Pierre-Arnoul de Marneffe

Holon Programming

A Survey

December 1973

Foreword

In this survey of the Holon Programming Concept, we shall try to collect together all the different notes we have written about this problem. All the texts about Holon Programming were written in French[1] (except the text for the ICS presentation),[2] and that precludes a diffusion of the concept, and at the same time, a welcome feedback of criticisms and comments to the author.

This survey was written during a stay at the IBM Federal Systems Center (Gaithersburg, Maryland). This stay was supported by the Belgian Fund for Scientific Research (F.N.R.S) with the help of the IBM-Belgium Donation.

[1] P.A. de Marneffe. *Holon Programming: definition du langage*. Privately circulated.

[2] P.A. de Marneffe et al. "Holon programming". In: *International Computing Symposium*. Ed. by A. Günther et al. Amsterdam, North Holland, 1973.

Abstract

The "Holon Programming Language" is designed to force the programmer to design programs according to a top-down methodology (structured programming). But, the language is designed to generate well-structured and efficient programs. The holon concept is explained; the Holon Language Syntax is described; and some further developments are presented: stretching the concept to concurrent processes, and the development of a holon operating system for a virtual memory computer.

Contents

List of Figures

Letter to the Author from Knuth

April 1, 1974

Prof. Pierre-Arnoul de Marneffe
Université de Liège
Service d'Informatique
Avenue des Tilleuls 59
B-4000 Liège, Belgium

Dear Prof. de Marneffe:

Thank you very much for sending me your survey of Holon Programming. I especially enjoyed your references to the non-computer literature (Koestler, Bernard-Shaw, Shanley, Mount Vernon, etc.) since computer scientists need to avoid insularity.

I believe you are making important strides toward the development of a new programming language. There still remain some unclear areas but you are obviously addressing the correct issues; the next thing to do (it seems to me) is to program several hundred examples!

For related reading I would suggest that you carefully study Ole-Johan Dahl's papers on SIMULA since his class concept is so close to the holon concept. Also I have just heard that Brian Randell of Newcastle has been working on a so-called PEARL system.

I found your report could have been improved if you had worked entirely with tree structures instead of converting to binary trees. The original tree structure is what is really relevant, and the Dewey notation for such structure is more directly suited to the operations you discuss. The binary tree is only a machine-oriented representation of the basic concept, the discussion should stay at a higher level.

Secondly, I found the report too preoccupied with details of implementation. The people by whom it is most important that this report be read are either able to visualize easily how to implement this sort of system, or else they are people who are not likely to care how it's implemented as long as it's handled sensibly. The important thing to stress is rather the conceptual issues of how holon programming differs from and improves on today's languages.

The example didn't come until page 100, while I expect most readers would have preferred to see it immediately. Since the program is almost self-explanatory, you can let it explain the language at the same time (integration of functions!).

page 66 of this document

Ideally there should be more examples of course.

The one example raises some interesting issues since the individual holons don't quite state their assumptions. In the very first holon, for example, it is not at all clear why you 'find first word starting character' instead of going right into 'find a word **etc**.' You must already have made a decision (a) that you wouldn't assume the text begins with a nonblank, (b) that there is going to be at this level an element of data representing the last-read character, (c) that the 'find a word' routine will already have its first character in hand, and (d) that there is no need to test for a message that has no words (only a full stop). As I recall when I was solving that problem, it took me a good five minutes to reach these decisions, during which I must have considered lots of alternatives. Once this step was made the rest of the program flowed naturally. My questions are: Where should we state these assumptions? Shouldn't we mention the existence of data representing the last-read character, even though we don't want to specify its detailed structure until later?

These issues seem to arise repeatedly and I haven't a first conclusion about what we ought to do. That's why I suggest working out hundreds of examples, as being the best kind of eating to prove the pudding at this stage. On the other hand creating the holon implementation itself is equivalent to working out quite a few examples.

Thanks again for showing me your stimulating work. I myself must get on with the writing of volume 4 of my series, so I have little energy to devote to the development of languages, but I will do my best to see that other people working in the area are kept informed of what you are doing.

