Massachusetts Institute of Technology

Laboratory for Computer Science

Cambridge, Massachusetts

Computation Structures Group Memo 209

Data Should Not Change: A Model for a Computer System[1]

by
Jack B. Dennis
July, 1981
Revised July 1985

Abstract: Our goal is to apply the concepts of functional programming and data flow computer architecture to the design of general purpose computer systems. We envision that such a computer system will support many users working at personal terminals and sharing information residing in a central system; the central system will hold all information in on line storage and provide access to it through a universal referencing mechanism using unique identifiers with uniform meaning for all users of the system. We advocate basing the design of such computer systems on a formal interpreter that provides an operational semantics of a data flow *base language* for the system. Furthermore, we propose that such systems be designed so that data is created, used and abandoned, but that *data never changes.* The base language presented here is determinate acyclic data flow program graphs with an apply operator, special operators to support *early-completion* construction of records, and a *guardian* construct for monitoring nondeterminate access to shared data.

Use of a data flow base language allows a large degree of concurrency in the execution of computations. This is consonant with the necessary trend toward distributed and data driven architectures for future computer systems.

---

[1]The ideas discussed in this paper were developed with support from the National Science Foundation, the U. S. Department of Energy, and the Advanced Research Projects Agency of the U. S. Department of Defense.

Introduction

The M. I. T. Computation Structures Group is applying the concepts of functional programming and data flow computer architecture to the design of general purpose computer systems. In the Vim project we are building an experimental computer system specified by a *base language* derived from data flow program graphs.

The Vim project is a continuation of work reported in [12] and [11], and has its origins in the work on "capabilities" I published jointly with my first doctoral student, Earl Van Horn [10]. The goal of our work on capabilities was to present a system model for a kind of ideal multiprogrammed computer system. Such a system would serve many users in a way that permitted sharing the products of their individual programming efforts without sacrificing the principles of program modularity—the ability to build program units which can be combined to form higher units, which in turn can be further combined.

Despite the passing of years, no practical computer system has achieved our original goals. This is not to say there have been no close calls. The essential requirements for modular programming in the context of one sequential process operating within a very large address space were developed and implemented in Multics [7, 5]. However, because each process in Multics runs in its own separate address space, programmers are obliged to go outside the coherent support of the virtual address space to share information among several processes or to coordinate concurrent activities of processes. The Unix operating system, although suffering from the address space limitations of the DEC PDP-11, introduced a very significant form of modular program construction using *pipes* [27]. Among the several systems designed around the concept of capabilities [19, 35, 32], I think the Chicago Magnum machine [19] was closest to realizing our original objectives, but the project was never completed. More recently, the IBM System 38 [23] has taken the courageous step of assigning a unique identifier to every object residing in the system and requiring that all accesses to an object be by means of its unique code. Unfortunately, in System 38 this beautiful idea is hidden under so many intricate facilities that it is difficult to discover whether any reasonable user-related goal has been achieved.

Let us speculate why the goals have been so hard to reach. From the user's viewpoint the failings of systems such as Multics and System 38 have been their inefficiency and the

complexity of the user interface.

I suspect that the complexity exists because the designers did not have in mind a complete, clean and simple model for the facilities to be provided to the user. Rather, certain characteristics believed to be desirable were provided by making minimal changes to the conventional structure of computer hardware. Since the ideal behavior of the system was never carefully formulated, there is a mismatch—the semantic gap—with the actual implementation. The user encounters complexity when a needed feature is affected by the gap.

I suspect the efficiency problems arise because simple variants of conventional computer architecture do not suffice to efficiently support the desired system behavior. Radically different structures may yield a superior result. Perhaps the mechanisms needed to implement a significantly improved user interface are at odds with conventional sequential instruction execution.

In a paper for the IFIP Congress 1968 [11], I argued that achieving "programming generality"—meaning general support for modular programming— in a computer system with a memory hierarchy spanning orders of magnitude in access time requires exploiting fine grain parallelism to achieve efficient operation. My reasoning starts from two assumptions concerning memory hierarchy and modular programming:

- Large application programs will be required by economics to operate with their information structures distributed among the levels of a hierarchical memory.
- In the construction of a large program, many parts of the program are written independently and in a way that the author of one module need not know the internal implementation details of other modules.

I expect the first assumption is agreeable to most readers in spite of the declining cost of memory chips. The second is the basis of modular software and is essential, in my view, to any significant progress in our ability to write large programs with less effort.

Our first conclusion is:

- The *computer system* (hardware and operating system), rather than the designer or user of a program module, must decide where information should reside

within the memory hierarchy.

The argument is that if the designer of one module is to make storage allocation decisions for another module, the first would have to know implementation details of the second, in violation of the requirements of modular programming. The next conclusion follows immediately and is generally accepted:

- The referencing of information structures by a program module must be by means of a virtual memory addressing mechanism.

The next point asserts the futility of attempting to anticipate the behavior of programs through any sort of analysis at execution time:

- Information can be moved toward more rapidly accessible storage in the memory hierarchy only *on demand*, that is, upon being referenced by an active computation.

The next step is probably the most controversial:

- The units of storage allocation should be the information units on which the primitive operations of the computer system are carried out, that is, instructions, scalar values, and records.

The argument in favor of this point is that choosing a larger unit of information movement will unavoidably result in moving to higher levels of memory information not being referenced by active computations. The cost is unnecessary information transfer and wasted memory. Arguments against generally assume present day hardware structures that favor the transmission of large blocks of data between memory levels. The choice of a small allocation unit can be made practical by new designs of auxiliary storage systems and new ways of coupling storage systems to the processing hardware. Such storage systems should be able to handle hundreds or even thousands of concurrent retrieval requests.

The conclusion is that parallelism should be exploited at the detailed level to provide a continuing large supply of concurrent retrieval requests to the storage system.

- Exploitation of fine-grain parallelism should be done to support modular programming with high efficiency in computer systems having a hierarchical memory.

Language-Based System Design

Dennis and Van Horn wrote their paper using the language and concepts of computer systems: processes, multiprogramming, segments of address space, etc. This was natural since the ideas grew out of their experience in working with the team of MIT faculty and staff that was developing Multics [9, 8].

In 1964 I decided to teach and do independent basic research rather than participate in the project to implement Multics. I realized that the interface a computer system presents to its users should not be described in "computer systems" terms, but rather in terms of concepts and notations appropriate for a high level programming language. To proceed otherwise requires description at several levels and fails to offer the coherence and clarity of a single uniform notation. The programming language should be so complete that users may express all elements of a computation concisely. It should not be necessary for the programmer to depart from the notation of a high level language to express any requirement of the application, be it a text editor or an airline reservation system. Thus there should be no independent "data base" facility, or facility for control of concurrent "tasks" invoked by means of "system calls", or system library procedures that, in turn, make such calls or, even worse, act on the hardware directly via machine instructions. To resort to such means leads to confusion from disparate representations, and the use of diverse mechanisms for the same objectives.

In language-based design a *base language* provides a standard model that separates issues of application programming from problems of computer system implementation. By *computer system* we now mean the combination of hardware, firmware, and software that realizes the base language. The challenge to the system implementor is to build the best realization of the specified base language within the state of the art. In the absence of the discipline of language-based design, system implementors generally build upward from the primitive facilities of some fixed hardware base, with no assurance that desired properties at the user interface are even possible to achieve, let alone realize with acceptable efficiency.

In this paper our objective is to sketch a specific base language proposal that addresses the expressive needs of a broad and general class of computer applications. Our design is

built on concepts from functional programming [4] and data flow computation [13, 3], extended to support input-output programming using streams, and to provide for managing transactions on data bases. We believe this base language could serve as the semantic model for a successful new class of computer systems.

Embodied in our base language proposal is a radical hypothesis: *no data ever changes.* In suggesting this, we do not mean that data is not created during computation, or that data does not eventually disappear. Rather, the collection of information held by the envisioned computer system is continually augmented by the creation of new information. At the same time, other information becomes inaccessible because all activities in the system have relinquished their rights to have access to it. This information is garbage and the storage it occupies may be made available to hold newly created information.

If data does not change, then some nasty problems in computer systems design become much simpler:

1. The problem of maintaining consistency of files in a distributed system is considered important and troublesome. If data never changes, there is no distributed update problem. In the proposed base language "updates" have the effect of creating a new version of an object and are handled by a *guardian* associated with the object.

2. Protection systems distinguish permission to read and permission to write an object. If data does not change, no writing is done and write permission is irrelevant. A very simple concept of protection is possible: if a program module does not contain an object and is not given the object as an input, it does not have access to that object. This requires no specific protection-related features in the user's programming language.

3. Critical sections are used to protect resources containing objects on which write actions may be performed. If no writing is done, there is no critical section problem. In our proposal the role of critical sections is assumed by the **NewNode** instruction which performs an atomic substitution, and is used only in the implementation of guardians.

We believe it is feasible to build practical computer systems in harmony with these principles. The sort of computer system envisioned would have these characteristics:

1. It supports many users working at personal terminals and sharing

information—data and procedural—residing in a central system.

2. The central system includes a storage hierarchy arranged so all information is held online and information currently in use is quickly accessible.

3. All information held online is accessible through a universal referencing mechanism using unique identifiers with uniform meaning for all users.

4. No data ever changes. That is, each unique identifier represents a mathematical value (perhaps a large data base, for example) that does not change during system operation. A "changing" data base is represented by a sequence of fixed "versions", each of which is an immutable object in the storage system.

5. Extensive concurrency of operation at the instruction level is realized by use of a data-driven instruction execution scheme.

6. The "operating system" is simply a "shell" program that handles user authentication, resource allocation, and accounting.

Base Language Model

For the discussion in this paper we will use data flow program graphs [13] as the base language representation of program modules. A more formal operational semantics for a similar program graph model is outlined in [17].

As it is convenient to use a conventional textual notation in presenting program examples, we will also use an *ad hoc* dialect of the functional programming language VAL [1, 31] as an illustrative source language. In this form, a program for the greatest common divisor (GCD) of two natural numbers is the following.

```
function GCD ( m, n: integer ) returns ( integer )
  if m = n then n
  elseif m < n then GCD ( m, n - m )
  else GCD ( m - n, n )
  endif
endfun
```

The language elements illustrated in this example are the conditional expression, recursion, and operations and tests on integers. The language we consider here does not include any iteration construct, for we prefer to express iterative computations as recursive function

definitions.

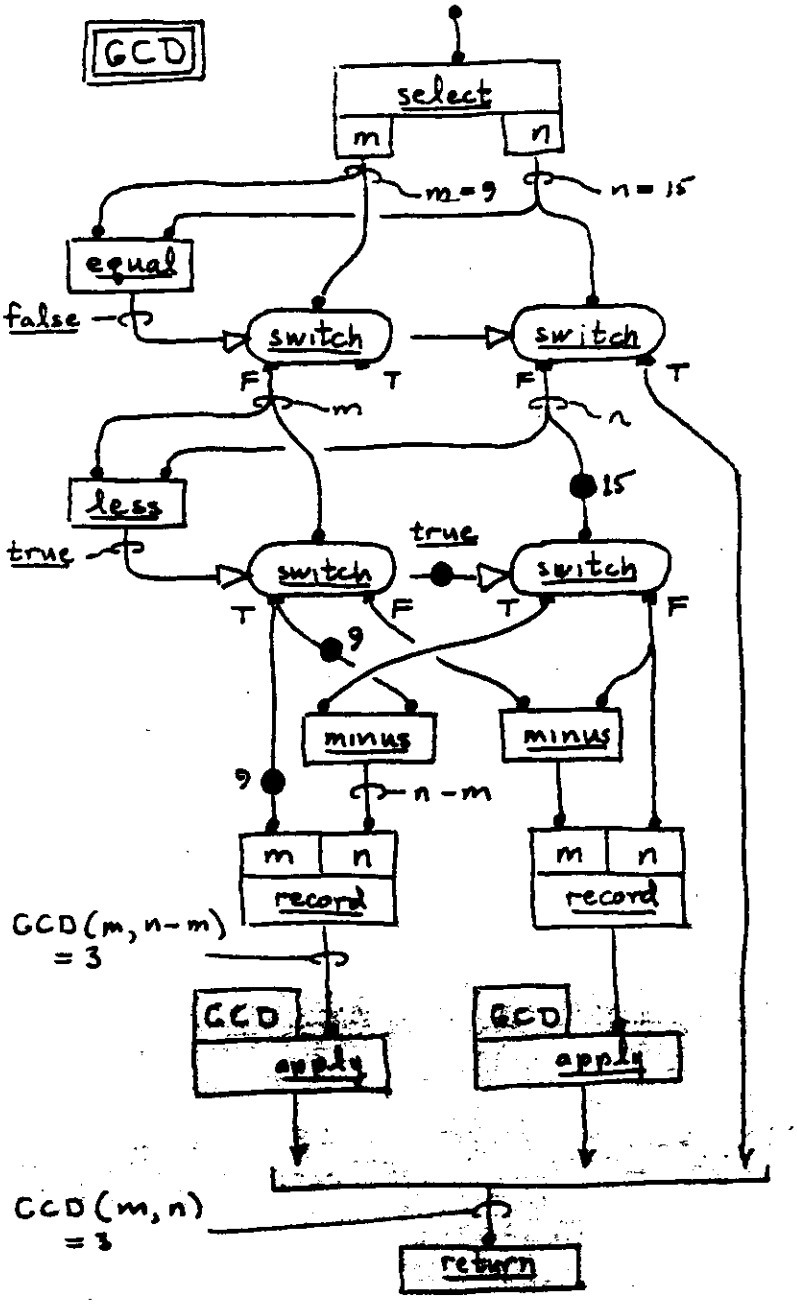

Fig. 1. Data flow program graph for the GCD function.
The tokens represent a stage in the evaluation of GCD(9, 15).

The data flow program graph shown in Fig. 1 is a base language representation of the GCD program. The status of computations in the base language model is represented by using a copy of the program graph for each activation of the corresponding function. How

far computation has progressed is indicated by the presence of tokens carrying information values on certain arcs of the program graph. The figure shows a stage in evaluating the outermost invocation in GCD (9, 15).

