SEMANTICS OF PARALLEL PROGRAM GRAPHS

BY

Robert M. Keller

UUCS - 77 - 110

July 22, 1977

This work was supported by the National Science Foundation through grant MCS77-09369.

Typing and drawing were done by Karen Evans.

SEMANTICS OF PARALLEL PROGRAM GRAPHS

by

Robert M. Keller
Computer Science Department
University of Utah
Salt Lake City, UT 84112

Abstract

A denotational model for networks of parallel processes is presented which generalizes the work of Kahn by using alternative data types, e.g. Lisp-like operators on trees. It is shown that the ordering aspect of data types plays a central role in determining how much parallelism can be present. It is also shown that this model allows an implementation-independent view of concepts related to the suspensions of Friedman and Wise and the lazy evaluator of Henderson and Morris. Finally, the cyclic structures of Kahn are shown to be eliminable through the use of recursion.

Keywords and Phrases: asynchronism, coroutine, denotational, graphs, Lisp parallelism, processes, recursion, semantics, streams, trees

CR Categories: 4.29, 4.32, 4.34, 5.21, 5.23, 5.24

Introduction

Techniques for the expression of parallelism within programs have received considerable attention in the past decade. Such techniques are not only useful in allowing the programmer to take advantage of a multiple processor facility; they also provide an effective way for specifying the essence of a computation without unnecessary sequencing, so that optimization can be accomplished without prior "de-sequencing."

Recent work of Kahn [Kahn 74, Kahn and MacQueen 76] forms the prime motivation for the current study. However, we hope to illustrate that our ideas also form a conceptual basis for the "suspension" notion in [Friedman and Wise 76] and the "lazy evaluator" of [Henderson and Morris 76].

The framework in which our ideas will be expressed is that of a network of operators. Such networks have been studied by numerous authors, e.g. [Karp and Miller 66], [Adams 68], [Seror 70], [Patil 67], [Patil 70], and [Kahn 74]. In all cases mentioned, the concern was with stream data types. That is, communication between operators was carried out by sending a stream of objects selected from a (possibly-infinite) set of atoms from one operator to another. Certain streams could also be designated as input streams or output streams.

Of the authors mentioned in the preceding paragraph, only Patil and Kahn dealt with "denotational" models, i.e. models in which the operators were specified in terms of their action on the entire history of a stream. The other authors dealt with "operational" approaches, in which an operator was defined in terms of state changes.

[Keller 77] observed that it was useful to generalize the network-of-operator approach to general data types, following [Scott 76] and [Vuilleman 73]. Keller also showed examples of the usefulness of specific alternative data types, such as bags. It is in this spirit which we wish to continue in the present paper. Specifically, we will show that the tree data type (mentioned only in passing in [Keller 77]) provides a useful basis for parallel networks of Lisp-like operators. The importance of selecting the proper ordering accompanying this data type is shown. It will also be shown that this data type generalizes the former notion of streams in a natural way, and provides a conceptual basis for the aforementioned schemes of [Henderson and Morris 76] and [Friedman and Wise 76].

Data Types

Following [Vuillemin 73], and [Scott 76], a data type is a set D with a partial order $\sqsubseteq$ on D and a least element $\perp$ in D, such that for any chain of elements in D,

$$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots$$

there is a unique limit (or least upper bound), denoted

$$\sqcup \{d_0, d_1, d_2, \ldots\}$$

If $\hat{d}$ denotes the above element, then its defining property is that

$$(\forall i)\ d_i \sqsubseteq \hat{d}$$

and if d is any element such that

$$(\forall i)\ d_i \sqsubseteq d$$

then necessarily $\hat{d} \sqsubseteq d$. Of course, we have $(\forall d \in D)\ \perp \sqsubseteq d$.

Roughly speaking, the elements of such a domain are the objects which will be computed. The ordering $\sqsubseteq$ represents the degree to which the computation of an object has been completed. That is, during the history of a computation, we may expect to observe at the output successively more-complete objects, i.e. a sequence

$$\perp \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq d_3 \sqsubseteq \ldots$$

The ultimate output is not the sequence above, but rather its least upper bound. Some computations may never actually complete, in which case we may get an infinite sequence of distinct objects with a limit which is in some sense infinite. However, a computation may also fail to produce any non-trivial output, so that the least upper bound is just the element $\perp$. It is sometimes intuitively helpful to think of the latter as the "undefined" object.

In this paper, we shall be especially interested in two data types. Both of these are based on a set A of atoms (which may well be infinite).

Definition The set of streams over A, denoted $\hat{A}$, is the set of all finite or countably semi-infinite strings of elements in A. Thus A contains the free-monoid A* and least upper bounds for any sequence of elements in A*. The ordering $\sqsubseteq$ is that of "prefix", i.e.

$$x \sqsubseteq y \quad \text{iff} \quad x = y \quad \text{or} \quad (\exists u) \quad xu = y$$

where juxtaposition denotes concatenation. The null string Λ is the least element. Thus an example of a chain and its limit is

$$\Lambda \sqsubseteq a \sqsubseteq ab \sqsubseteq aba \sqsubseteq abab \sqsubseteq \ldots \sqsubseteq (ab)^{\omega}$$

Definition The set of finite-height trees (fht's) over A is defined as follows:

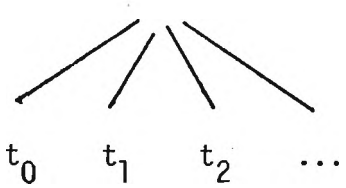
(i) Λ is an fht (the "null" tree)

(ii) Any element of A is an fht (an "atomic" tree)

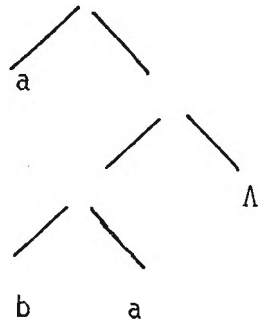
(iii) If $t_0, t_1, t_2, \ldots$ is any countable sequence of fht's, then the bracketed combination $\langle t_0, t_1, t_2, \ldots \rangle$ is an fht.

(iv) Extremal clause.

We will be mostly concerned with binary trees, in which the sequence $\langle t_0, t_1, \ldots \rangle$ is just $\langle t_0, t_1 \rangle$. For purposes of clarification, we sometime represent the tree $\langle t_0, t_1, t_2, \ldots \rangle$ as



For example, the tree <a, <<b, a>, Λ>> would be exhibited as



Note The set of fht's does not form a data type, as limits do not exist. We introduce it merely to simplify the presentation of the set of trees, which does form a data type.

We define an ordering $\sqsubseteq$ on the above set as follows:

(i) $\Lambda \sqsubseteq t$ for any fht t

(ii) $a \sqsubseteq t$ for $a \in A$ iff $t = a$

(iii) If $t_0 \sqsubseteq t_0'$, $t_1 \sqsubseteq t_1'$, $t_2 \sqsubseteq t_2'$, ...
then $\langle t_0, t_1, t_2, \ldots \rangle \sqsubseteq \langle t_0', t_1', t_2', \ldots \rangle$

(iv) Extremal clause.

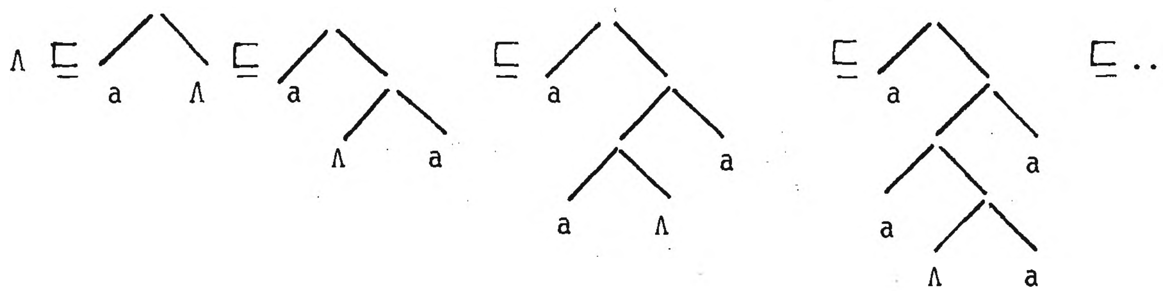
Definition The set of trees over A, denoted $\tilde{A}$, is defined as follows:

(i) Any finite-height tree over A is a tree over A.

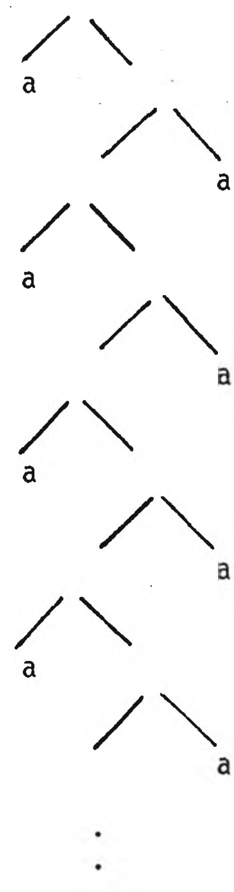
(ii) If $t_0 \sqsubseteq t_1 \sqsubseteq t_2 \sqsubseteq \ldots$ then the limit of this sequence is a tree over A.

(iii) Extremal clause.

<u>Example</u> Notice that if a is an atom, then according to the above definitions,



and following this branching pattern, we have as a limit the infinite tree



Definition A tree is called complete if none of its leaves are Λ. Otherwise it is called incomplete.

Notice that an infinite tree can be complete. The previous example is one instance.

Observation If t is complete, then $t \sqsubseteq t'$ iff $t = t'$.

This observation follows from the fact that a complete tree has only atoms as leaves, and atoms are $\sqsubseteq$ themselves only.

A computation will produce trees from the root outward. One may think of the presence of Λ at a leaf as denoting the possibility that the tree may be extended by a computation at that leaf. Once a leaf becomes an atom, there is no possibility of extending it from that node. Hence a complete tree must be the ultimate result of a computation, although the converse is not necessarily true.

For future reference, we define two special kinds of trees, "skinny" trees, and "flat" trees.

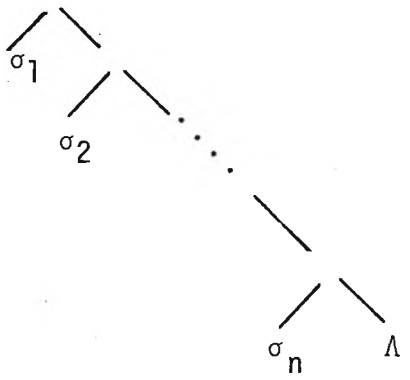
Definition A skinny tree is a binary tree such that if $\langle t_0, t_1 \rangle$ is a sub-tree, then t_0 is an atom. A flat tree is either the null tree or an atomic tree.

Hence every flat tree is also skinny.

Observation For any set of atoms A, there is a subset of the skinny trees over A which is isomorphic to the set of streams over A. That is, a finite stream

$$\sigma_1 \ \sigma_2 \cdots \sigma_n$$

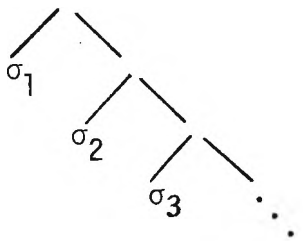
corresponds to the incomplete skinny tree



and an infinite stream

$$\sigma_1 \sigma_2 \sigma_3 \cdots$$

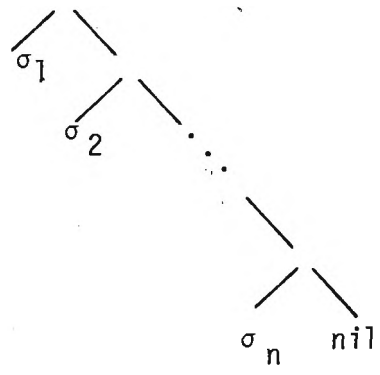
corresponds to the complete skinny tree



Note

The representation discussed above is related to the Lisp representation of "lists". The latter, however, use a special atom ("nil") as a terminating character, i.e. the list

is the complete tree



The distinction between streams and lists is important because we want operators to be "continuous" functions (defined in the next section). While a function which concatenates streams is not continuous unless there is only a single atom, it is easy to see that there is a continuous function which concatenates lists. Furthermore, unlike [Kahn 74] and [Weng 75], we can easily handle streams of streams, streams of streams of streams, etc. to any desired level of nesting.

Note Infinite trees allow the representation of so-called "circular lists" as in Lisp. Every such list corresponds to an infinite tree, but there are infinite trees which do not correspond to any circular list.

Before proceeding to the next section, we should also observe that if D_1 and D_2 are data types with orderings $\sqsubseteq_1$ and $\sqsubseteq_2$ respectively, then $D_1 \times D_2$ is a data type with

$$(d_1, d_2) \sqsubseteq (d_1', d_2') \text{ iff } d_1 \sqsubseteq_1 d_1' \text{ and } d_2 \sqsubseteq_2 d_2'.$$

Operators

According to [Scott 76], computable functions are "continuous" operators on data-types. As shall be seen, this manifesto certainly has high intuitive content for the data types of interest here.

Definition A function $f: D_1 \to D_2$ is continuous if for any chain in D_1

$$d_0 \sqsubseteq d_1 \sqsubseteq d_2 \sqsubseteq \ldots$$

we also have a chain in D_2 (this is called the monotonicity condition)

$$f(d_0) \sqsubseteq f(d_1) \sqsubseteq f(d_2) \sqsubseteq \ldots$$

and furthermore

$$f(\sqcup\{d_0, d_1, d_2, \ldots\}) = \sqcup_2\{f(d_0), f(d_1), f(d_2), \ldots\}$$

The implication of continuity is that when a function is supplied with "more nearly complete" input, the output can become "no less complete." The importance of this condition for operators on sequence domains was observed in [Patil 70] and [Kahn 74].

