A MODEL FOR FUNCTIONAL REASONING IN DESIGN*

P. Freeman and A. Newell
Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania, U.S.A.

Abstract

A model of the design process is developed in two stages, corresponding to the task environment of design and the activity of posing and solving design problems. Use of the model with top-down and bottom-up disciplines is discussed. An example of the design of an object using a semi-automated design system based on the model is presented. Several issues raised by the model's qualitative aspects, its suitability to automated design, and lines for further development are discussed.

Introduction

We wish to understand "the process of devising artifacts to attain goals," as Herbert Simon (19) recently characterized design. Our interests include the design of programming and computer systems, the intellectual processes that constitute designing, and the actual processes used by humans in designing.

Informal knowledge about design is abundant. Entire professions--engineering, programming and architecture--take design as a central professional aim. However, little reliable information exists about how design is accomplished. Most works, e.g. (3), exposit design methodology to instruct the novice, being based on informal observation of design and participation in its practice, overlaid with attempts to rationalize its methodology. These works provide a useful initial approximation. We learn strategies such as top-down and bottom-up, that careful specification of design goals is a must, and that careful evaluation is equally important. But this provides only a beginning. Of works on the psychology of design, there is even less (though see 7 and 8).

What does exist, in quantity and quality, is work on formalizing design. This occurs both as mathematical models and as computer programs for classes of design tasks. The bulk of this work fits the following constraint formulation:

> Find a point x in a space X such that x satisfies the constraints $C_i(x)$ and maximizes an objective function F(x).

Most mathematical work takes X to be Euclidean, so that a mathematical programming problem of some type is obtained (e.g., 20). Much computer work follows suit, being practical algorithms for solving such problems (e.g., 5). A small amount, mostly experimental work in artificial intelligence, considers more complex spaces, such as all arrangements of a set of objects in a two-dimensional room (9), (11).

The generality and utility of this formulation belies the difficulty of specifying problems in its terms. To formulate completely the design, say, of an operating system appears to be essentially impossible. All aspects of the formulation contribute to the difficulties: defining the space of possibilities; formulating the constraints; obtaining all the constraints in advance; and creating a reasonable objective function. Evidence from Eastman (7) and considerations brought forth by others (e.g., 1) agree in indicating that humans do not design using this approach. Even when the space seems well defined, the constraints emerge continually throughout the design, rather than being available all at once.

Following Eastman's evidence, one could profitably consider relaxing the above constraint-formulation: permitting the space, the objective function or the constraints to change or to become progressively defined throughout a design. Instead, we wish to follow a different clue: the tendency of humans to design in terms of functions.

Reasoning in terms of functions--functional reasoning, as we shall call it--appears to be ubiquitous. We often name things by the functions they provide: a machine for washing clothes is a "washing machine;" a man who sets switches for trains is a "switchman." We give advice in functional terms: "If your man is attacked, defend it." We even write definitions of function terms by using others:

> wash: To cleanse, using water or other liquid, usually with soap detergent, bleach or the like, by immersing, dipping, rubbing or scrubbing (2).

* This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) and is monitored by the Air Force Office of Scientific Research.

Only a modest amount of scientific study has been devoted to functional reasoning. Psychology has had a continuing concern with functional fixity (6), the observed tendency of humans to select objects for consideration in problem solving in terms of their functional descriptions (thus, proving to be poor problem solvers if becoming fixated on inappropriate descriptions). This work mainly demonstrates that humans do indeed reason in terms of functional descriptions.

Among artificial intelligence programs, GPS (10) and related programs (e.g., the Heuristic Compiler, 18) offer the most explicit model of functioning reasoning. GPS differences operate as function terms; they describe a situation in its relation to a goal and operators in relation to how they affect situations. Matching the functional description of a situation's requirements to the functional description of what an operator provides selects out a subset of all possible operators for consideration. Thus, in an action task (transformation of a situation into a desired one) GPS provides at least one paradigm of functional reasoning.*

Nothing indicates that functional reasoning is a total scheme, sufficient to carry out complete solutions. In GPS it only provides one strand of means-ends analysis. GPS also applies actual operators to actual situations and observes the results. Thus, unlike schemes for reasoning that attempt to map problems into a single formalized representation (e.g., those in the Advice Taker tradition (14), (12)), functional reasoning may be only a partial technique that must be combined with others to achieve a complete problem solving system.

This paper is limited to describing and illustrating a model for functional reasoning in design. Evidence for the model must come from studies of human reasoning or from the construction of design systems that incorporate the model. Neither is presented here. A semi-automated design system based on the model is being developed (by PF) and has been used in the main illustration presented later. It will be reported on in a separate publication.

We start by describing the task environment in which design can take place. We first give the simplest possible model and then augment it with various complexities. The next section deals with posing design problems and solving them. Finally, a detailed example is developed.

---

* The success of GPS-like programs in describing human problem solving offers additional evidence for the role of this paradigm (15), (16).

The Basic Model for the Design Task Environment

The basic (or qualitative) model of a design task environment consists of a set of structures and a set of functions such that:

P1: Each structure provides a set of functions.

P2: For each function it provides, a structure requires a set of functions.

P3: A functional connection can occur between two structures if one provides a function required by the other.

P4: A constructed structure consists of a set of structures (its parts) and a set of functional connections between them such that:

1) The functions provided are those provided by the parts that are not consumed in functional connections.

2) The functions required are those required by the parts that are not provided by a functional connection.

We call this the qualitative model because nothing is said about how many structures of a given type may exist or how much of a function is provided or required. These additional stipulations are augmentations to the model, to be discussed in the next section. We illustrate below each of the propositions, drawing most but not all examples from computer systems.

Structures provide functions

The most obvious examples are those where an object has already been named by its functions:

A core memory provides memory.
A store instruction stores information.

The apparent banality of these statements arises purely from the use of functional names as can be seen in these examples:

A drum provides memory.
The BOZ instruction provides a change in program control.

This common usage of functional names for structures does lead to more serious confusion of what are functions and what are structures, however:

> An iteration-code requires an initialization, an iteration-action, a termination-test and an exit.

The iteration-code is a structure--a sequence of instructions. We name it "iteration-code" because all we know about it is that it provides the function of iteration. If we had called it "Sam," there would have been no confusion between structure and function.

Proposition P1 asserts that more than one function can be provided by a single structure:

> The drum can hold user files or resident systems.

> The conditional branch instruction can provide either loop control or switching between two mutually exclusive control paths.

In all these examples there appears to be a single capability--a single function--that underlies the multiple use. A drum provides memory; holding user files and resident systems are simply two subvariates of the memory function. The branch provides for the single function of changing program control; it suffices for either loop control or path splitting.

