5

Implementation of EDEN Interpreter

There are no strict rules for the behaviour of a definition manager. The evaluation of definitions can be eager, or lazy. The automatic re-evaluation can be enabled, or disabled, data driven or demand driven. No matter how the definition manager internally functions, it must ensure the values of formula variables are up-to-date. Perhaps it is the only rule for the definition manager to follow. I call it the *"Definition Manager's Rule of Thumb"*.

Definition Manager's Rule of Thumb:

Always evaluate a formula when the definition's value is referenced (if the value is not up-to-date).

5.1 Definitions As Functions

The simplest definition manager treats definitions as functions. This method satisfies the Rule of Thumb because we always evaluate the function to get the value. Thus the definition:

```
F is A + B;
```

can be compiled to Pascal code (assuming that F, A and B represent integers):

```
function F: integer;
begin
        F := A + B;
end;
```

In this way, the evaluation of definitions is *"demand-driven"*. This means that the definition is evaluated only if a statement which refers to the value of this definition is executed. This method is all right until we consider the implementation of actions which are procedures that must be automatically executed without a demand (i.e. a request from the user).

In addition, each reference to a definition causes the function to be evaluated again even though there is no change in the values of its source variables since its last evaluation.

5.2 Data Dependency

To tackle the problems described in §5.1, we employ the technique of *"data dependency management"*. The system builds a dependency graph of variables (read/write variables (RWV) or formula variables (FV)) according to the definitions and action specifications. (See §3.1.3)

From this graph, the system can determine whether a new definition is cyclic by searching a chain of dependencies associated with this variable. If such dependencies form a cycle, the definition is rejected. This test ensures that all definitions are non-cyclic.

As in the "definition as function" method, a definition can be compiled into function code (for faster execution) which is stored as part of the formula variable. If a definition has to be evaluated, this code will be executed to update the value of the data register (DR).

A boolean flag, UpToDate variable, is associated with each variable to indicated if the DR is up-to-date. If it is TRUE, the DR contains an up-to-date value of the FV and hence there is no need to evaluate the function code again. Otherwise, the function code is executed; the result of evaluation will be stored in the DR and the flag UpToDate will be set to TRUE. If the flag is FALSE, the variable is not updated or it is undefined (i.e. no value or formula is associated to the variable).

The assignment or re-definition given by the user can make the stored value of a RWV or the value of the DR of a FV obsolete. The scheme of evaluation of definitions can be logically divided into three phases:

Phase 1: Trigger

The UpToDate flags of all FV's, that depends on the variable, are marked FALSE.

Phase 2: Schedule

All marked FV's are sorted into topological order consistent with the acyclic dependency graph.

Phase 3: Evaluation

The definitions of the sorted FV's are evaluated in sequence.

It is sometimes useful to delay the evaluation of a definition until all the values of its source variables are defined and evaluated as discussed in §3.1.4. To implement such a delay of evaluation of definitions, we test whether all the source variables are up-to-date. If all the source variables of a formula variable are up-to-date (i.e. we have enough information to compute the definition), we evaluate it; otherwise we simply ignore it and leave its UpToDate flag marked FALSE.

The EDEN interpreter includes a special variable `autocalc` that can be used for controlling the delay evaluation mechanism. We can delay all evaluations by blocking the evaluation phase. If the value of the RWV `autocalc` is 0, the system skips the evaluation phase. If the value of `autocalc` is set to 1 (causing a trigger), the system schedules and evaluates the marked FV's in the normal fashion, i.e. resumes delayed evaluations.

In §3.3, we have explained that action specifications can be implemented as FV's whose procedure codes are stored in the formula expression (FE) and whose DR's are left unused. Thus the same scheme of evaluation of definitions applies to actions, too. The execution of actions can be delayed in the same way as definitions.