Another way of understanding continuity is that a continuous function is one which is monotone and which is defined on infinite objects by taking the limit of the definition on finite objects. For example, the operator cons on finite trees is just

$$\underline{\text{cons}}\ (t_0, t_1) = \langle t_0, t_1 \rangle$$

This operator is monotone according to the ordering given in an earlier definition, since

$$(t_0, t_1) \sqsubseteq (t_0', t_1')$$
$$\Longrightarrow t_0 \sqsubseteq t_0' \text{ and } t_1 \sqsubseteq t_1'$$
$$\Longrightarrow \langle t_0, t_1 \rangle \sqsubseteq \langle t_0', t_1' \rangle$$

We then extend the definition of cons to infinite trees, by representing any such trees t, t' as limits:

$$t = \bigsqcup\{t_0, t_1, t_2, \ldots\}$$

$$t' = \bigsqcup\{t_0', t_1', t_2', \ldots\}$$

and define

$$\underline{\text{cons}}\ (t, t') = \bigsqcup\{\langle t_0, t_0'\rangle, \langle t_1, t_1'\rangle, \langle t_2, t_2'\rangle, \ldots\}$$

This definition still simply means that cons forms a new tree from two trees by combining their roots to get a new root.

We now extend other familiar functions to the domain of trees in general, and observe that all of these extensions are continuous:

$$\underline{\text{car}}(t) = \begin{cases} t_0 & \text{if } t = \langle t_0, t_1\rangle \\ \underline{\text{error}} & \text{if } t \text{ is an atom} \\ \Lambda & \text{otherwise (i.e. } t = \Lambda) \end{cases}$$

$$\underline{\text{cdr}}(t) = \begin{cases} t_1 & \text{if } t = \langle t_0, t_1\rangle \\ \underline{\text{error}} & \text{if } t \text{ is an atom} \\ \Lambda & \text{otherwise (i.e. } t = \Lambda) \end{cases}$$

$$\underline{\text{atom}}(t) = \begin{cases} \underline{\text{true}} & \text{if } t \text{ is an atom} \\ \underline{\text{false}} & \text{if } t \neq \Lambda \text{ and } t \text{ is not an atom} \\ \Lambda & \text{otherwise (i.e. } t = \Lambda) \end{cases}$$

(Here we assume atom maps into a domain which contains the atoms true and false.)

$$\underline{\text{atomeq}}(t_0, t_1) = \begin{cases} \underline{\text{true}} \text{ if } \underline{\text{atom}}(t_0) = \underline{\text{true}} \text{ and } \underline{\text{atom}}(t_1) = \underline{\text{true}} \text{ and } t_0 = t_1 \\ \underline{\text{false}} \text{ if } \underline{\text{atom}}(t_0) = \underline{\text{true}} \text{ and } \underline{\text{atom}}(t_1) = \underline{\text{true}} \text{ and } t_0 \neq t_1 \\ \underline{\text{false}} \text{ if } \underline{\text{atom}}(t_0) \neq \Lambda \text{ and } \underline{\text{atom}}(t_1) \neq \Lambda \text{ and } \underline{\text{atom}}(t_0) \neq \underline{\text{atom}}(t_1) \\ \Lambda \text{ otherwise} \end{cases}$$

$$\underline{\text{if}}\ldots\underline{\text{then}}\ldots\underline{\text{else}}(t_0, t_1, t_2) = \begin{cases} t_1 \text{ if } t_0 = \underline{\text{true}} \\ t_2 \text{ if } t_0 = \underline{\text{false}} \\ \underline{\text{error}} \text{ if } \underline{\text{atom}}(t_0) = \underline{\text{false}} \\ \Lambda \text{ otherwise} \end{cases}$$

The reader should note that the continuity of the above functions does indeed coincide with the intuitive notion of computability, with Λ connoting "undefined" or "undecided." For example, atom(t) is supposed to tell whether tree t is atomic. If t is atomic, then atom can report true, since atomic trees are complete; there will be no further input to change the answer. On the other hand, if t is a tree which is not null and not atomic, then it is not an atom, and the answer false can be reported in confidence, since no tree "more nearly complete" than t will change this answer. But if t is Λ, it is always possible that future information might make it either an atom or a non-atom, so the answer must remain undecided until such information is provided.

Types of Parallelism

There are several means of introducing parallelism into an otherwise-sequential system. We list a gross classification of these means:

1. Introduce inter-operator parallelism
2. Introduce intra-operator parallelism
3. Modify data-types (and, correspondingly, the operators applied to them) to permit greater concurrency.

We now elaborate on this classification.

It has long been recognized that one way to introduce parallelism is to construct the system as a network of operators, where all operators can potentially compute concurrently, subject to the presence of data being sent from other operators. This technique is explored in many of our references, e.g. [Karp and Miller 66], [Patil 67], [Adams 68], [Patil 70], [Seror 70], [Kahn 74], [Ritchie and Thompson 75], etc. However, as mentioned, this inter-operator parallelism exploits only one of several possibilities.

Another possibility is to modify the operators slightly. For example, [Kleene 52], [Manna and McCarthy 70], [Paterson and Hewitt 70] discuss "parallel" versions of operators, such as the "if...then...else" operator. As defined earlier,

$$\underline{\text{if}}...\underline{\text{then}}...\underline{\text{else}}...(\Lambda, t_1, t_2) = \Lambda$$

That is, if the evaluation of the "if" condition does not produce a non-trivial answer, then the entire computation does not produce an answer, because it can never be decided whether the answer should be t_1 or t_2. However there is one case where an answer can still be given consistently, and that is when $t_1 = t_2$. This suggests that the evaluation of t_1 and t_2 proceed concurrently with evaluation of the condition, so that if the

computations of t_1 and t_2 should happen to complete and the two are equal, then the answer t_1 can be given as the result of the entire computation. This revised operator is then defined as follows:

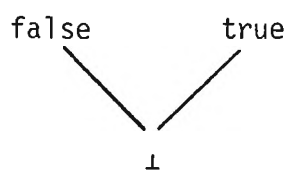
$$\underline{\text{parallel if}}\ldots\underline{\text{then}}\ldots\underline{\text{else}}(t_0, t_1, t_2) = \begin{cases} t_1 \text{ if } t_0 = \underline{\text{true}} \\ t_2 \text{ if } t_0 = \underline{\text{false}} \\ \underline{\text{error}} \text{ if } \underline{\text{atom}}(t_0) = \text{false} \\ t_1 \text{ if } t_1, t_2 \text{ complete and } t_1 = t_2 \\ \Lambda \text{ otherwise} \end{cases}$$

As we shall see, it is possible to invent another version of this operator which gives even more information when $t_0 = \Lambda$.

Finally, we mention the effect data types can have on concurrency. Obviously there can be many data types with a common domain, the difference among the types lying in the associated orderings. For example, any data-type can be re-configured into a <u>flat</u> data type, i.e. one in which

$$x \sqsubseteq y \quad \text{iff} \quad x = y \text{ or } x = \perp$$

For example, the following represents such an ordering $\sqsubseteq$:



Thus, in a flat data type, we either have complete information about an object, or no information. Now since it is the objects in a domain which are transmitted from one operator to another, what we are saying is that a non-flat data type, such as our trees, permits a meaningful semantic description of <u>incremental computation</u> whereas a flat data type does not. For example, [Manna 74] discusses Lisp-like operators on lists, without explicitly defining an associated ordering. We could infer, from his

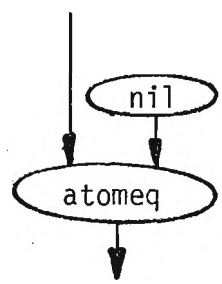
use of a flat data type for the natural numbers, that a flat data type is intended for lists as well, and he does not consider incremental computation.

To see how incremental computation allows greater concurrency, we only have to note that with the former, pieces of an object can be present before the object itself is assembled, and these pieces can be operated on concurrently. Some examples will be forthcoming after we describe networks in more detail.

Networks of Operators

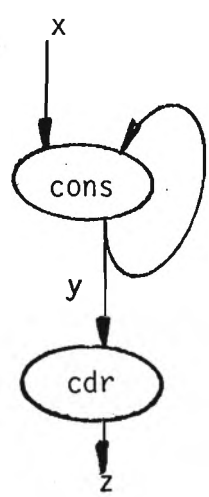
By a network, we mean a directed graph, the arcs of which correspond to data types and the nodes of which correspond to continuous functions on those data types. As indicated in [Keller 77], the semantics of such a network (even if cycles are present) can also be described by a continuous function. Put another way, such a network is determinate, even if parallelism is present.

To give a trivial example, consider the Lisp-like function null which gives output true if the input is the atomic tree nil, false if the input is a non-null tree other than nil, and Λ otherwise. A network which computes this function is shown below



where the node marked "nil" is a constant function which produces the atom nil.

Such networks would be uninteresting, were it not for the possibility of cycles and "recursion." A simple example of a cyclic network is the following:



If the input is x in this example, then the output is a z which satisfies, for some y,

$$y = \underline{cons}(x, y)$$

$$z = \underline{cdr}(y)$$

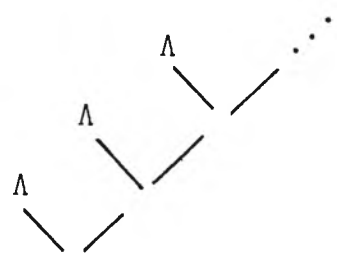
Under least fixed-point semantics (see [Keller 77] for greater detail), we would have

$$y = \underline{cons}(x, \underline{cons}(x, \underline{cons}(x, \ldots)))$$

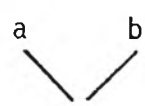
$$z = \underline{cdr}(y) = \underline{cons}(x, \underline{cons}(x, \ldots))$$

For example, if the input x were

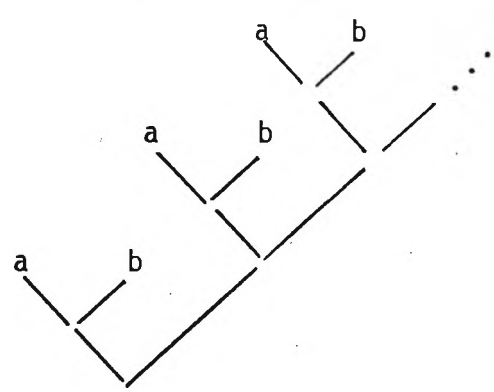
then the output would be the (incomplete) infinite tree



If the input x were



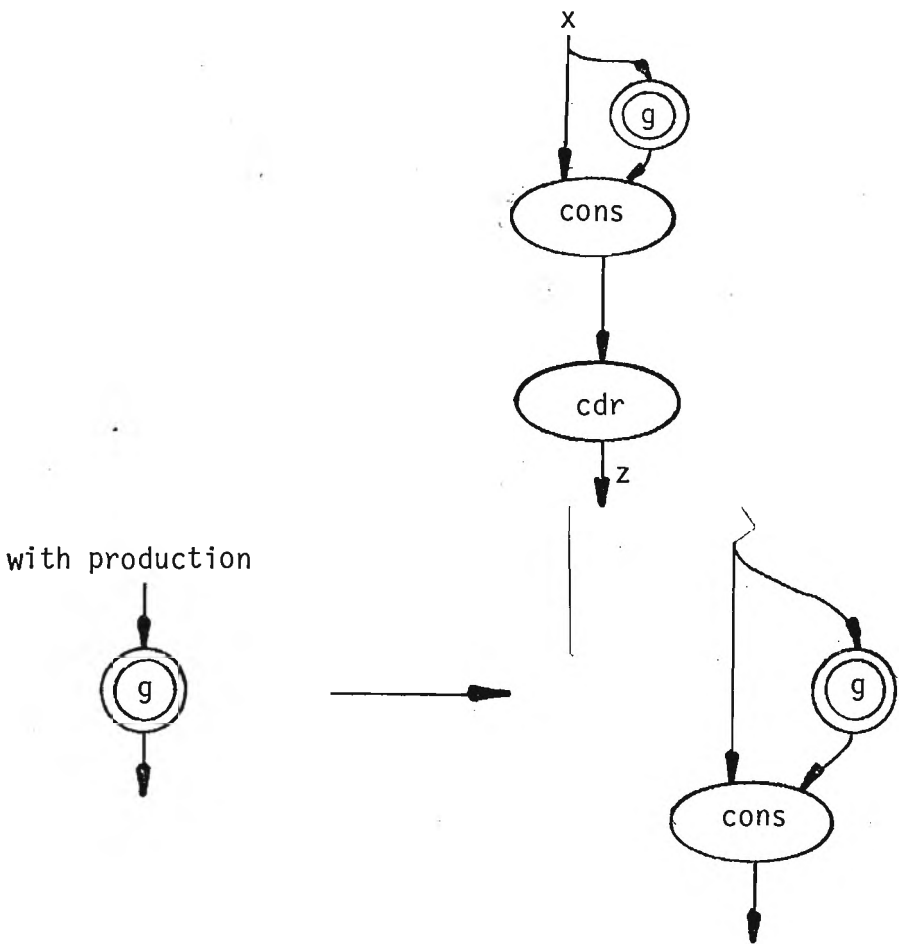
then the output would be the complete infinite tree



and so on.

Note that if a flat ordering were being used, the output for any input would necessarily be Λ.

By a network with recursion, we mean one in which certain nodes (called auxiliary nodes) are named, but not given a priori function definitions. Instead, these names are associated with productions in a "graph grammar" which indicates a substitution to be made for the node when the "appropriate time" comes. For example, consider the network



Here we have used double circles on auxiliary nodes for greater emphasis. The semantics of this network (again see [Keller 77] for more detail) are that the output z in response to input x is that z such that

$$z = \underline{cdr}(\underline{cons}(x, g(x)))$$

where g is the least function satisfying

$$g(x) = \underline{cons}(x, g(x))$$

By "least function," we here imply that the ordering $\sqsubseteq$ on data types D induces an ordering on the continuous functions on those data types, namely

$$f \sqsubseteq g \quad \text{iff} \quad (\forall d \in D) \quad f(d) \sqsubseteq g(d)$$

Again we refer the reader to [Keller 77] for a more detailed explanation. In particular, it is therein shown what is implicit in the above discussion: that any network of continuous functions, possibly with productions, is determinate , in the sense that its semantics is given by a function, and moreover that function is continuous. Thus, parallel activity can occur without concern over the affect of relative timing of operators on the ultimate result.