The model moves from one configuration to another by the execution or "firing" of enabled nodes (also called *actors* or *instructions*) of a program graph copy. A node of a data flow program graph is "enabled" if tokens are present on each input arc. An enabled node is eventually "fired", removing tokens from its input arcs and placing a token carrying the computed result value on each output arc.

The tokens on a program graph belong to a single "activation" of the function the graph represents. Since the graph is acyclic, it follows that there can never be more than one token on any arc. Execution of the **apply** actor is performed by creating a copy of the program graph that represents the function argument of **apply** and placing a token carrying the operand value on the input arc of the program graph copy. When a token appears on the output arc of a program graph instance, the value it carries is placed on the output arc of the appropriate **apply** actor of the calling program graph, and the copy of the applied graph is abandoned. Further on we will see that the graph copy must be retained because activity may continue after the return instruction has been executed.

Streams

The language elements introduced so far allow us to write only programs which receive inputs, compute, then produce output. It is not possible to represent programs that overlap computation with input or output, or that process input and output concurrently. By adding streams of data as a class of data types usable in writing programs, programmers may readily express computations in a way that allows input/output overlap and producer/consumer schemes of computation [33].

A *stream* is a sequence of values in which the individual values are all of the same type. A stream may be an unending sequence. The basic operations for streams are **cons**, **first**, **rest** and **empty**, which satisfy this relationship:

```
if empty (s)
    then s = [ ]
    else s = cons ( first (s), rest (s) )
```

As an example of programming with streams, we show a module that receives a stream of integers and separates them into two result streams containing the even elements and the odd elements of the input stream, respectively:

```
function Distribute ( InStream: stream [integer] )
    returns ( stream [integer], stream [integer] )
  let EvenStream, OddStream := Distribute ( rest (InStream) );
      Element := first (InStream);
  in
      if even (Element)
      then cons (Element, EvenStream), OddStream
      else EvenStream, cons (Element, OddStream)
  endlet
endfun
```

How should we represent streams and stream operations in the base language, that is, in terms of tokens, program graphs and firing rules? If we represent a stream by the sequence of tokens successively present on an arc of a program graph, then our observation that at most one token ever occupies an arc in any instance of a program graph would not hold and it would be difficult to design execution rules that ensure safety (no overwriting of operands) and liveness (freedom from deadlock). One alternative is to distinguish the tokens by means of labels, as Arvind has proposed in the "unraveling interpreter" [3]. For our base language, we have chosen to represent streams by "early completion" data structures, as discussed below.

Data Structures

We have found that use of "early completion" data structures solves certain problems in expressing concurrent computation: processing streams of data and implementing data base guardians. The ideas presented here are based on [15, 33, 16] and have drawn inspiration from [24, 25, 20, 21, 22, 26].

A general purpose language should support a variety of data structures, such as arrays,

lists and records. For our present purposes, it will suffice to illustrate the principles using binary trees. These data structures are simple in their properties, yet very general. In fact, binary trees, in the form of list structures, are used as a universal form of data in the programming language Lisp.

From the programmer's viewpoint, binary trees form a simple recursive record type

type Tree = **record** [L, R: Value]

where the Value type is a union type containing binary trees, as well as the scalar types of the base language. The basic operations on binary trees are *Create*, which forms a node from two element values, *Left* and *Right* which select the left and right elements of a tree, and the test *Tree* where *Tree* (x) = **true** if the value x is a tree. These operations satisfy

if *Tree* (t) then t = *Create* (*Left* (t), *Right* (t))

We imagine that a token on an arc of a program graph can just as well denote a binary tree (of arbitrary extent) as a scalar value, and this is the image we wish users of the base language to have for understanding the meaning of programs.

However, one of our objectives is to allow expression of stream-oriented computations such that consumption of a stream by one module may proceed concurrently with production of the stream by another. If we represent streams by binary trees, this means that it must be possible to pass the tree on to a user module before the tree has been completely constructed. We include exactly this feature in our base language by supporting what we call "early completion" binary trees. The early completion idea is closely related to concepts of eager and lazy evaluation that have been widely studied [20]. Here, we regard the concepts as an implementation mechanism that must be designed so that the desired properties of the user language—the properties of streams—are achieved.

To discuss early completion binary trees, we augment our semantic model for base language programs as follows. To the collection of program graph instances we add a *heap*, a multi-rooted, acyclic, directed graph in which each node is either

1. a leaf node with an associated scalar value;

2. a node with out-degree two and labels L and R on the out-going arcs.

An example of a heap is shown in Fig. 2a. Note that each node in the heap either represents an associated scalar value, or represents a binary tree whose L- and R- elements are the values represented by the nodes reached over the corresponding arcs leaving the given node. The values represented by nodes α, β and γ of the heap are shown in Fig. 2b.

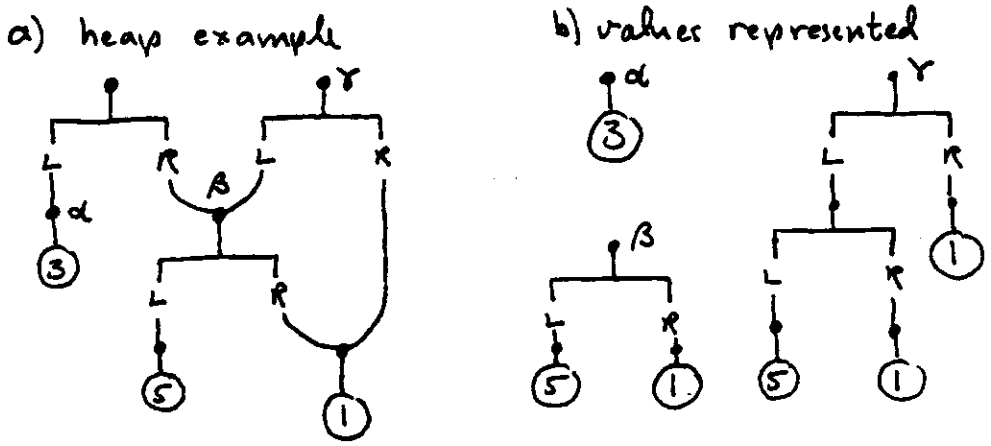

Fig. 2. Illustration of values represented in a heap.

One further type of heap node is required to implement "early completion" structures: a *queue* to hold requests for elements of binary trees yet to be constructed. Such a request occurs when a *select* actor (**Left** or **Right**) attempts to access a node for which no value has been constructed. The elements held by a queue are called *targets* and identify the instruction instances to which the node value must be sent once it becomes available.

Each target consists of three parts: a unique identifier of the program graph instance in which the target instruction resides; the index number of the target instruction within the program graph; and a small integer that specifies which operand of the target instruction is being supplied.

The base language instructions used in implementing binary trees are:

MkNode, MkLft, MkRht, Left, Right

In Fig. 3, the effect of executing each of these instructions is defined by transition rules for the graph/heap model. (The rules for **MkRht** and **Right** are analogous to those given for

MkLft and **Left**.) A binary tree is represented in the heap by a node with left and right elements, each of which is either a value or a queue. An element is a value once the component of the binary tree has been constructed and made part of the heap by execution of a **MkLft** or **MkRht** instruction. An element is a queue from the moment its parent node is created by a **MkNode** instruction, until the queue is replaced by a value. Initially, the queues are empty. On each execution of a **Left** or **Right** instruction for which the selected component is a value, that value is the result; if the selected component is a queue, the targets of the instruction (as determined by its output arcs) are entered in the queue.

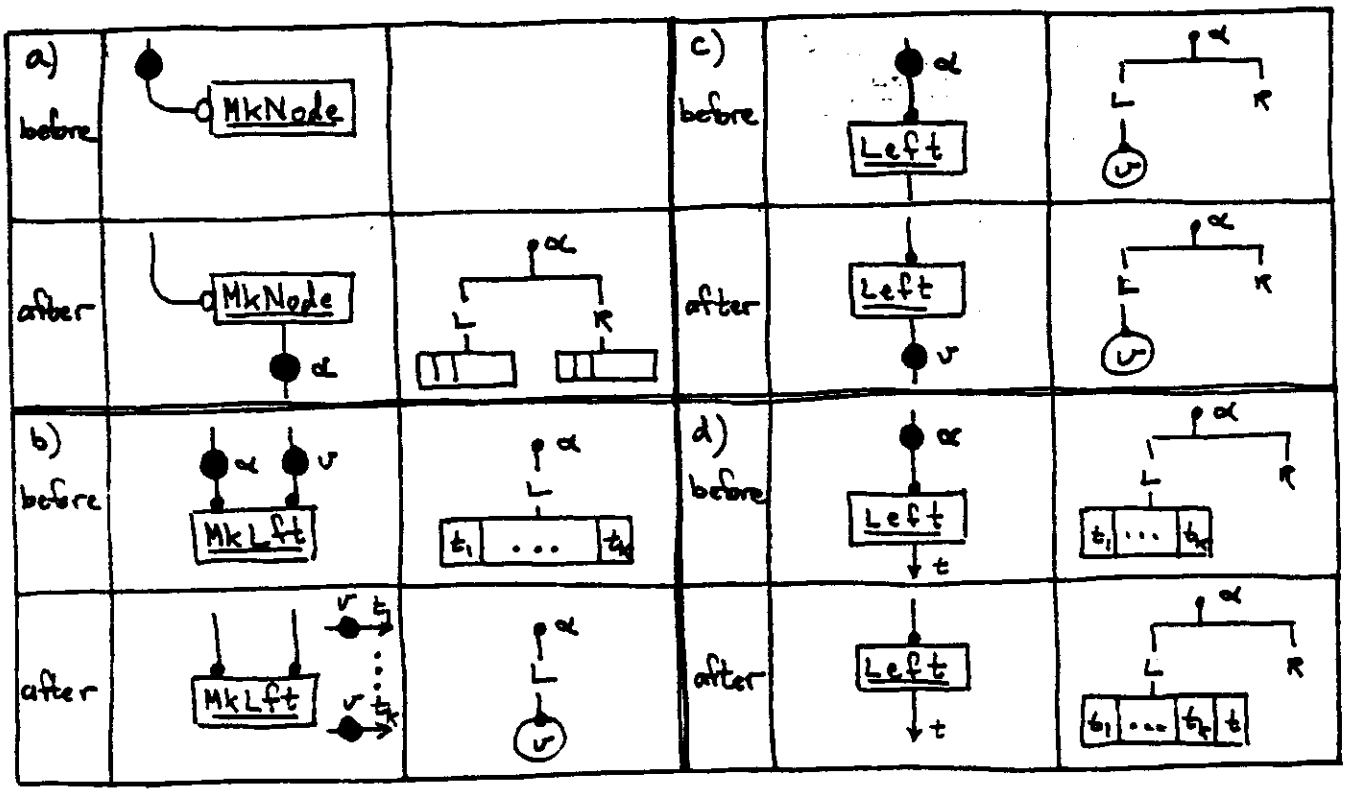

Fig. 3. Base language transitions for early completion binary trees.

Fig. 4 shows the implementation of the binary tree operations in terms of base language instructions.

The reader will note that, under the assumptions of our model and with only the instructions introduced so far, we have adhered to our principle that no data changes: there is no way of altering the value of any element of a binary tree in a way that can affect the result computed by a program. An important consequence is that there is no way to build a

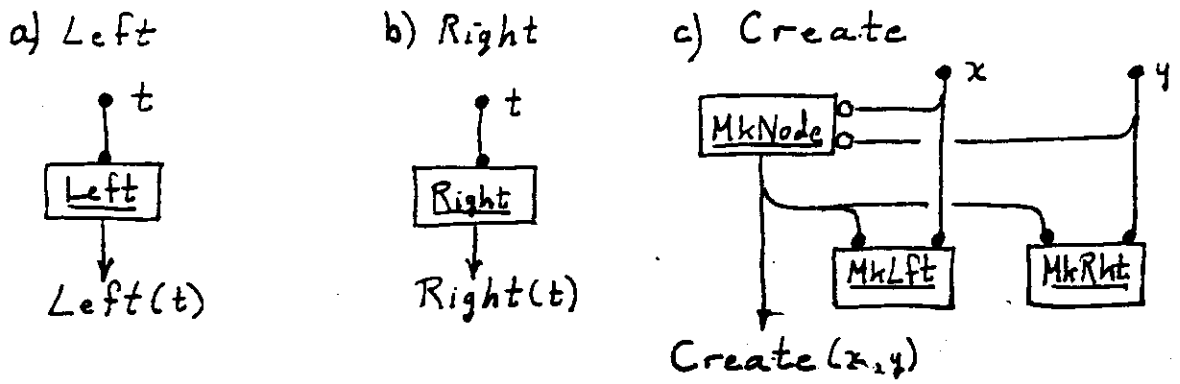

Fig. 4. Base language implementation of binary tree operations.

cycle in the heap. The acyclic property of the heap is secure and a reference count implementation of storage reclamation may be used [14]. We find this attractive as it avoids interruptions of computation and appears to permit more efficient implementation of the heap on a memory hierarchy.

Fig. 5 is a program graph which is a possible translation of the Distribute function into base language. The small open circles at each **MkNode** instruction indicate that arrival of a token supplies a necessary signal for enabling each instruction.

