLIST



CHECK

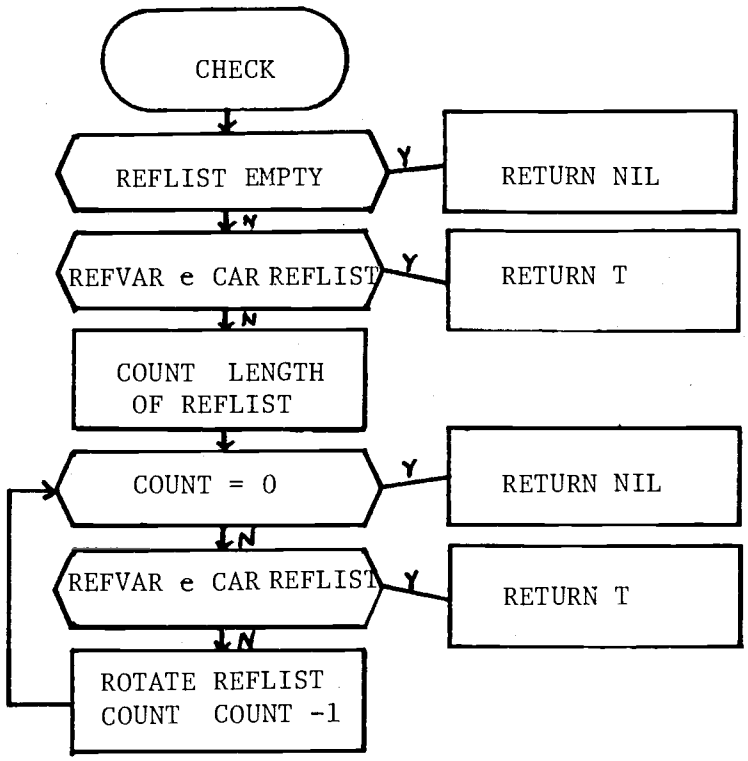
```
(DEFPROP CHECK
 (LAMBDA(REFVAR)
  (PROG (COUNT)
        (COND ((NULL REFLIST)NIL)
              ((EQ REFVAR (CAAR REFLIST)) T))
        (SETQ COUNT (LENGTH REFLIST)
  A     (COND ((EQ COUNT 0)(RETURN NIL)))
        (COND ((EQ REFVAR (CAAR REFLIST))(RETURN T)))
        (ROTATE)
        (SETQ COUNT (SUB1 COUNT))
        (GO A)))
EXPR)
```

PURPOSE: see if reference variable is in REFLIST

PARAMETER: reference variable

CALLED BY: package functions

COMMENTS: if reference variable is found, REFLIST is left with REFVAR in CAR and T is returned. If not found, REFLIST is left as it was, and NIL is returned.



COMMENT

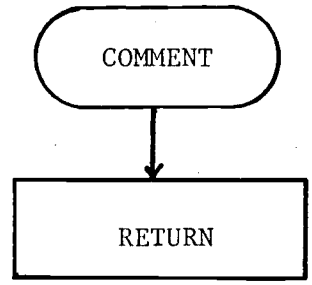
```
(DEFPROP COMMENT
 (LAMBDA (L) NIL)
FEXPR)
```

PURPOSE: allows the user to insert comments into the body of the function

PARAMETER: comment string

CALLED BY: user program

COMMENTS: none



PUTLAB

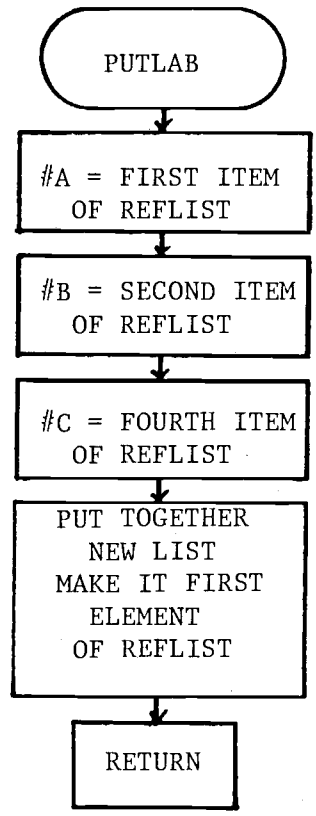
```
(DEFPROP PUTLAB
 (LAMBDA(LABEL)
  (SETQ #A (CAAR REFLIST))
  (SETQ #B (CADAR REFLIST))
  (SETQ #C (CADDDAR REFLIST))
  (SETQ REFLIST (APPEND (LIST #A #B LABEL #C))(CDR REFLIST))))
EXPR)
```