This singularity, however, need not be:

> A pencil can print characters, punch holes in paper, serve as a core for a ball of string, and tamp down pipe tobacco.

Different aspects of a single object provide the structural basis for each separate function. Other structures can be found that provide each function separately, but not the others (in any reasonable way): In the example above a typewriter, a three-hole punch, a cardboard cylinder, and a pipe tool, respectively.

With designed objects (e.g., a magnetic tape) there is a strong tendency for a single function to be dominant, with all others seen as variant, or even deviant:

> A magnetic tape provides memory or a photogenic visage for public relations.

Inside of a structure the situation is otherwise, and parts are carefully constructed to have multiple function (e.g., general registers in a central processor).

The structures in the examples above are physical objects (drums and pencils) and abstract objects (instructions). Another important class of structures in computer systems are relations:

> Adjacency of cells in memory provides a two-way association between two items of information (their contents).

A structure may provide a given function for any number of objects:

> A single real-time clock provides time-of-day for all user programs.

In general there is no limit on the number of structures that can use a provided function.

Structures require functions

The most obvious examples involve general requirements.

> A physical object requires space.

> A program requires primary memory space.

Typically a structure requires the conjunction of several functions:

> A compiler requires primary memory, secondary memory, input-output, and a processor.

> A generator requires initialization, a way to obtain a candidate from the generator state, a next state, and a termination test.

The second example has another instance of using function terms to name structures. The termination test might seem to be a structure. After all, we know that any test is a piece of code. But this is because we presume a unique structure for the required function of testing for termination. In fact, generator terminations can be provided by a memory protect system that detects out-of-bounds memory accesses.

The converse of one structure providing many functions is that in general many structures can be found to provide a given function:

> Tape, disk, and core provide memory.

> Printers, terminals, and card punches provide output.

The choice of one of the structures providing a function is the central activity of qualitative design.

Proposition P2 is more specific than we have demonstrated. Functions are required by a structure in order to provide specific functions:

> A program requires primary memory space to execute.

This is most clearly seen when a structure has different functional requirements for the

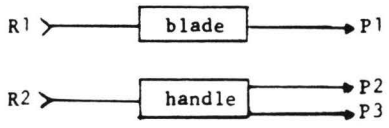
different functions it can provide:

> A compiler provides compiled programs and also source language debugging. It requires a resident symbol table to provide the source language debugging, but not to provide compiled programs.

Construction of structures

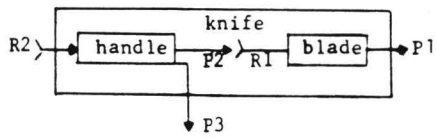
Propositions P3 and P4 describe how structures can be combined into new structures. By choosing a set of structures so that the functional requirements of some are satisfied by others, some functions are left for external usage. Consider a functional description of a knife:

Component structures:



R1: Requires being held.
R2: Requires being held.
P1: Provides cutting.
P2: Provides for holding a narrow object.
P3: Provides being holdable by a human hand.

Constructed structure:



A functional connection exists between the blade's requirement for being held (R1) and the handle's provision of that function (P2). The blade's provision of the cutting function (P1) is not consumed in a connection and is thus provided by the constructed structure; likewise, the handle's requirement of being held (R2) is not satisfied and is thus a functional requirement of the knife.

This example illustrates a basic property of construction: Once a functional connection is established, some of the functions involved may disappear (e.g., the ability of the handle to hold a narrow object). They become internal to the new structure, so to speak.

The functional description of a knife just given is incomplete. More is required to cut than just having a blade and a handle (even with a hand). There must be motion aimed in the right way and with sufficient pressure; the material from cutting must be removed; the blade must be sharp; and so on. A functionally described structure is never taken as complete and always admits of further refinement and modification.

Augmentations of the Model

The model in the previous section was the simplest one that seems to capture the essence of the relations between structure and function in design. The situation can be complicated by various restrictions:

> Define the functional specification of a structure to be the functions provided and required by it along with whatever restrictions apply (as defined below).

P4. A constructed structure must obey the restrictions on its individual subparts after the effects of the functional connections are accounted for.

P5. A structure may be subject to a supply law that limits the number of structures of a given functional specification that are available and/or can be constructed.

P6. A structure may be subject to a capacity law that limits the functions that can be provided.

P7. Functions may be quantified so that an amount of a function may be provided or required (measured in some units); these amounts may be subject to capacity laws.

P8. A structure may be subject to an input-output relation connecting the functions it provides and those it requires (either by amounts for quantified functions or mere existence for unquantified functions--compare compiler example above).

As elaborate restrictions are applied, the problem of design gradually converts from one of purely qualitative specification into one that gives the full details of the structures involved and requires quantitative mathematical treatment.

Supply laws

Often only a limited supply of a given type of structure is available, especially in ad hoc

design efforts:

> In constructing a raft to get across a river, exactly two large felled logs of particular characteristics are available.

These logs can be described in terms of the functions they provide and require, but one must not assume from such descriptions the availability of an indefinite number of logs.

When design is done in a commercial enterprise, supply restrictions often do not exist. Any number of structures of a given type can be used, the restrictions being expressed in terms of the costs of obtaining them.

Capacity laws

The most pervasive form of restriction is on the ability of a structure to provide a function for more than one (or several) structures, or to provide a function given that it is providing another.

> A wall plug may provide any electrical device with power, but only one at a time.
>
> If a conditional branch instruction is used for a loop, it cannot also be used for path-splitting.

Capacity laws are not quantitative, but rather an expression of the logical character of the structure in providing its functions. Almost any pattern of interrelation can exist, but the most usual are:

> Single-function capacity: A structure can provide one and only one of its functions at a time.
>
> Single-structure capacity: A structure can provide a function to one and only one structure at a time.

There are also strong time dependent effects as well as irreversible ones:

> A bomb may provide an explosion only once.
>
> A pawn may be queened only once.

Quantitative functions

Many homogeneous quantities occur in functional descriptions: power, space, memory, processing, channel capacity, energy, time, etc. The amounts of these functions that can be provided are frequently subject to capacity laws:

> A file directory can be kept in primary memory, on the swapping drum, or on the secondary file.
>
> - If it is kept in primary memory, the users will have a smaller job area.
> - If it is kept on the swapping drum, fewer users can be logged in simultaneously.
> - If it is kept on the file, the time to obtain files increases.

The first two consequences derive from the capacity law on the amount of memory that is a form of conservation law (i.e., what is provided to one structure is not available for others). The third arises from the fundamental law that a function (obtaining files) cannot be performed until its required functions (obtaining file addresses) is provided.

Input-output relations

The amount of a function provided can be tied to the amount of the functions required by an input-output relation:

> The number of users logged on depends on the amount of primary memory available.
>
> The amount of cutting depends on the sharpness of the blade.