Records

Records as used in programming languages are easily represented by binary trees: the translator maps the record field names into sequences on the alphabet { L, R } in a manner that is consistent for each record type. The record constructor operation builds the appropriate binary tree, and record field selection is done by the appropriate series of **Left** and **Right** instructions. Note that early completion of our record constructor operation follows from our implementation of binary trees. Therefore we use records freely in our program examples. This also provides a useful interpretation of functions that accept multiple arguments or yield multiple results. We suppose that a multi-argument function is

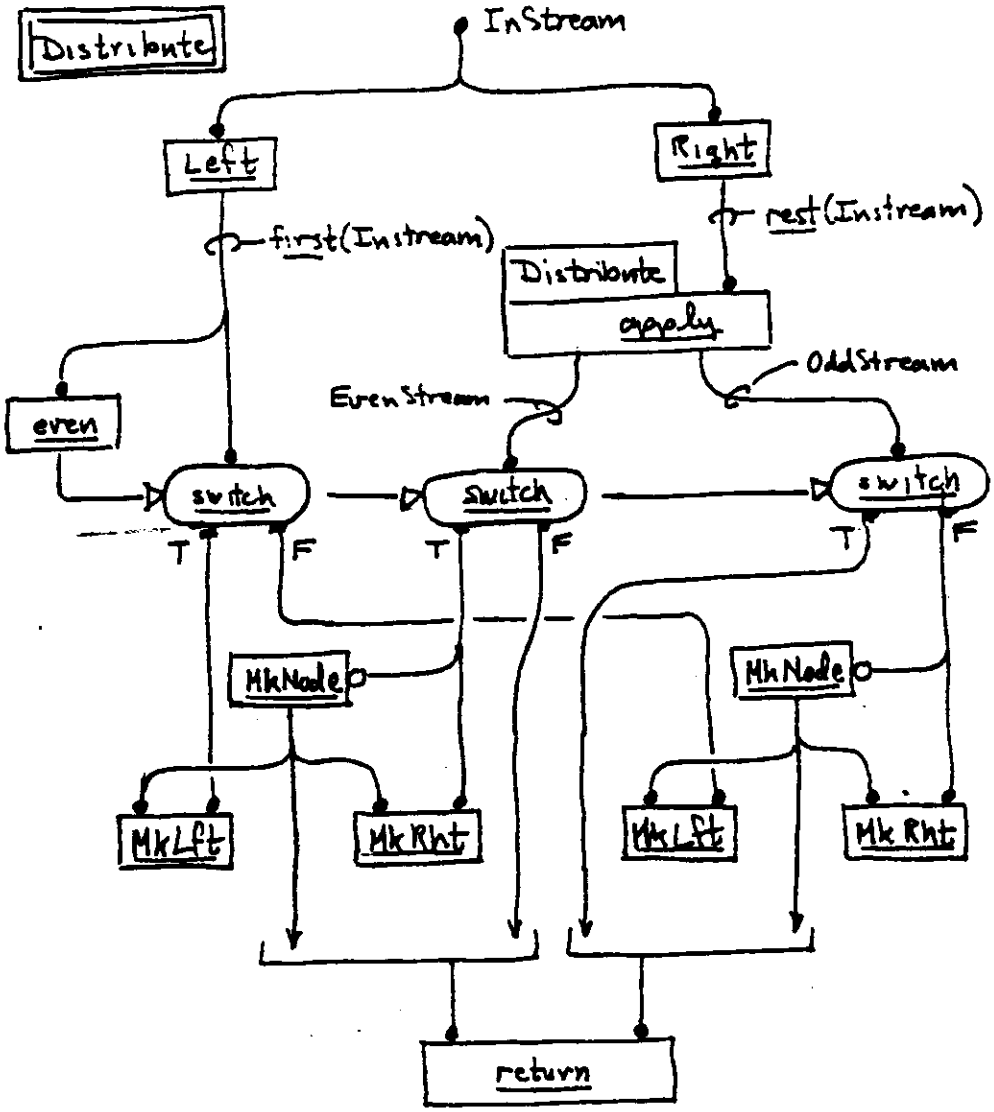

Fig. 5. The Distribute function in base language.

implemented as a program graph that accepts as its only input a record with one field for each argument of the function. Similarly, the program graph for a function with multiple results produces a record of result values. Assuming early completion for the record constructor, function evaluation will commence with the arrival of any argument value, and a result may be sent to its target instruction before all results of function application have been generated.

Data Base Transactions

Our base language would not be complete without provision for expressing transactions on data bases. Data base transactions are a form of nondeterminate computation in that the relative arrival time of independent requests for transactions generally affects the results. Of two agents requesting the last seat on an airline flight, one will win and the other lose, but either possible outcome is correct. Programs having this sort of behavior cannot be written using the source language elements introduced so far. Only programs that functionally map input histories into output histories can be expressed.

To express nondeterminate computations, use of the nondeterminate merge operation has been suggested [6, 15, 28, 29]. This operation produces an output stream in which the elements of two (or more) input streams are arbitrarily interleaved. Here we suggest using a form of the manager construct described by Arvind [2, 3].

A *guardian* is syntactically similar to a function, but its single argument may be thought of as a *Command* that the guardian perform some specific transaction. The single result produced by a guardian may be thought of as its *Answer* to the command. The body of the guardian is an expression denoting a function that maps a stream of Commands into a stream of Answers. Each invocation of the guardian presents a value of type Command to its body. A group of concurrent invocations creates many Command values that are merged into a stream of Commands which is the input stream of the body expression. Operation of the body produces a stream of Answers which is matched with the Command stream to determine the invocation for which each Answer is the result value.

In the following we show how a guardian can be written for a simple data base and how its implementation in the base language can support concurrency of access and update operations. As our example, we use the integer set data type with two operations: Search and Insert. These are representative of the access and update operations, respectively, that would be implemented for a more elaborate data base. We represent a set of integers as a chain of records using a distinguished union type [1, 30].

```
type Set = oneof [
  empty: null;
  nonempty: record [
    element: integer;
    rest: Set ] ]
```

We assume the list is maintained so the integers are kept in increasing order. This representation permits us to show how the operations would work in a hierarchical structure.

Using the **tagcase** construct of VAL, which selects over the alternatives of a **oneof** type, the Search and Insert operations may be written as follows:

```
function Search ( S: Set; n: integer ) returns ( boolean )
  tagcase S
  tag empty: false;
  tag nonempty:
    if S.element = n
    then true
    elseif S.element > n
    then false
    else Search ( S.rest, n )
    endif
  endtag
endfun
```

```
function Insert ( S: Set; n: integer ) returns ( Set )
  tagcase S
  tag empty:
    make Set [ nonempty: record [
      element: n;
      rest: S ] ];
  tag nonempty:
    if S.element = n then S
    elseif S.element > n then
      make Set [ nonempty: record [
        element: n;
        rest: S ] ]
    else
      make Set [nonempty: record [
        element: S.element;
        rest: Insert ( S.rest, n ) ] ]
    endif
  endtag
endfun
```

Each of these functions recurses down the chain of records until it succeeds or fails. Base language program graphs for Search and Insert are shown in Figs. 6 and 7.

Our data base, an integer set, will be held by a guardian whose Commands and Answers are of types defined as follows:

```
type Command = oneof [
  search: integer;
  insert: integer ]

type Answer = oneof [
  search: boolean;
  insert: null ]
```

The guardian for our data base is programmed as in Fig. 8.

Now let us look at the representation of the guardian in the base language. As before, streams are represented as early completion binary trees and the Transact function has a

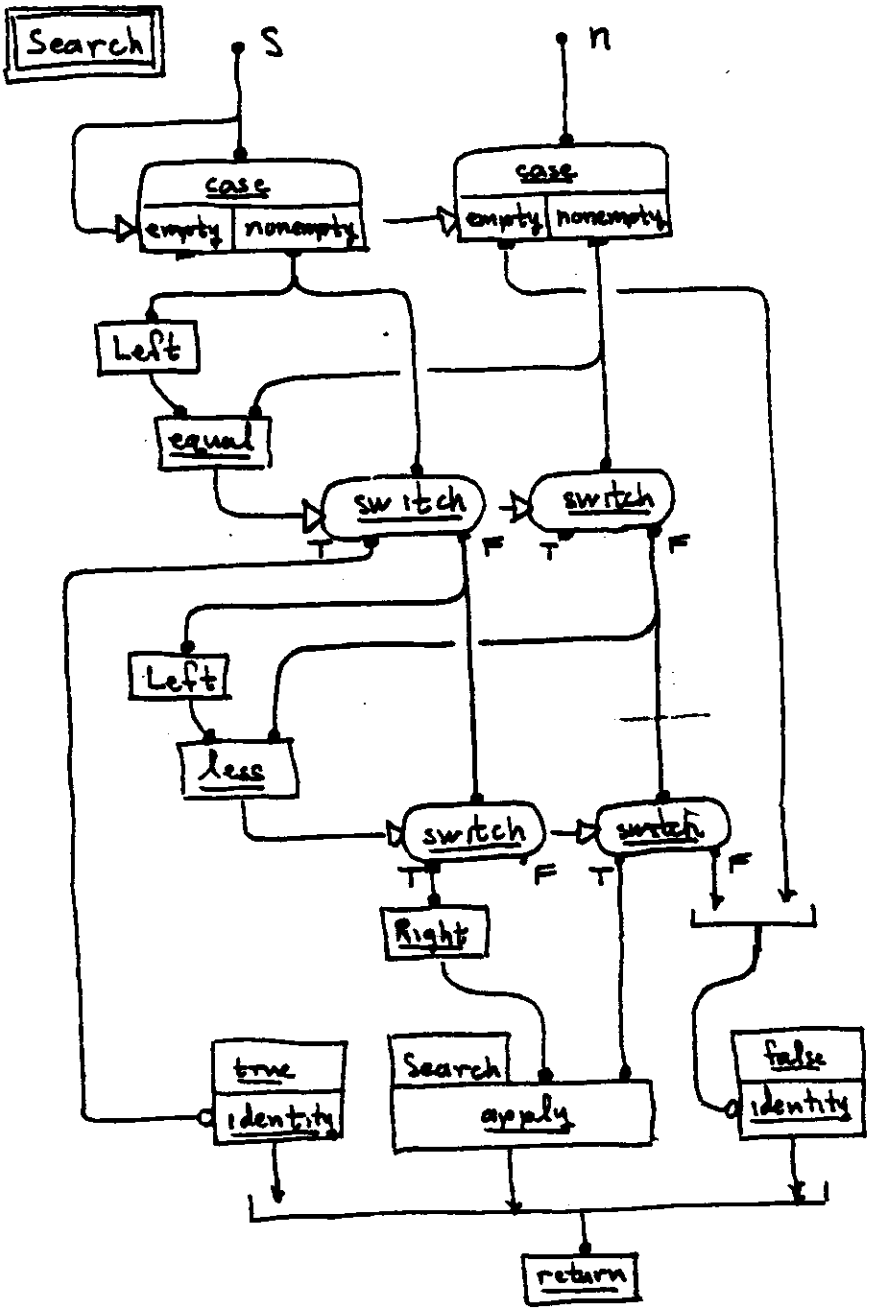

Fig. 6. The Search function in base language.

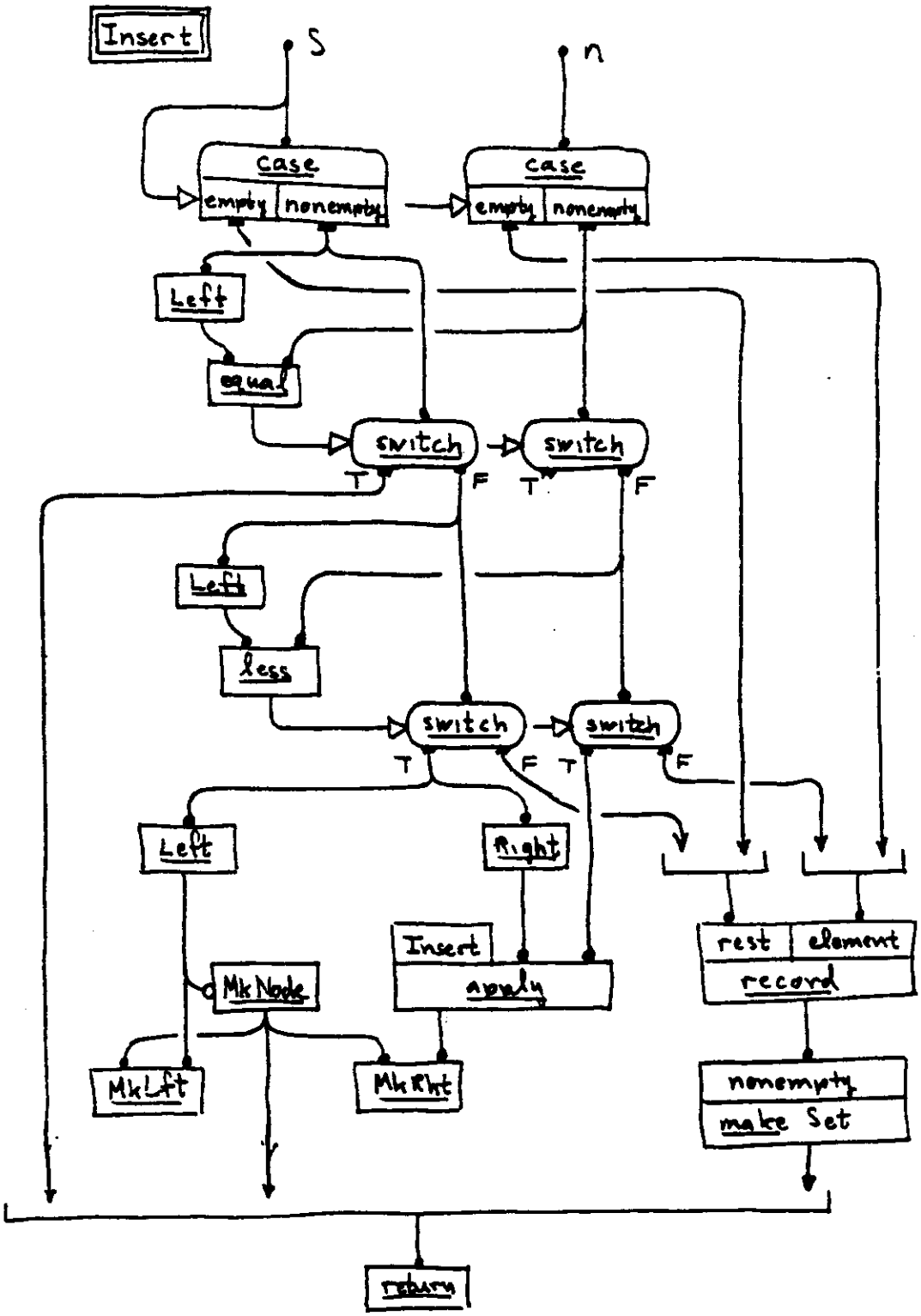

Fig. 7. The Insert function in base language.

```
guardian IntSet ( Cmd: Command ) returns ( Answer )
function Transact (
    CmdStream: stream[Command];
    BeforeData: Set )
    returns ( stream [Answer] )
  let
    C: Command := first ( CmdStream );
    A: Answer,
    AfterData: Set :=
      tagcase C
      tag search:
        make Answer [search: Search ( BeforeData, C )],
        BeforeData;
      tag insert:
        make Answer [insert: nil ],
        Insert ( BeforeData, C );
      endtag;
    AfterStream: stream [Answer]
      := Transact ( rest (CmdStream), AfterData );
  in
    cons ( A, AfterStream )
  endlet
endfun
  let InitData := make Set [ empty: nil ];
    AnsStream: stream [Answer]
      := Transact ( Cmd, InitData );
  in
    AnsStream
  endlet
endguard
```

Fig. 8. Textual coding of the IntSet guardian.

program graph implementation (Fig. 9) similar to that given earlier for the Distribute function.