PURPOSE: store return point label into first sublist of REFLIST

PARAMETER: label to be stored

CALLED BY: SUSPEND function

COMMENTS: does not check to see that proper sublist is in CAR REFLIST



PUTVAL

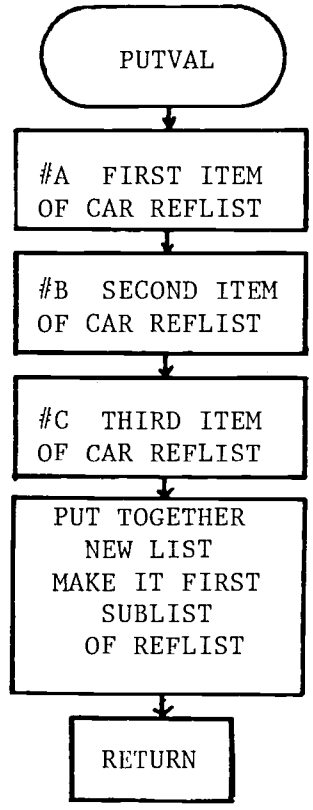
```
(DEFPROP PUTVAL
 (LAMBDA(LIST)
  (SETQ #A (CAAR REFLIST))
  (SETQ #B (CADAR REFLIST))
  (SETQ #C (CADDAR REFLIST))
  (SETQ REFLIST (APPEND (LIST (LIST #A #B #C LIST))(CDR REFLIST))))
EXPR)
```

PURPOSE: put list of variable names and values into first sublist of REFLIST

PARAMETERS: function name

CALLED BY: user program

COMMENTS: no checking for correct entry in CAR REFLIST



RESUME

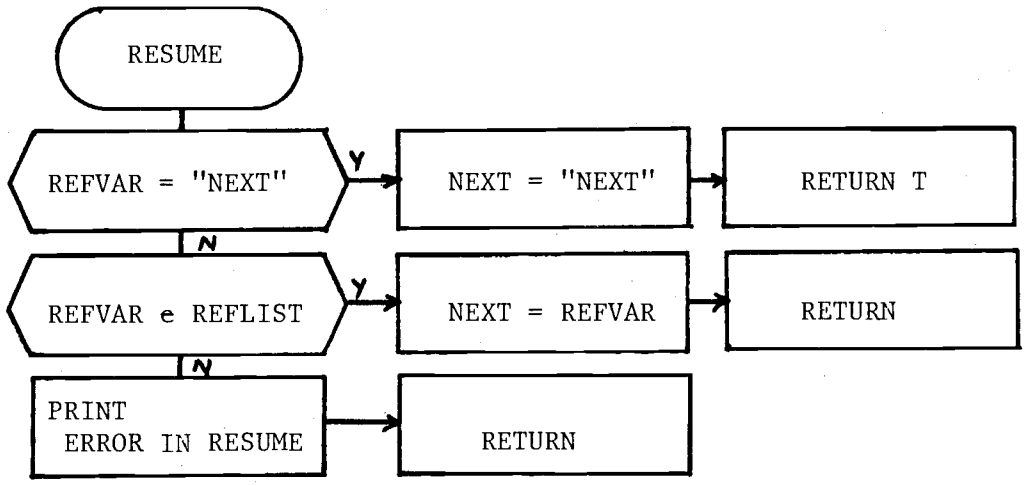
```
(DEFPROP RESUME
 (LAMBDA(REFVAR)
  (COND ((EQ REFVAR (QUOTE NEXT))
         (SETQ NEXT (QUOTE NEXT))
         (RETURN T))
        ((CHECK REFVAR)(SETQ NEXT REFVAR))
        (T (PRINT (QUOTE (ERROR IN RESUME - REFVAR NOT FOUND))))))
EXPR)
```