Since the evaluation of definitions can be delayed, it is possible that we reference the value of a definition whose DR is not up-to-date. By the Rule of Thumb, the system must force the evaluation neglecting what the status (e.g. the value of `autocalc`) is. For example, the following definition F indirectly refers to the value of y through the function call g:

```
F is g(1);
func g { return $1 * y; }
y is 2;
```

In this example, after the function g was defined (and before y is defined), F is evaluated (note that the function definition of g does not specify the dependency of g on y, so the scheduler cannot determine the proper order of evaluation). While F is evaluating, y is still marked "not up-to-date". The system must decide how to handle this situation. A sensible way to handle it is to return the value `@` (undefined) for y and continue the evaluation. In this case, the multiplication results in `@` and thus g and F would returns `@`. However, in some cases, e.g. the `?:` (if-then-else) operators, the system cannot handle `@` as an argument and produces a run-time error.

To avoid this, the programmer can either re-arrange the definitions (e.g. interchange the definitions of g and y), or enclose all definitions with the following statements: "`autocalc = 0;`" and "`autocalc = 1;`". In the latter case all evaluations are suppressed until `autocalc` becomes 1; at that time, all definitions are defined.

5.3 Data Template

By modifying the *"data dependency"* technique, we could build a concurrent definition manager. Each variable is a *"data template"* in this method. These templates are connected by communication channels in the same way as the dependency graph; each node in the dependency graph represents a data template and each edge represents a **bi-directional** channel.

The data templates represent variables rather than operators as in some data-flow machines, such as **Lucid**[WA85]. In addition, the data templates do not send their data to other templates unless they receive an enquiry from other templates.

Each data template functions independently. Each template can be implemented as a process or even a processor.

Here we employ the message passing technique which is used in some object-oriented programming languages, such as **SmallTalk**[GR83], for sharing data.

The test for acyclicity can be carried out as described in §5.2 above. The path searching can also be carried out concurrently.

We present a scheme for the concurrent definition manager below:

Phase 1: Trigger

- When the value of a data template (variable) becomes obsolete:
  ¿ sends MARK messages upward.

- When a template receives a MARK message:
  if its UpToDate flag is FALSE, then
  if there are upward templates,
  ¡ passes the MARK message upward,
  else

¬ sends a ACKNOWLEDGE message to the sender;
otherwise, (i.e. already marked obsolete)
√ sends a ACKNOWLEDGE message to the sender.

- When a template receives all ACKNOWLEDGE messages:
  if it has received a MARK message (i.e. not the initial template),
  f sends a ACKNOWLEDGE message to the sender of the MARK message,
  else (i.e. the initial template)
  ≈ sends a EVALUATE message to itself.

Phase 2: Evaluation

- When a template receives a EVALUATE message:
  If there was a EVALUATE message,
  ignores the message
  else
  Δ sends the EVALUATE messages upward.

  Then, it starts computing the current value of the definition (if it is an action, it executes the procedure), and if the value of a variable is required in the evaluation,
  « it sends a QUERY message to that template and waits for the ANSWER message.

- When a template receives a QUERY:
  if UpToDate is TRUE,
  » it sends a ANSWER message, along with the value of DR, to the sender of the QUERY message.
  else
  remember the QUERY message.

- After the evaluation is completed:
  the template stores the result in the DR and sets the UpToDate to TRUE.

  Then if there are some un-answered QUERY messages,
  ... it sends the ANSWER messages, along with the value of the DR, to the senders of these QUERY messages.

Figure 5-1, shows a trace of the management scheme. An assignment or a (re-)definition specified by user causes an initial trigger. As in the "data dependency management" method, the triggers propagate upward. After all related templates are triggerred, the evaluation phase starts (c.f. Figure 5-1-c). Note that the delay of evaluation is achieved by waiting for the answer (c.f. « in the description of the scheme).

Notice that since templates work concurrently, the messages are passed asynchronously.