The implementation problem for the guardian reduces to that of receiving Commands asynchronously from concurrently operating activities and forming them into a stream for input to the Transact function. How this may be done is illustrated in Fig. 10. A separate

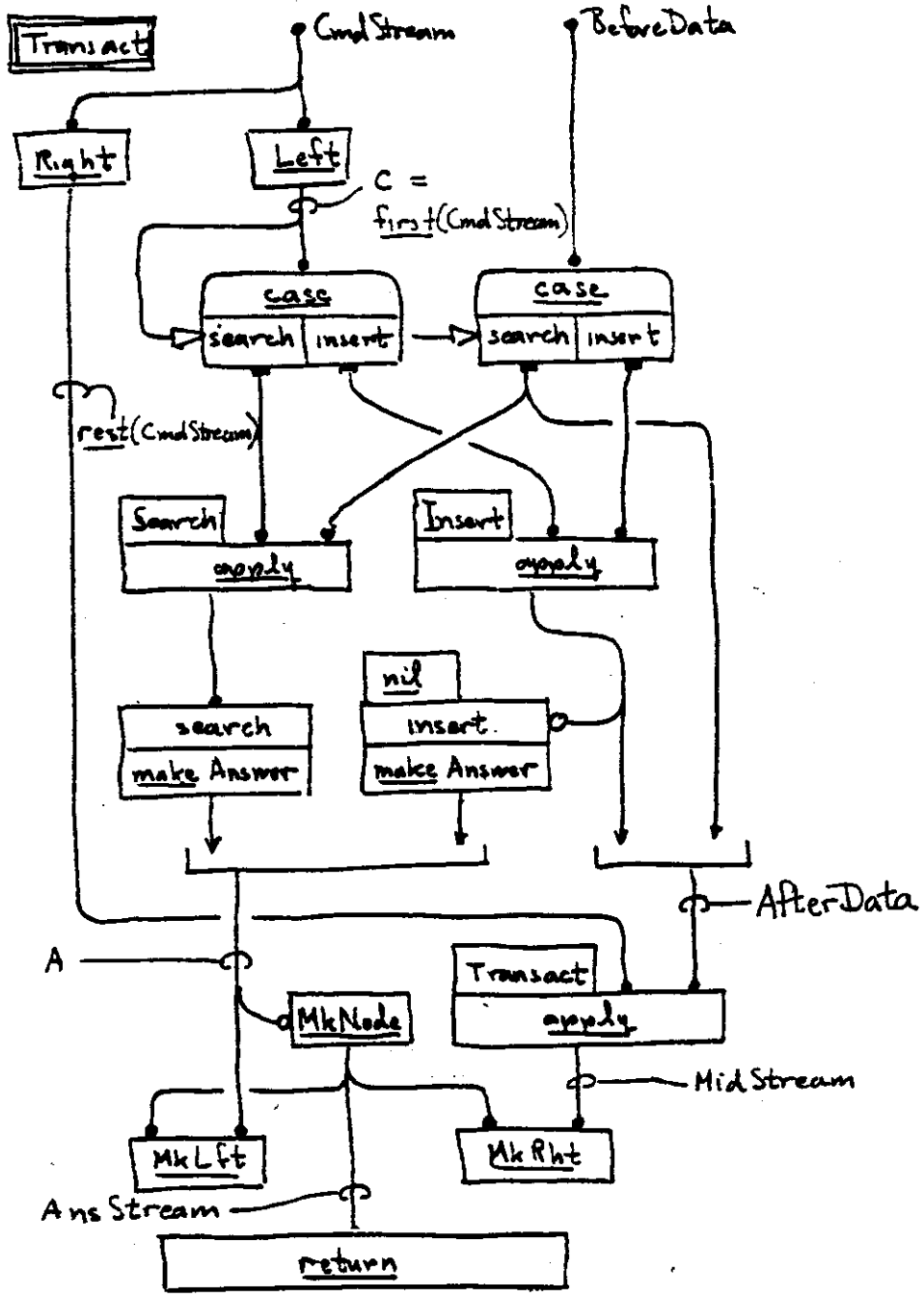

Fig. 9. The Transact function in base language.

copy of the program graph IntSet (Fig. 10a) is created for each invocation of the guardian. The body of the guardian is implemented by the program graph shown in Fig. 10b, which is activated exactly once by a token denoting the stream of tagged Commands (Entries) to be processed by the Transact function. Initially, this stream is empty. For each invocation of IntSet a record of the form

```
type Entry = record [
  argument: Command;
  return: Target ]
```

is created and appended to the stream. The *return* field of an Entry is a Target which specifies the instruction to which the response to the Command is to be sent. This process is illustrated in Fig. 11 which shows successive states of the heap for one invocation of IntSet. Initially, the guard holds the unique identifier α of the empty tail of the stream of command entries. In Fig. 11a, the **MkNode** instruction has obtained the unique identifier β for the new (empty) tail of the stream and the command entry record *e2* is waiting at the **MkLft** instruction. In the next step (Fig. 11b) the **NewNode** instruction performs the key operation of the proposed implementation. It substitutes β for α in the guard and passes α to its successors. The process is completed by execution of the **MkLft** instruction which places the entry record *e2* in the stream as the L-component of node α, and the **MkRht** instruction which installs the R-link from node α to node β.

In the presence of many concurrent invocations of IntSet there may be many concurrent executions of the **NewNode** instruction, all referring to the same guard. By making executions of the **NewNode** instruction atomic events, we can ensure that the stream of Commands is constructed to represent some specific order of arrival of Commands at the IntSet guardian.

Now let us observe what happens when many Search and Insert commands are sent to IntSet concurrently. As fast as commands arrive, IntSet appends them to the stream being processed by Transact. There is not likely to be a significant bottleneck here since the limiting time per transaction is just the execution time of the **NewNode** instruction.

Next consider how Transact will process the stream of commands. Immediately after

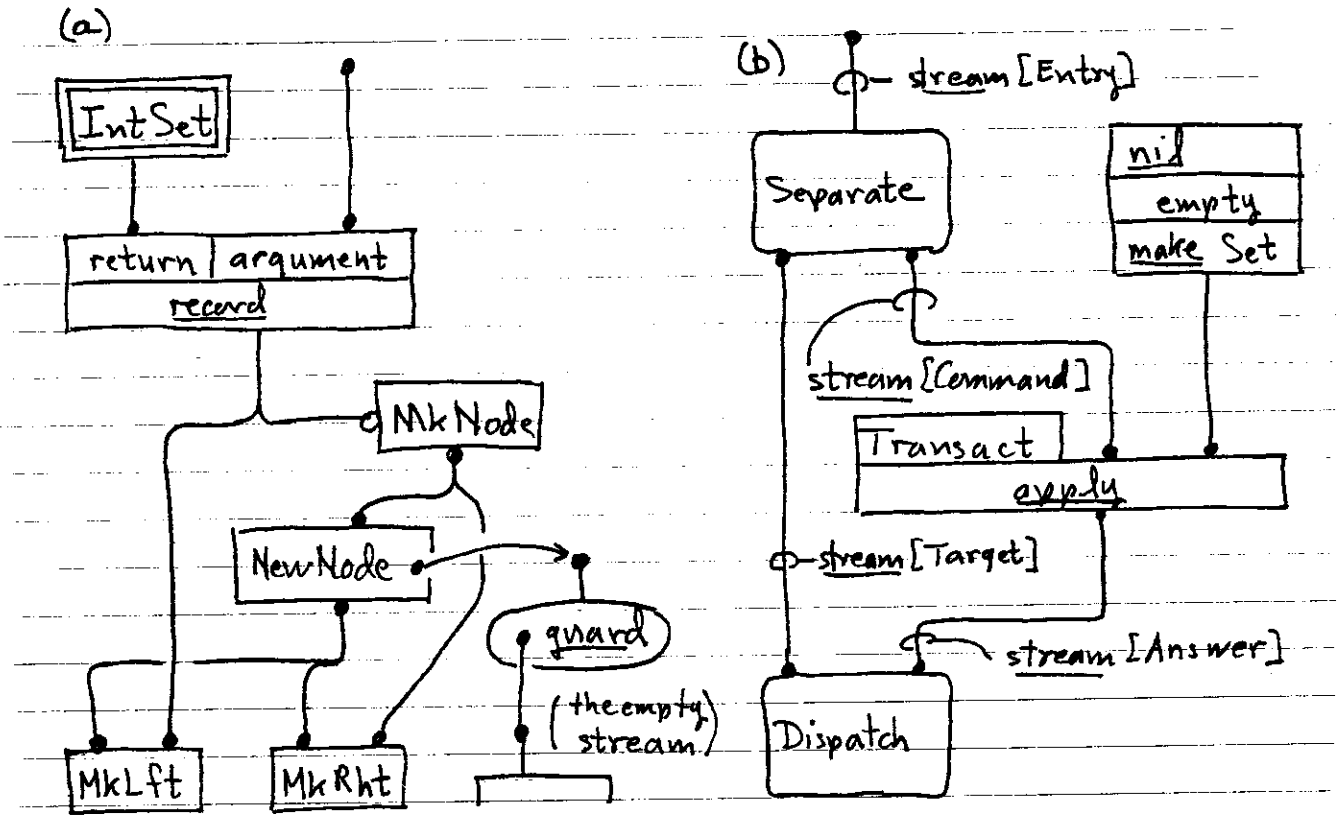

Fig. 10. Base language implementation of the IntSet guardian.

initiating a search command Transact may process further commands because even a subsequent Insert command will not affect the data seen by the Search function (data never changes!). After initiating an insert command, Transact will wait only a few instruction times before processing subsequent commands because the Insert function returns its Set result (an early completion structure) without waiting for inner activations of Insert to complete. Thus many Search commands and even many Insert commands can be active at once. The essential synchronization between Insert commands and crowds of Search commands is accomplished by the queuing mechanism built into the binary tree implementation.

In general, a significant data base will take the form of a broad tree structure. We have sketched how many access and update transactions may be processed concurrently over the depth of the tree. In addition it is desirable to allow concurrency of transactions that involve non-overlapping subtrees. In the case of access transactions, this presents no problem. To permit concurrent update of independent subtrees, two approaches are

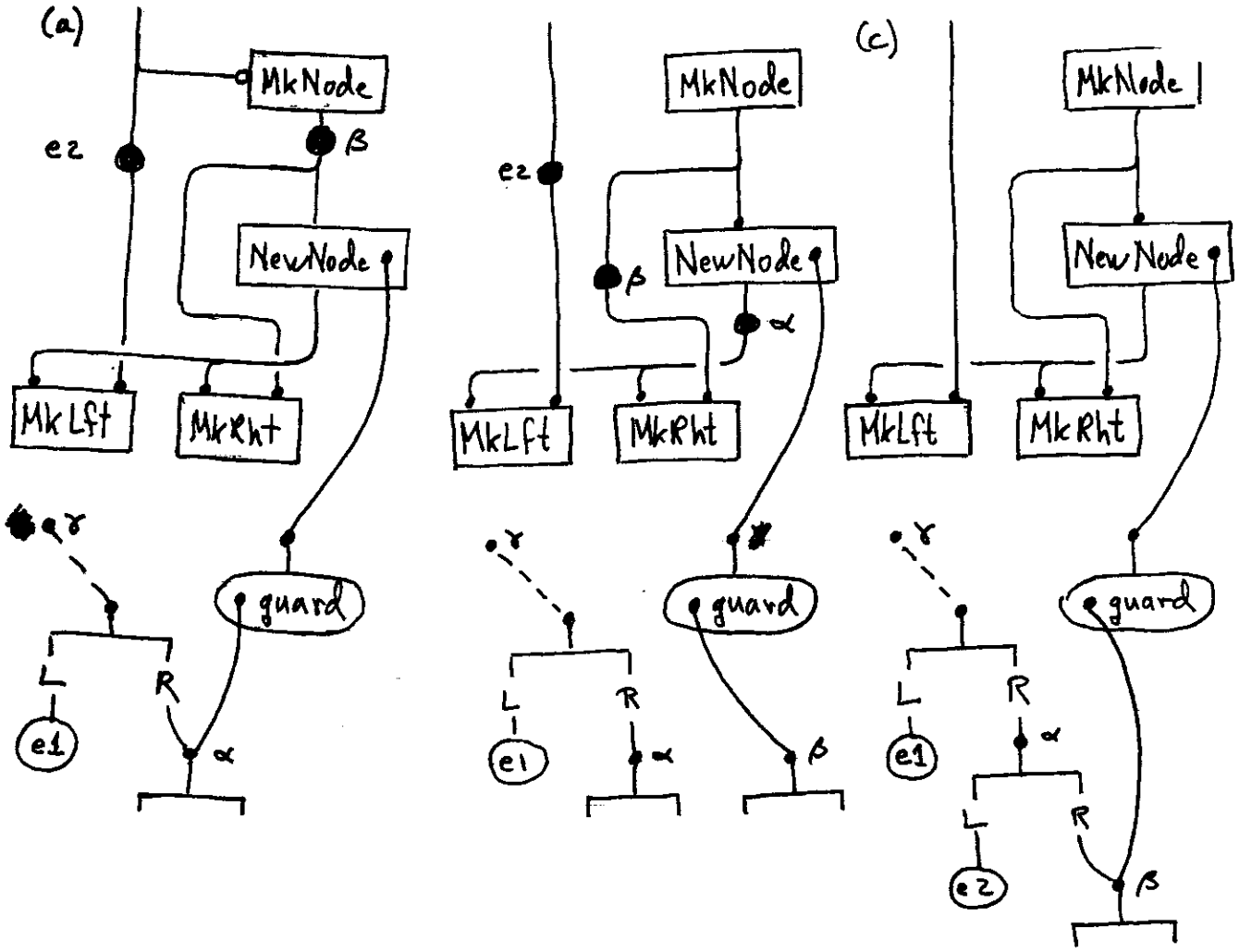

Fig. 11. Base language transitions illustrating use of the **NewNode** instruction.

possible within the framework we have presented. One way is for the data base to be a large record in which each independent subtree is a separate component. "Updating" a subtree amounts to creating a new record in which a new subtree is substituted at the appropriate field. If the record is an early completion record, the processing of further transactions, including updates to other subtrees, may begin once processing of the first update has begun and its (incomplete) result returned. A second approach is to treat the independent subtrees as distinct data bases, that is, let the data base be a record in which each component is a guardian that holds the current version of one independent subtree.