PURPOSE: set NEXT variable for continuation of a process after suspension

PARAMETER: reference variable

CALLED BY: user program

COMMENTS: Sets NEXT variable as a flag for START, which uses it to find the appropriate process



ROTATE

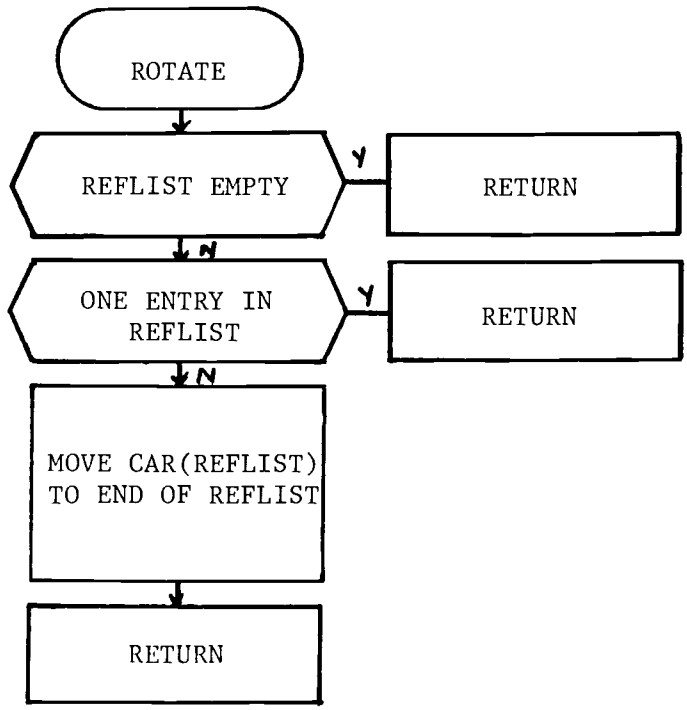
```
(DEFPROP ROTATE
 (LAMBDA NIL
  (COND ((NULL REFLIST) NIL)
        ((NULL (CDR REFLIST)) NIL))
  (SETQ REFLIST (APPEND (CDR REFLIST)(LIST (CAR REFLIST)))))
EXPR)
```

PURPOSE: rotate reflist if two or more entries are present

PARAMETER: none

CALLED BY: CHECK routine or user program

COMMENTS: none



START

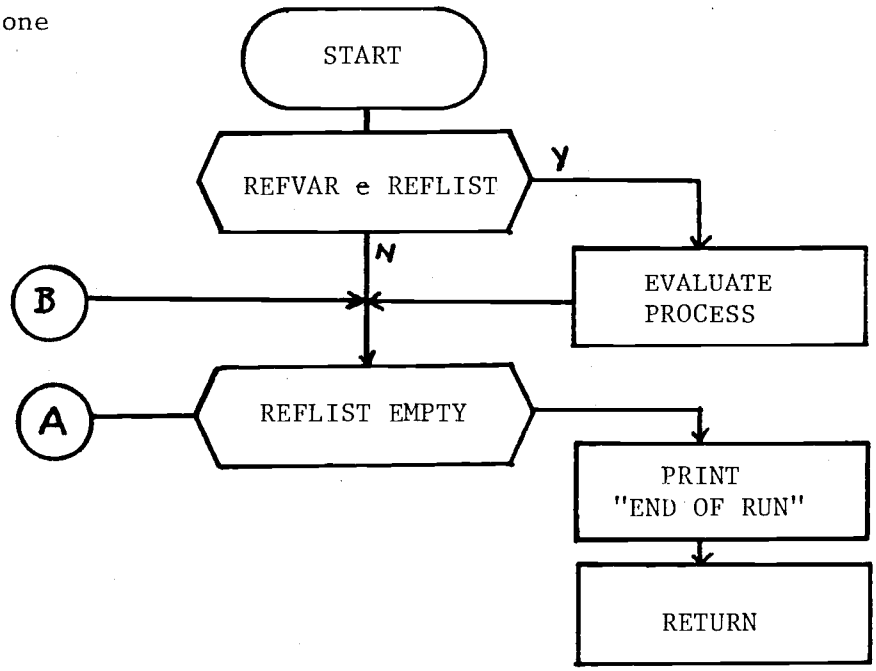
```
(DEFPROP START
 (LAMBDA(REFVAR)
  (PROG NIL
        (COND ((CHECK REFVAR)(EVAL (LIST (CADAR REFLIST) (QUOTE
                                          (CADDAR REFLIST))))))
  A     (COND ((NULL REFLIST)(PRINT (QUOTE (END OF RUN)))(RETURN T))
              ((EQ NEXT (QUOTE NEXT))
               (ROTATE)
               (SETQ NEXT (QUOTE NEXT))
               (EVAL (LIST (CADAR REFLIST)(QUOTE (CADDAR REFLIST))))
               (GO A))
              ((CHECK NEXT)(SETQ NEXT (QUOTE NEXT))
               (EVAL (LIST (CADAR REFLIST)(QUOTE (CADDAR REFLIST))))
               (GO A)))))
EXPR)
```