Sincerely,

Donald E. Knuth
Professor

P.S. Is there a place in Belgium whose postal code is B-6700 like the Burroughs computer?

Your Royal Academician thinks he can get the style of Giotto without Giotto's beliefs, and correct his perspectives in the bargain. Your man of letters thinks he can get Bunyan's conviction or Shakespear's apprehension, especially if he takes care not to split his infinitives. And so with your Doctor of Music, who, with their collections of discords duly prepared and resolved or retarded or anticipated in the manner of the great composers think they can learn the art of Palestrina from Cherubini's treatise.

— GEORGE BERNARD SHAW
Letter to Arthur Bingham Walkley

Structured Programming and Present Programming Languages

It was contended by Dijkstra that: "... the art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible...."[3] His letter to the ACM's Editor about the "harmful GOTO statement"[4] was already an attempt to avoid "chaos" in programming. A full explanation of a methodology to design well-formed programs appeared for the first time in his "Notes on Structured Programming."[5] It is assumed in the next sections that the reader is familiar with the notions of structured programming and top-down programming. (If not, useful references are Dijkstra's above-mentioned "Notes," Wirth,[6] Woodger,[7] and Mills[8]).

[3] E.W. Dijkstra. *EWD316: A Short Introduction to the Art of Programming*. Technical University Eindhoven, Aug. 1971.

[4] E.W. Dijkstra. "Go to statement considered harmful". In: *Communications of the ACM* 11.3 (1968), pp. 147–148.

[5] E.W. Dijkstra. *EWD249: Notes on Structured Programming*. Technical University Eindhoven, 1970.

[6] N. Wirth. "Program development by stepwise refinement". In: *Communications of the ACM* 14.4 (Apr. 1971), pp. 221–227.

[7] M. Woodger. "On semantic levels in programming". In: *IFIP Congress (1)*. 1971, pp. 402–407.

[8] H.D. Mills. *Mathematical foundations for structured programming*. Tech. rep. FSC-72-6012. Gaithersburg, MD: IBM Federal Systems Division, Feb. 1972.

Many people use the structured programming methodology on purpose to write large programs. The unconvinced reader of Dijkstra's "Notes" was surely struck by the publication of practical results.[9,10] However, the use of the structured programming methodology with the present programming languages has, in our opinion, several drawbacks.

[9] F.T. Baker. "Chief programmer team management of production programming". In: *IBM Systems journal* 11.1 (1972), pp. 56–73.

[10] F.T. Baker. "System quality through structured programming". In: *Proceedings of the December 5–7, 1972, Fall Joint Computer Conference, part I*. 1972, pp. 339–343.

A) The programming cannot use the same language for the design and the implementation of a program. The design phase of the program is performed by using a design language which allows only the writing of structured programming control statements (*ifthenelse*, *dowhile*, *repeatuntil*, *case*, *sequence*). Those control statements link together a series of abstract levels, the definition of which is postponed. But, when all the levels are defined in this way, the programmer must undertake the mapping of the designed program into an executable one. This last one must be written in a "machine-understandable" language. But this mapping is a nontrivial and error-prone operation.

B) Another problem is the correspondence between each level of the structured program and an implementation unit for this level. Woodger was the first one, in my opinion, to point out the lack of a suitable tool for implementing semantic levels (Woodger's semantic level is similar to Dijkstra's pearl[11]).

[11] See Section 14 of Dijkstra's *Notes*.

It is impossible to give to the programmer a "rule of thumb" for the relation between semantic level and implementation unit: suppose you teach a programmer to structure a program by a stepwise process, introducing a lot of levels—and that is the right way—you can't advise the programmer to implement each level as a procedure, because that will generate a very inefficient program.[12] On the other hand, you can't allow the programmer to follow the rule: create levels as you introduce procedures in conventional programming. Application of this rule will not produce structured programs! Because:

[12] J. Horning et al. "Process structuring". In: *ACM Computing Surveys (CSUR)* 5.1 (1973), p. 26.