Sharing and Protection

Now let us consider issues of sharing and protection that arise because our computer system will have many users who wish to work cooperatively, but also wish to be protected from the consequences of accidental or deliberate interference from other users.

We adopt the idea that if information owned by user A is made available to user B and user B decides to pass it on to user C, there is no point in attempting to prevent or block such action. In any case, once B has the ability to read the information, B can construct a copy of it to pass on to C as though B had created it. If the information is a function and B is given the ability to invoke it and can use it as a module in building systems to be shared, then B may pass on the ability to use the function to anyone he chooses. As in the real society outside, security of data in a computer system ultimately rests on trusting relationships between people, and the prospect of punishment if one is discovered to have used or passed on information unethically. (Here we are not discussing systems that enforce constraints on access to "classified" information.)

Here we consider how, in our base language model, we can arrange for user A to authorize some user B access to a subsystem constructed by user A (an income tax adviser program, for example). An illustration of the envisioned organization of information in the computer system is shown in Fig. 12. Our assumptions:

1. There is a root node of the heap which is accessible to any activation by performing the instruction **Root**. From the root any activation may access the following information structures:

    a. *WhoIs Directory*: A directory containing information about users, maintained and certified by the computer system Manager.

    b. *Subsystem Directory*: An information structure in which each entry has two components: a "help" file giving information about the subsystem and its use; and an *entry function* which is called by any user desiring to make use of the subsystem.

2. Each user has a *User Directory*, a private information structure to which access is given when a person logs onto the system as the user. User directories are inaccessible from the root.

3. The WhoIs Directory and the Subsystem Directory are held by guardians which accept access commands from any user but accept update commands only from named users authorized by the Manager to install new information.

4. An activation of a function is owned by the owner of the function activation from which it was created. Thus each user of the system has a well defined tree of activations in any snapshot of the system. (Note carefully that our concept of *ownership* for this discussion applies only to activations, and not to data structures or program modules (functions).)

5. There is an instruction **Owner** which yields the name of the owner of the function activation in which the instruction is executed.

The entry function can determine the guaranteed identity of the caller by using the **Owner** instruction, and can request and check a password from the caller, if desired, before returning the requested subsystem to the caller. The caller may then use the subsystem as though it were an entry in his own directory.

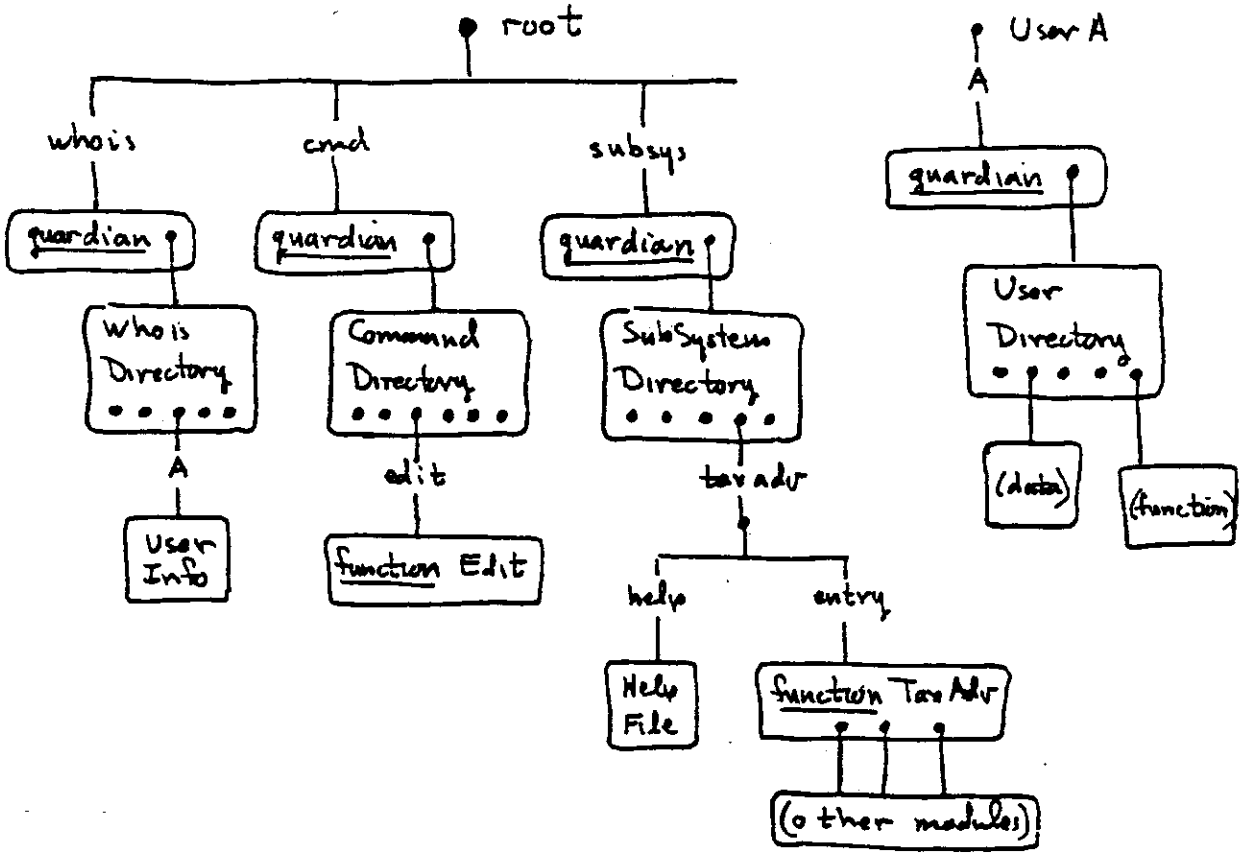

Fig.12. Suggested organization of information in the envisioned computer system.

Experimental Realization

In the Computation Structures Group of the MIT Laboratory for Computer Science we have begun construction of an experimental computer system according to the principles presented here. The system will eventually be implemented by writing microcode for an interpreter of the base language, as outlined in an earlier paper [17]. Some further details of the proposed implementation have been described [18]. At present a preliminary version of the interpreter has been implemented which runs on a commercial Lisp machine.

We imagine that these ideas might eventually be realized on a larger scale in the following form: One or more central system configurations would hold the bulk of information storage and perform all intensive numerical computations, at least. The architecture of each of the central systems would be patterned after the work of Ken Weng [16, 34]. It is possible to design such a distributed system so it is coherent; that is, it is functionally equivalent to a single central system of larger size.

Users would communicate with the system from conventional terminals or from personal computers or workstations. In the latter case, the workstation may be designed so that its memory acts as a cache holding those portions of the centrally stored information relevant to the current activities of the user. Information created at the terminal is immediately sent to the central system for safe keeping and potential sharing with other users. So long as no guardian is sent to a remote terminal, information at the workstation is never inaccurate -- there is no distributed update problem. Information in the terminal's memory may always be discarded in favor of new or more pertinent information without concern that any data could be lost.

References

1. Ackerman, W. B. and Dennis, J. B. VAL -- A Value - Oriented Algorithmic Language: Preliminary Reference Manual. Technical Report TR-218, Laboratory for Computer Science, MIT, Cambridge, MA 02139, June, 1979.

2. Arvind, Gostelow, K. P., and Plouffe, W. "Indeterminacy, Monitors and Dataflow". *Operating Systems Review 11*, 5 (November 1977), 159-169. Special issue: "Proceedings of the Sixth ACM Symposium on Operating Systems Principles".

3. Arvind, Gostelow, K. P., and Plouffe, W. An Asynchronous Programming Language and Computing Machine. Technical Report 114a, Department of Information and Computer Science, University of California, Irvine, California, December, 1978.

4. Backus, J. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs". *Communications of the ACM 21*, 8 (August 1978), 613-641.

5. Bensoussan, A., Clingen, C. T., and Daley, R. C. The Multics Virtual Memory. Proceedings of the Second Symposium on Operating System Principles, October, 1969, pp. 30-42.

6. Brock, J. D., and Ackerman, W. B. An Anomaly in the Specifications of Nondeterminate Packet Systems. In *Formal Description of Programming Concepts*, Lecture Notes in Computer Science 107, Springer-Verlag, Berlin, Heidelberg, New York, 1981, pp. 252-259.

7. Corbato, F. J. Multics -- the first seven years. AFIPS Conference Proceedings, 1972, pp. 571-583.

8. Daley, R. C., and Dennis, J. B. "Virtual memory, processes and sharing in Multics". *Communications of the ACM 11*, 5 (May 1968), 306-312.

9. Dennis, J. B. "Segmentation and the design of multiprogrammed computer systems". *Journal of the ACM 12*, 4 (October 1965), 589-602.

10. Dennis, J. B., and van Horn, E. C. "Programming Semantics for Multiprogrammed Computations". *Communications of the ACM 9*, 3 (March 1966), 143-155.

11. Dennis, J. B. Programming Generality, Parallelism and Computer Architecture. In *Information Processing 68*,
North-Holland Publishing Company, Amsterdam, 1969. The argument was presented in the conference talk and appears in the complete paper: Computation Structures Group Memo 32, Laboratory for Computer Science, MIT, Cambridge, MA 02139, 1968.

12. Dennis, J. B. On the Design and Specification of a Common Base Language. Proceedings of the Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, New York, April, 1971.

13. Dennis, J. B. First Version of a Data Flow Procedure Language. In *Programming Symposium: Proceedings, Colloque sur la Programmation*, B. Robinet, Ed., Lecture Notes in Computer Science 19, Springer-Verlag, 1974, pp. 362-376.

14. Dennis, J. B. On Storage Management for Advance Programming Languages. Memo 109, Computation Structures Group, Laboratory for Computer Science, MIT, Cambridge, MA 02139, November, 1974.

15. Dennis, J. B. A Language Design for Structured Concurrency. In *Design and Implementation of Programming Languages: Proceedings of a DoD Sponsored Workshop*, J. H. Williams and D. A. Fisher, Eds., Lecture Notes in Computer Science 54, Springer-Verlag, 1977.

16. Dennis, J. B., and K.-S. Weng. An Abstract Implementation for Concurrent Computation with Streams. Proceedings of the 1979 International Conference on Parallel Processing, August, 1979, pp. 35-45.

17. Dennis, J. B. An Operational Semantics for a Language with Early Completion Data Structures. In *Formal Description of Programming Concepts*, Lecture Notes in Computer Science 107, Springer-Verlag, Berlin, Heidelberg, New York, 1981, pp. 260-268.

18. Dennis, J. B., Stoy, J.E., and Guharoy, B. Vim: An experimental multiuser system supporting functional programming. Proceedings of the International Workshop on High-Level Computer Architecture 84, Association for Computing Machinery, May, 1984, pp. 1.1-1.9.

19. Fabry, R. S. "Capability-based addressing". *Communications of the ACM 17*, 7 (July 1974), 403-412.

20. Friedman, D. P., and Wise, D. S. CONS Should Not Evaluate its Arguments. In *Automata, Languages, and Programming*, University Press, Edinburgh, 1976, pp. 257-281.

21. Friedman, D. P., and Wise, D. S. Applicative Multiprogramming. Technical Report 72, Computer Science Department, Indiana University, Bloomington, Indiana, December, 1978.

22. Hewitt, C. E., Attardi, G., and Lieberman, H. Specifying and Proving Properties of Guardians for Distributed Systems. In *Semantics of Concurrent Computation*, G. Kahn, Ed., Lecture Notes in Computer Science 70, Springer-Verlag, Berlin, Heidelberg, New York, 1979, pp. 316-336.

23. IBM. *IBM System/38 Technical Developments.* IBM General Systems Division, 1978.

24. Kahn, G. The Semantics of a Simple Language for Parallel Programming. Information Processing 74: Proceeding of the IFIP Congress 74, 1974, pp. 471-475.

25. Kahn, G., and MacQueen, D. Coroutines and Networks of Parallel Processes. Information Processing 77: Proceedings of IFIP Congress 77, August, 1977, pp. 993-998.

26. Keller, R., Lindstrom, G., and Patil, S. A loosely-coupled applicative multi-processing system. Proceedings of the National Computer Conference, June, 1979, pp. 613-622.

27. Knowlton, W., et al. "Papers on the UNIX Operating System". *Bell System Technical Journal 57*, 6 (July-August 1978).

28. Kosinski, P. R. A Straightforward Denotational Semantics for Non-Determinate Data Flow Programs. Conference Record of the Fifth ACM Symposium on Principles of Programming Languages, January, 1978, pp. 214-221.

29. Kosinski, P. R. Denotational Semantics of Determinate and Non-Determinate Data Flow Programs. Technical Report TR-220, Laboratory for Computer Science, MIT, Cambridge, MA 02139, May, 1979.