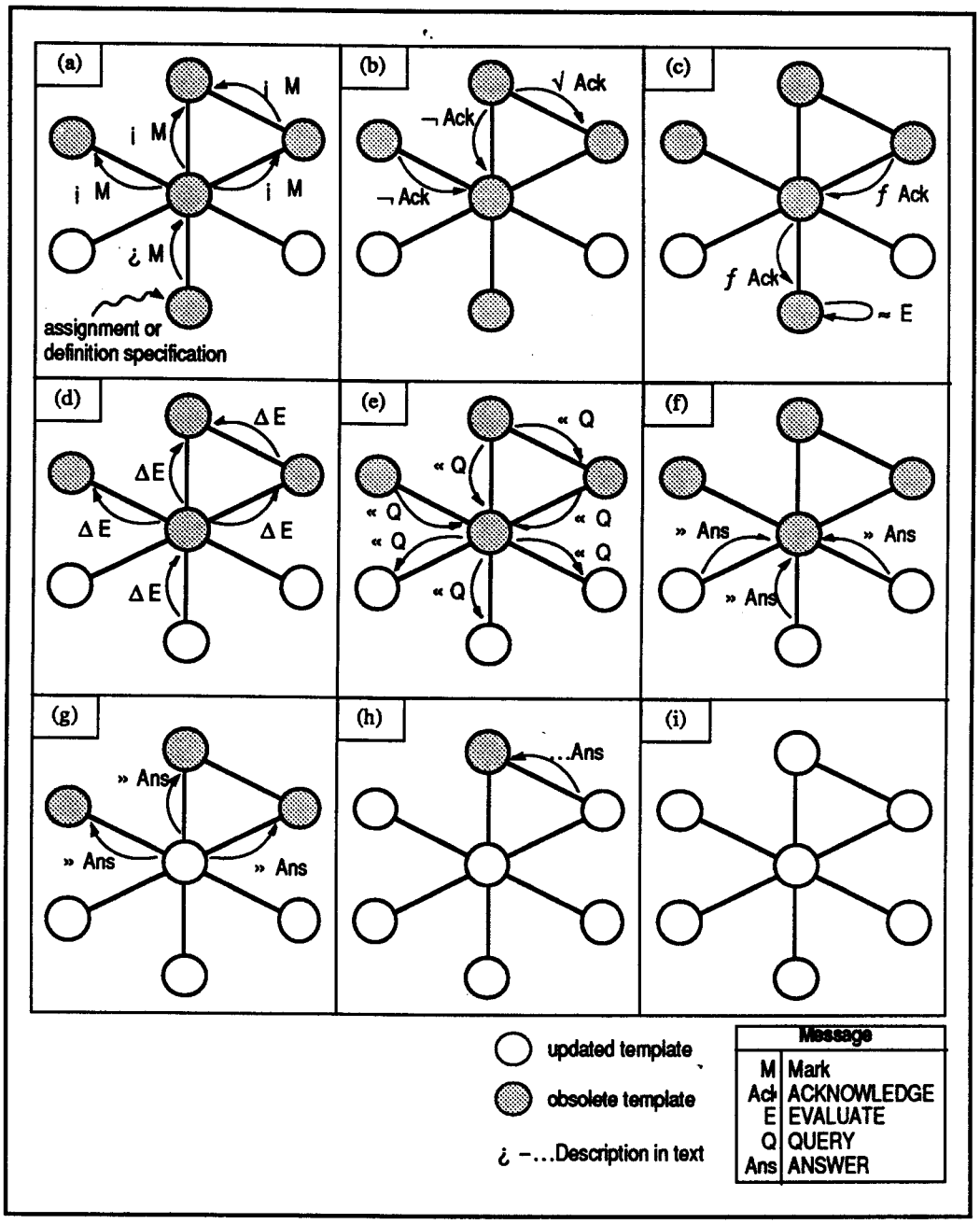


Figure 5-1: A trace of the definition manager scheme

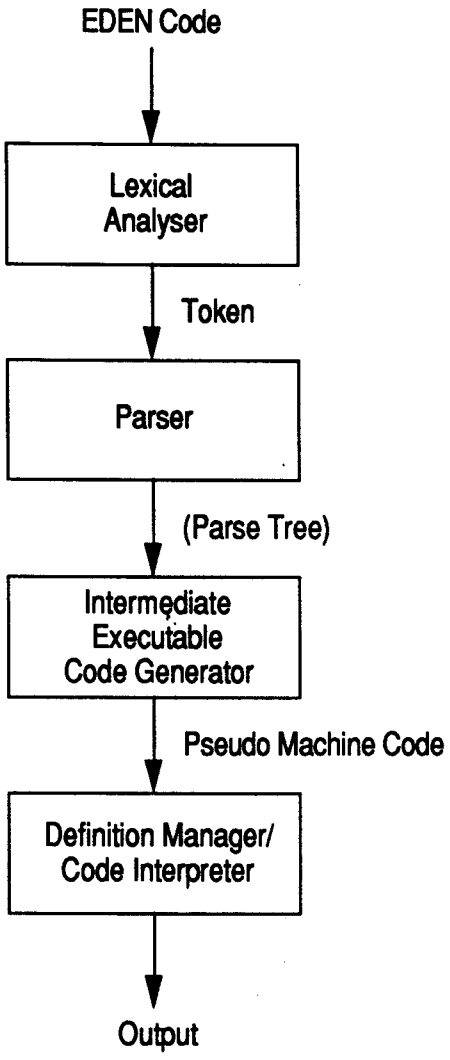
Of course, many details have not been considered here, but the basic scheme is stated above. This scheme is similar to which has been developed in **Coral** [SM88].

5.4 Implementation

The "data dependency management" scheme is simple and feasible on conventional computer architecture. Thus the actual implementation of the EDEN interpreter is based on this scheme.

The rest of this chapter outlines some major components of the EDEN interpreter.

The EDEN interpreter consists of: a parser, an intermediate code generator, and a definition manager/code interpreter.



We have implemented the interpreter on VAX† and on a Sun workstation‡. Both of these machines run the UNIX operating system.

5.4.1 Lexical Analyzer

The lexical analyzer reads in EDEN code and converts it into a sequence of lexical tokens. This stream of tokens is passed to the parser for grammatical checking.

If the input is an identifier, the lexical analyzer looks it up from the symbol table. If no such identifier is found in the table, the lexical analyzer creates an entry in the table, and presumes that this identifier denotes a RWV. Otherwise, the lexical analyzer generates a suitable token according to the matched symbol (e.g. a RWV, a definition, a user-defined function, etc.).

5.4.2 Symbol Table

All the meanings of the identifiers (except the keywords) are stored in the symbol table. An identifier can be the name of a RWV, a formula variable, a function (or procedure), or an action specification.

5.4.3 Parser

The parser analyses the sequence of tokens. If it is a valid EDEN statement, the parser calls the code generator to generate intermediate machine codes. Then it will call the code interpreter to execute the intermediate code.