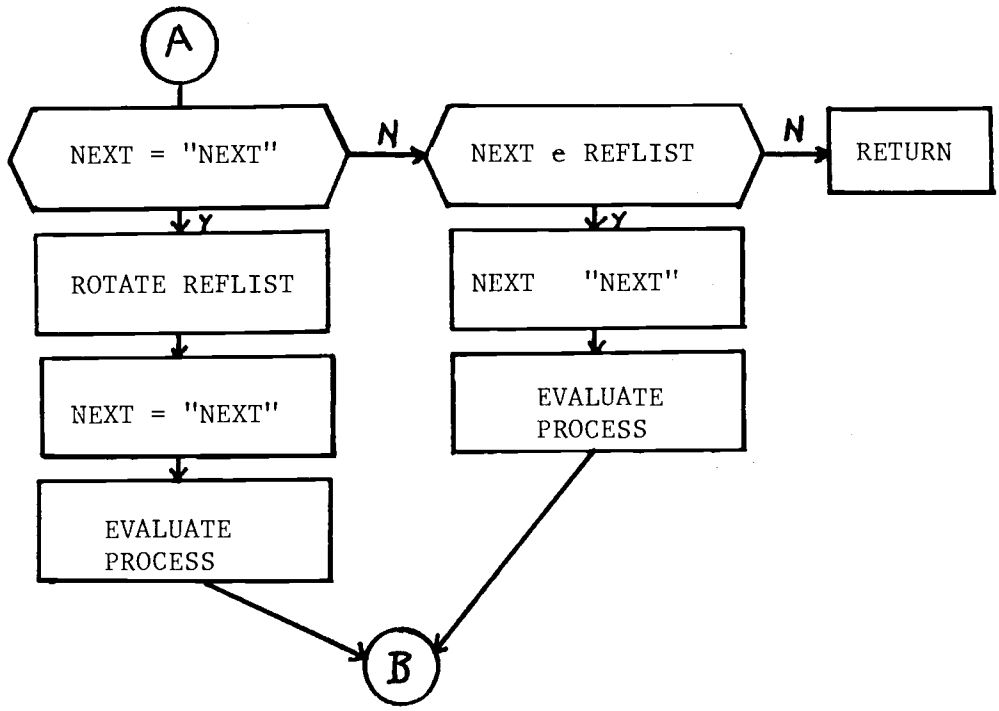
PURPOSE: begin the run and control execution

PARAMETER: reference variable

CALLED BY: user

COMMENTS: none




A
NEXT = "NEXT"
N
NEXT e REFLIST
N
RETURN
Y
Y
ROTATE REFLIST
NEXT "NEXT"
NEXT = "NEXT"
EVALUATE PROCESS
EVALUATE PROCESS
B

SUSPEND

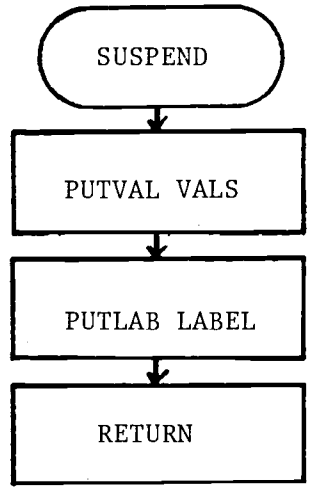
```
(DEFPROP SUSPEND
 (LAMBDA(VALS LABEL)
  (PUTVAL VALS)
  (PUTLAB LABEL))
EXPR)
```