30. Liskov, B. H. et al.. *CLU Reference Manual.* Lecture Notes in Computer Science 114, Springer-Verlag, Berlin, Heidelberg, New York, 1981.

31. McGraw, J. R. "The VAL language: Decription and Analysis". *Transactions on Programming Languages and Systems 4*, 1 (January 1982), 44-82.

32. Needham, R. M., and R. D. H. Walker. The Cambridge CAP computer and its protection system. Proceedings of the 6th ACM Symposium on Operating Systems Principles, November, 1977, pp. 1-10.

33. Weng, K.-S. Stream-Oriented Computation in Recursive Data Flow Schemas. Technical Memo TM-68, Laboratory for Computer Science, MIT, Cambridge, MA 02139, October, 1975.

34. Weng, K.-S. An Abstract Implementation for a Generalized Data Flow Language. Technical Report TR-228, Laboratory for Computer Science, MIT, Cambridge, MA 02139, 1979.

35. Wulf, W., et al. "HYDRA: The kernel of a multiprocessor operating system.". *Communications of the ACM 17*, 6 (June 1974), 337-345.

definitions.

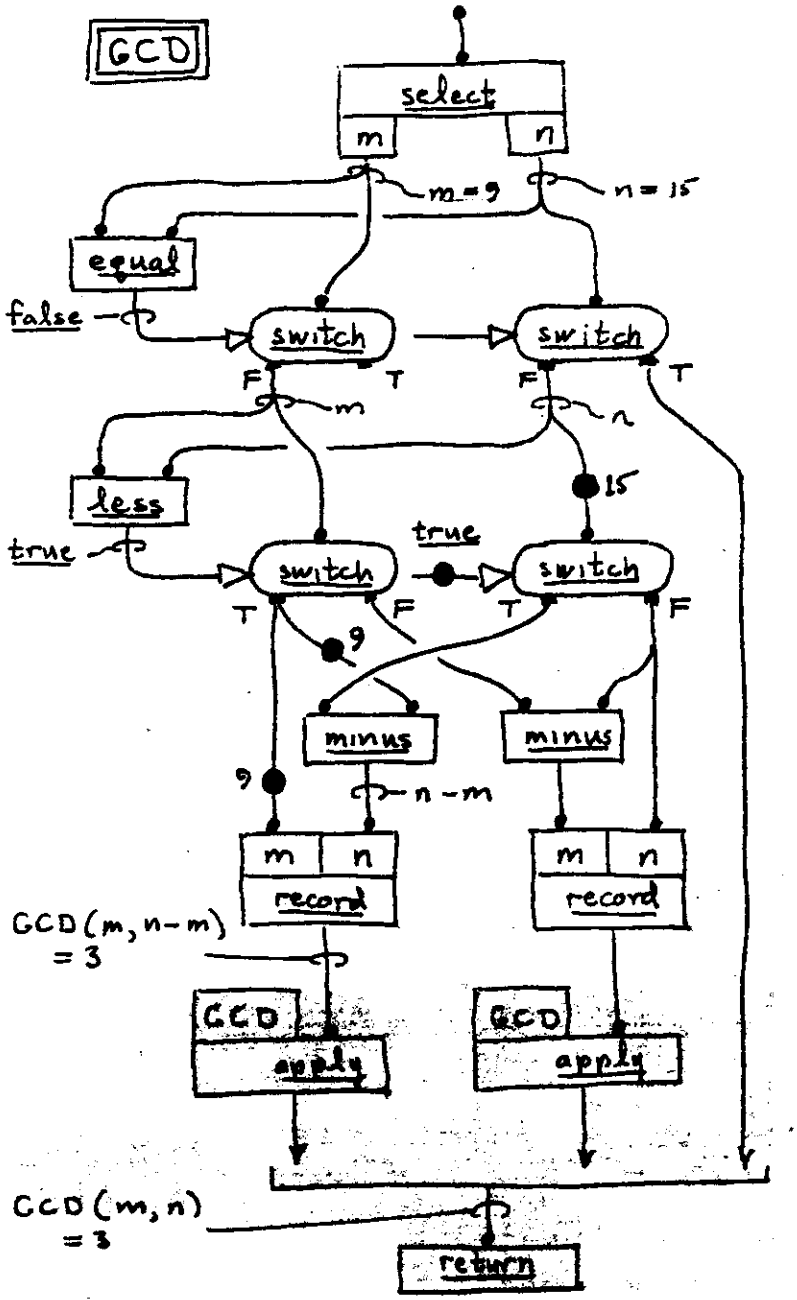

Fig. 1. Data flow program graph for the GCD function.
The tokens represent a stage in the evaluation of GCD(9, 15).

The data flow program graph shown in Fig. 1 is a base language representation of the GCD program. The status of computations in the base language model is represented by using a copy of the program graph for each activation of the corresponding function. How

2. a node with out-degree two and labels **L** and **R** on the out-going arcs.

An example of a heap is shown in Fig. 2a. Note that each node in the heap either represents an associated scalar value, or represents a binary tree whose **L**- and **R**- elements are the values represented by the nodes reached over the corresponding arcs leaving the given node. The values represented by nodes α, β and γ of the heap are shown in Fig. 2b.

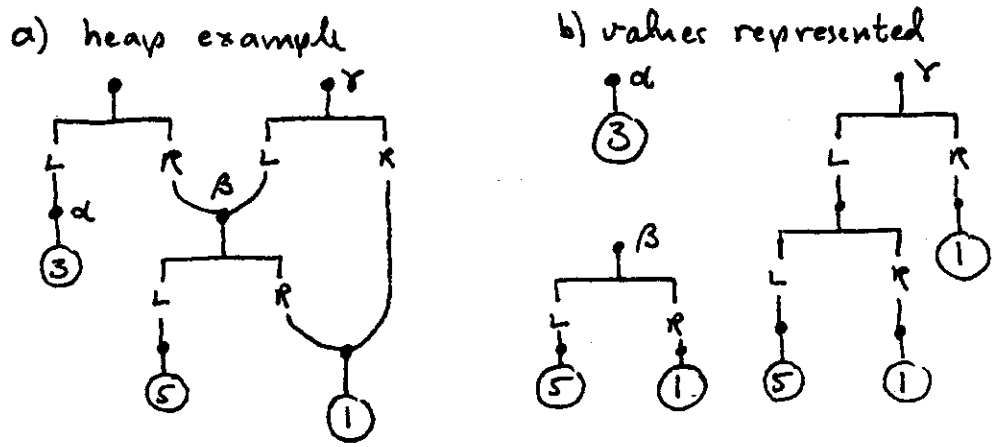

Fig. 2. Illustration of values represented in a heap.

One further type of heap node is required to implement "early completion" structures: a *queue* to hold requests for elements of binary trees yet to be constructed. Such a request occurs when a *select* actor (**Left** or **Right**) attempts to access a node for which no value has been constructed. The elements held by a queue are called *targets* and identify the instruction instances to which the node value must be sent once it becomes available.

Each target consists of three parts: a unique identifier of the program graph instance in which the target instruction resides; the index number of the target instruction within the program graph; and a small integer that specifies which operand of the target instruction is being supplied.

The base language instructions used in implementing binary trees are:

MkNode, MkLft, MkRht, Left, Right

In Fig. 3, the effect of executing each of these instructions is defined by transition rules for the graph/heap model. (The rules for **MkRht** and **Right** are analogous to those given for

MkLft and **Left**.) A binary tree is represented in the heap by a node with left and right elements, each of which is either a value or a queue. An element is a value once the component of the binary tree has been constructed and made part of the heap by execution of a **MkLft** or **MkRht** instruction. An element is a queue from the moment its parent node is created by a **MkNode** instruction, until the queue is replaced by a value. Initially, the queues are empty. On each execution of a **Left** or **Right** instruction for which the selected component is a value, that value is the result; if the selected component is a queue, the targets of the instruction (as determined by its output arcs) are entered in the queue.

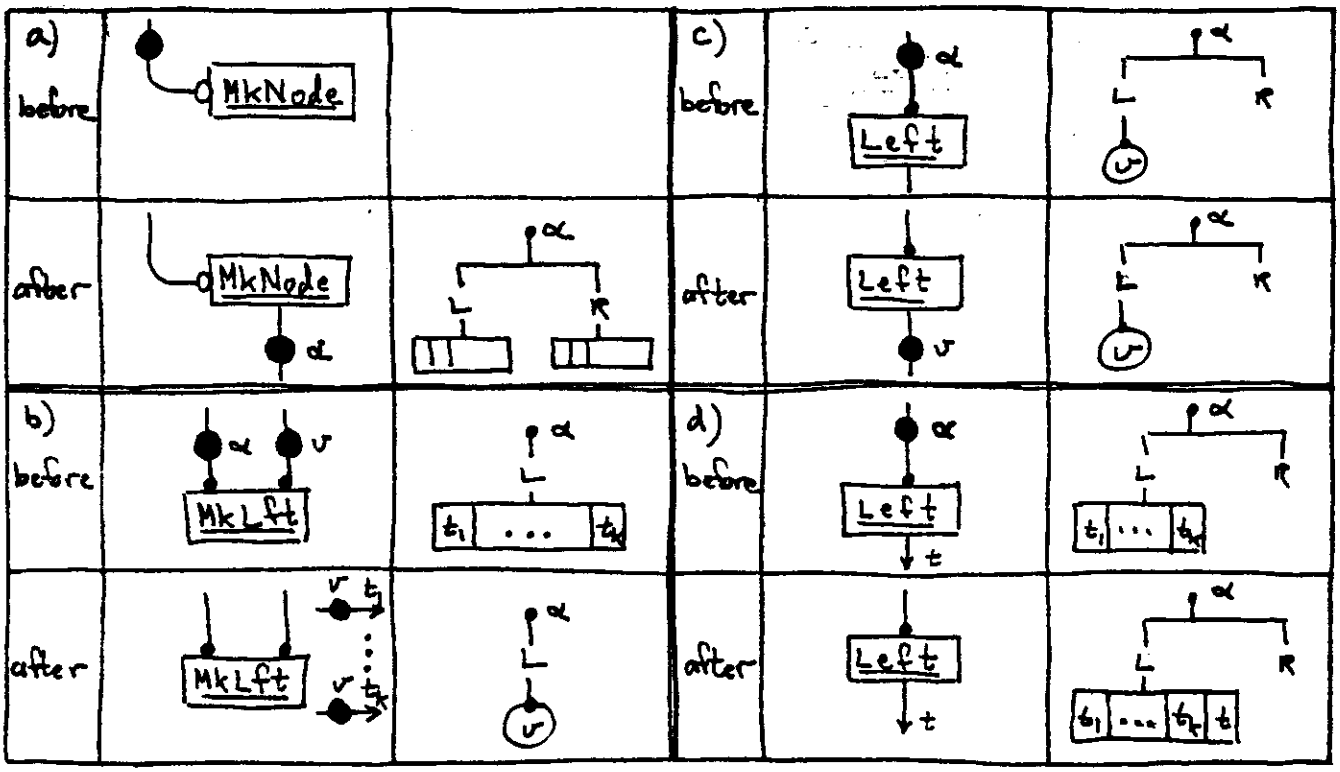

Fig. 3. Base language transitions for early completion binary trees.

Fig. 4 shows the implementation of the binary tree operations in terms of base language instructions.

The reader will note that, under the assumptions of our model and with only the instructions introduced so far, we have adhered to our principle that no data changes: there is no way of altering the value of any element of a binary tree in a way that can affect the result computed by a program. An important consequence is that there is no way to build a

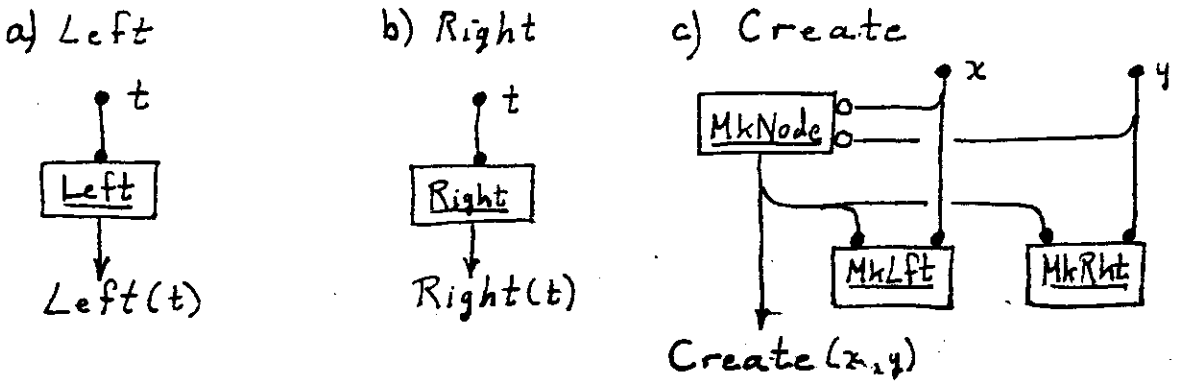

Fig. 4. Base language implementation of binary tree operations.

cycle in the heap. The acyclic property of the heap is secure and a reference count implementation of storage reclamation may be used [14]. We find this attractive as it avoids interruptions of computation and appears to permit more efficient implementation of the heap on a memory hierarchy.

Fig. 5 is a program graph which is a possible translation of the Distribute function into base language. The small open circles at each **MkNode** instruction indicate that arrival of a token supplies a necessary signal for enabling each instruction.