The interpreter does not have a semantic analyzer because the language is loosely typed. The interpreter checks data type clashes at run-time rather than at compile-time.

5.4.4 Intermediate Executable Code Generator

The intermediate executable code generator is a set of functions which generates some pseudo executable codes. The executable codes can either be executed immediately, or stored (as the formula of a formula variable, the procedure code of an action specification, or so forth) for execution later.

---

† VAX is a trademark of DEC.
‡ Sun workstation is a trademark of SUN Microsystem Ltd.

The pseudo machine is a simple stack-based sequential computer. This pseudo machine has some advanced instructions (e.g. list operations) which distinguish this machine from ordinary machines. In fact, if necessary, we can generate native machine code to enhance the execution speed. However, at this experimental stage, we are quite satisfied with the interpretation speed.

5.4.5 Definition Manager/Code Interpreter

The definition manager/code interpreter executes the pseudo code. It is responsible for the run-time memory management (e.g. memory allocation in string and list operations, stack and heap memory allocation in function calls), and definition management.

The definition manager checks whether a "*new/modify definition*" instruction, for instance, causes a cyclic definition conflict. More importantly, it maintains definitions and invokes action procedures.

Some pseudo instructions activate the definition manager. For example, the assignment operators (=, +=, ++, etc.) make the values of some variables obsolete; hence these operators have to call the manager to maintain the affected definitions (or actions).

5.5 Definition Management Scheme

Although we follow the "data dependency management" scheme briefly described in §5.2, many details can be varied in the actual implementation. These variations may affect the behaviour, especially the order of execution of action procedures, of the interpreter.