PURPOSE: save execution environment and return point label

PARAMETER: reference variable

CALLED BY: user program

COMMENTS: none



TERMINATE

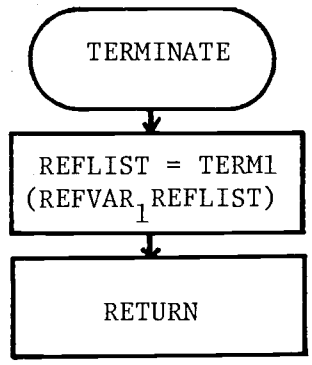
```
(DEFPROP TERMINATE
 (LAMBDA(REFVAR)
  (SETQ REFLIST (TERM1 REFVAR REFLIST)))
EXPR)

(DEFPROP TERM1
 (LAMBDA(REFVAR REFLIST)
  (COND ((NULL (CHECK REFVAR)) REFLIST)
        (T (CDR REFLIST))))
EXPR)
```

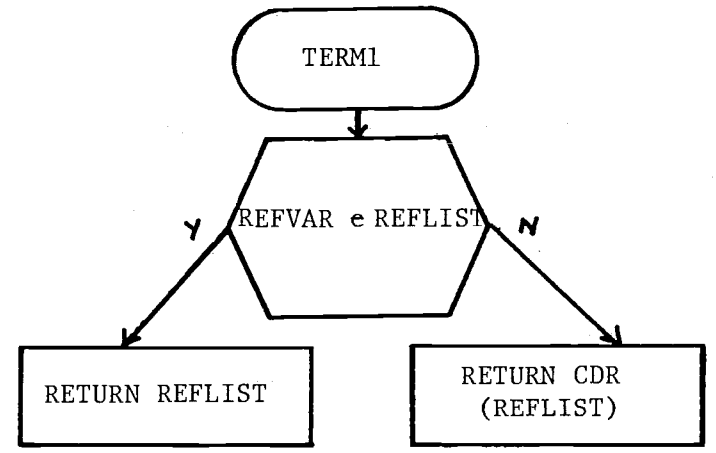
PURPOSE: terminates process associated with reference variable

PARAMETER: reference variable

CALLED BY: user program

COMMENTS: no error if reference variable not found in reference list





Appendix B - Definitions

An argument is an independent variable used by a function. In terms of its application, it is a particular value supplied to the function at the time the function is to be executed.(10)

An atom is a unit which cannot be divided further. In LISP, an atom may be a variable or a function name and is represented by a sequence of characters containing no blanks.(10)

A binding is a one-to-one correspondence between formal parameters and actual parameters. A LISP binding consists of the assignment of a pointer to the value of a variable.(10)

A dotted pair is a special kind of two element list. A normal two element list (A B) in LISP contains two words of memory. The first half of the first word contains a pointer to a variable location A, and the second half contains a pointer to the second word of the list. The second word is divided in half as well. The first half points to the location B, and the second half points to NIL, the list terminating symbol. In the case of the dotted pair (A . B), only one word of memory is contained in the list. The first half of the word points to the location A, while the second half points to B.(10)

An element is an atom, a list of atoms, or a list of lists.(12)

An environment is that part of the world which the processor can directly sense or alter.(11)

A function is a unit of computer code which performs some action

and returns a single value to its caller.

An instance is an example or a case of an object. Given an object "vehicle", two instances of it are "car" and "bus". The word "instance" is used to distinguish between different occurrences of the same process, each having different sets of local variable values.

A list is an ordered collection of data. Recursively defined, a list is an atom or a list of lists.

A parameter is a characteristic element which governs the actions of a function.

A processor is a pair (D, I) where D is a physical device which can be placed in specified initial states and I is an interpretation of its physical status. Each sequence of states from an initial state is a computation of the processor.(9)

A simulation is an imitative representation of the functioning of an object or system.

A stand is a group of plants growing in a continuous area.

