STRUCTURING

Eric C.R. Hehner
University of Toronto

Abstract: Structuring can be defined independently of what is being structured, and can be applied profitably to more than one domain. Using one mechanism to structure both values and assignments, we obtain equivalents for a variety of data and control structures. Structuring assignments is preferable to structuring control: the former is more conducive to a mathematical style of programming while the latter is more conducive to tracing.

Key Words and Phrases: language design, data structures, control structures, recursion.

CR Category: 4.20.

0. Introduction

Programming languages today have a variety of data structures and control structures that are intended to be convenient for programming. For example, PL/I provides character strings (a structure of characters), arrays, and the "structure" (known in some languages as a "record") for structuring data, and "if" statements, definite (or indexed) iteration, indefinite iteration (the "while" loop), and of course, the "go to" for structuring control. In each case, it is quite clear whether the construct is a data structure or a control structure. Two assumptions are implicit: that a variety of structures serves us better than one, and that structures are appropriate for data or for control but not for both. This paper questions these two assumptions. It suggests that a well-chosen structuring mechanism, defined independently of what is being structured, can provide the good things we now enjoy, while keeping the number of basic language constructs to a minimum. We first exhibit one such mechanism, then discuss its merits.

1. The Structure

To begin with, we shall give ourselves some simple values: the two logical values true and false; numeric values such as 3 and 3.3E3; character values such as 'A'; and a facility for defining new simple values, such as *red*, *blue* and *green*, as in Pascal. Whether these values are truly "simple" will not be debated; it is a convenient choice for the moment. Character strings are not considered simple, and are not included in the above. We give ourselves also a handful of logical, numeric and comparison operators for forming expressions.

Next, we give ourselves a simple assignment:

identifier := expression

Note that the left side may not be an array element. (Declaration of variables, though important, is irrelevant at present.)

Finally, we give ourselves a naming notation for values and assignments. For example,

pi: 3.14

means that wherever *pi* appears, it will stand for 3.14, and

increment: $i := i + 1$

means that wherever *increment* appears, it will stand for the assignment $i := i + 1$. To distinguish *increment* from a conventional label, we emphasize that the above statement does not increment i, but only gives a name to the act of incrementing i.

Our structure is simply a set of ordered pairs. The first member of each pair is called an index, the second an element, of the structure. Indices are values. (It may seem appropriate to restrict the indices of a given structure to one type, or to a contiguous subrange of a type, but such restrictions are irrelevant to this paper.) An expression of an index may include variables, although our examples will use only constants. The elements of a structure constitute that which is being structured; we shall consider structured values and structured assignments.

2. Structured Values

Define a map as a structure whose elements are values. For example, the map

map 1→11, 2→12, 3→13 □

has indices 1, 2 and 3, and elements 11, 12 and 13. A map is itself a value, and can be used in expressions, in assignments, or as indices or elements. It may be given a name by the same

notation used to give a simple value a name. If M is or denotes a map, and i is or denotes an index of M, then $M[i]$ denotes an element of M corresponding to index i (if this element is to be unique, the indices must be unique). Other operations on maps may be defined as deemed necessary; for example, if M is a map and i and e are values, then the expression

$$M \text{ with } i \to e$$

denotes a map like M except that $M[i] = e$.

Certain special cases of maps deserve special notations. When the indices are the integers 1 to n, the notation may be abbreviated by listing only the elements, in order of index, enclosed in angle brackets. The map of the previous paragraph may be expressed as

$$\langle 11, 12, 13 \rangle$$

This gives us a kind of array. If the elements are characters, e.g. ⟨'A', 'B', 'C'⟩, we may abbreviate further, e.g. "ABC". This gives us character strings. The equivalent of records (or PL/I structures) are formed, not as an abbreviation, but simply by using suitable programmer-defined values as indices:

```
map name    → "HEHNER",
    address → map city    → "TORONTO",
              country → "CANADA" □ □
```

It is convenient to group indices that have equal elements, so that

$$3, 4, 5 \to 50$$

stands for three index-element pairs. In this connection, and for arrays, the abbreviation n to m can stand for $n, n+1, \ldots, m$ whenever n and m denote appropriate values.

By introducing a name for the index, we can express a group of elements in terms of their indices. By specifying a range for the index (i.e. a domain for the map), we obtain either redundancy that is useful for checking that all indices are present, or the ability to specify a restricted else; for example,

```
map i ε ⟨1 to 100⟩
    3, 5, 10 → 50,
    11 to 20 → i*2,
    else     → 0 □
```

is a sparse array containing thirteen non-zero and eighty-seven zero elements. The generalization to maps with more than one index (multi-dimensional arrays) is straightforward.