We also note that solution z of the preceding equations is the same as that of the following system:

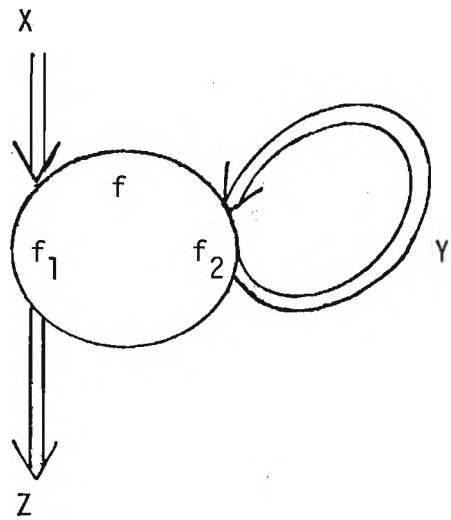
$$z = \underline{cdr}\ (g(x))$$

$$g(x) = \underline{cons}(x, g(x))$$

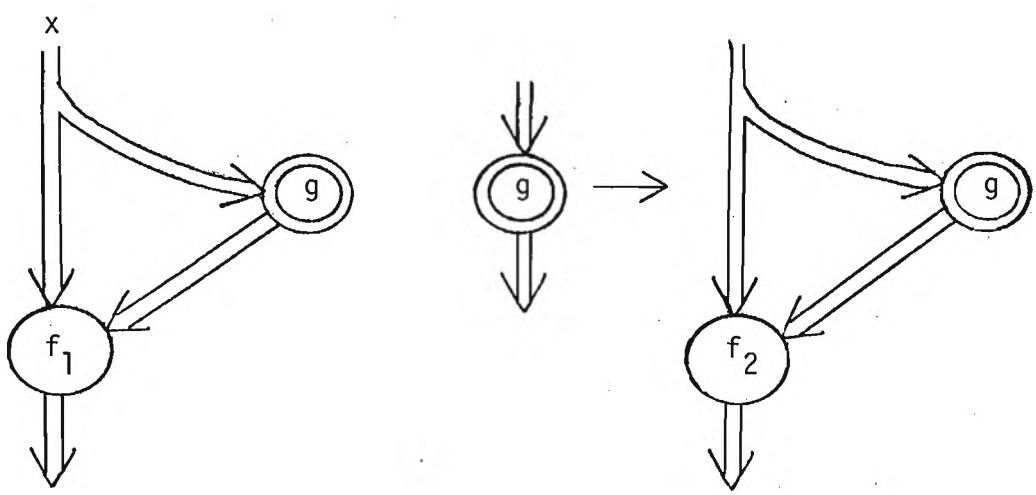
by virtue of the second equality. In fact, if we identify g(x) with y in the earlier cyclic, but non-recursive, example,we see that the two systems have the same solution. This is no accident, for it turns out that we can always get rid of cycles, at the expense of possibly introducing recursion.

Proposition For every network of continuous functions, there is an acyclic network (possibly with additional auxiliaries) which computes the same functions.

Proof Locate within the network a cutset Y of arcs, i.e. a set, the removal of which makes the network acyclic. We then depict the network as follows:



Here f represents the acyclic portion of the network. f_2 represents the function computed on the cutset arcs with respect to the input and f_1 represents the function computed on the non-cutset arcs. We then introduce a new auxiliary, say g, and observe that the following system is equivalent to the original, insofar as the function computed at the output (f_1) is concerned.



To see why this works, the original network has

$$Z = f_1(X, Y)$$

$$Y = f_2(X, Y)$$

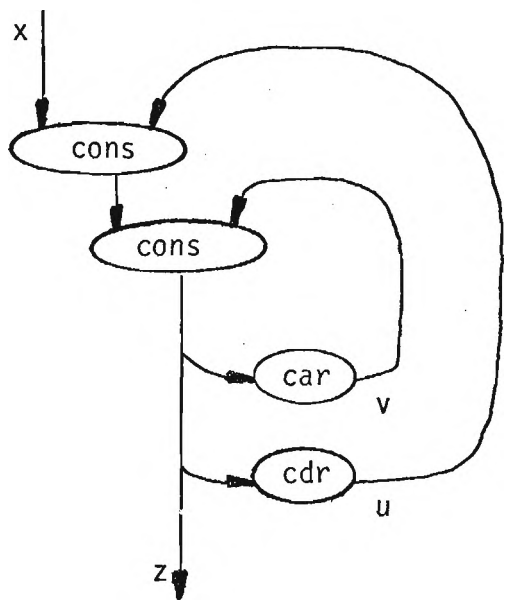
while the newly constructed network has

$$Z = f_1(X, g(X))$$

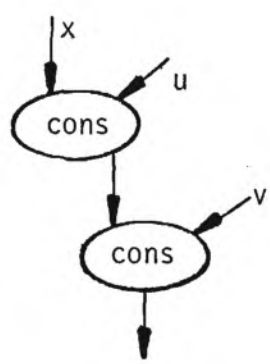
$$g(X) = f_2(X, g(X))$$

Thus, identify Y with g(X) to see the correspondence.

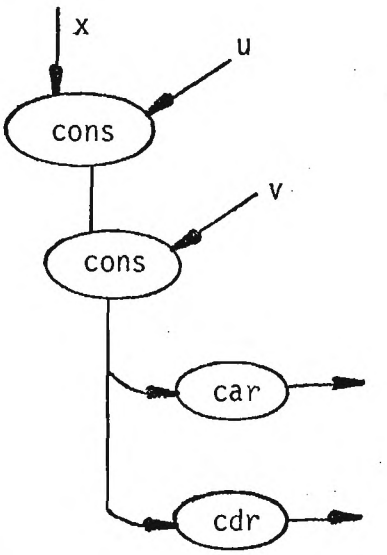
Example Consider the following contrived cyclic network.



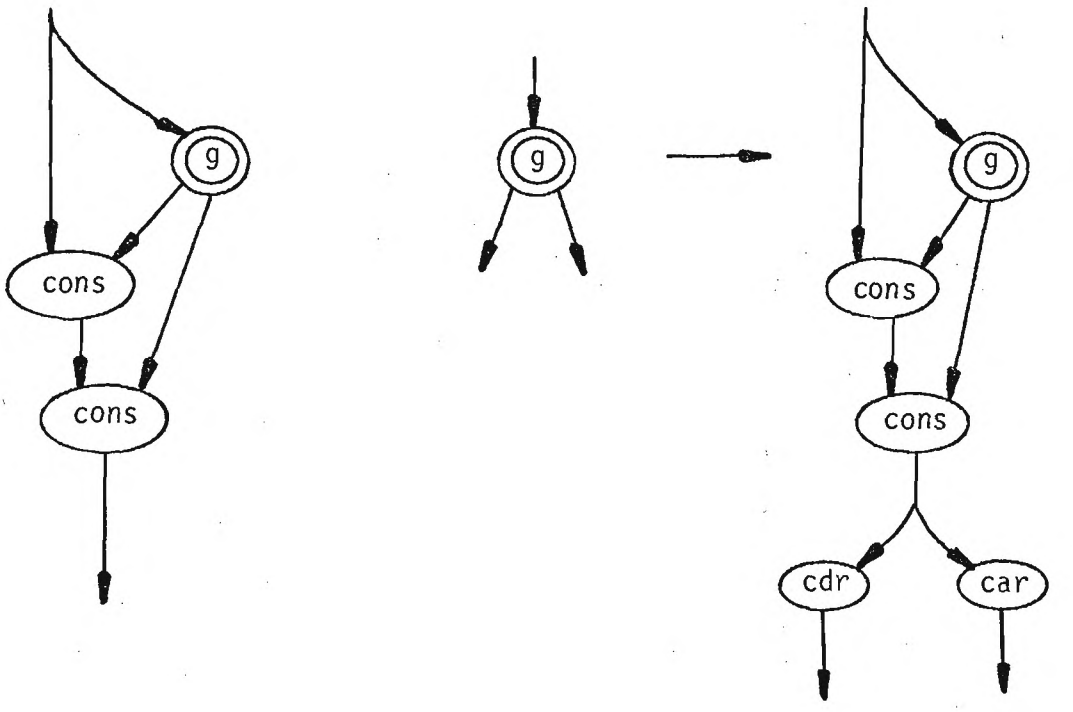
f_1 is the function which relates z to (x, u, v) and choose as f_2 the function which relates (u, v) to (x, u, v). We then have for f_1:



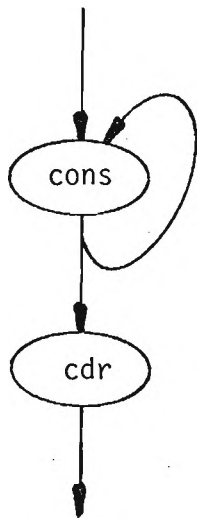
and for f_2:



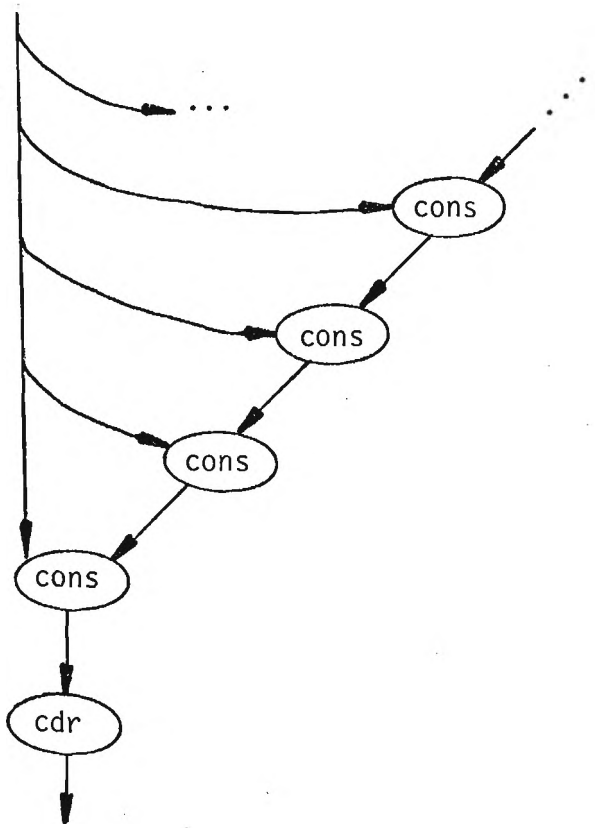
so that the acyclic network is:



Note The construction above indicates that a way to see the effect of cycles is by "backward unraveling" which in this case is equivalent to repeated expansion of the auxiliary graph. Returning to an earlier and simpler example,



backward unraveling using the recursive form would give

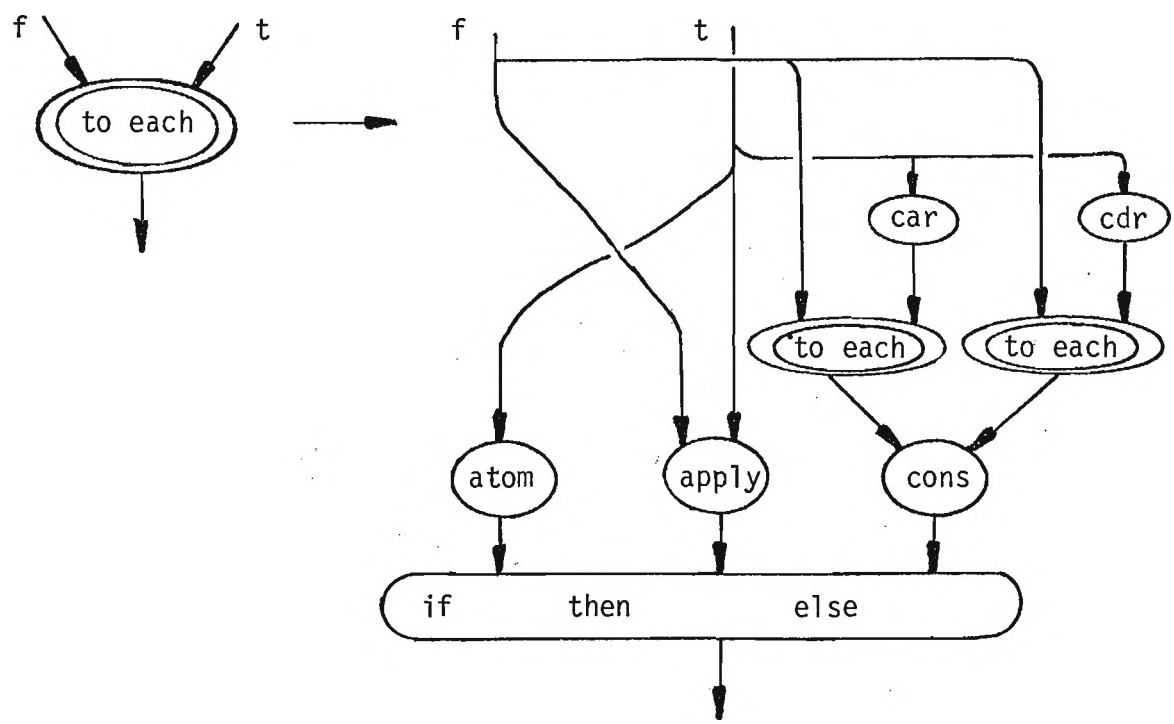


and the infinite tree which is generated can be clearly ascertained.