Summary

Functional reasoning in design does not operate at some particular level in the range of restrictions. Rather, different domains of design (different technologies) require the use of specific types of restrictions. Further, no consistency occurs within a design situation (e.g., if some supply laws are used, it does not follow that there are supply laws on everything). Complexity is added only for the critical structues that need it, other components being left as simple as possible.

We have laid out these augmentations to provide context for the qualitative model and to emphasize that casting the basic model in the simplest of forms was deliberate. This paper will focus on the qualitative model, since it contains the basic notions.

Design with the Model

The model given in the previous sections lays out an environment. Within this a variety of design problems can be posed. The most fundamental one is:

> Given: A set of structures and and their functional specifications.

Construct: A structure with desired functional specifications.

The fundamental problem here is not one of optimization, but one of feasibility--to find a structure that can be composed from the available structures and has the desired properties (expressed functionally). Variations can be generated by insisting that certain structures be used, by asking for minimum structure, by finding all possible designs rather than just one, and so on.

Given a design problem, different methods can be formulated for attempting it. Top-down schemes start with the desired functions and work back toward the structures that are available. Bottom-up schemes start with the structures available, constructing successively larger structures until one is found that has the desired specifications. Most-critical-component-first schemes posit functional specifications of structures that appear (on whatever grounds) to be critical to the final design and then design these first.

These methods all have a combinatorial, heuristic-search character. At each stage of the design a set of possible actions is available to advance the design, one of which must be selected, which then leads to a new situation of partial design. As in other such problems, the set of alternatives is generally large enough so that brute force search (e.g., breadth first) cannot possibly succeed. However, let us formulate the basic bottom-up and top-down methods, since they bring out some important points.

Basic bottom-up method

The elements of the problem space consist of sets of structures (with functional specifications). A new structure can be constructed from any set of existing structures that can form functional connections. The functional specifications for the new structure can be determined via postulates P4 and P4'.

A structure satisfies the desired functional specifications if its provided functions include those desired and if for each desired function the required functions match exactly those desired.* Thus the basic bottom-up method is exactly heuristic search in terms of the operations that are specified in the task environment.

Basic top-down method

Working top-down is not just working backward from the desired functions toward the given structures. It also attempts to bind the design as little as possible at each stage, taking only necessary steps, given the structures actually available, and coming as close as possible to deriving the design.

A typical step in a top-down design might look like:

> Synchronization of the drum requires a buffer memory.
>
> Stipulate a memory that is write-by-word and read-by-bit.

Only a single essential feature of the memory is specified, rather than an actual available memory (e.g., a specific drum). The specification (the stipulation) is structural, for it is not necessary really that memory have this property to produce synchronization. The step is a perfect one in a top-down design if, in fact, for the available structures, all solutions to the design problem will have this structure.

Stipulation can appear to be entirely functional:

> A value of the Bessel function must be obtained.
>
> Stipulate a process that searches a table for it.

Further specification has occurred, for an alternative could have been chosen:

> Stipulate a process that recomputes it on demand.

We recognize the difference between a search and a computation (as intended here), but the distinction is given in functional terms (though lurking in the background are some structural distinctions).

In a top-down design one is continually forced to ask "What structures will provide this function?". The answer one wants in order to delay binding the design is one that makes only very general commitments. Thus, an answer such as

> "CLA instructions and input operations retrieve information."

to the question "How can we obtain information?" is not very useful during most stages of a top-down design. On the other hand, an answer of the form:

> Information can be obtained by knowing it, computing it, or searching for it.

provides an incremental binding of the design

* The asymmetry arises because there is no obligation to use provided functions.

that permits further detailing without overcommitting the designer. Such generalized function-structure laws* appear to be quite important, carrying much of the burden of top-down reasoning.

A top-down design iterates between picking structures to supply functions and generating new functional requirements from the chosen structures. Successful termination of the process occurs when all functional requirements are satisfied by parts of the designed structure (through functional connections) and all desired functions are provided. This corresponds to a backwards working heuristic search procedure.

The result of a successful design is a constructed structure whose components are either given structures or are reducible (recursively) thereto. Such a structure may still be distant from a real physical object that will perform as desired. To have designed a knife as a blade and a handle, or even a blade, handle and fastener (of blade to handle), is not yet to have a knife. One might insist that a design task is not well posed unless the given structures correspond to actual physical objects. Successively more detailed restrictions could be added to the functions, until an adequate physical model of the situation would be implicit in the functional description. Reasoning would more and more involve the details of these restrictions and less and less the matching of functions required to functions provided, which is the heart of functional reasoning. Actually, it appears impossible (even if it were desirable) that an elaboration of functions into finer and finer categories could suffice for expressing the intricacies of actual physical technologies (e.g., metallurgy or polymer chemistry). However, such an elaboration might prove successful for extremely artificial and discrete technologies, such as computer programming.

The result, then, of a functional design remains a plan for the realization of an actual object. To implement the plan requires enlarging the reasoning to include working with the actual physical structures or with symbolic models of them, observing the effects of manipulation, correcting the design, testing it, and so on. This is in agreement with the paradigm embedded in GPS; and we shall not explore it further here.

* The example given here is actually a function-function law, of a type that further elaborates a function term. There is a similar type of relation that deals with the inclusion of one function in another (e.g., hi-speed, low-speed, and multiplexed transmission of information are all transmissions of information). For simplicity, the initial version of the model does not deal with this issue. A more complete version will deal with direct relationships between functions (i.e. with no intervening structure) since they are an important part of design.

An Example of Qualitative Design: A Symbol Table

We will not attempt an example of a complete design, involving the application of a design method to a task environment to obtain a solution. Although this must be done to explore the problem solving aspects of an automated design system, that seems less crucial initially than exploring the framework itself. We will attempt an example that provides:

- an explicit rendition of a design task environment;
- the posing of a real design problem in this environment;
- the description of a path through the design space that would have resulted in the solution given;
- the exhibition of a functional assignment of structures that satisfies the posed design problem;
- an indication of how a complete physical specification could be obtained;
- an indication of how alternate choices in the design could have led to other solutions.

We chose as the example the design of a two-way symbol table for a programming language. This is a common programming structure, and many standard solutions to it exist: sequential search, binary search, logorithmic or tree search, and hash addressing. It is a fairly simple, though real, design task and permits us to stay within the confines of the paper and manual analysis.

No extensive discussion exists in the literature that explores thoroughly the design of symbol tables, describing trade-offs in terms of the possible properties. There does not even exist, so far as we know, a complete formulation of the design problem, say in terms of a space of possibilities, the constraints and the objective function. Our example is not meant to fill any of these gaps, but only to illustrate our model of functional reasoning in design.