In some cases, a compiler may represent a map as a sequence of pairs of values; in other cases only the elements need storage. In still other cases, maps may be compiled into code for computing values from indices. As an example in which the latter is suitable, consider the following recursive definition of limited factorial.

```
factorial: map n ε ⟨0 to 99999⟩
               0     → 1,
               else  → n*factorial[n-1] □
```

The reader may, at this point, feel drowned in notation. But he should not feel drowned in concepts or language features. Different programmers will have different data structuring requirements, and different notations and abbreviations that seem convenient for their purposes. No programming language can hope to provide all such notations, nor should it attempt to. Instead it should provide a general structure, and a mechanism for specializing it to an individual's needs, and for making convenient abbreviations. This paper does not discuss such mechanisms. It suggests that the "map" can easily be specialized to provide a variety of programmers' needs, while maintaining an economy of concepts within the programming language. It also suggests certain specializations that seem needed often enough to deserve special notations within the language.

3. Structured Assignments

Define a group as a structure whose elements are assignments. For example,

```
group 1 → x := 11;
      2 → y := 12;
      3 → z := 13 □
```

A group is itself an assignment; it is executed by executing its elements in the order they are written.* (This may be a slight abuse of the term "assignment"; it is used here to mean a notation for associating some variables with some values, rather than exactly one variable with one value.) A group may be given a name by the same notation used to give a simple assignment a name. If G is or denotes a group, and i is or denotes an index of G, then $G[i]$ denotes an element of G corresponding to index i. (Once again, if the element is to be unique, the indices must be unique. Without this restriction, we can build Dijkstra's guarded command sets [1] by using logical expressions as indices.)

Thus one builds a group and selects elements from it the same way one builds and selects elements from a map. Most of the same abbreviations are relevant. When the indices are the integers 1 to n, or when we are uninterested in the indices, the elements may simply be listed, giving us the usual "do group". For example, the group of the previous paragraph may be expressed as

```
do x := 11;
   y := 12;
   z := 13 □
```

*The alternative of executing elements in order of increasing index was considered, but rejected for two reasons: we did not want to restrict the indices to be of an ordered type; the for construct for this alternative was more complicated. One alternative remains in contention: that groups should not be executed; a separate syntactic construct is then required for sequencing.

If G is a group, then $G[i]$ is evidently a case statement (either the Algol W kind, or the Pascal kind), for which a more familiar notation is

$$\underline{\text{case}}\ i\ \underline{\text{of}}\ G$$

As a further specialization, we introduce the notation $\underline{\text{if}}$ ... $\underline{\text{then}}$ ... $\underline{\text{else}}$ ... $\square$ to abbreviate a two-element group, whose indices are $\underline{\text{true}}$ and $\underline{\text{false}}$, from which we are selecting one of the elements. For example,

$$\begin{array}{l}\underline{\text{if}}\ b\\ \quad \underline{\text{then}}\ x := 5\\ \quad \underline{\text{else}}\ y := 7\ \square\end{array}$$

abbreviates

$$\begin{array}{l}\underline{\text{group}}\ \underline{\text{true}} \rightarrow x := 5;\\ \quad\ \ \underline{\text{false}} \rightarrow y := 7\ \square\ [b]\end{array}$$

Similarly, $\underline{\text{if}}$ ... $\underline{\text{then}}$ ... $\square$ abbreviates the special case when the $\underline{\text{false}}$ alternative is null. The $\underline{\text{else}}$, which was so useful in the previous section for specifying sparse arrays, has an analogous role in groups used as case statements.

By introducing a name for the index and grouping indices, we have a $\underline{\text{for}}$ construct. The notation

$$\begin{array}{l}\underline{\text{for}}\ i\text{: } 1\ \underline{\text{to}}\ 3,\ 5\ \underline{\text{do}}\\ \quad s := s + i\ \square\end{array}$$

merely abbreviates

$$\begin{array}{l}\underline{\text{group}}\ i \in \langle 1\ \underline{\text{to}}\ 3,\ 5\rangle\\ \quad 1\ \underline{\text{to}}\ 3,\ 5 \rightarrow s := s + i\ \square\end{array}$$

which in turn abbreviates

$$\begin{array}{l}\underline{\text{group}}\ 1 \rightarrow s := s + 1;\\ \quad\ \ 2 \rightarrow s := s + 2;\\ \quad\ \ 3 \rightarrow s := s + 3;\\ \quad\ \ 5 \rightarrow s := s + 5\ \square\end{array}$$

The generalization to more than one index, which in the previous section gave us multi-dimensional arrays, here gives us the effect of nested $\underline{\text{for}}$ constructs, but written as a simple construct. Notice that the indices are not variables, so no question of assignment involving indices within the construct is raised.