1) Structuring programs by procedures leads to programs structured by "large steps." This way may easily produce incorrect programs.[13]

[13] P. Henderson et al. "An experiment in structured programming". In: *BIT Numerical Mathematics* 12.1 (1972), pp. 38–53.

2) The programmer is influenced by the language which he usually uses.[14] But, it must be well-understood that he is influenced not only by the language syntax but also by the idiosyncrasies of the compiler and the computer he uses. Programmers avoid language statements which lead to unclear compiler error messages. In his use of procedures, the programmer is influenced by the relative cost of the procedure overhead on his computer.

[14] E.W. Dijkstra. "The humble programmer". In: *Communications of the ACM* 15.10 (Oct. 1972), pp. 859–866.

C) When a structured program is translated into a program written in a conventional language (i.e., not designed for structured programs only), errors may cut right across the layers, because these are only abstractions (Ven Der Poel,[15] page 51).

[15] P. Naur et al., eds. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. 1969.

With all these drawbacks in mind, it may seem sensible to design a language which would force the programmer to design Well-structured programs. The language must help the programmer to write programs without any computer architecture in mind. The language structure must permit automatic translation of a structured program into an efficient one. We shall try to design such a language by the use of the "Holon" concept.

The "Holon" Concept

We shall show that the close connection of the "Holon" concept to the structured programming principles justifies its use in the language design. A systematic use of this concept will help to suppress that shortcomings of the use of the present programming languages for writing structured programs.

The "Holon" concept has been introduced in biological and behaviour sciences by Koestler.[16] This concept proceeds from work of Simon.[17] It is used for instance to analyse complex living organisms or complex social systems.

[16] A. Koestler. *The Ghost in the Machine*. Macmillan, 1968.

[17] H.A. Simon. "The architecture of complexity". In: *Proceedings of the American philosophical society* 106.6 (1962), pp. 467–482.

This neologism is from Greek "holos" (ὅλος), i.e., whole, and the suffix "-on" meaning "part." A holon is a "part of a whole." The reader is forewarned to not mix up the holon concept with the "module" one. The difference will clearly appear, we hope, in the next paragraphs.

According to the holon concept, the complex systems can survive only if they are holons. A complex system must be a hierarchic ordering of functions (holons) with the following features:

A) "Hierarchy": Each holon is composed by other holons which are "refinements" of the former holon. These holons are submitted to some rigid rules; they perform the "detail" operations which, put together, compose the function of the former holon.

B) "Tendency to Autonomy": Each holon has the ability to check its "environment" before performing its function; it may modify its behaviour according to test results.. There exists a clear distinction between a "state description" and a "process description."[18] The holon function corresponds to the process description; but it checks if the environment upon which it acts, belongs to its expected state description.

[18] Ibid.

C) "Tendency to Integration": The holon integrates with other holons in the hierarchy according to a flexible strategy. This integration must be understood as a will to close cooperation with the other holons for the emergence of a "tougher" and more efficient component.

A full description of the holon concept appears in Koestler's *The*

Ghost in the Machine, chapters three to eight. Only the main features were described in this section. In the next section, we will display the relation between this concept and the design of structured programs.

Holons and Structured Programming

Our statement is:

1) The holon model is an attractive one for a programmer using structured programming;

2) The model may be mapped into a useful and implementable language.

We shall review each of the three holon features to uncover the corresponding programming features.

Hierarchy

Anyone, familiar with the structured programming concepts, will immediately understand the connection between the hierarchic embedding of holons and the structured programming levels.

Tendency to autonomy

First of all, this feature must be related to the drawback pointed out by Van Der Poel (see section 1.(c)): errors may cut right across the layers.[19] The "tendency to autonomy" may be understood as checking properties at an appropriate level. These properties must hold for any correct execution of the program.

[19] Naur et al.

Structured Programming may be described in terms of set theory;[20] the correctness of the program is based upon assertions about functions, the values of which are computed at each level. For instance, if the programmer is using Hoare's theorems,[21] the correctness is guaranteed if the data always fit the hypothesis (assertions before each function).

[20] Mills.

[21] Dijkstra.

Where? When? How do we do this check? Answers to these three questions will be given in the following section.