The design task environment

The basic class of structures to be used in the design, called programming structures, consists of collections of data-structures with

programs that operate on them. Programming structures, by the execution of one or more of their programs, provide (possibly many) functions to other programming structures in which they are embedded.

We adopt two general conventions of design practice, which are substantially honored throughout the programming world:

> Principle of separated functions. There exists a distinct program that provides each of the functions provided by a programming structure.
>
> Principle of uniform control. There exists a programming language (including therein the notion of a set of programming conventions) that for all programming structures provides the functions of:
>
> (1) Designating the program to be executed at each moment.
>
> (2) Designating the operands (data-structures) for a program.
>
> (3) Communicating the operands to a program to be executed.
>
> (4) Communicating the results of an executed program.
>
> (5) Loading programs and their operands into space required for their operation.

The first principle is simply that one designs a separate subroutine for each kind of thing one can do with a data structure, rather than evoking the actions in various linked and contingent combinations. It does not imply that only such programs should exist; the usual programming structure has many subroutines that are internal to it. The second principle is simply to have uniform conventions for calling routines, passing parameters, providing space in primary memory, etc. Both principles cover functions that must be provided for any programming structure to operate. They permit the design of a particular programming structure to attend exclusively to the processing required to achieve the specific functions desired.

Both program and data are structures and lead to a kind of dual functional description. We can describe programs, giving the function performed by each subroutine (ultimately, each instruction). Such a functional description will be complete: the functioning of each of the instructions is the necessary and sufficient condition for the programming structure to provide specified functions. The data does not seem to enter in. Equally, we can describe data structures, giving the functions of the contents of each subpart and their relations in address space. Such a description will be almost complete, for almost every instruction in the program will operate only by the grace of some aspect of the data.* Thus, a functional description of a programming structure will appear to present everything twice. We recognize this by taking the function of data as permitting various processing functions to be provided by particular programs (ultimately, instructions). For example, an address relationship may "permit computation" of some result.

Figure 1 shows the actual structures available for our example: cells, addresses, the relation of contents-of, and sets of instructions. We write programs in a simple Algol-like notation. Figure 2 illustrates the functions provided by the program structures and those permitted by the data structures. The functional requirements of the structures are not shown here since they are well-known to all programmers.

Figure 3 shows the code and data structure for the simple symbol table whose design we shall illustrate. It consists of a one-dimensional array with each entry in the symbol table taking two adjacent cells. The external symbol is entered in the first cell of a pair, the internal address in the second. There are three routines: R1 finds the address associated with an external symbol; R2 finds the symbol associated with an internal address; and R3 enters a new symbol-address pair into the table. For the accessing operations a search is made of the table, testing the first word of a pair if the symbol is given or the second word of the pair if the address is given. If a match is found, then the other member of the pair is returned as the result. To establish a new pair, it is simply loaded into the table at the high end.

Several simplifications have been introduced. We ignore all failure conditions, either that a requested entry is not in the table or that the table is too full to receive a new entry. We do not provide for the removal or modification of existing entries so that R3 becomes much simplified. These simplifications are made to keep the example within bounds.

It is necessary to impose a system of function terms. Figure 4** defines the functional

* The failure of completeness derives from programs accomplishing some things without resort to a data structure.

**The material in Figures 4, 5 and 6 is output from XDA, a semi-automated design system based on our model. The system is being built (by PF) and used on the PDP-10; it is constructed on L*(F) -- a kernel system-building system.

provisions of the structures to be used in the design example. Referring to Table 1 for an explanation of the notation used by XDA, Figure 4 tells us that the function OBTAIN (with internal name F1) can be provided by three different structures: a KNOW STRUCTURE, a COMPUTE STRUCTURE, or a SEARCH STRUCTURE (with internal names S1, S2, and S3). Additionally, S1 (the KNOW STRUCTURE) is marked PRIMITIVE, meaning that for the present we need not worry about its functional requirements.

Although the function terms listed in Figure 4 (OBTAIN, KNOW, COMPUTE, etc.) are meaningful to the reader, their role in this design must depend entirely on their functional specifications. These consist of the function provision laws given in Figure 4 and the functional requirements given in Figure 5. The first line of Figure 5 states that S2 requires the function F3, lines 4 and 5 state that S5 requires F3 and F9, etc. (The full form of a name (e.g. SEARCH STRUCTURE) or its internal name (e.g., S3) can be used interchangeably).

The system of functional description that this scheme leads to is quite simple and reasonably abstract compared to the actual programming structures of Figure 1. Nonetheless, it is adequate for the purposes of the example and serves to define a small but complete design task environment.

The design problem

We can pose the problem of designing a symbol table as follows:

Given: The structures available in the design task environment.

Construct: A structure (called a symbol table) that provides for:

(1) Obtaining the internal address associated with a presented external symbol.

(2) Obtaining the external symbol associated with a presented internal address.

(3) Associating a presented external symbol and internal address.

In accordance with our earlier remarks, we have removed other functions normally associated with a symbol table.

The problem is cast in terms of a set of three functions to be provided and none to be required. From the principles stated earlier, we can take it that there will be a program for each of the three functions, and that the only design problem is what is the nature of these three programs and the data structures on which they work. The issue of how an external symbol and an internal address are presented is not of concern, being part of the surrounding programming system.

The design task environment presented is not at all a special task environment for symbol tables. The structures and functions provided are general. With a few additions, a complete order code could be built up and we could then propose other problems to be solved in the same environment.

Design path

As already discussed, one can follow a variety of design strategies while employing functional reasoning to go from a design goal to a complete design. Although the strategy used can be very important, our purpose here is to illustrate the reasoning and its product -- not the pattern it follows. Consequently, no particular significance should be attached to the order in which the following design unfolds.

XDA was used to go through an essentially complete sequence that achieves a design for the object specified above within the design task environment given. Figure 6 shows an initial portion of the design trace output of XDA as it was used to develop this example. Using the notation of Table 1 and noting that the designer's input is underlined to distinguish it from XDA's output, we see that first the structure to be designed is defined by stating the functions it is to provide. This structure and its required functions are placed in nodes of the design representation (D1, D2, etc.). The designer is then given a choice as to which part of the design he wants to work on and a set of possible structures are presented to him for his choice.* He may stipulate a structure to supply the function in question at this point in the design (other structures might be used to supply the same function elsewhere in the design). Then its functional requirements are entered into the design and the cycle repeats. This proceeds (assuming the designer does not alter the prompting sequence of XDA) until a primitive structure is stipulated to provide a needed function, terminating that branch of the design.