Finally, the notation $\underline{\text{while}}$ ... $\underline{\text{do}}$ ... $\square$ may be used to abbreviate a recursively defined group. For example,

$$\begin{array}{l}\underline{\text{while}}\ i < 10\ \underline{\text{do}}\\ \quad i := i + 1\ \square\end{array}$$

abbreviates a group, say G, defined as

$$\begin{array}{l}G\text{: } \underline{\text{if}}\ i < 10\\ \quad\ \ \underline{\text{then}}\ i := i + 1;\\ \quad\quad\ \ G\ \square\end{array}$$

Those who consider iteration to be simpler than recursion may, at this point, be unhappy. The next section is intended to change the viewpoint of those people.

4. Understanding Structured Assignments

When we read or write programs, we have always maintained a thinly disguised "program counter" or "instruction pointer"; we are always aware of the flow of control. For example, at the end of a loop we know that control or execution goes back to the beginning of the loop. We have no problem understanding the $\underline{\text{go to}}$: execution continues at the indicated statement (understanding programs containing $\underline{\text{go to}}$s is another matter).

Recursive flow of control, in general, involves a stack of return addresses, and so it is more complicated than looping. In current programming languages it is further complicated by making it inseparable from procedures, which introduce a new local scope each invocation. In this paper, recursion was introduced independently of local scope; even so, defining a $\underline{\text{while}}$ loop by way of recursion may seem to be defining something simple by way of something complex.

A program should not be understood in terms of a particular implementation of a language, nor by tracing an execution of the program. These two premises are generally well accepted, and need no defense in this paper. Yet in almost every programming text, control structures are explained only by explaining how to trace an execution according to some implementation.

Selection ($\underline{\text{if}}$ or $\underline{\text{case}}$) is usually explained by saying that control jumps to one of the alternatives, and from there to the end of the construct. In this paper, the notation $G[i]$ is said to denote an element of G. Apart from the subtle change in attitude, the use of the word "jump" may sometimes be misleading. For example, with the constant definition

$$\textit{devices}\text{: } 4$$

the structured assignment

$$\begin{array}{l}\underline{\text{if}}\ \textit{devices} < 10\\ \quad \underline{\text{then}}\ A\\ \quad \underline{\text{else}}\ B\ \square\end{array}$$

can be compiled simply as A, giving conditional compilation without any new language feature. We introduced the $\underline{\text{for}}$ construct as an abbreviation for a sequence of assignments, rather than as a "loop". The important point is not the change in terminology, but the change in thinking: from jumping control to structured assignments. One could never invent a $\underline{\text{go to}}$ as a structured assignment.

An explanation of recursion that involves activation records or return address stacks is irrelevant, confusing, and often wrong. The best implementation of the recursive group in the previous section is exactly the same as the implementation of the $\underline{\text{while}}$ construct that abbreviates it[6]. The proper explanation of a recursive construct, or of a $\underline{\text{while}}$ construct, involves the principle of mathematical induction[3].

It is sometimes objected that an average person cannot be expected to understand the principle of induction, or to apply it to programming. If that were true, it would not be an argument against the use of induction in programming, but against the use of average people as programmers. In fact, average people understand the principle perfectly well, although informally. Given a positive integer, and enough time, an average person believes he can count from 1 to the given integer. For large enough integers, that belief is not based on the experience of having done so, but on an implicit understanding of the principle of induction.

It is often easy to see that a recursive construct works for n = 0, and that if it works for n = k - 1, it will work for n = k. The common mistake is asking (or explaining) how it works for n = k - 1. This mistake is made for one of two reasons: (a) failure to assume the inductive hypothesis; induction requires that we prove an implication, not the hypothesis of the implication. (b) curiosity about the implementation; an explanation of the implementation should come only after the semantics are understood, not as an explanation of the semantics. For either reason, this mistake leads to an effort to understand by tracing, and to the poor man's induction: "If it works for n = 1, 2 and 3, then that's good enough for me.".

Mathematical semantics refers to the characterization of programming language statements by their effect on program state (the mapping from variables to values). It has developed as a technique for defining programming languages[9]. In this view, the time sequence of events that occur during program execution is irrelevant: a statement is a mathematical function from states to states. Unfortunately, may of the statements it has been used to characterize were designed to control execution. By structuring assignments, rather than control, we are taking the view that is consistent with mathematical semantics.