A state variable is an elementary quantity which can assume certain well-defined values.(9)

A state variable set is a set of state variables.(9)

Appendix C - Support Functions Used in the Example

F11

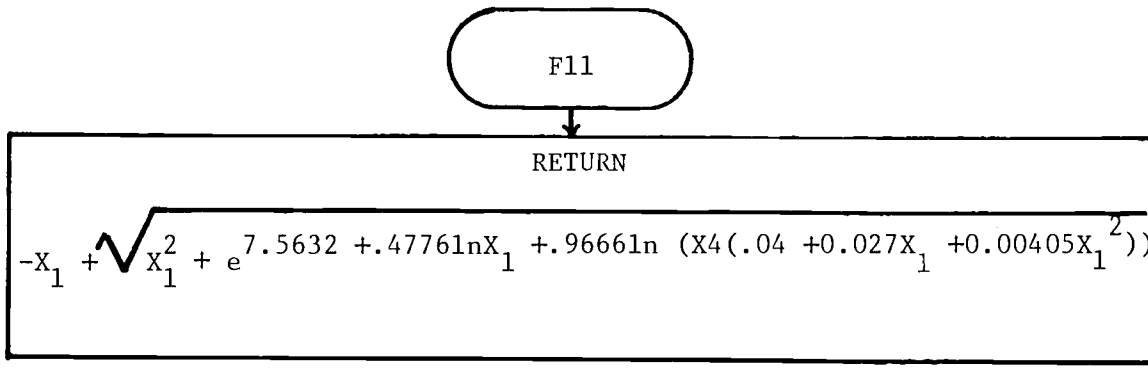
```
(DEFPROP F11
 (LAMBDA(X1 X4)
  (*PLUS (MINUS X1)
         (SQRT
          (*PLUS (*TIMES X1 X1)
                 (EXP
                  (PLUS 7.5632699
                        (*TIMES 0.47760000 (LOG X1))
                        (*TIMES -0.96660000
                                (LOG
                                 (TIMES X4
                                        (PLUS 0.40000000E-1
                                              (*TIMES 0.26999999E-1 X1)
                                              (TIMES 0.40499999E-2 X1X1
))))))))))
EXPR)
```

PURPOSE: flow function for box #1 of tree model

PARAMETERS: diameter of tree at breast height, number of trees/acre

CALLED BY: tree model process

COMMENTS: from FLEX-form



F22

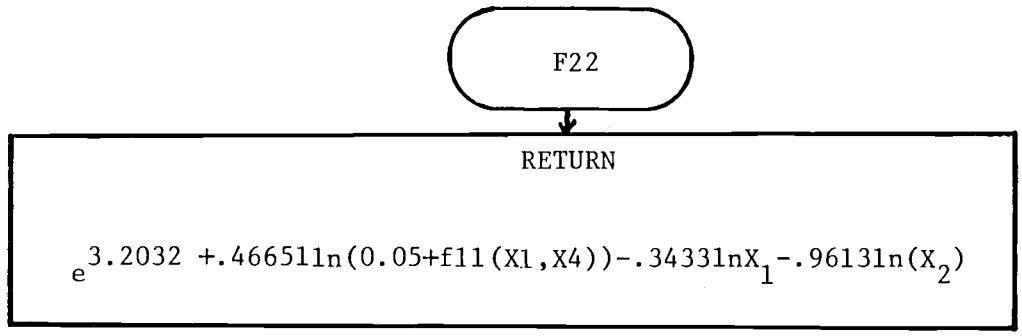
```
(DEFPROP F22
 (LAMBDA(X1 X2 F11VAL)
  (EXP
   (PLUS 3.2032000
         (*TIMES 0.46649999 (LOG (*PLUS F11VAL 0.5E-1)))
         (*TIMES -0.34632999 (LOG X1))
         (*TIMES -0.96129999E-1 (LOG X2)))))
EXPR)
```