Figure 7 portrays graphically the design path that was followed in the complete design trace of XDA (the path that produced the design

*The designer may make a variety of responses, besides those suggested by the system, that permit him complete freedom in choosing a design sequence. Additionally, the design may be backed up, new functions and structures not in the data base may be defined and used in the design, and the system can be requested to perform more complicated searches for structures supplying a given function.

of routine R1 is omitted since it is exactly analogous to the design of R2 -- the left-most branch of the tree). The time order of the trace can be reproduced by following the branches depth-first in left to right order (a top-down strategy was used for simplicity). The tokens in the nodes are the shorthand names of functions and structures and the node is labeled with its name from XDA's trace. A function node below a structure node indicates the function is required by the structure; a structure node below a function node indicates the structure can provide the function. If a structure node is a triangle, then the structure was considered but not stipulated for the design. If a node is square, the structure is terminal and a name of the form Si/(Di) is attached to indicate the actual structure that was stipulated for the design, as shown in Figures 6 and 8.

Starting at the top we note that the symbol table (S16) has three functional requirements (F24, F25 and F26) that are the three functions it is to provide. To follow the design of the retrieval mechanism for (F24) we move down the left subpath. F24 has three types of structures (S1, S2 and S3) that can provide it. Since the path goes through S3 that means we chose to use a search structure (S3) in the design.

From XDA's data base (built up in Figures 4 and 5) we learn that a search structure (S3) has only one functional requirement, search (F4), and that two structures (S7 and S8) can provide the function of search. We chose a generator searcher (S7). It in turn has three functional requirements (F11, F12 and F13). S8 was considered as a candidate to provide F4 but was not chosen. The design trace in Figure 6 shows this sequence more clearly.

The path continues in this way, splitting into subpaths each time two or more functional requirements appear for a single structure. At the terminal nodes we have designated particular pieces of primitive structure (for this design) that go to make up the designed object.

A Solution of the Design Problem

Figure 3 shows a completed symbol table (data structures plus operators) that is an informal solution. Any programmer would verify that it performs the required functions. In terms of the model, however, it is not a complete solution for we have not demonstrated what structures provide what functions and what structures permit other structures (e.g., programs) to operate as needed.

Figure 8 presents the completed design by associating the terminal structures from the design trace with the actual structures used in the solution presented in Figure 3. The first column has the code names of the terminal structures. The second column presents the functions provided by the structures as developed in the design trace; it consists of all functional requirements above the structure in the design path up to and including the first function that is a member of a set of two or more functional requirements of a structure. The third column is the informal definition of the structure given in the design trace and the last column shows the corresponding structure used in Figure 3. As noted above, the data structures play a role of permitting various functions by the program structures and thus do not appear in Figure 8.

Full Design

The disparity between the functional descriptions of the terminal objects and the actual structures used illustrates the point made above that the output of a functional reasoning design process is essentially a plan requiring further implementation. The structures used are not only some distance from the structures specified but some critical elements are missing and must be supplied by some process to effect an implementation. To wit, the control structures that tie together the various pieces of program and the exact constraints on the data structures (e.g., size of cell, adjacency) are not present.

The distance between the final result of the functional reasoning in the example and a physical specification of the constructed object does not seem to be an unbridgeable chasm. Aside from the fact that many important designs (e.g., flowcharts and blueprints) leave out many important details (the use of language constructions and the principles of carpentry), there are at least two possible courses of action.

One could try to carry the functional reasoning process down further to obtain more concrete specifications. Analysis of the requirements of individual instructions in terms of their need for operands and/or adjacent instructions permits a more detailed specification. Likewise, data structures could be analyzed in terms of relations between their parts. Even though this approach could carry the design further than our example goes, it is not clear it could do the entire job.

The other approach that bears investigation is to consider the output of functional reasoning to be a plan that is input to a heuristic compiler. Matching on the functional descriptions of instructions and data structures would be used to implement the plan. Clearly, such a procedure would have to have available operators for manipulating and testing the physical structures so assembled. The details have not been worked out, but Figure 8 clearly presents a set of well-defined problems of the general form "Build a structure out of

the available physical structures (as in Figure 1) that performs this well-defined function", which would be amenable to heuristic compiler techniques.

Alternative Designs

Figure 7 plainly shows that only one of many possible paths has been chosen. Most, of course, lead to no design at all. For example, if the association function had been designed as a hashing function and the retrieval operations as we now have them, we clearly would not have a symbol table.

One common type of symbol table employs a variant of binary search in which an inspection of a candidate indicates a narrowed range from which to obtain the next candidate. This is what we have called a guided searcher (S8). Thus, by choosing S8 instead of the generator searcher (S7) at design node D7, we would have obtained a symbol table of that design (assuming we made appropriate changes in the association function as well).

To obtain a hash storage scheme instead of the relational storage we used, we could have picked the structure of name location storage (S15) at design node D40, this being the kind of structure that computes the address of a storage cell from the name of the presented item. Then in the retrieval operation we would have had to choose a compute structure (S2) at design node D5 in order to compute directly the location of the target object.

Discussion

We have now presented a model of functional reasoning in design and illustrated at least the central tenets of it. A number of aspects have been left dangling or received no attention at all. They can receive no adequate treatment here, but we will attempt to state some of them briefly.

Relation to constraint-formulation

We took note of the constraint-formulation at the beginning of the paper, because it appears to be the form towards which design problems tend as they become formalized. There is more than one way to look at the relation of functional reasoning to the constraint-formulation. One view takes functional reasoning as a special subspecies of the constraint-formulation. The kinds of constraints are simple logical ones, saying that connections of various kinds must exist. As we move toward the various restrictive laws (supplies, capacities, etc.) a few additional simple constraints can be handled (e.g., these laws are mostly expressible as constant or bounded sums, as in linear programming). But no really intricate constraints can be handled. Thus, functional reasoning is sort of a poor man's non-technical constraint satisfaction scheme. It should be replaced by more adequate formulations wherever possible.

An alternative view sees functional reasoning as a planning scheme to be used in connection with more accurate procedures. This is the view implicit in our treatment of the example, where we carried the functional description only to a certain detail and then used a more precise formulation.

On the initial structuring of designs

One of the peculiarities of many design problems is that they create structure out of nothing, so to speak. They appear in this sense to be open problems. It can often be observed in human design that a structure is placed on the design problem within a few minutes, or even fractions of a minute, of obtaining the problem. Pure functional reasoning seems to be a plausible candidate for the mechanism whereby this initial creation of structure occurs.

The functionally described structures are not unlike the kinds of descriptions of structure people seem to have initially.* The reasoning involved, which is highly associational (bouncing back and forth between functions and structures using function terms as the linkages), is well suited for rapid reasoning which could put together new structures never before known to the designer. This role for functional reasoning in design is consistent with viewing it as a planning, initial approximation instrument. This aspect also emphasizes the qualitative model, rather than models with substantial restrictions added which make reasoning more complex.