Tendency to integration

This third holon feature is the most important one. With the help of an adequate implementation, the programmer will not have to consider anymore the choice of an implementation unit (open or closed procedure) for each level. This choice will be automatic and tailored for the computer which will execute the program. The choice is left to an automatic system because the relative cost of an open or closed procedure is a machine-dependent parameter. The cost is expressed as a tradeoff between time and space.

In this way, the programmer designs his program in terms of a set of embedded functions, and the mapping of the program description in a "machine" program is outside his control. The only thing he knows is that the generated machine-program will be as efficient as possible for the used computer.

Datatest and Invariant

Now, we return to the three questions about the level properties checking: Where? When? How?

I must confess that it takes a long time to fully understand the implications of the questions and the corresponding answers. The holon concept assumes that a complex system must check its environment to stay alive. It can be made by the check of the similarity between the real environment and a "standard" one.

Where?

First, consider a holon A. The function of this holon A is made up by the execution of other holons, for instance: B, C, D. These three holons (B, C, D) are the first level of the hierarchic structure of the holon A. They may be composed of other holons, according to the hierarchic embedding of holons.

We say that the execution of holon A is "sensible" only if his "input environment" is a "legal" one. There exists a set of predicates, the values of which are *true* if the environment is a "legal" one. We call this set of predicates: the datatest.

Now, consider the three holons (B, C, D), which are the first level of the hierarchic decomposition of A. Suppose that the execution of A implies the executions of B, C and D, in this order. After each execution of one of these holons, the environment of A is modified. Nevertheless, there exists a set of predicates (upon this environment), the values of which must be *true* after each execution. If not, the modified environment will not belong to the range of holon A, after completion of the execution of A. We call this set of predicates: the invariant.

When?

When do we check the predicates of the datatest and invariant?

The definitions of these two sets give well-defined positions in the holon: the datatest at the beginning, and the invariant after each inner holon.

But each set (invariant or datatest) can be segregated in three subsets:

1) A subset of predicates which can be checked at compile-time (for instance—data types). The extent of this subset depends on the designed syntax of the language.

2) There are predicates, the truth of which may be proved by the programmer. The proof can be a formal or informal one relying upon an analysis of the structure of the program. This means that the programmer can prove the truth of these predicates, whatever the input to the program.

3) There remains a set of predicates, the values of which will be computed at run-time.

The main question is: why does the third set of predicates exist? It exists because the programmer wants to increase the confidence in the program correctness, the value of the output, and the program robustness.

How?

The datatest and the invariant will be made up by predicates belonging to the third set. They are used to trap illegal inputs which will cause havoc. We fear these inputs because we are unable to trace the consequences of such inputs on our program, or because we can't decide if this kind of input will or will not appear during the program lifetime.

For instance, the description of many solutions of some problems are preceded by the foreword: "this solution works perfectly provided. . . "; there is room for a datatest or an invariant.

It may be argued that we don't need a special syntax in the language to implement this kind of "fire-wall." That is true; but, in our opinion, casting the program hypothesis in a dedicated syntax unit will catch the attention of the program reader (and user) to the exact kind of hypothesis which was made during the program design.

Many readers of previous reports about "Holon Programming"[22] have suggested that there is no need for a distinction between datatest and invariant. I introduced the invariant because a holon can have a way to verify its output. On the other hand, we can use the invariant to build a "fire-wall" when we merge several predefined holons to construct a new one. From time to time, we make some kind of bottom-up construction, even in a top-down design.

[22] Marneffe.

The datatest may help by introducing levels in the tests we made. For instance, everyone knows the problem of checking the subscripts in array computation: Algol-Compilers made (generally) a run-time checking of the computed subscript expressions. This slows down the computation, but on the other hand helps for avoiding errors. Nevertheless, that may discourage a programmer to prove the programs he is writing. For instance, if you write a procedure to invert a matrix, and you prove (formally or informally) that this procedure outputs the correct inverse matrix, you may consider that the subscripts' run-time checks are a waste of time. The only check you need is the checking of the squareness of the input matrix. In a holon program, we will have a check of the matrix shape at the "invert matrix" level, without checks of subscripts in the inner levels. [This last point implies that there must exist functions to test the components of a data structure as "finding the bounds of the range of an array subscript," etc. . .]