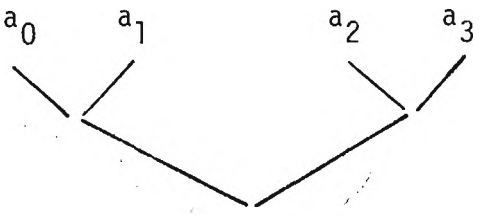
Note The proposition just proved shows that cyclic structures are not an essential aspect of networks of parallel operators, as suggested in [Kahn and MacQueen 76] (Section 1.3, first paragraph). However, our transformation preserves only the input/output function, not each internal function. In practice, cyclic structures may be much more "efficient", if they allow recursion to be avoided.

Note A similar transformation for removing loops from flowcharts was presented in [McCarthy 63]. However flowcharts are not an instance of our parallel program graphs, and our transformation is distinct from McCarthy's. On the other hand, McCarthy's transformation could be applied to a flowchart to get a system of recursion equations, which is an instance of our model.

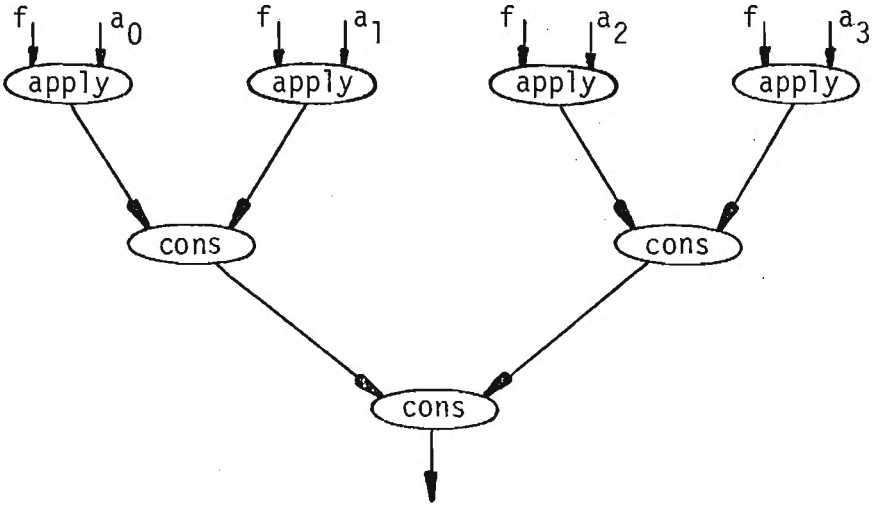
We now proceed to some more practical examples. The first is a parallel version of the usual Lisp function which applies a function f each leaf atom of a tree t:



Here we have taken the liberty of introducing a new data type, namely functions on atoms. An intermediate result of applying a function f to a tree such as



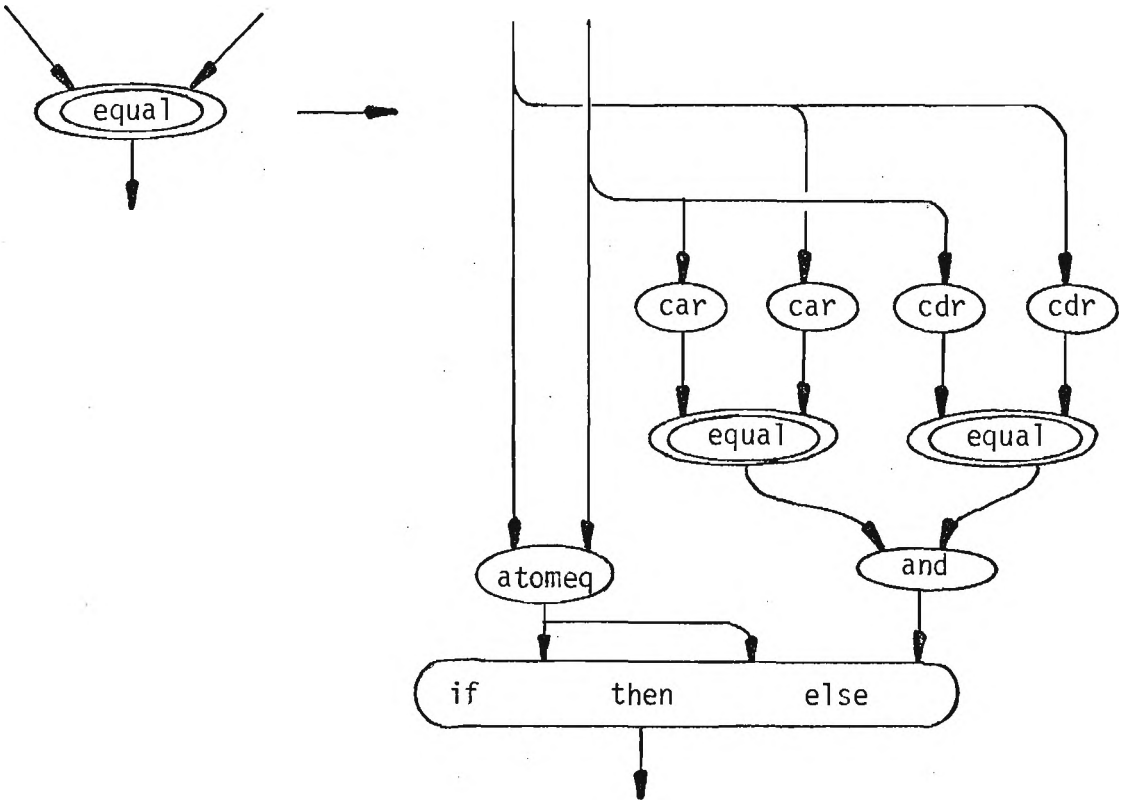
is (removing some superflous nodes):



so that f is indeed applicable to each leaf simultaneously.

Another example is the function equal, defined by

$$\underline{\text{equal}}(t_0, t_1) = \begin{cases} \underline{\text{true}} & \text{if } t_0, t_1 \text{ complete and } t_0 = t_1 \\ \underline{\text{false}} & \text{if } t_0, t_1 \text{ complete and } t_0 \neq t_1 \\ \Lambda & \text{otherwise} \end{cases}$$



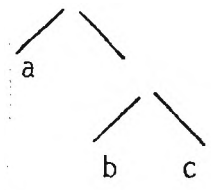
Where and is defined by the table

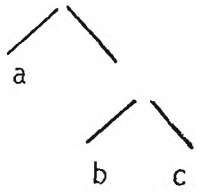
and	Λ	false	true
Λ	Λ	false	Λ
false	false	false	false
true	Λ	false	true

(This is the "parallel" version of and.)

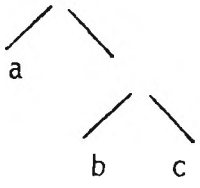
There is an extended version of equal (called eqtree) which permits the extraction of even more information about the arguments. This version constructs a tree which will have true at a leaf if the sub-trees of the arguments at nodes corresponding to that leaf are complete and equal, false at a leaf if the corresponding sub-trees are complete but unequal and one or both is an atom, and Λ if one of the corresponding sub-trees is incomplete.

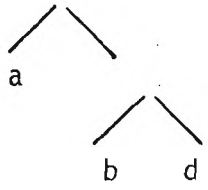
Thus, eqtree gives the right-hand value when applied to the left-hand pair:

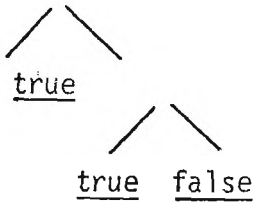


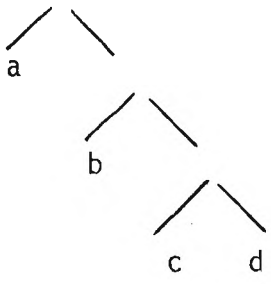


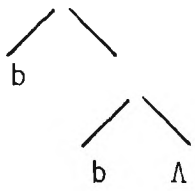
true

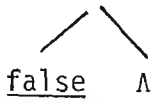


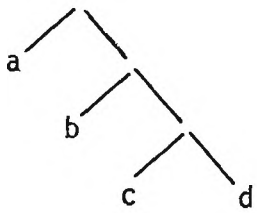


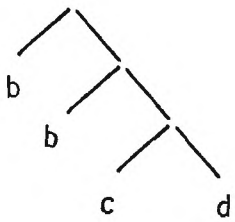


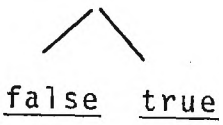




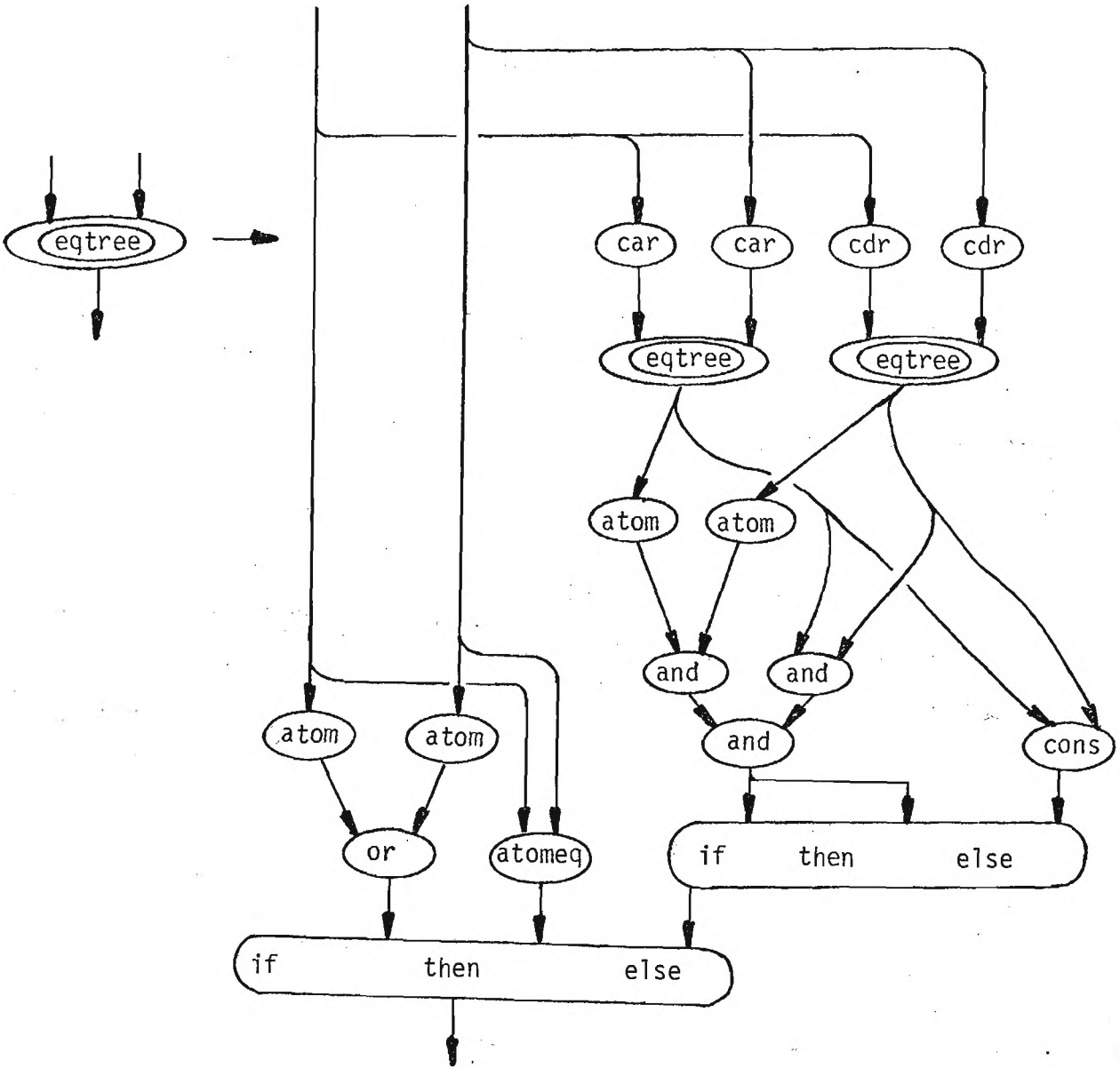








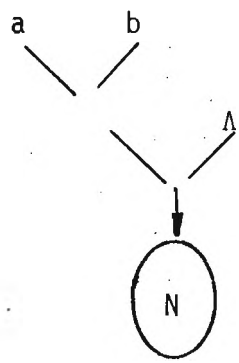
The production is:



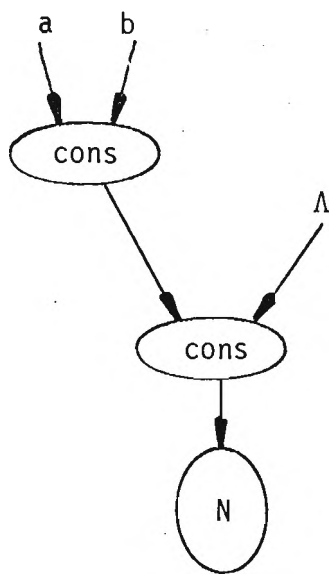
This example exploits all three types of parallelism: inter-operator, intra-operator, and data-type.

Computation

Having described the intended semantics of a parallel program graph, let us describe one notion of "computation" in such a graph. For concreteness, we assume that we have the following base functions: atom, atomeq, null, cons, car, cdr, if...then...else, and possibly productions for auxiliary nodes representing recursion. We assume for simplicity that any input trees are represented by cons operators, i.e.