Large memory structure

It goes almost without saying (though we have not said it yet) that real designs require a large memory of structures and functional laws between them. This point has been urged in connection with almost every attempt to reason about the real world, and this paper offers no new evidence for it. The qualitative model can be viewed as a sort of retrieval net for indexing and organizing a large memory, and much of its power should only become apparent in such contexts.

The relation to predicate calculus models

We do not yet understand the relationship of this model of functional reasoning to attempts to formulate problems in a formal calculus (12), (13), (17). It appears that all

*Sometimes they also have a highly specific and elaborate structure clearly evoked from having already known it.

reasoning schemes rely ultimately on mechanisms for matching expressions and instantiations of forms -- for that is what is available in information processing systems. Thus all systems have a sort of brotherhood under the surface. The formal calculi offer great precision and thus appear to be modelling the structure of situations. But when they are applied to non-formal situations (i.e., not to already formalized areas such as group theory, lattice theory, or simple puzzles) the toy models that are constructed (i.e., the baby axiomatizations) are sufficiently gross caricatures of reality that they may in fact be nothing but a vehicle for the sort of functional reasoning discussed here.

Universal non-model

The uniformity of functional reasoning across all domains prompts the conjecture that it is a sort of non-model* of each particular domain. That is, it is a scheme of reasoning that is adapted to the needs of the reasoner, not to the details of the domain. It is applied universally to all domains. Whatever is picked up is reflected in the reasoner's problem solving; whatever is too intricate is lost. The major degree of freedom available to make the model adaptive to a particular domain is the choice of the function terms that are to be applied in that domain. The set of functional terms appears not to be derivable from the structural domain, so that they constitute an importation or construction for a domain. For instance, they can reflect past experience with solving problems in that domain, so that a particular set of function terms serves, in part, as a memory of past solutions.

Systems of functional description

Function terms do not generally occur in isolation. They form systems for a given domain. For example, GPS has a set for logic: add and delete terms; increase and decrease numbers of terms; change signs, connective, position, and grouping. These cover the domain: if no such term applies, then the given situation is already the desired one. Relations other than partitioning hold between function terms (e.g., inclusion). It is clear that the efficacy of a scheme of functional reasoning depends on the set of terms chosen and their relations to each other (e.g., see GPS on the Tower of Hanoi (10)). The importance of the nature of the descriptive system available on a problem space has been emphasized by others, most notably by Banerji (4). It deserves extensive treatment.

*It is, of course, a model of any domain it is used for. Our use of "non-model" is to emphasize its lack of responsiveness to the details of any particular domain.

Conclusion

This paper provides an initial attempt to set out a model of a specific type of reasoning. It does not capture all that happens when humans design using function terms. But only by attempting an explicit model for some of the more obvious aspects of functional reasoning, will it be possible to discover the additional phenomena that exist.

References

1. Alexander, C. Notes on the Synthesis of Form, Harvard University Press, 1964.

2. American Heritage Dictionary, 1969.

3. Asimow, M. Introduction to Design, Prentice-Hall, 1962.

4. Banerji, R. B. Theory of Problem Solving: An Approach to Artificial Intelligence, American Elsevier Publishing Company, 1969.

5. Dejka, W. and D. C. McCall. "A Study in the Design of a Practicable Timeable Bandpass Filter Using Mathematical Programming," Proc. of the IEEE 1970 Systems Science and Cybernetics Conference, p. 267.

6. Duncker, K. "On Problem-Solving," Psychology Monographs, 58, 5, 1945.

7. Eastman, C. M. "Explorations of the Cognitive Processes in Design." Computer Science Department, Carnegie-Mellon University, AD671158, 1968.

8. Eastman, C. M. "Cognitive Processes and Ill-defined Problems: A Case Study from Design." Proc. of First Joint International Conference on Artificial Intelligence, Washington, D. C., 1969.

9. Eastman, C. M. "Problem solving strategies in Design." EDRA 1: Proceedings of the Environmental Design Research Association Conference, H. Sanoff and S. Cohn (eds.) Chapel Hill, N. C., June 1969. School for Design, North Carolina State University, 1970.

10. Ernst, C. W. and A. Newell. GPS: A Case Study in Generality and Problem Solving, Academic Press, 1969.

11. Grason, J. "Methods for the Computer-Implemented Solution of a Class of 'Floor Plan' Design Problems," Ph.D. Thesis, Carnegie-Mellon University, AD 717756, 1970.

12. Green, C. "Application of Theorem Proving to Problem Solving," Proc. of First International Joint Conference on Artificial Intelligence, Washington, D. C., 1969.

13. McCarthy, J. "Programs with Common Sense," in Semantic Information Processing, M. Minsky (ed.), MIT Press, 1968.

14. McCarthy, J. and P. J. Hayes. "Some Philosophical Problems From the Standpoint of Artificial Intelligence," in Machine Intelligence 4, B. Meltzer and D. Michie (eds.), American Elsevier Publishing Co., Inc., 1969.

15. Newell, A. and H. A. Simon. "Computers in Psychology," in Handbook of Mathematical Psychology, vol. 1, John Wiley, pp. 361-428, 1963.

16. Newell, A. and H. A. Simon. Human Problem Solving, Prentice-Hall, in press, 1971.

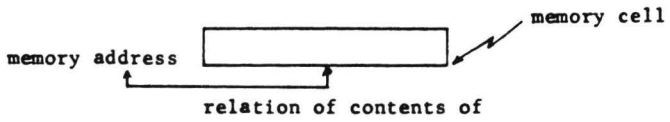
17. Pople, H. E. Jr. A Goal Oriented Language for the Computer, Ph.D. Thesis, Carnegie-Mellon University, 1969.

18. Simon, H. A. "Experiments with a Heuristic Compiler," JACM 10,4, October, 1963.

19. Simon, H. A. The Sciences of the Artificial, MIT Press, 1969.

20. Wilde, D. J. and C. S. Beightler. Foundations of Optimization, Prentice-Hall, 1967.

A set of memory cells of the form:



An ALGOL-like programming language:

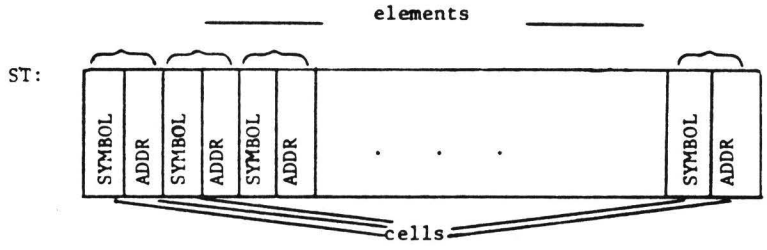
- identifiers
- assignment operator
- simple arithmetic expressions
- arrays
- special procedure test(A,B) returns true if A=B
- simple conditional: iftrue
- goto, labels, declarations, delimiters as needed

Note: Addresses are positive integers and symbols are sequences of characters.