PURPOSE: flow function for box #2 at tree model

PARAMETERS: trunk diameter at breast height, value of F11 flow function

CALLED BY: Tree model process

COMMENTS: from FLEX-form



F33

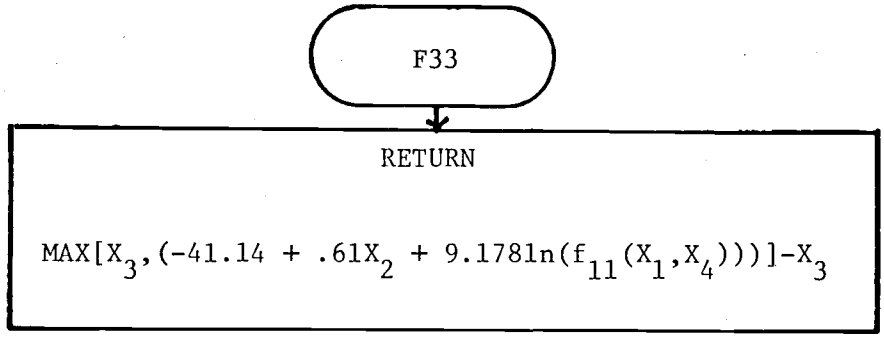
```
(DEFPROP F33
 (LAMBDA(X1 X2 X4)
  (*PLUS (MINUS X3)
         (MAX X3
              (PLUS -41.139997
                    (*TIMES 0.60999999 X2)
                    (*TIMES 9.1779998
                            (LOG
                             (TIMES X4
                                    (PLUS 0.40000000E-1
                                          (*TIMES 0.26999999E-1 X1)
                                          (TIMES 0.40499999E-2 X1 X1))))
)))))
EXPR)
```

PURPOSE: flow function for box #3 of tree model

PARAMETERS: height of tree, height of tree crown base, value of flow function F11

CALLED BY: tree model process

COMMENTS: taken from FLEX-form



F44

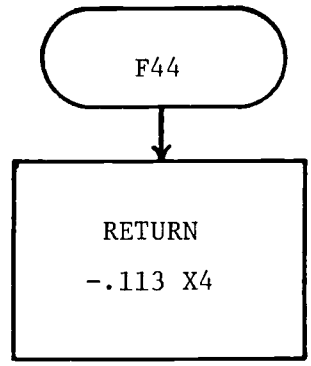
```
(DEFPROP F44
 (LAMBDA (X4)
  (*TIMES -0.113 X4))
EXPR)
```

PURPOSE: flow function for box #4 of tree model

PARAMETER: number of trees/acre

CALLED BY: tree model process

COMMENTS: from FLEX-form



SQRT

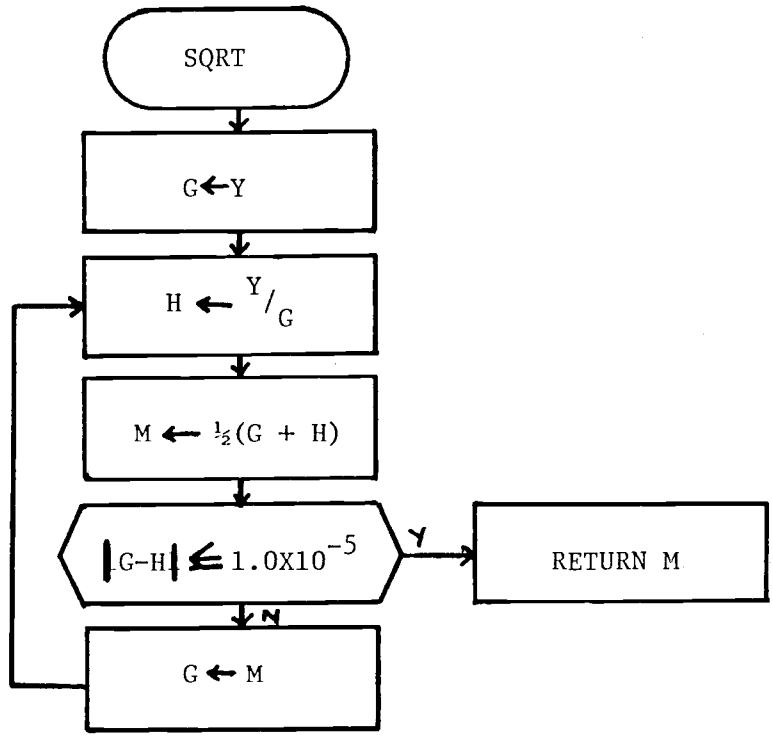
```
(DEFPROP SQRT
 (LAMBDA (Y)
  (PROG (M G H)
        (SETQ G Y)
  A     (SETQ H (*QUO Y G))
        (SETQ M (*TIMES 0.5 (*PLUS G H)))
        (COND ((*LESS (ABS (*DIF G H)) 1.0E-5) (RETURN M))
              (T(SETQ G M)(GO A)))))
EXPR)
```