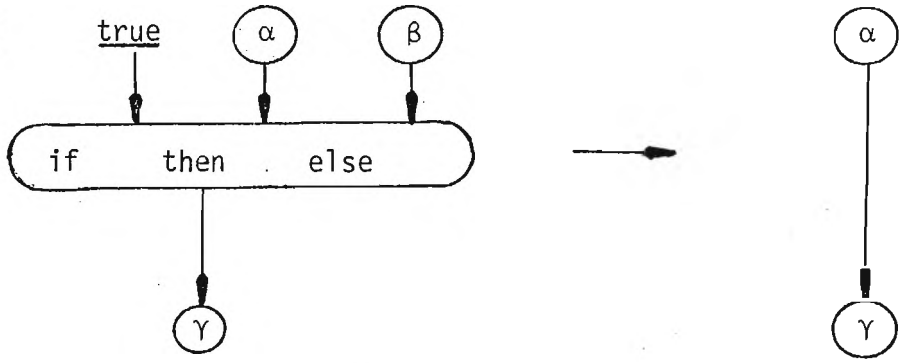
where N is a network of operators, is represented by the extended network



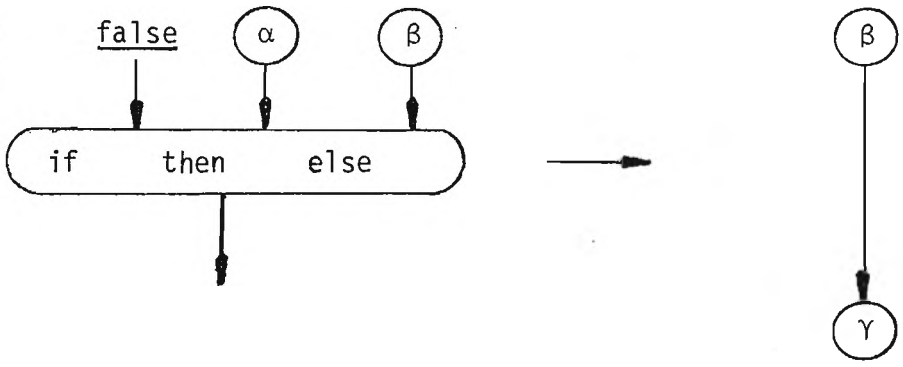
We also assume that the graph is acyclic (without loss of generality, by an earlier proposition).

A computation then involves applying a series of productions, until no further productions can be applied. We include, first of all, those productions (called productions of type 0) corresponding to the expansion of auxiliaries. We also include the following types of productions, where α, β, and γ represent connections to the rest of the network.

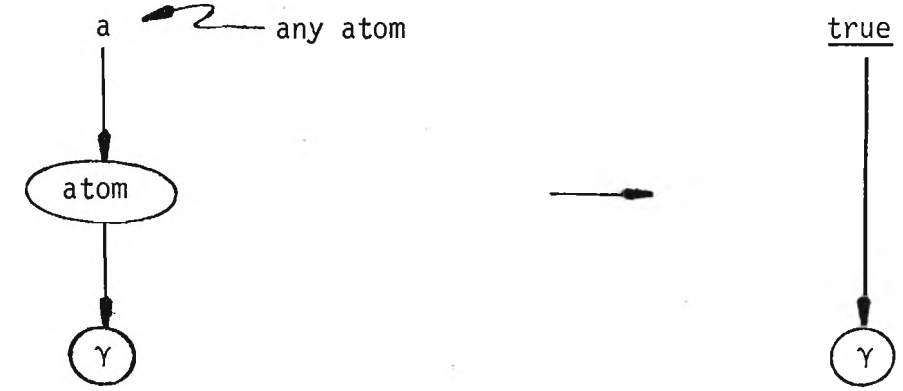
(1a)



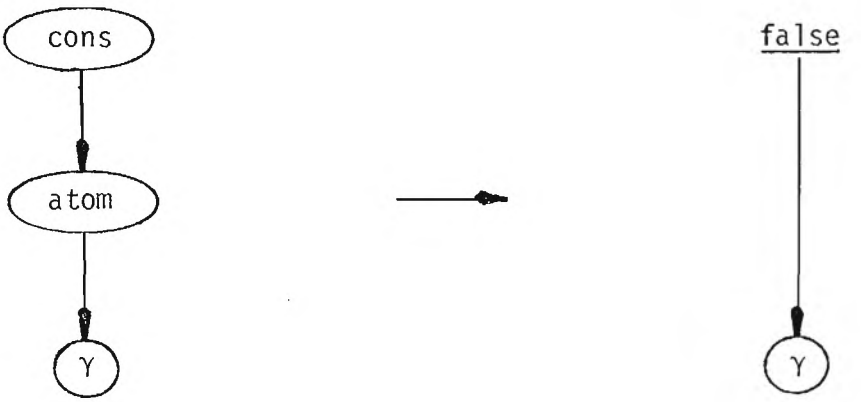
(1b)



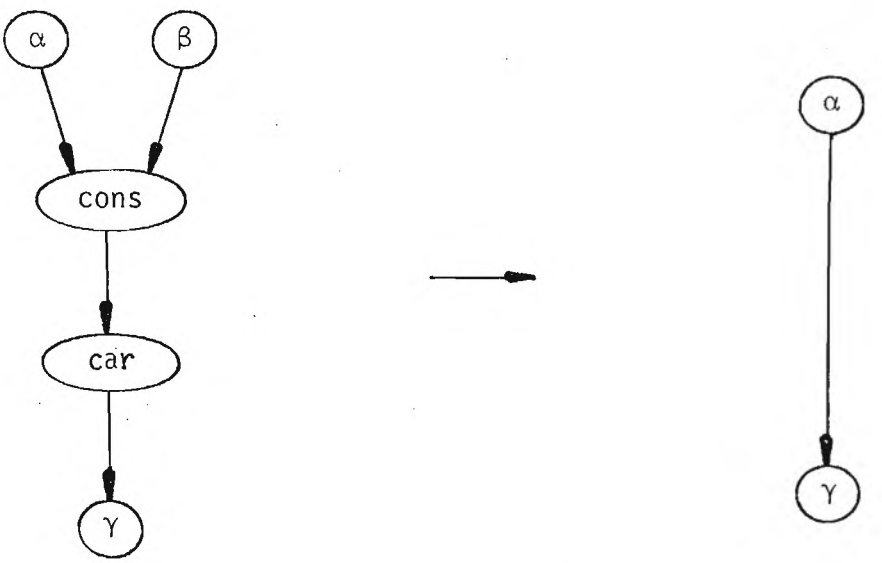
(2a)



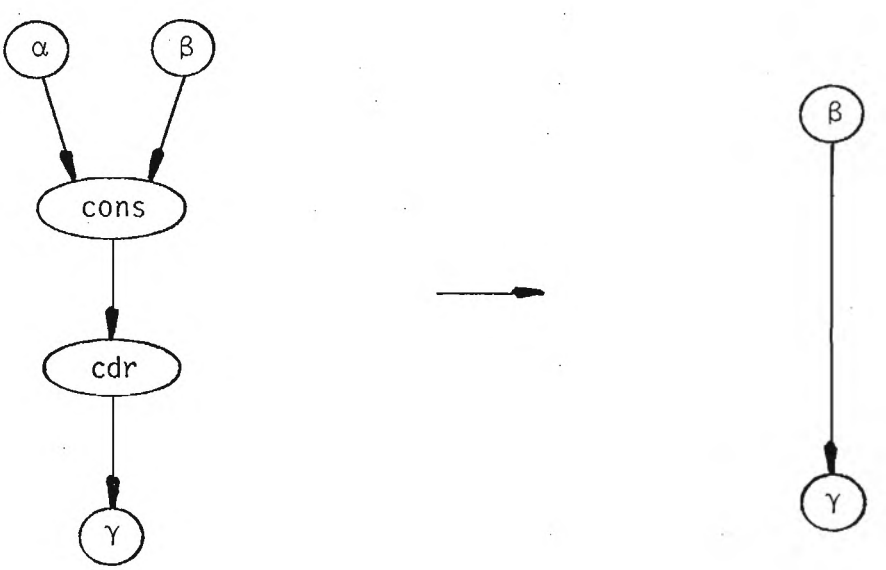
(2b)



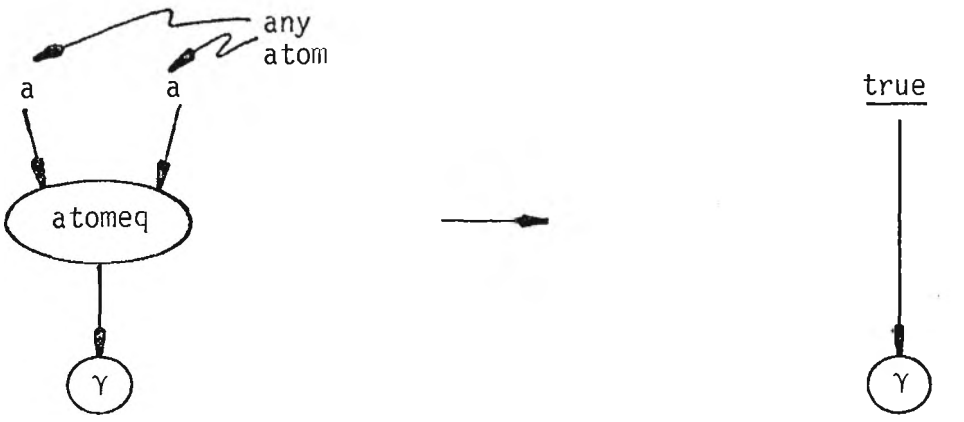
(3a)



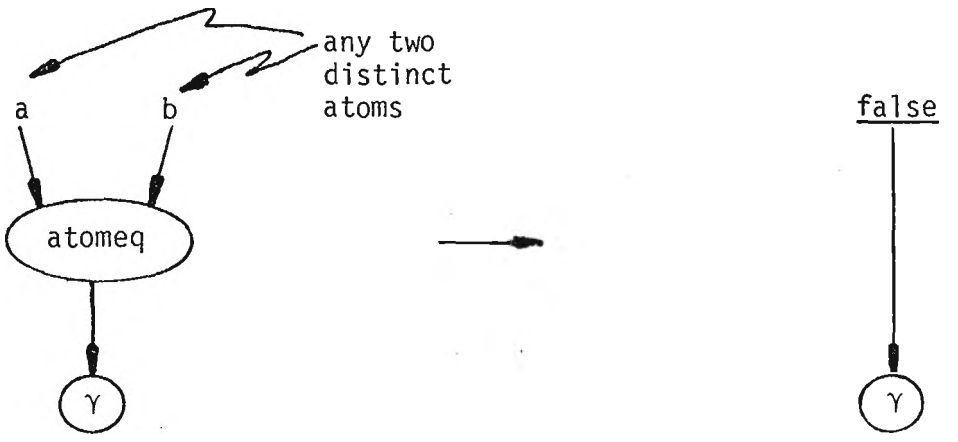
(3b)



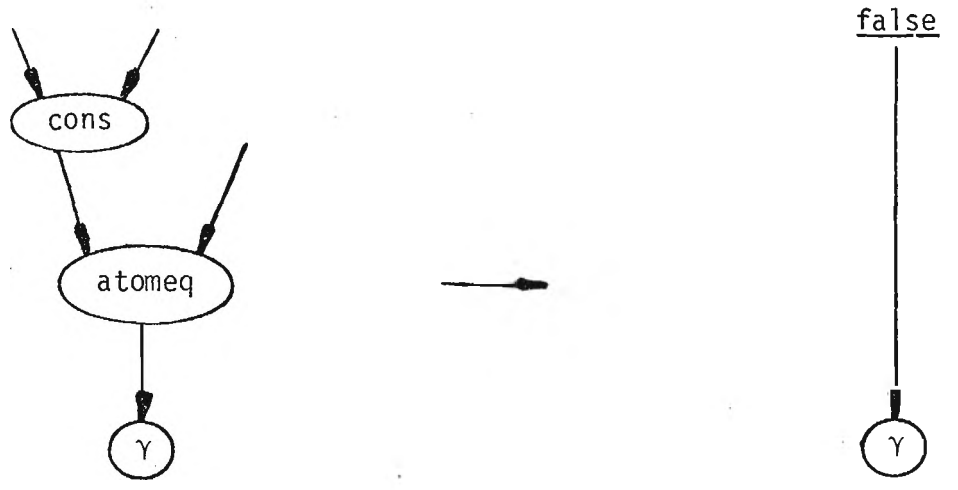
(4a)


any
atom
a
a
atomeq
γ
true
γ

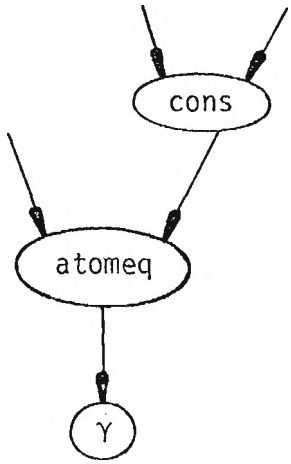
(4b)


any two
distinct
atoms
a
b
atomeq
γ
false
γ

(4c)