5. Exits

The while loop, considered as a construct denoting repetitive execution, has one exit: at its head. And that exit is only a "single-level" exit that cannot be used to terminate execution of several nested loops at once. Intermediate and deep exits have been proposed in various forms; for arguments pro and con see [7]. Suppose $A1$ is defined as

```
A1: while b1 do
        A2;
        A3 □
```

and action (assignment) $A2$ within $A1$ is defined as

```
A2: wnile b2 do
        A4;
        if ¬ b3
           then exit A1 □;
        A5 □
```

where the notation exit $A1$ means that we jump to the statement following $A1$. The benefit of this construct is efficiency: with only single-level exits, one needs to perform a test within $A1$, just prior to $A3$, to determine whether to execute $A3$ and continue $A1$, or to exit $A1$. The problem with this construct is a loss of clarity: an examination of $A1$ would lead one to conclude wrongly that $A2$ and $A3$ are executed repeatedly until $b1$ becomes false.

As structured assignments, rather than structured control, loops and exits are unacceptable. The equivalent recursive definitions are as follows.

```
A1: if b1
       then A2 □

A2: if b2
       then A4;
            if b3
               then A5;
                    A2 □
       else A3;
            A1 □
```

The recursions may be implemented by simply branching to the appropriate label; we thus have the efficiency of the deep exit without inventing a new language feature. And the condition under which $A3$ is executed is clear at a glance. The loop-with-exit has the curious property that when there is more to be done,one says nothing,but when there is no more to be done, one says something: exit. The recursive version is more straightforward; when there is more to be done, one specifies it, and when there is not, one says nothing.

6. Uniform Referents

When programming, if one decides that one needs to select a value from a class of values, one can use the notation $F[x]$ where F denotes the class, and x is used to select the desired element of F. One may then decide to implement F either as a map, or, if one needs to introduce local names and variables and to use assignments in the production of the desired value, as a function (procedure that returns a value and does not affect global variables). The notation $F[x]$ must not prejudice the choice of implementation of F. A change in the implementation of F that does not change the meaning of the program, such as substituting a map for a function or vice versa, should not require changing the notations involving F throughout the program. The above principle of language design is (a version of) the "principle of uniform referents"[8].

We now have a new application for the principle. If one decides that one needs to select an action (assignment) from a class of actions, one can give the class a name, say G, and denote selecting the desired action by $G[x]$. This notation should allow G to be implemented either as a group, giving us a "case statement" implementation, or as a routine (procedure that

affects global variables and does not return a value).

7. Conclusion

The point of this paper is not the notations used to present the structuring mechanism; when the reader finds them awkward or unpleasant he is invited to change them. Nor is the point the particular structuring mechanism chosen; it is not claimed that the mechanism in this paper is adequate or suitable for all purposes.

There are two main points. One is that structuring can be defined independently of what is being structured, and can then be applied profitably to more than one domain. A well-chosen general structure that can be specialized in a variety of ways is preferable to the myriad of seemingly unrelated structures we now live with. The other main point is that structuring assignments is preferable to structuring control. The former is more conducive to a mathematical style of program composition, while the latter is more conducive to tracing. Arguments in favour of structuring values rather than storage have been presented elsewhere[2,4].

Acknowledgement

I am indebted to Jim Horning, Nigel Horspool and Brian Clark for discussions and suggestions that lead to this paper.

References

1. Dijkstra, E.W. A Discipline of Programming. Prentice-Hall (1976).

2. Hehner, E.C.R. A simple view of variables and parameters. (to appear.)

3. Hoare, C.A.R. An axiomatic basis for computer programming. Comm. ACM 12(10) (October 1969).

4. Hoare, C.A.R. Recursive data structures. Report STAN-CS-73-400, Stanford University (1973).

5. Kieburtz, R.B. Programming without pointer variables. ACM Conference on Data, Salt Lake City (March 1976).

6. Knuth, D.E. Structured programming with go to statements. ACM Computing Surveys, 6.4 (December 1974).

7. Ledgard, H.F., Marcotty, M. A genealogy of control structures. Comm. ACM 18(11) (November 1975).

8. Ross, D.T. Uniform referents: an essential property for a software engineering language. in Tou, J.T. (ed.) Software Engineering, Academic Press (1970).

9. Scott, D. and Strachey, C. Towards a mathematical semantics for computer languages. J. Fox (ed.), Computers and Automata, John Wiley, pp. 19-46 (1972).