PURPOSE: extract square root of a number

PARAMETER: number

CALLED BY: User program

COMMENTS: Newton's method



LOG

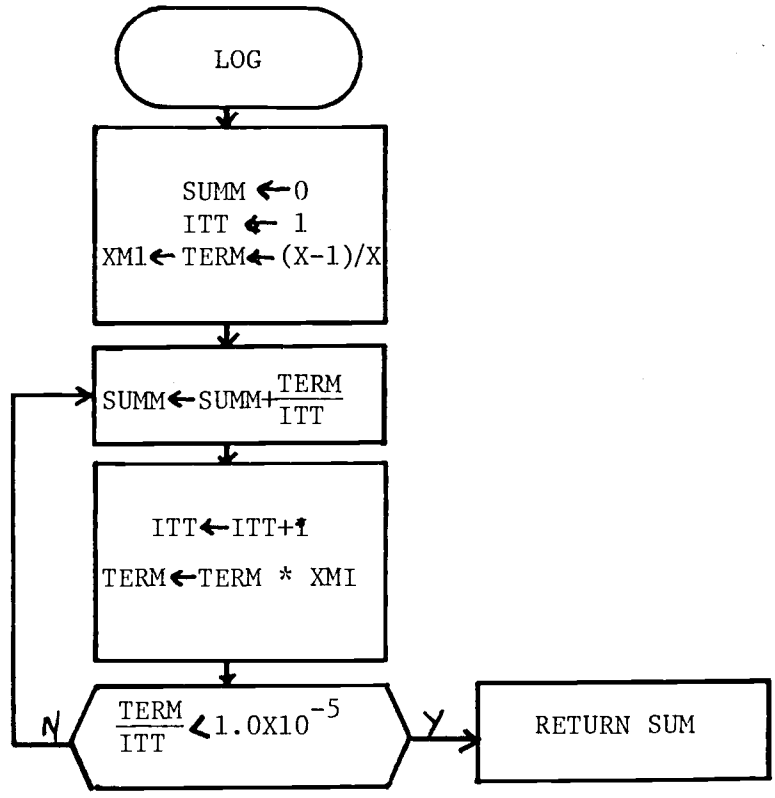
```
(DEFPROP LOG
 (LAMBDA(X)
  (PROG (XM1 TERM SUMM ITT)
        (SETQ SUMM 0.0)
        (SETQ ITT 1.0)
        (SETQ TERM (*QUO (SUB1 X) X))
        (SETQ XM1 TERM)
  A     (SETQ SUMM (*PLUS SUMM (*QUO TERM ITT)))
        (SETQ ITT (ADD1 ITT))
        (SETQ TERM (*TIMES TERM XM1))
        (COND ((*LESS (*QUO TERM ITT) 1.05-5) (RETURN SUMM)) (T (GO A)))
))
EXPR)
```

PURPOSE: extract natural log of a number

PARAMETER: positive number

CALLED BY: user program

COMMENTS: Taylor series expansion



EXP

```
(DEFPROP EXP
 (LAMBDA(X)
  (PROG (TERM SUMM ITER)
        (SETQ SUMM 1.0)
        (SETQ ITER 1.0)
        (SETQ TERM (*QUO X ITER))
  A     (SETQ SUMM (*PLUS SUMM TERM))
        (SETQ ITER (ADD1 ITER))
        (COND ((*LESS TERM 1.05-5) (RETURN SUMM)))
        (SETQ TERM (*TIMES TERM (*QUO X ITER)))
        (GO A)))
EXPR)
```

PURPOSE: exponential function e^x

PARAMETER: number

CALLED BY: user program

COMMENTS: Taylor series expansion