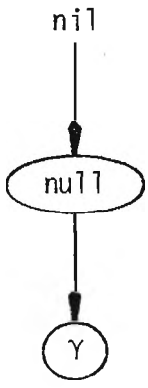

cons
atomeq
γ
false
γ

(4d)


cons
atomeq
γ

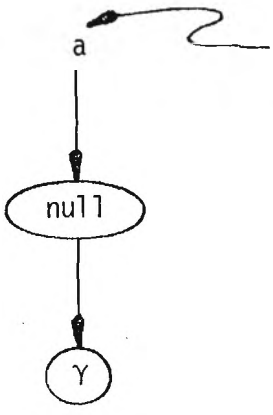

false
γ

(5a)


nil
null
γ

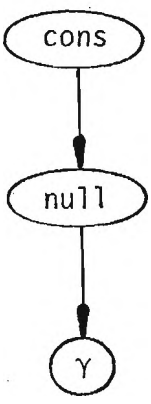

true
γ

(5b)


a
any atom other than nil
null
γ


false
γ

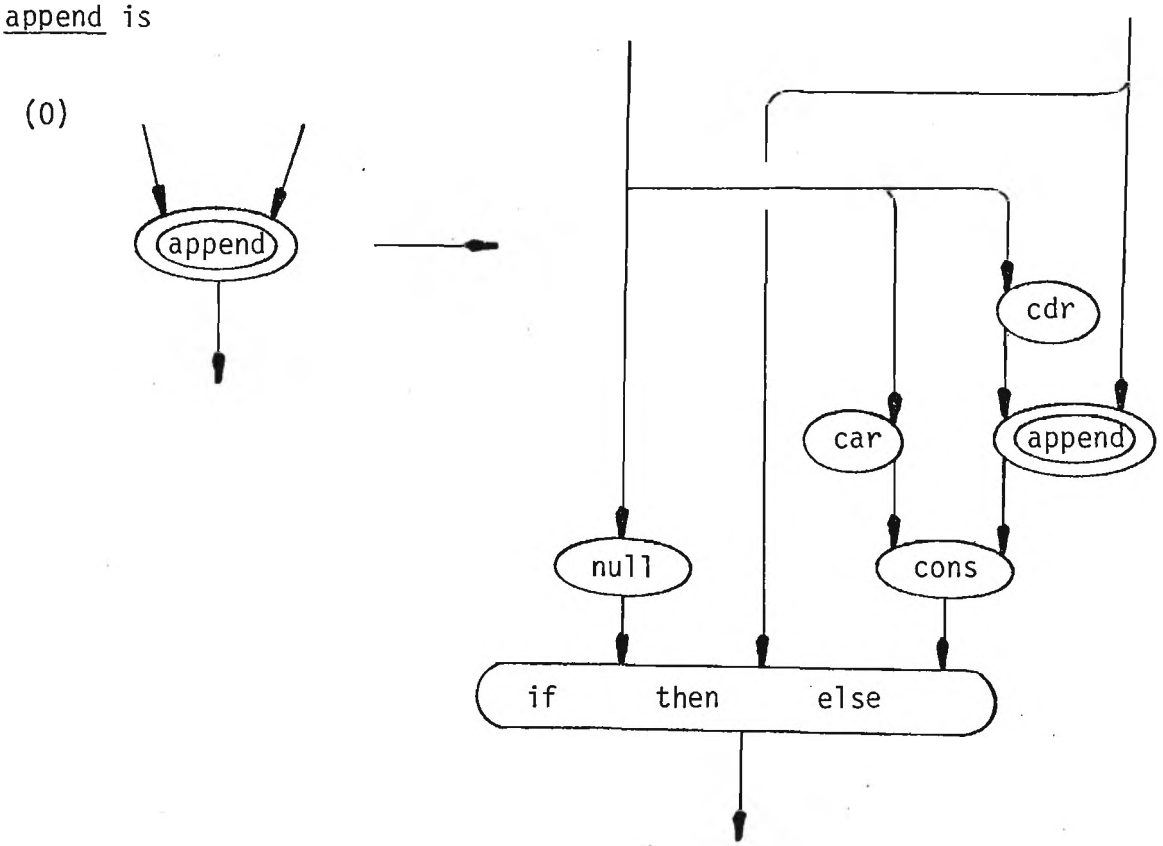
(5c)


cons
null
γ

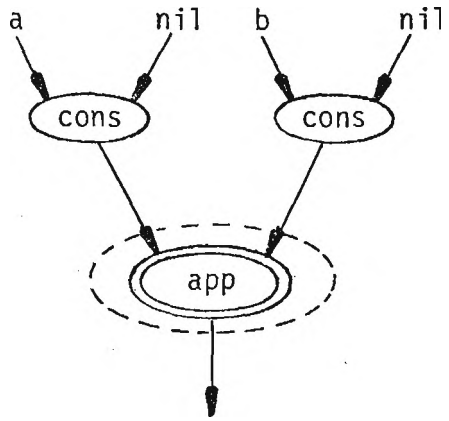

false
γ

In addition to defining what productions can be used, efficient evaluation demands a strategy for the order in which productions are to be applied. The strategy which is most attractive resembles one called "call-by-need" [Wadsworth 71] and proceeds by giving priority to productions "nearest" the output arcs, in hopes of avoiding the expansion of nodes not essential to the ultimate result. In addition, the same result is not re-computed, thanks to the use of a graph rather than a tree to represent the program.

We illustrate by an example in which a recursively-defined function append is applied to the two lists (a) and (b). The production for append is

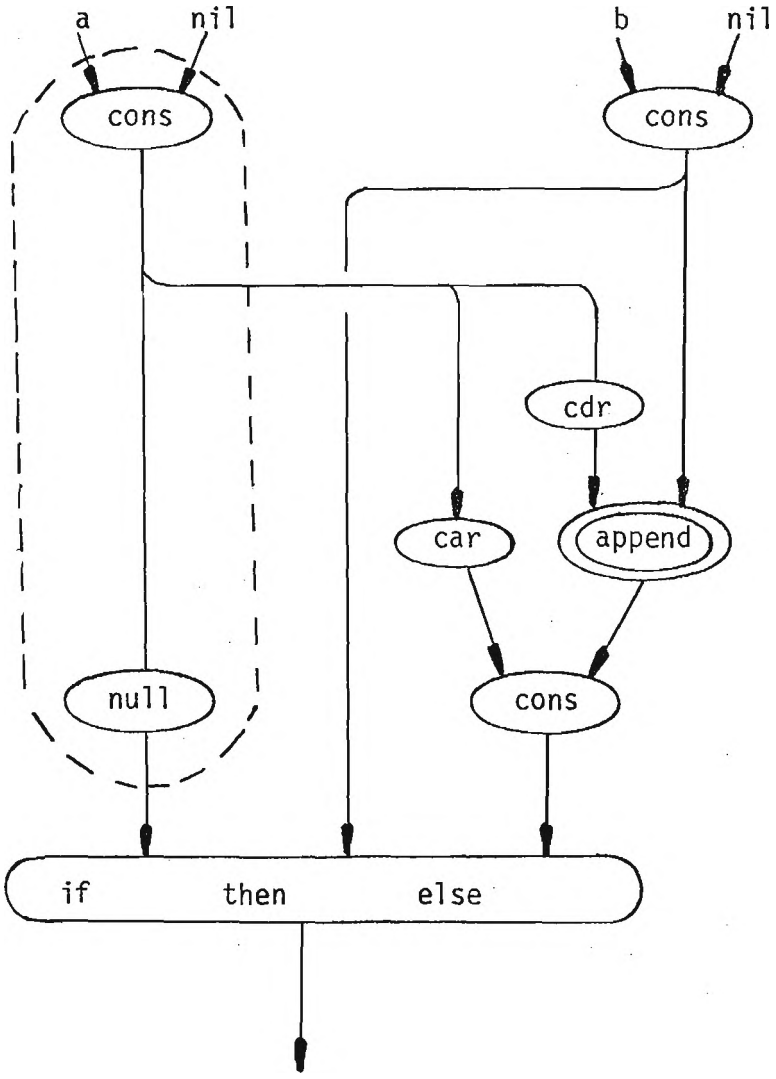


Therefore the graph initially is

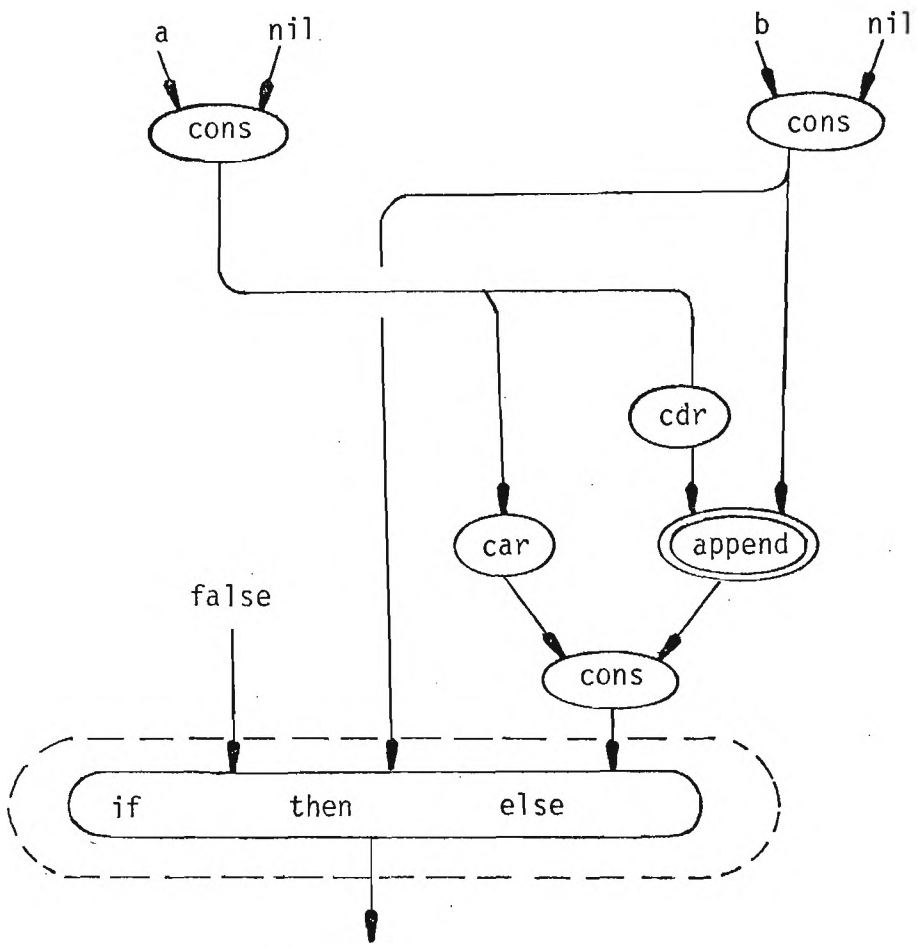


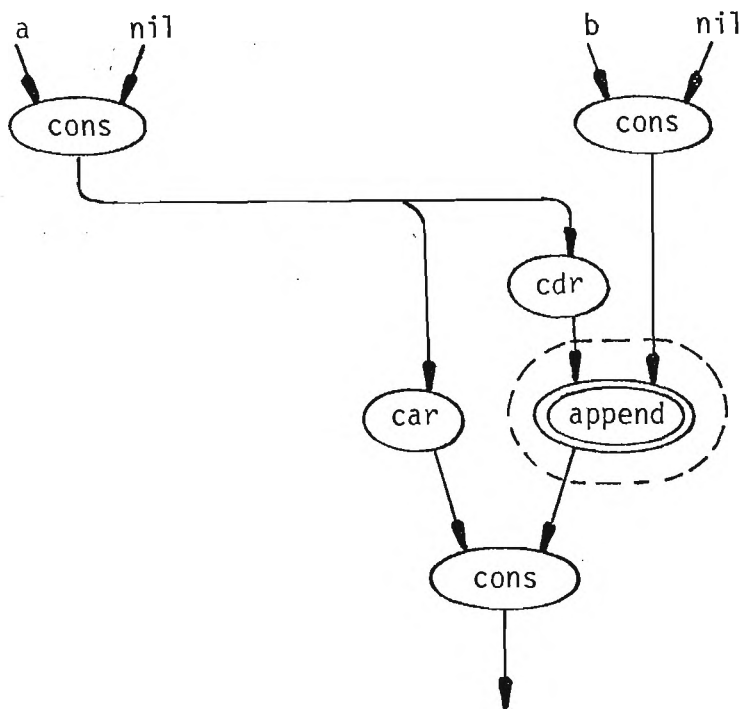
(0)

For readibility, we have encircled the sub-graph which is the antecedent of the production next applied