The order of execution of action procedures is our major concern because action procedures are intended to produce visual feedback or whatever to the user. The variation in the order will directly affect the appearance and synchronization of the output.

The order of evaluation of definitions is less likely to be a problem because the operators (including user-defined functions) used in a definition are supposed to have no side-effect (or interference). Although the EDEN language does not oblige the user to write functions with no side-effects as functional languages do, the user is advised to write pure functions (i.e. without any side-effect).

The topological sorting algorithm used by the scheduler determines the order in which triggered definitions/actions are evaluated/performed. We implemented two algorithms to study the behaviour of the interpreter.

5.5.1 Depth First Scheme

In this scheme, we traverse the graph in a *depth first* order. That is, we go as deep as possible before we backtrack to other branches. The following algorithm describes the scheme.

Trigger phase:

```
1.    procedure MARK (V)      /* mark variable V is obsolete */
2.        for all U ∈ V.Target
3.            if U.UpToDate == TRUE then
4.                U.UpToDate = FALSE
5.                MARK(U)
6.            end if
7.        end for
```

Evaluation phase:

```
1.    procedure EVALUATE (V)
2.        if (AUTOCALC == TRUE) then

3.            if (not V.UpToDate and  ∧  U.UpToDate ) then
                                      U∈ V.Source
4.                V.value = V.function()        /* evaluate V*/
5.                V.UpToDate = TRUE
6.                for all U ∈ V.Source
7.                    EVALUATE(U)
8.                end for
9.            end if
10.       else
11.           remember V   /* so that we can restore evaluation later */
12.       end if
```

In this scheme, no topological sort is actually performed. The condition test

$\bigwedge_{U \in V.Source} U.UpToDate$ in step 3 of the evaluation phase determines whether a

definition has enough information to compute. If there is not enough information, the definition will simply be ignored since this definition will be encountered again later (after all source variables are updated).

In this scheme, definitions and actions are treated in an analogous fashion; evaluating a definition corresponds to executing an action procedure.

Example: Figure 5-2 shows a dependency graph of some definitions. The trace of the Depth First Scheme appears in Figure 5-3. The values of a, b and c are obsolete initially.

The order of evaluation is:

a d b c j g f e h i k

In this scheme, if the boolean variable *AUTOCALC* is FALSE, a reference to the definitions is saved so that we can resume calculation after *AUTOCALC* becomes TRUE.