Figure 1: Structures Available for Building Symbol Table

set of cells	PERMITS	generate
memory address	"	recognize
memory cell	"	store
cell contents	"	recognize
contents-of relation	"	obtain
cell adjacency relation	"	retain
symbol	"	recognize
identifier	"	obtain
assignment operator	PROVIDES	store
arithmetic expression	"	obtain
array	"	retain
test(A,B)	"	recognize
variables	"	know
program structure	"	know

Figure 2: Functions Provided by Programming Structures




```
procedure R1(symbol, address); array ST;
     i := 0;
     R1.1: test(symbol, ST[i]);
     iftrue goto R1.2;
     i := i + 2;
     goto R1.1;
     R1.2: address := ST[i + 1]; end

procedure R2(address, symbol); array ST;
     i := 1;
     R2.1: test(address, ST[i]);
     iftrue goto R2.2;
     i  := i + 2;
     goto R2.1;
     R2.2: symbol := ST[i - 1]; end

procedure R3(symbol, address); array ST;
     top := top +2;
     ST[top] := symbol;
     ST[top + 1]  := address; end
```

Figure 3: A Completed Symbol Table

Structure and Function Names:

CELL, ELEMENT ADDRESS, OBTAIN CANDIDATE, SEARCH FOR IDENTITY, ...

S1, S2, ... F1, F2, ... used internally and for shorthand (assigned automatically by XDA the first time used)

Design Nodes:

D1, D2, D3, ...

Operators:

%PRIMITIVE%	Mark the preceding structure as not needing any functional specification
,	List separator.
&	List separator.
*	Input statement terminator.
si:	A new structure with name si follows.
fi:	A new function with name fi follows.
Si: A B C	Si is the internal name of the structure A B C.
Fi: D E	Fi is the internal name of function D E.
A --> B	A provides B.
A <-- B	A is provided by B.
A > B	A requires B.
A << B	A is required by B.
A == B	Design node A is specified by B.
A = B	Design terminal A is specified by B.

Table 1: Notation Used by XDA System

```
f1: OBTAIN <-- s1: KNOW STRUCTURE %PRIMITIVE% ,
        s2: COMPUTE STRUCTURE ,
        s3: SEARCH STRUCTURE *
f2: KNOW <-- s1: KNOW STRUCTURE *
f3: COMPUTE <-- s4: OPERATOR THAT PRODUCES RESULT ,
        s5: OPERATOR THAT PRODUCES STRUCTURE FROM WHICH CAN COMPUTE ,
        s6: OPERATOR THAT COMPUTES CONDITIONALLY *
f4: SEARCH <-- s7: GENERATOR SEARCHER ,
        s8: GUIDED SEARCHER *
f5: GENERATE <-- s9: GENERATOR *
f6: RECOGNIZE <-- s10: UNIQUE STRUCTURE MATCHER ,
        s11: UNIQUE RELATION MATCHER ,
        s12: EXTREME MEASURE SEARCHER *
f7: RETAIN <-- s13: STORE OPERATOR %PRIMITIVE% ,
        s14: RELATIONAL ITEM STORAGE ,
        s15: NAME LOCATION STORAGE *
f8: STORE <-- s13: STORE OPERATOR *
```

Figure 4: Structures and Provided Functions (input to XDA)

```
s2: COMPUTE STRUCTURE >> f3: COMPUTE *
s3: SEARCH STRUCTURE >> f4: SEARCH *
s4: OPERATOR THAT PRODUCES RESULT >> f9: KNOW OPERATOR *
s5: OPERATOR THAT PRODUCES STRUCTURE FROM WHICH CAN COMPUTE >>
        f9: KNOW OPERATOR & f3: COMPUTE *
s6: OPERATOR THAT COMPUTES CONDITIONALLY >> f6: RECOGNIZE &
        f9: KNOW OPERATOR *
s7: GENERATOR SEARCHER >> f10: GENERATE CANDIDATES &
        f11: RECOGNIZE TARGET & f12: OBTAIN FROM TARGET *
s8: GUIDED SEARCHER >> f13: OBTAIN INITIAL CANDIDATE &
        f14: OBTAIN NEXT CANDIDATE FROM CURRENT CANDIDATE &
        f11: RECOGNIZE TARGET &
        f12: OBTAIN FROM TARGET *
s9: GENERATOR >> f15: OBTAIN INITIAL STATE &
        f16: OBTAIN CANDIDATE FROM STATE &
        f17: OBTAIN NEXT STATE FROM STATE *
s10: UNIQUE STRUCTURE MATCHER >> f18: OBTAIN UNIQUE STRUCTURE &
        f19: OBTAIN DIFFERENCE *
s11: UNIQUE RELATION MATCHER >> f20: OBTAIN UNIQUE RELATION &
        f21: COMPUTE RELATION *
s12: EXTREME MEASURE SEARCHER >> f4: SEARCH & f6: RECOGNIZE *
s14: RELATIONAL ITEM STORAGE >> f22: OBTAIN STORAGE OBEYING RELATION &
        f8: STORE *
s15: NAME LOCATION STORAGE >> f23: COMPUTE NAME OF STORAGE FROM ITEM &
        f8: STORE *
```

Figure 5: Functional Requirements (input to XDA)

```
DESIGN SYMBOL TABLE *
S16: SYMBOL TABLE (D1) --> F24: OBTAIN ADDRESS GIVEN SYMBOL &
        F25: OBTAIN SYMBOL GIVEN ADDRESS &
        F26: RETAIN SYMBOL ADDRESS PAIR *
CHOOSE FROM F24 F25 F26 TO WORK ON:  F24
OBTAIN <-- S1 S2 S3
F24: OBTAIN ADDRESS GIVEN SYMBOL (D2) == S3: SEARCH STRUCTURE *
S3: SEARCH STRUCTURE (D5) >> F4: SEARCH (D6)
CHOOSE FROM F4 TO WORK ON:  F4
SEARCH <-- S7 S8
F4: SEARCH (D6) == S7: GENERATOR SEARCHER *
S7: GENERATOR SEARCHER (D7) >> F10: GENERATE CANDIDATES (D10) &
        F11: RECOGNIZE TARGET (D8) & F12: OBTAIN FROM TARGET (D9)
CHOOSE FROM F10 F11 F12 TO WORK ON:  F12
OBTAIN <-- S1 S2 S3
F12: OBTAIN FROM TARGET (D9) == S2: COMPUTE STRUCTURE *
S2: COMPUTE STRUCTURE (D11) >> F3: COMPUTE (D12)
CHOOSE FROM F3 TO WORK ON:  F3
COMPUTE <-- S4 S5 S6
F3: COMPUTE (D12) == S4: OPERATOR THAT PRODUCES RESULT *
S4: OPERATOR THAT PRODUCES RESULT (D13) >> F9: KNOW OPERATOR (D14)
F9: KNOW OPERATOR (D14) == S1: KNOW STRUCTURE (D15)
PLEASE DEFINE AN S1 AT (D15) THAT PROVIDES F9 F3 F12
S1/(D15) = ADD ONE TO TARGET AND FETCH
CHOOSE FROM F10 F11 TO WORK ON:  F11
        .
        .
        .
```