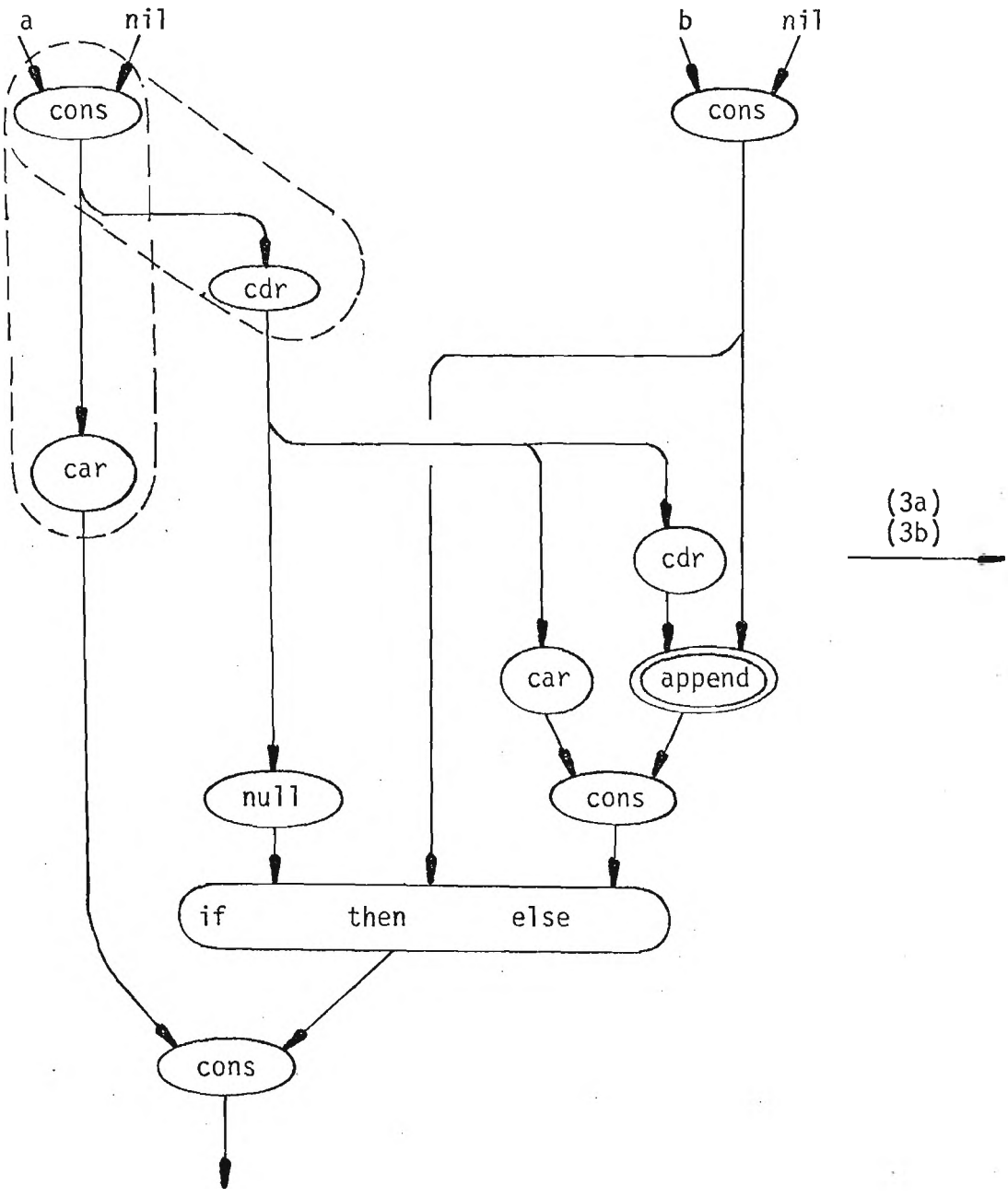


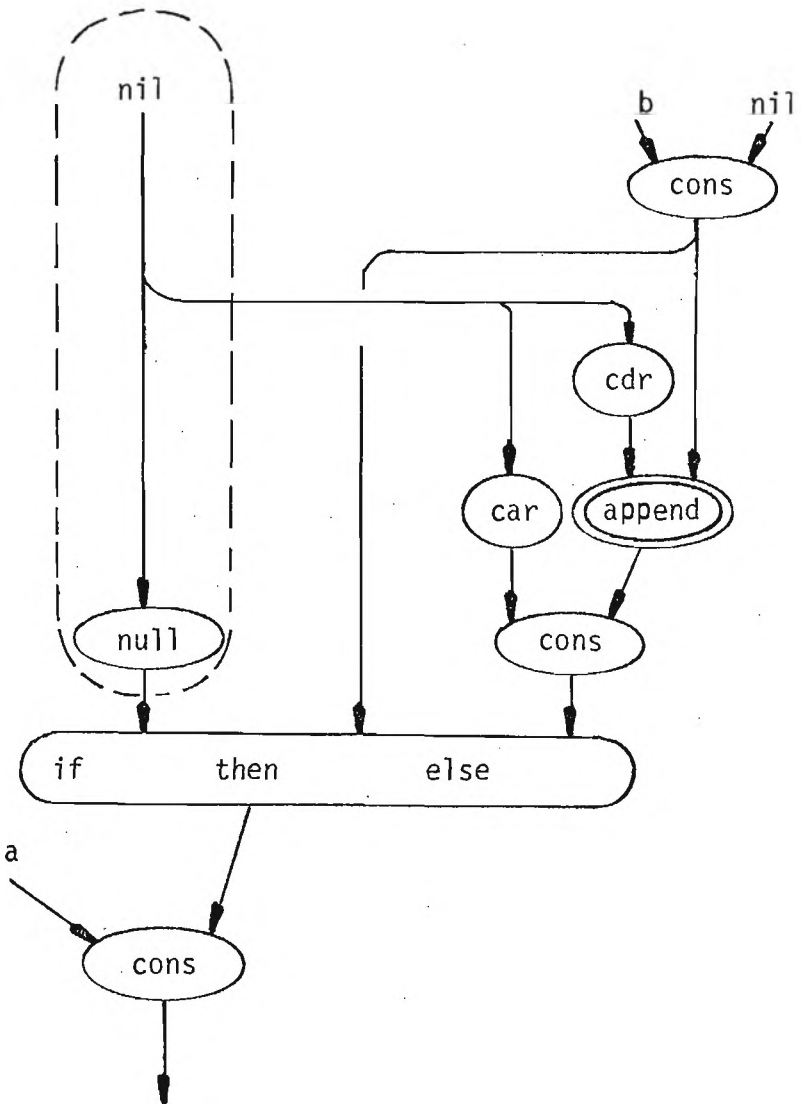
(5c)

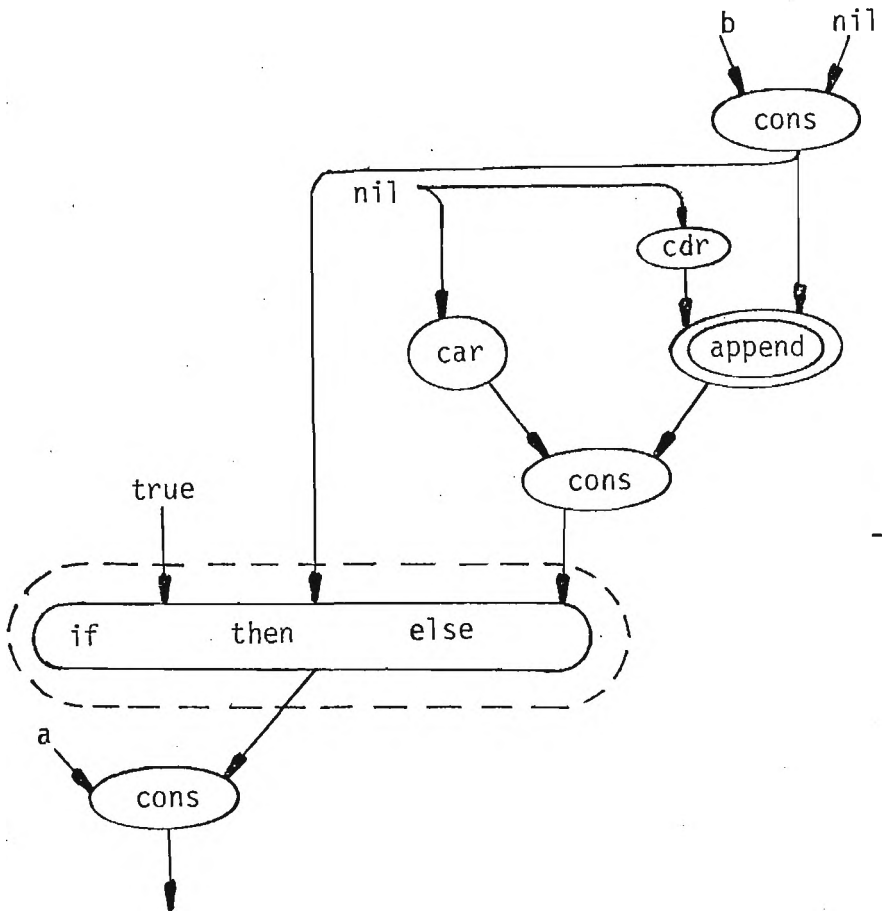

a
nil
b
nil
cons
cons
cdr
car
append
(1b)
false
cons
if
then
else

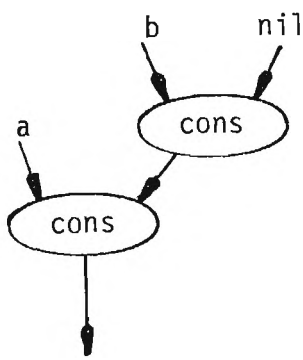

a
nil
b
nil
cons
cons
cdr
car
append
(0)
cons

Here we yield to temptation and apply two productions consecutively:



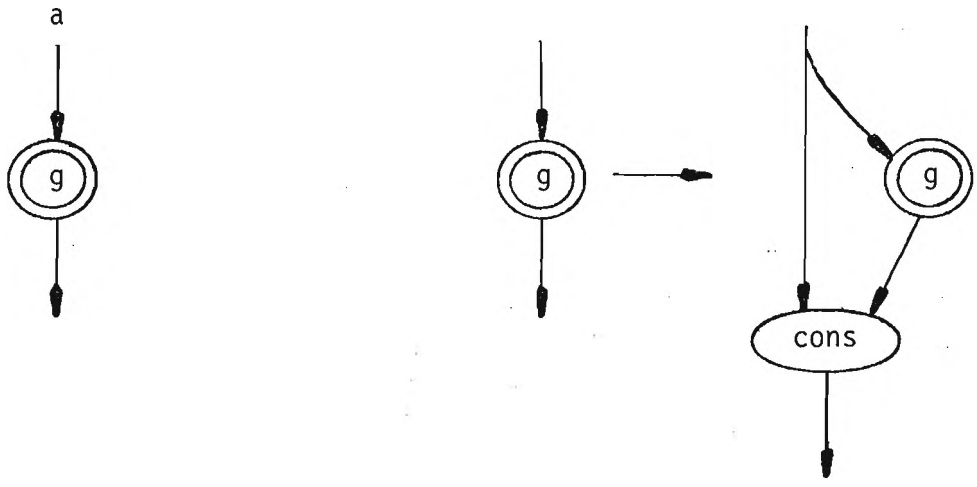

nil
b
nil
cons
cdr
(5a)
car
append
null
cons
if
then
else
a
cons





the end result,
representing the list (a b).

Although the preceding example is informative, there is some redundancy, in that we have, in a sense, violated our call-by-need rule. That is, we kept applying productions until we could go no further. However this approach would be dangerous with a system such as



since we would never stop expanding.

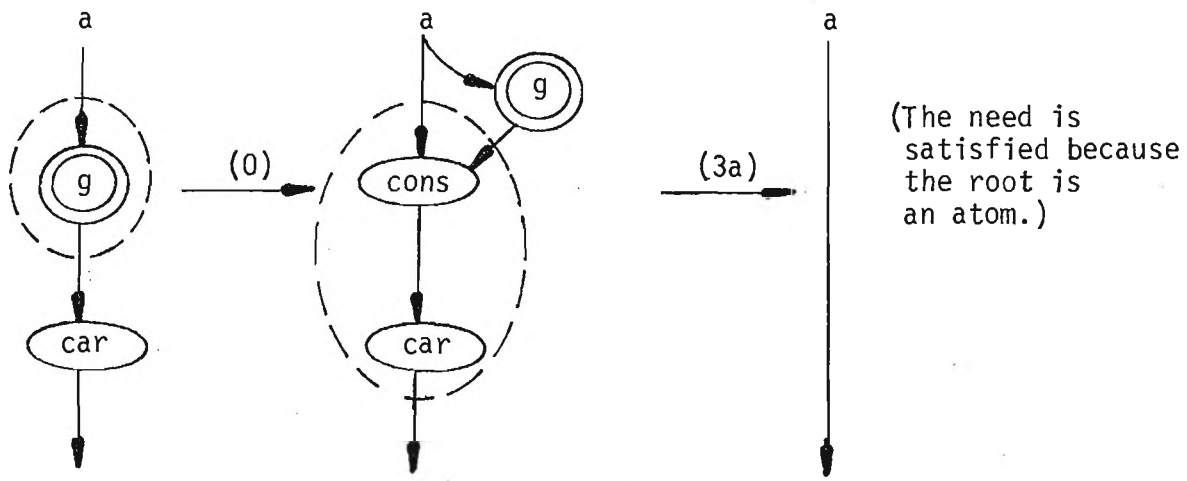
What we need is some way to indicate "need" in the graph. To do this, we must assume that there is an interface to the output which tries to print same by applying car and cdr to discern the tree structure, or by applying the routine printatom to an atom to print its value. Thus a depth-first printing of the tree could be accomplished by

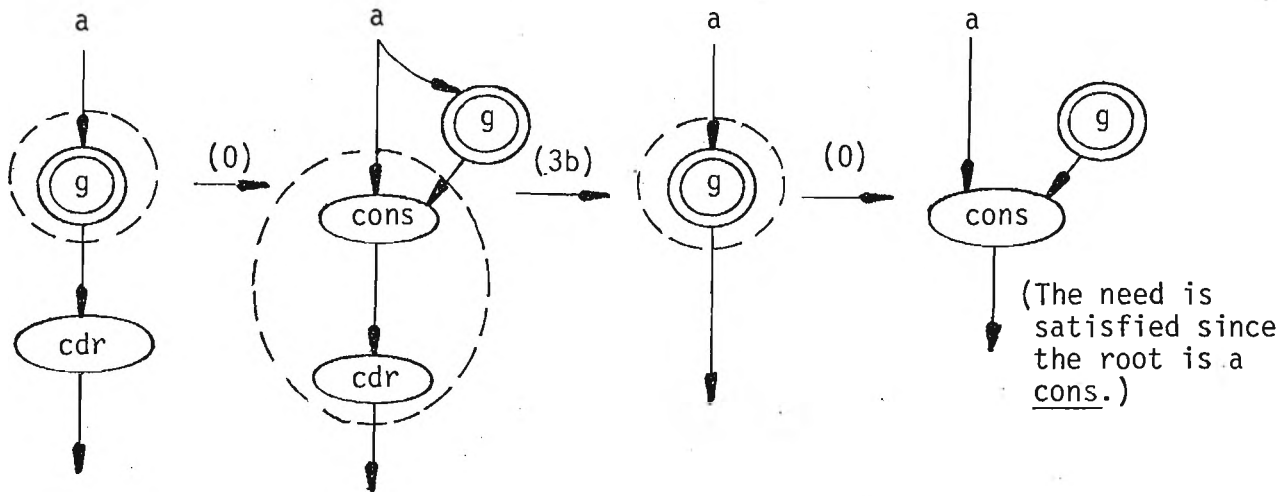
```
print(t):  if atom(t) then printatom(t)
           else print(car(t)); print(cdr(t)) fi
```