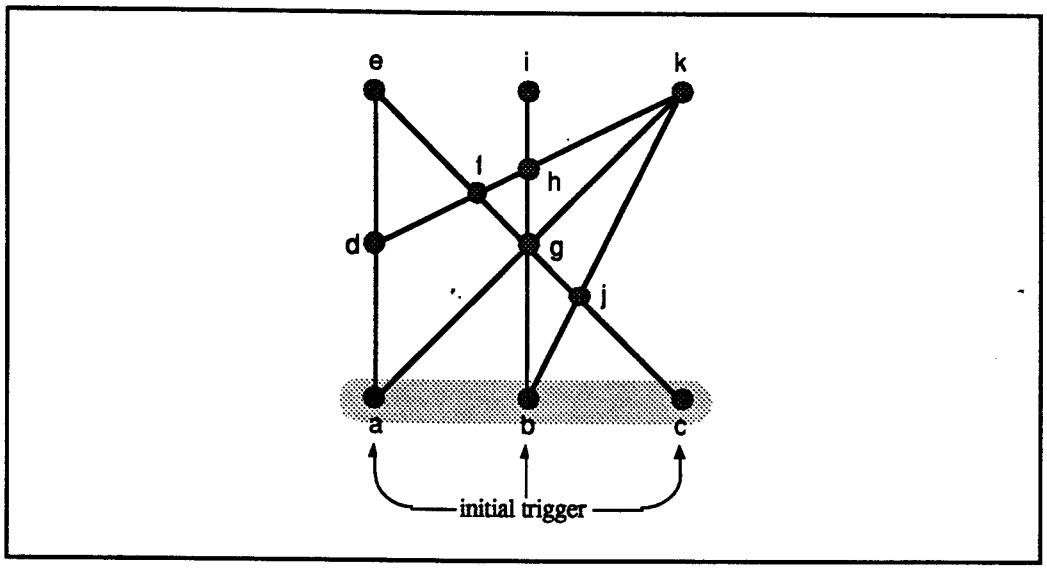


Figure 5-2: The dependency graph of a set of definitions

5.5.2 Breadth First Scheme

Unlike the Depth First Scheme, the Breadth First Scheme uses a non-nested algorithm to schedule the evaluation. The references to the definitions are put into a queue in breadth first order. Repeated entries are discarded except the last entry. After sorting, the definitions are evaluated in sequence.

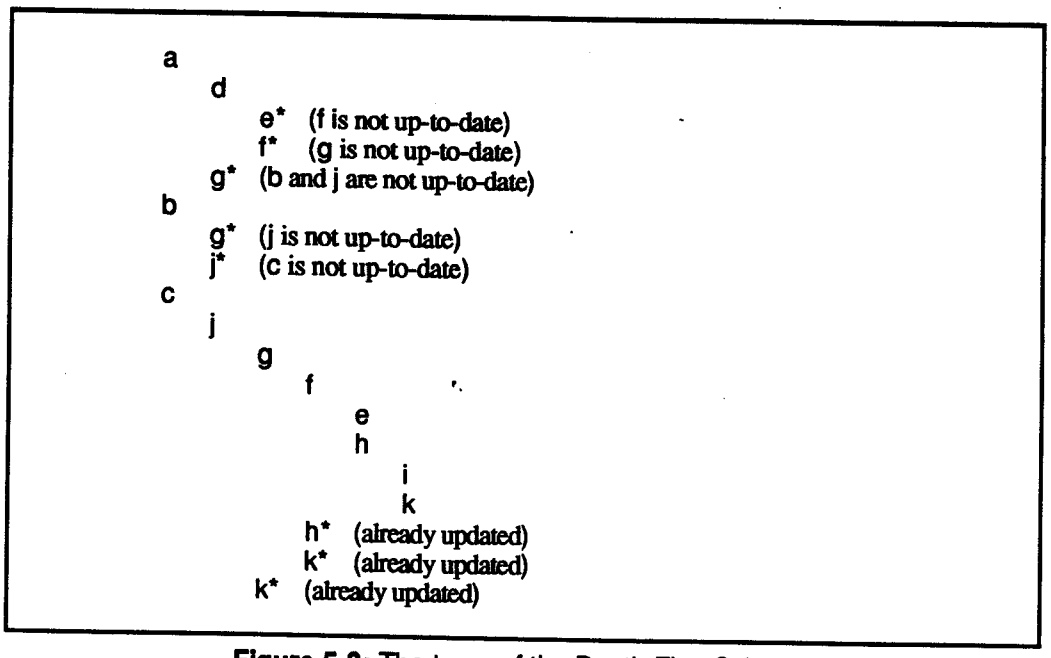


Figure 5-3: The trace of the Depth First Scheme

*The evaluation of those definitions marked with * are ignored since either the information is not sufficient or they are up-to-date.*