Figure 6: Initial Portion of XDA Design Trace
(Designer's responses to system are underlined.)

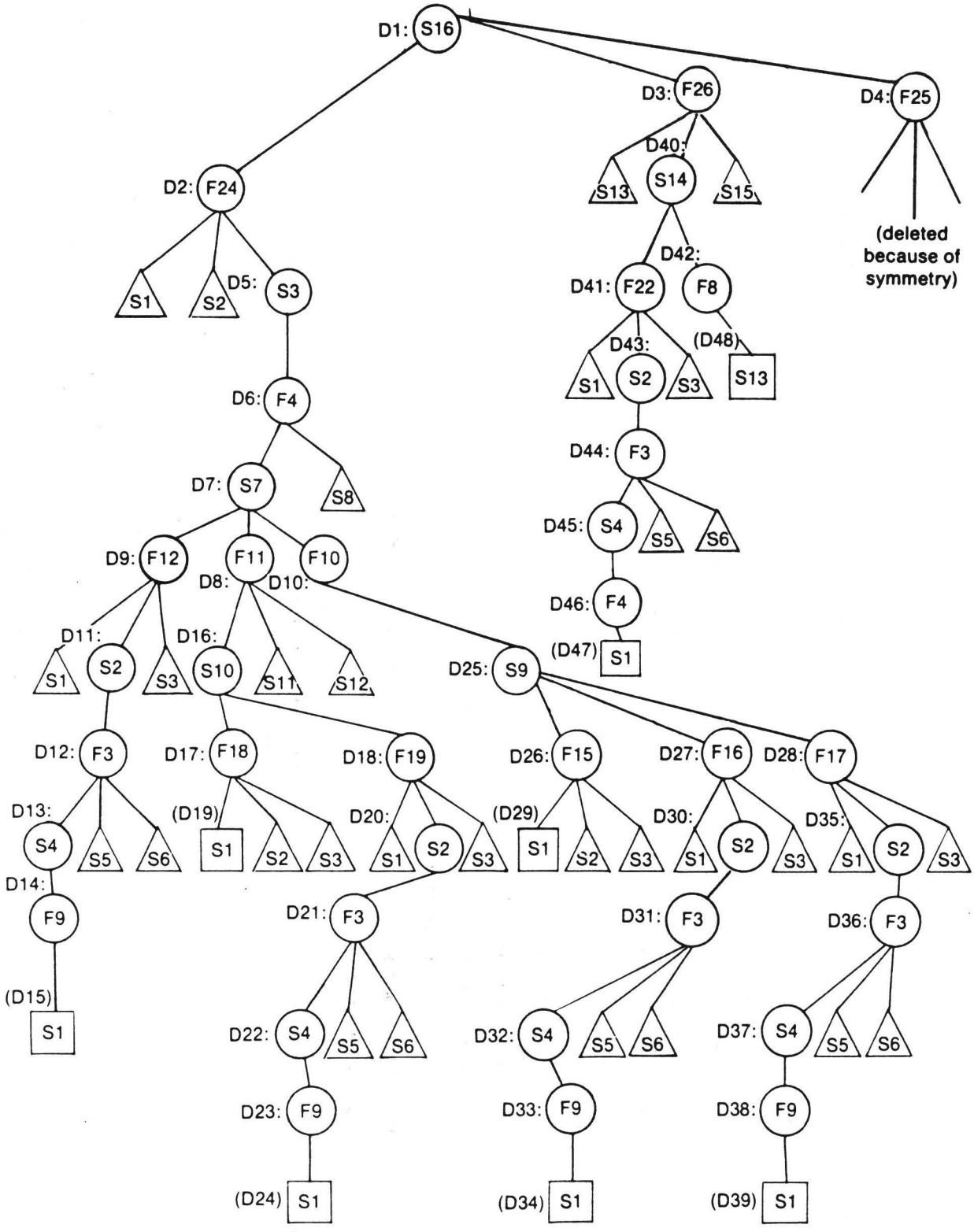


Figure 7: Outline of XDA Design Trace for Symbol Table (Structures considered △s and chosen Ⓢ, resulting functional requirements Ⓕ, and design terminals [s].

Design Terminal	Functions Provided	Definition Given in Design Trace	Structure Used in Actual Solution
S1/(D15)	F9: KNOW OPERATOR F3: COMPUTE F12: OBTAIN TARGET	ADD 1 TO TARGET AND FETCH	address:= ST[i + 1];
S1/(D19)	F18: OBTAIN UNIQUE STRUCTURE	BIT PATTERN OF GIVEN SYMBOL	symbol
S1/(D24)	F9: KNOW OPERATOR F3: COMPUTE F19: OBTAIN DIFFERENCE	FETCH CONTENTS OF CANDIDATE AND TEST FOR IDENTITY WITH UNIQUE STRUCTURE.	test(address, ST[i])
S1/(D29)	F15: OBTAIN INITIAL STATE	GET FIRST WORD ADDRESS OF TABLE.	i := 1
S1/(D34)	F9: KNOW OPERATOR F3: COMPUTE F16: OBTAIN CANDIDATE FROM STATE	STATE IS CANDIDATE.	ST[i]
S1/(D39)	F9: KNOW OPERATOR F3: COMPUTE F17: OBTAIN NEXT STATE FROM STATE	ADD 2 TO STATE.	i := i + 2
S1/(D47)	F9: KNOW OPERATOR F3: COMPUTE F22: OBTAIN STORAGE OBEYING RELATION	ADD 2 TO CURRENT TOP ADDRESS.	top := top + 2
S13/(D48)	F8: STORE	STORE SYMBOL IN FIRST CELL, ADDRESS IN SECOND CELL.	ST[top] := symbol ST[top + 1] := address

Figure 8: Functional Assignments for Symbol Table Design