Records

Records as used in programming languages are easily represented by binary trees: the translator maps the record field names into sequences on the alphabet { **L, R** } in a manner that is consistent for each record type. The record constructor operation builds the appropriate binary tree, and record field selection is done by the appropriate series of **Left** and **Right** instructions. Note that early completion of our record constructor operation follows from our implementation of binary trees. Therefore we use records freely in our program examples. This also provides a useful interpretation of functions that accept multiple arguments or yield multiple results. We suppose that a multi-argument function is

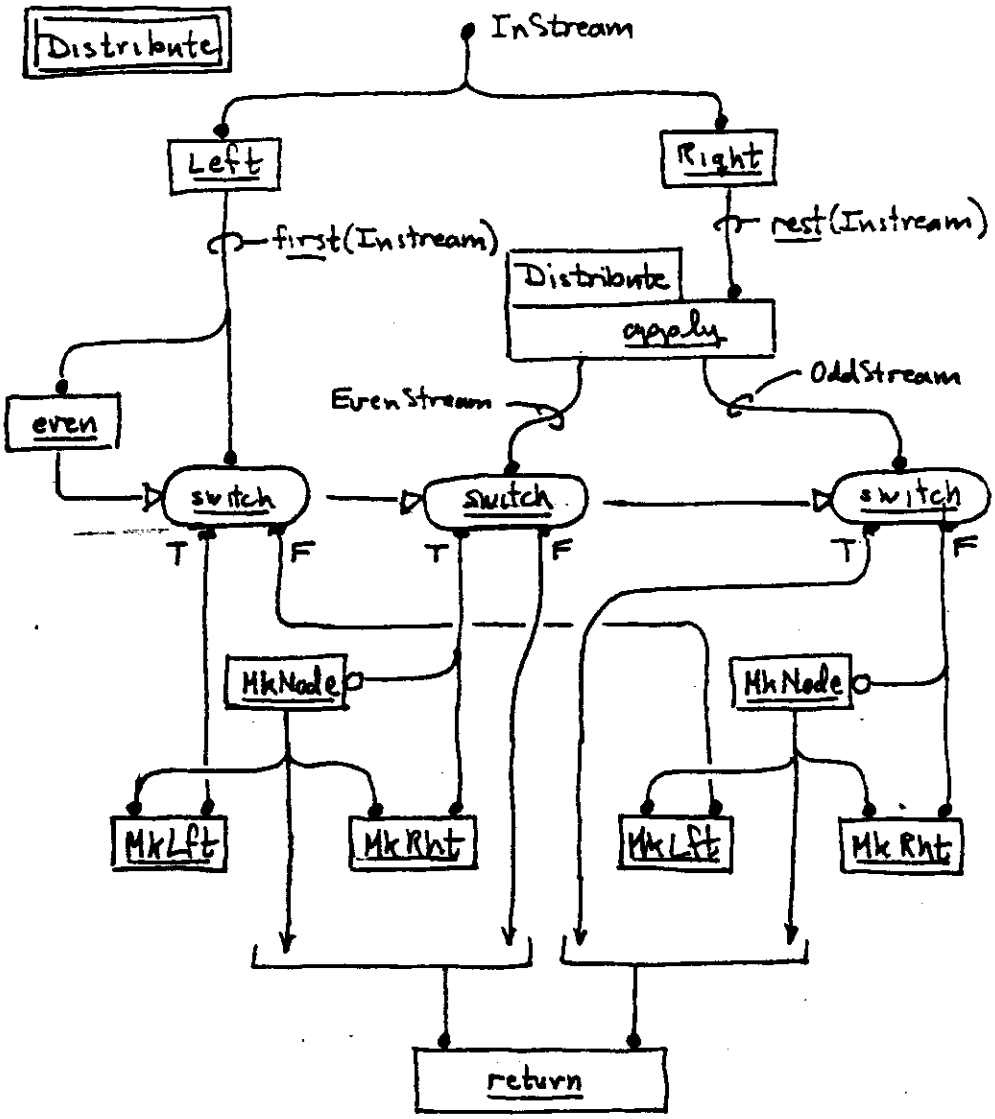

Fig. 5. The Distribute function in base language.

implemented as a program graph that accepts as its only input a record with one field for each argument of the function. Similarly, the program graph for a function with multiple results produces a record of result values. Assuming early completion for the record constructor, function evaluation will commence with the arrival of any argument value, and a result may be sent to its target instruction before all results of function application have been generated.

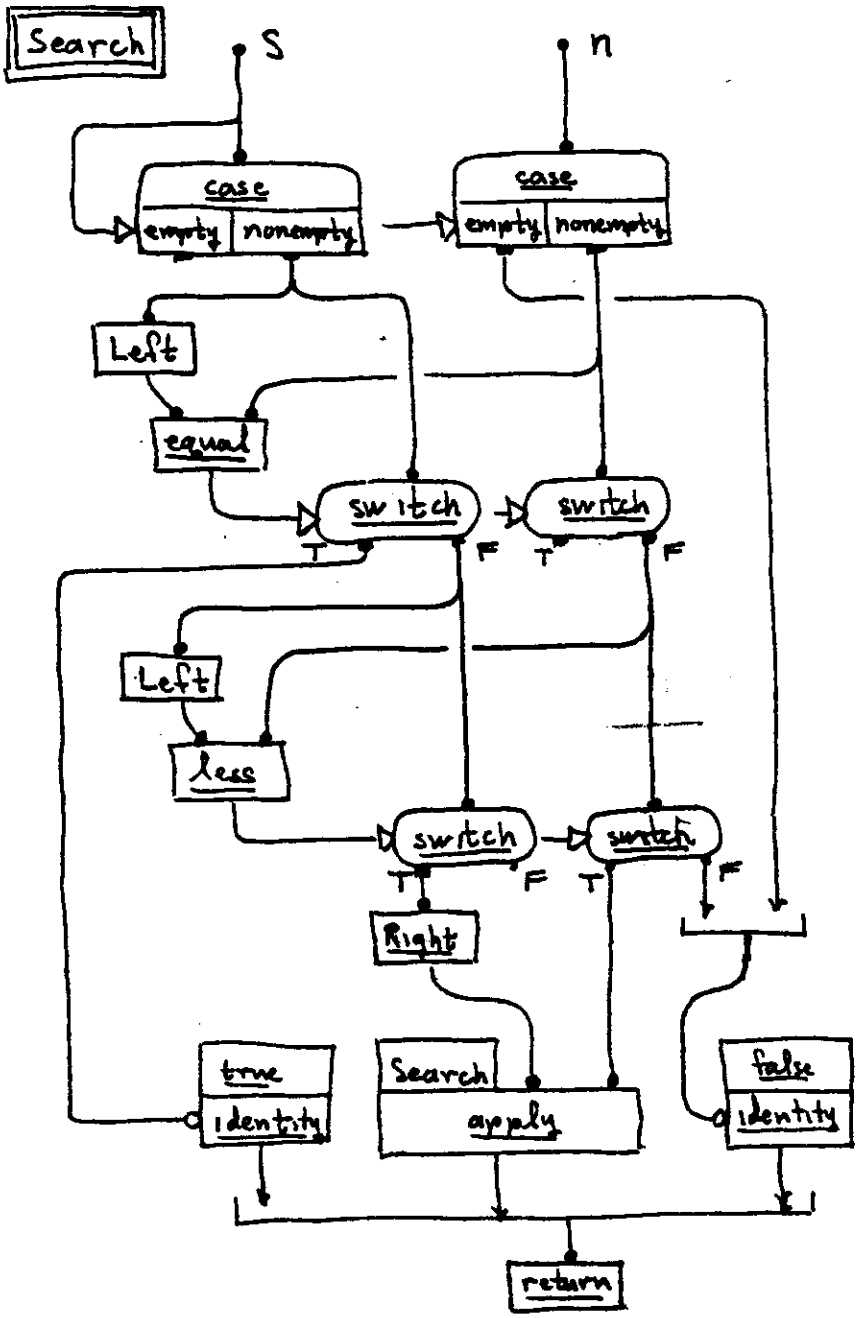

Fig. 6. The Search function in base language.

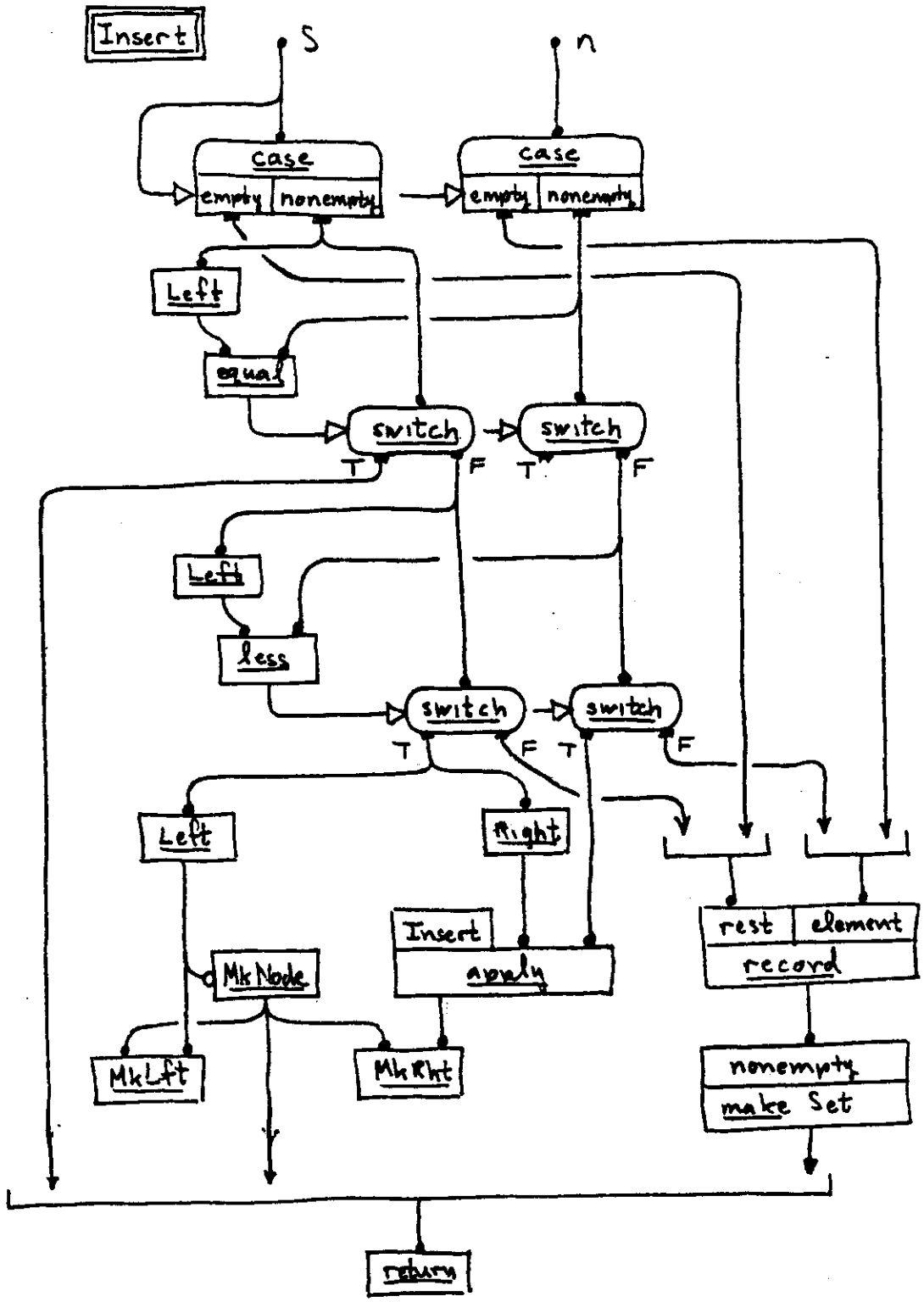

Fig. 7. The Insert function in base language.

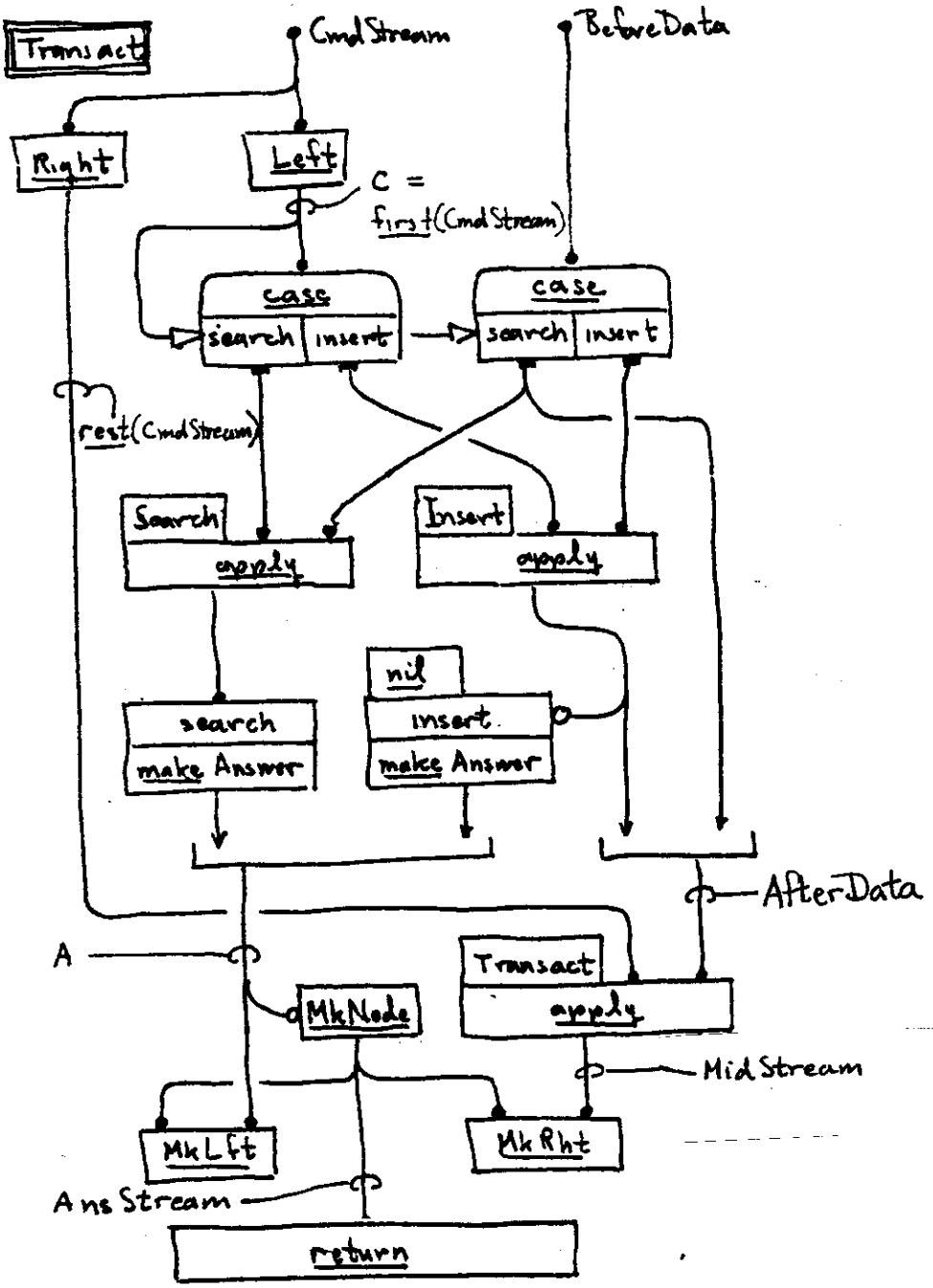

Fig. 9. The Transact function in base language.