On the other hand, we could devise procedures to print the tree breadth-first, to print any leaves below a certain level, etc. All such procedures generate needs at the output arc(s) of the network. Productions are applied to the network as long as needs are unsatisfied. Thus, for the network above

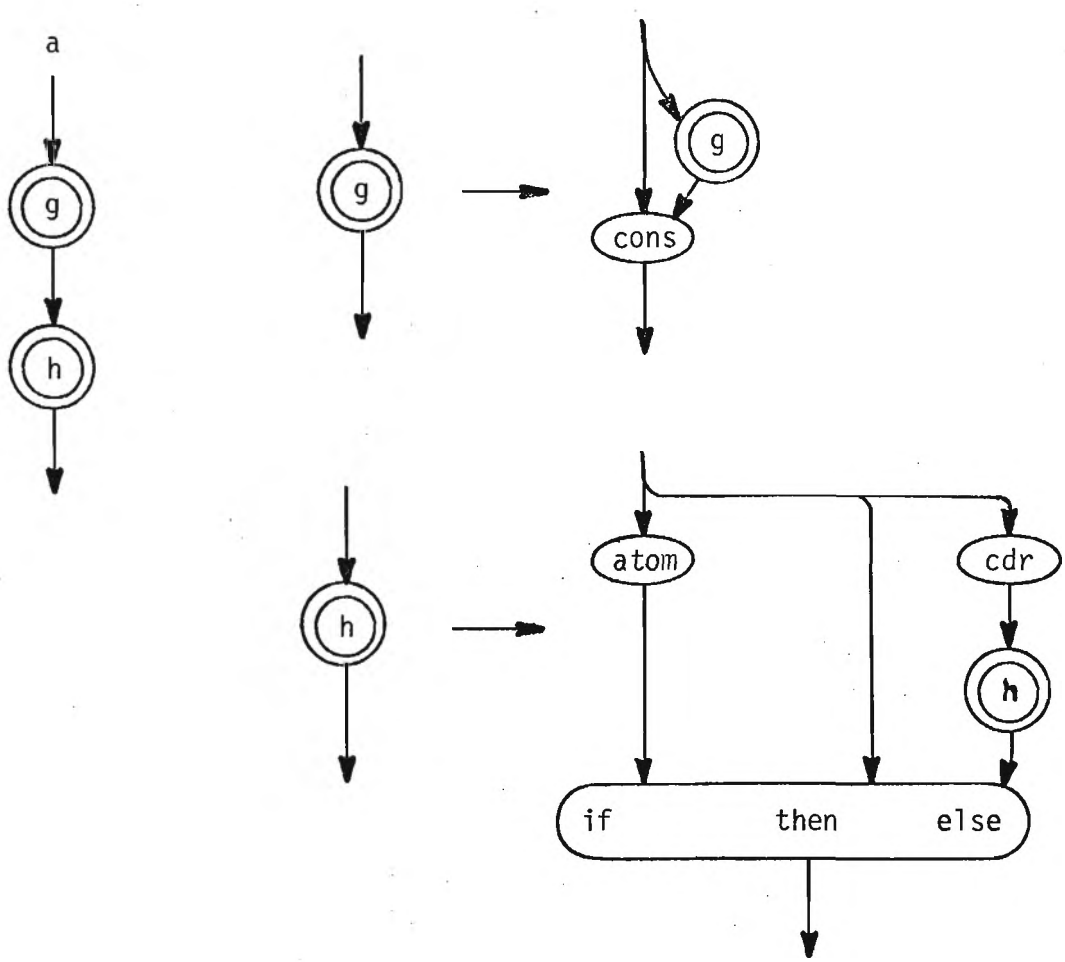
no productions would be applied until a need producing function such as car were applied. This would result in the graph on the left below, which would undergo the productions shown.

We say that the need is satisfied when the root node of the tree becomes either an atom or a cons, the latter indicating that the result is a tree with further sub-trees. Of course, further need can then be generated by applying car or cdr. Two examples are shown below.





We conjecture that this approach eliminates infinite expansions which are unnecessary because of lack of need. However, it does not eliminate infinite computations arising from a need which can never be satisfied, as in the example below. Of course, it is undecidable whether a network has this property.



Implementation

It is not our intention to describe the implementation of our model in detail. From the preceding section, it is clear that an implementation can be designed in which the computation graph itself is maintained in the form of a graph data structure. For the data type of trees discussed, we also noted that the data could be incorporated directly into the graph. In fact, a more extensive data type, that of directed acyclic graphs, could also clearly be so incorporated. An implementation need only select and perform the productions which were exhibited in the preceeding section. Each potential antecedent determines the production uniquely, so there is a choice only regarding where the antecedent is in the graph. As mentioned, it is prudent to choose those antecedents nearest the root, for they have to have been chosen eventually anyway, and by choosing them first, there is a good chance that other parts of the graph will go away. In reality, of course, "going away" means that a sub-graph becomes unreachable from the root, whence the space it uses can be returned to free storage.

We conjecture that the productions given achieve the least fixed point for any given set of input values, in the sense that if we call print on the output, we will "print" the entire output tree (although this may take forever). It is easy to show that the rules are sound, i.e. they preserve the least fixed point. And we conjecture that they are complete, i.e. print will indeed print the entire tree and not get stuck somewhere because of some rule we have omitted.

It seems worthwhile to ponder the relationship between the above proposal and other suggested implementations, namely [Kahn 74], [Henderson and Morris 76], and [Friedman and Wise 76].

Kahn notes that his model is capable of representing a network of sequential processes interacting via streams of data. Each process can execute arbitrary instructions on private data, and selected types of instructions on streams: send a character out on a stream, or wait for the next character in a stream and read it. The analog in our model seems to be instructions which either send pointers to other processes, or wait for selected pointers on input ports. A pointer would in general point to some node in the graph. For example, cons would be a process which waits for each of a pair of pointers, then sends a pointer to this pair as a unit; car would wait for a pointer, which presumably points to a pair of pointers, the left of which would be sent as its result. Notice that we do not require that pointers point to complete objects. To do so would mean that we get weaker semantics as a result.

It is also interesting to note that whereas the merge module (see [Keller 77]) is not implementable using only the send and wait primitives, neither is the parallel if...then...else module implementable by primitives which wait for a fixed set of pointers. Both require some form of "multiplexing". Space does not permit a more precise description of the phenomenon to which we are alluding.

The implementation of [Henderson and Morris 76] is done in terms of a cell memory with pointers. In a sense, it is equivalent to the one proposed here. The contribution we offer is that by viewing the system as a graph, the meaning of the transformations can be understood much more readily. The implementation of [Friedman and Wise 76] is also similar. However, we have no need, as they do, to introduce such notions as "suspensions". Our semantics are presented in a more implementation independent fashion than either of those above.

Conclusion and Miscellaneous Remarks

We have presented a graph model for parallel computation and its semantics. We have also characterized three distinct ways in which parallelism may be introduced in terms of the semantics of the model. Our results form a proper extension of earlier work by Kahn. We cannot claim that our model subsumes those of [Henderson and Morris 76] and [Friedman and Wise 76], for those models also allow lambda expressions and bindings within the data type. However, our model displays inherent parallelism more clearly than do the above and avoids awkward Lisp-like syntax. Furthermore, we have given a precise semantics for the tree data type, separating it from implementation-oriented discussions given previously, and shown the importance of the data-type ordering in relation to parallelism.

Finally, by keeping the definition of semantics sufficiently separated from data type, we can easily incorporate other types of operators, even those with "internal memory".

We have also shown that, given recursion, the cyclic structures of [Kahn 74] are eliminable. (However, it is yet unclear whether this is also the case for indeterminate programs, [Keller 77].)

In view of a long history of discoveries and re-discoveries of related concepts (e.g. [Kleene 52], [McCarthy 60], [Brown 62], [Conway 63], [Zadeh and Desoer 63], [Landin 64], [Landin 65], [Böhm 66], [Karp and Miller 66], [Patil 67], [Constantine 68], [Tesler and Enea 68], [Rodriguez 69], [Patil 70], [Seror 70], [Stoy and Strachey 72], [Milner 73], [Hewitt, et al. 74], [Kahn 74], [Dennis 74], [Ritchie and Thompson 75], [Weng 75], [Scott 76], [Kosinski 76], [Keller 77]), the contribution we claim most worthwhile is yet another viewpoint which will aid in unifying the diverse areas of programming language semantics and parallel computation.

References

[Adams 68] D.A. Adams, A computation model with data flow sequencing. Stanford University, Computer Science Dept., Tech. Rept. CS117 (1968).

[Böhm 66] C. Böhm, The CUCH as a formal description language, in T.B. Steel (ed.) Formal language description languages, North-Holland (1966).

[Brown 62] G. Brown, A new concept in programming, in M. Greenberger (ed.), Management and the computer of the future, Wiley (1962).

[Burge 75] W.H. Burge, Recursive programming techniques. Addison-Wesley (1975).

[Chamberlin 71] D.D. Chamberlin, The single-assignment approach to parallel processing, AFIPS Proc., 263-269 (Fall 1971).

[Conway 63] M.E. Conway, Design of a separable transition-diagram compiler. CACM, 6, 396-408 (1963).

[Constantine 68] L.L. Constantine, Control of sequence and parallelism in modular programs, AFIPS Proc., 409-414 (Spring 1968).

[Dennis 74] J.B. Dennis, First version of a data flow procedure language, in B. Robinet (ed.), Programming Symposium, Lecture Notes in Computer Science, 19, 362-376.

[Friedman and Wise 76] D.P. Friedman and D.S. Wise, CONS should not evaluate its arguments, in Michaelson and Milner (eds.), Automata, Languages, and Programming, 257-284, Edinburgh University Press (1976).

[Henderson and Morris 76] P. Henderson and J.H. Morris, Jr., A lazy evaluator. Proc. Third ACM Conference on Principles of Programming Languages, 95-103 (1976).

[Hewitt, et al. 74] C. Hewitt, et al. Behavioral semantics of non-recursive control structures, in B. Robinet (ed.), Programming Symposium, Lecture Notes in Computer Science, 19, 385-407.

[Kahn 74] G. Kahn, The semantics of a simple language for parallel programming. Proc. IFIP '74, 471-475 (1974).

[Kahn and MacQueen 76] G. Kahn and D. MacQueen, Coroutines and networks of parallel processes. IRIA Rept. 202 (Nov. 1976).

[Karp and Miller 66] R.M. Karp and R.E. Miller, Properties of a model for parallel computations: Determinacy, termination, queueing. SIAM J. Appl. Math, 14, 6, 1390-1411 (Nov. 1966).

[Keller 77] R.M. Keller, Denotational models for parallel programs with indeterminate operators. University of Utah, Dept. of Computer Science, Tech. Rept., 77-103 (1977).

[Kleene 52] S.C. Kleene, Introduction to Metamathematics. Van Nostrand (1952).

[Kosinski 76] P.R. Kosinski, Mathematical semantics and data flow programming. Proc. Third ACM Conference on Principles of Programming Languages, 175-184 (1976).

[Landin 64] P.J. Landin, The mechanical evaluation of expressions, Computer J., 6, 4, 308-320 (Jan. 1964).

[Landin 65] P.J. Landin, A correspondence between Algol 60 and Church's Lambda-Notation, CACM, 8, 2, 89-101 (Feb. 1965) and 8, 3, 158-165 (Mar. 1965).

[Manna 74] Z. Manna, Mathematical theory of computation, McGraw-Hill (1974).

[Manna and McCarthy 70] Z. Manna and J. McCarthy, Properties of programs and partial function logic, Machine Intelligence, 5, 27-37 (1970).

[McCarthy 60] J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, I. CACM, 3, 4, 184-195 (April 1960).

[McCarthy 63] J. McCarthy, Towards a mathematical science of computation. IFIP '62, Proc., 21-28 (1963).

[Paterson and Hewitt 70] M. Paterson and C. Hewitt, Comparative schematology, Rec. Project MAC Conference on Concurrent Systems and Parallel Computation, 119-128 (1970).

[Patil 67] S. Patil, Parallel evaluation of lambda-expressions. Unpublished manuscript (Jan. 1967).

[Patil 70] S. Patil, Closure Properties of interconnections of determinate systems. Proc. Project MAC Conference on Concurrent Systems and Parallel Computation, 107-116 (June 1970).

[Ritchie and Thompson 75] D.M. Ritchie and K. Thompson, The Unix time-sharing system. C. ACM, 17, 7, 365-381 (July 1975).

[Rodriguez 69] J.E. Rodriguez, A graph model for parallel computation. MIT Project MAC Tech. Rept. TR-64 (1969).

[Scott 76] D. Scott, Data types as lattices. SIAM J. Comput., 5, 3, 522-587. (Sept. 1976).

[Seror 70] D. Seror, DCPL: A distributed control programming language. Tech. Rept. UTEC-CSc-70-108, Dept. of Computer Science, University of Utah (Dec. 1970).

[Steele and Sussman 76] G.L. Steele, Jr. and G.J. Sussman, Lambda: The ultimate imperative. MIT AI Lab., Memo. 353 (March 1976).

[Stoy and Strachey 72] J.E. Stoy and C. Strachey, OS6 - An experimental operating system for a small computer. Computer J., 15, 3, 195-203 (1972).

[Tesler and Enea 68] L.G. Tesler and H.J. Enea, A language design for concurrent processes. AFIPS Proc., 403-408 (Spring 1968).

[Vuillemin 73] J.E. Vuillemin, Proof techniques for recursive programs, Stanford Rept. CS-73-393 (Oct. 1973).

[Wadsworth 71] C.P. Wadsworth, Semantics and pragmatics of the lambda-cadenlus. Thesis, University of Oxford (1971).

[Weng 75] K.-S. Weng, Stream-oriented computation in recursive data flow schemes, MIT Laboratory for Computer Science, Tech. Memo. 68 (Oct. 1975).

[Zadeh and Desoer 63] L.A. Zadeh and C.A. Desoer, Linear system theory. McGraw-Hill (1963).