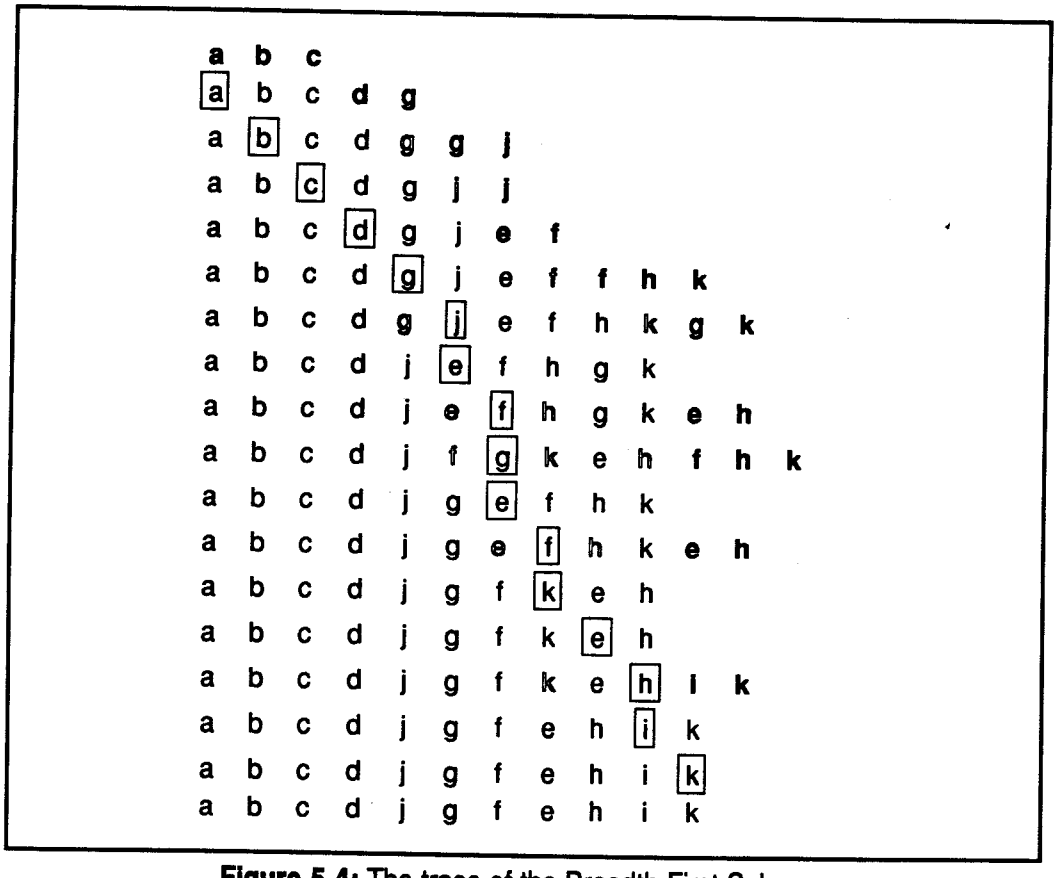


Figure 5-4: The trace of the Breadth First Scheme

The boxed *letter denotes the current definition of concern. The* **bold** *letter denotes the reference that is appended at the end of the queue. The repeated reference that is going to be discarded is printed in* **outline** *typeface.*

Example: Consider the same example in Figure 5-2, the trace of the Breadth First Scheme is shown in Figure 5-4.

In this scheme, we can separate the definitions and actions into two queues — one for the definitions and one for the action specifications. We always evaluate the definitions before executing the action procedures. Thus we can separate the mathematical model and the feed-back module. Also, since we have ordered the evaluations of definitions and executions of actions in a queue, the definitive system is non-interleaving (c.f. §2.2).

5.5.3 Comparison of The Two Schemes

Consider the following EDEN program:

```
func dummy
{
   auto     i;          /* local variable */
   for (i=1; i <= 10; i++)
      V = i;
}

proc Print_V : V
{
   write(V, ' ');
}

F is dummy();
```

The Depth First Scheme produces the output:

```
1 2 3 4 5 6 7 8 9 10
```

since every time the assignment "`V = i`" is executed, the action `Print_V` is executed immediately.

On the other hand, the Breadth First Scheme produces:

```
10
```

because the system must finish evaluating the definition `F` before executing the action `Print_V`. Therefore, when `Print_V` is executed, the value of `V` is `10`.

Thus the Depth First Scheme is good at monitoring the change of values; while the Breadth First Scheme is more efficient in producing the final result.

Another difference between the two schemes is the amount of the run-time memory used. Because the Depth First Scheme uses a nested calling mechanism, it takes more memory than the Breadth First Scheme does.

However, the Depth First Scheme does not have the time-consuming sorting phase; we should expect the execution speed of the Depth First Scheme is better than that of Breadth First Scheme though this has not been tested empirically.