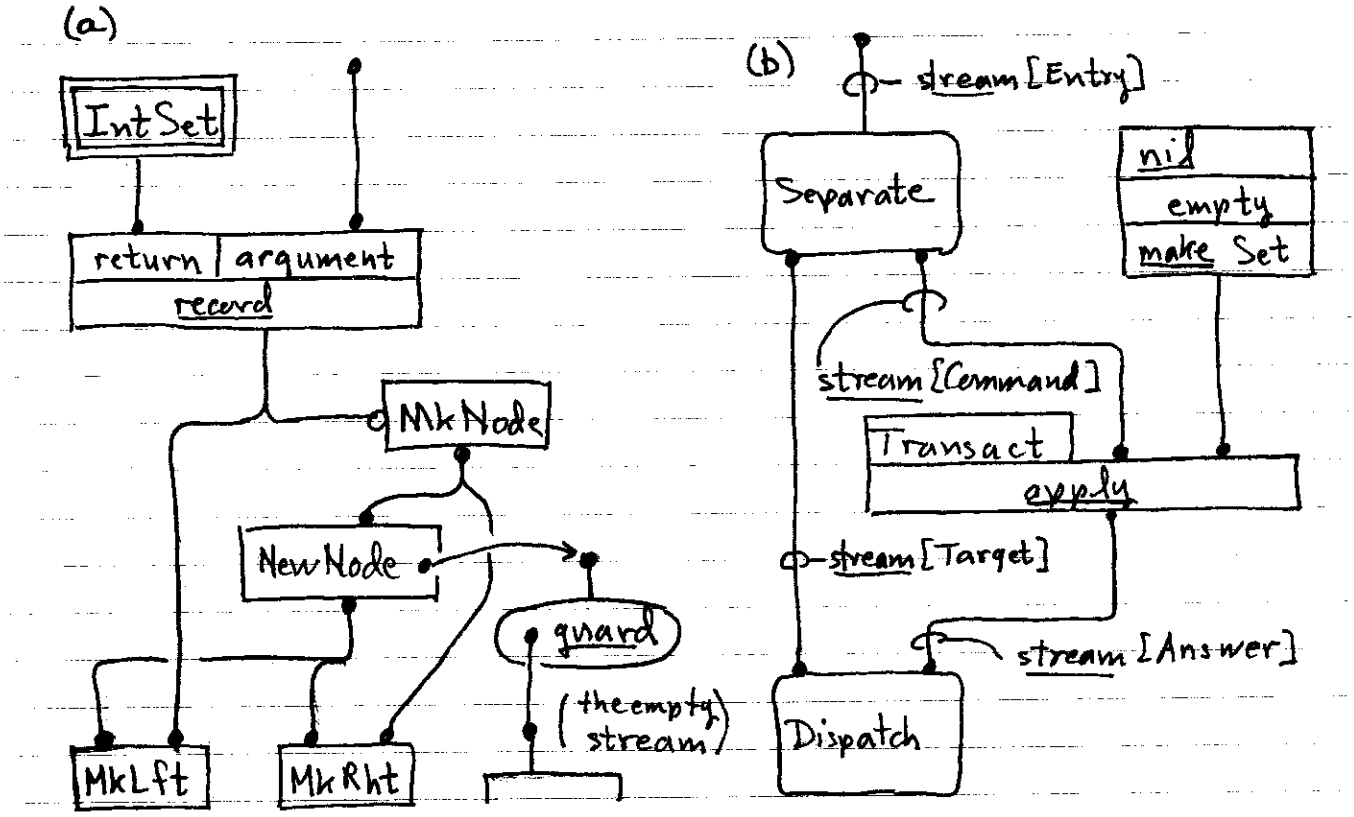

Fig. 10. Base language implementation of the IntSet guardian.

initiating a search command Transact may process further commands because even a subsequent Insert command will not affect the data seen by the Search function (data never changes!). After initiating an insert command, Transact will wait only a few instruction times before processing subsequent commands because the Insert function returns its Set result (an early completion structure) without waiting for inner activations of Insert to complete. Thus many Search commands and even many Insert commands can be active at once. The essential synchronization between Insert commands and crowds of Search commands is accomplished by the queuing mechanism built into the binary tree implementation.

In general, a significant data base will take the form of a broad tree structure. We have sketched how many access and update transactions may be processed concurrently over the depth of the tree. In addition it is desirable to allow concurrency of transactions that involve non-overlapping subtrees. In the case of access transactions, this presents no problem. To permit concurrent update of independent subtrees, two approaches are

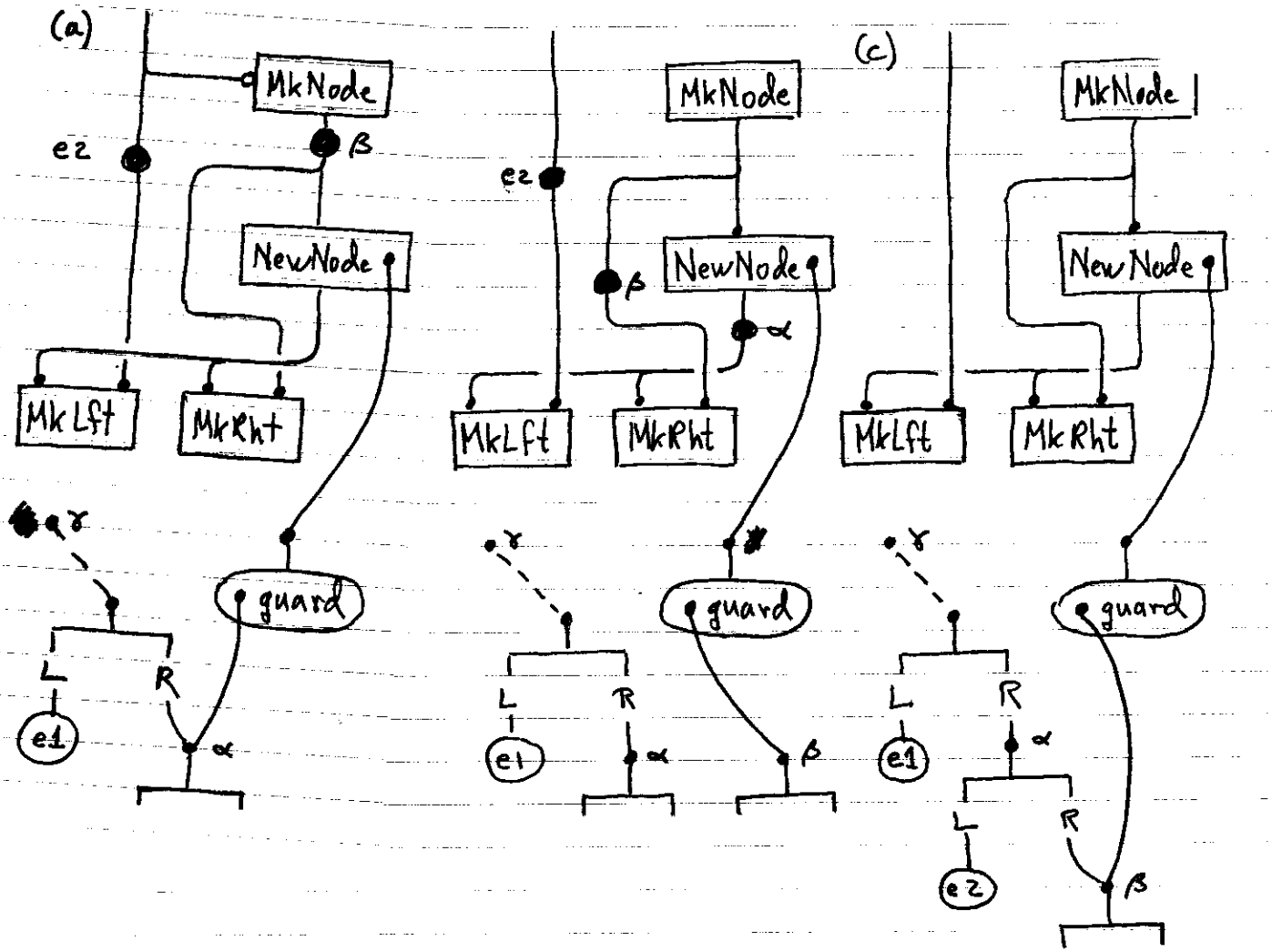

Fig. 11. Base language transitions illustrating use of the NewNode instruction.

possible within the framework we have presented. One way is for the data base to be a large record in which each independent subtree is a separate component. "Updating" a subtree amounts to creating a new record in which a new subtree is substituted at the appropriate field. If the record is an early completion record, the processing of further transactions, including updates to other subtrees, may begin once processing of the first update has begun and its (incomplete) result returned. A second approach is to treat the independent subtrees as distinct data bases, that is, let the data base be a record in which each component is a guardian that holds the current version of one independent subtree.

3. The WhoIs Directory and the Subsystem Directory are held by guardians which accept access commands from any user but accept update commands only from named users authorized by the Manager to install new information.

4. An activation of a function is owned by the owner of the function activation from which it was created. Thus each user of the system has a well defined tree of activations in any snapshot of the system. (Note carefully that our concept of *ownership* for this discussion applies only to activations, and not to data structures or program modules (functions).)

5. There is an instruction **Owner** which yields the name of the owner of the function activation in which the instruction is executed.

The entry function can determine the guaranteed identity of the caller by using the **Owner** instruction, and can request and check a password from the caller, if desired, before returning the requested subsystem to the caller. The caller may then use the subsystem as though it were an entry in his own directory.

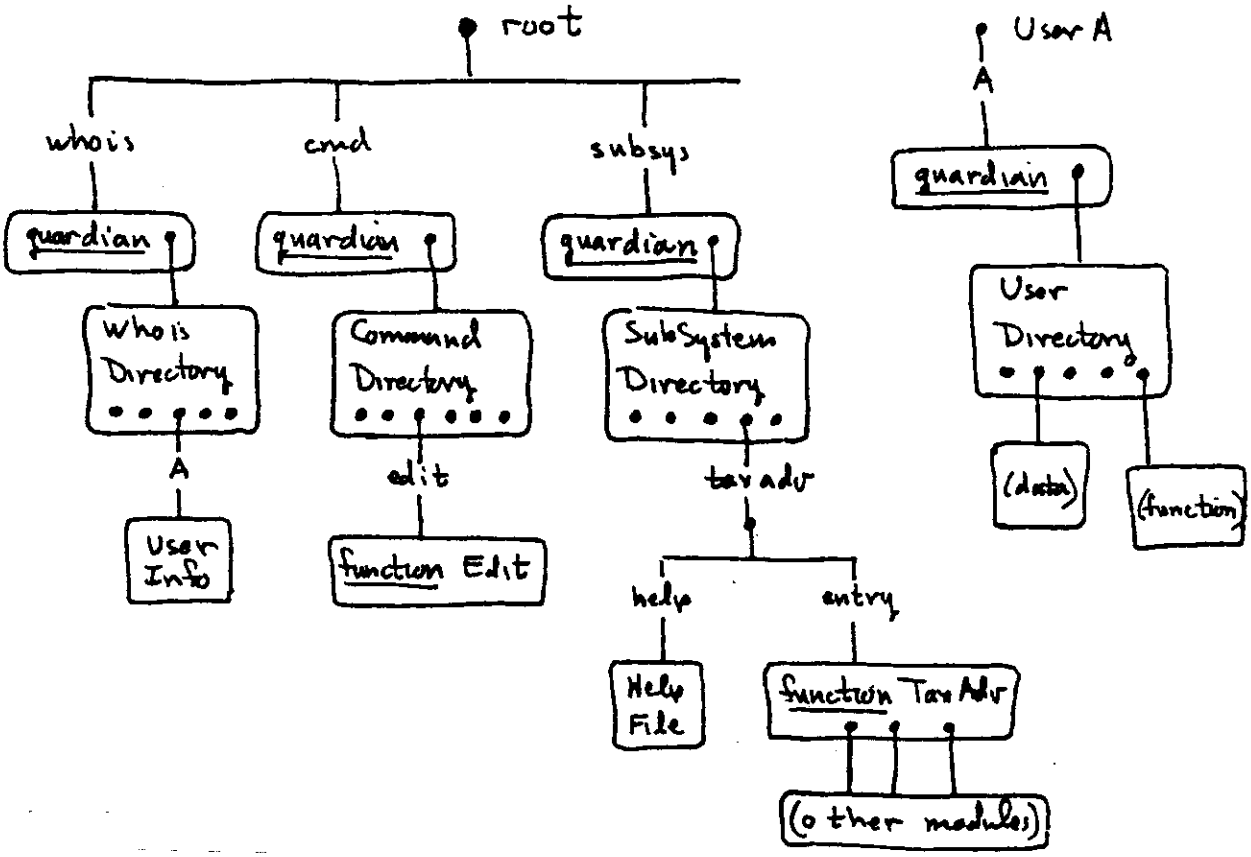

Fig.12. Suggested organization of information in the envisioned computer system.