Remark: The main point with the holon's tendency to autonomy is that the operations check (at some level) the correctness of the type of the data upon which they are operating. The following example appears in Koestler's book: when a bird is building a nest, it follows a "program" made up of a set of well-defined operations; it is not only by applying the operations in the right sequence but also by applying the right operation sequence to the right materials that the nest is built. For instance, the search and use of hard material (twigs) must precede the catch and use of soft material (wool, moss). For people with a structured programming mind, this description of a nest building will look like a structured program!

The Holon Language Syntax

To define the language syntax we will use the BNF notation.

Transcriber's note: the author used an idiosyncratic variation of Extended Backus-Naur form. The grammar that appears here is written using the convention that the superscript $^{+}$ denotes the appearance of one or more syntactic elements and '[...]' delimits optional elements.

Three kinds of syntax

We will describe three kinds of syntax:

1) The external syntax of the language,
2) The internal syntax,
3) The modified external syntax.

The reader may puzzle about the third syntax; as a matter of fact this syntax will look like the first one but with some abbreviations to ease the handwriting (and in this last word, the prefix "hand" is important). The modified syntax is a mapping of the external one in function of the kind of tools we use to write the program.

The external syntax

Programs, holons and names

⟨program⟩ ::= ⟨holon⟩
⟨holon⟩ ::= ⟨holon name⟩⟨holon body⟩
⟨holon name⟩ ::= ⟨plain holon name⟩
    | ⟨holon name with parameters⟩
⟨plain holon name⟩ ::= ⟨element of the unreserved symbols set⟩$^{+}$
⟨holon name with parameters⟩ ::=
    ⟨element of the unreserved symbols set⟩$^{+}$
    | (⟨left identifier bracket⟩⟨identifier⟩⟨right identifier bracket⟩)$^{+}$
⟨element of the unreserved symbols set⟩ ::=
    "any symbol except a reserved symbol"

This last definition is not a BNF one, we used this informal sentence to shorten the definition; as a matter of fact, the symbols set is not defined in holon programming; you may use anything (ideograms if you will), but there are some symbols which are reserved.

In the symbols set you use, you must reserve the following symbols:

1. the identifier bracketing symbols (we use '%' for left and right identifier brackets),

2. the connective symbol (we use ';'),

3. the reserved words used in the language statements (we set reserved symbols in **bold** font).

The following examples are four valid holon names; the last one is not a statement, it is a holon name.

- Invert matrix %A% with bounds %N% %M%
- Search a free record
- Is the printer ready?
- $X := X + 1$

Remarks

1. ⟨identifier⟩ will be defined inside the data structures section.

2. Two holon names are said to be "identical" if the symbols of the two names match after the following operations:

   (a) Leading and trailing blanks are discarded,

   (b) Strings of consecutive blanks are replaced by one blank symbol,

   (c) The symbols between a left and its corresponding right identifier brackets are ignored.

   The following holon names are "identical":

   - Invert matrix %X% result in %Y%
   - Invert matrix %bobby% result in %ok%

   The following are not:

   - Invert matrix %A%
   - Invertmatrix %4%

3. There is not a limit to the length of a holon name; the holon name may become a full explanatory text.

Holon body

⟨holon body⟩ ::= [⟨datatest block⟩][⟨invariant block⟩]⟨function block⟩
⟨function block⟩ ::= ⟨holon statements block⟩
    | ⟨holon terminal language statements block⟩

The important fact of the holon body syntax is the alternative for the writing of the ⟨function block⟩: either like a ⟨holon statements block⟩ or like a ⟨holon terminal language statements block⟩; inside a holon there is no mixing ⟨holon statement⟩ (see further definitions) and ⟨holon terminal language statement⟩ (see the external syntax).

Boolean holon

If a holon name appears at well-defined positions in holon statements (these positions will be defined in the holon statements syntax), this holon is called a "boolean holon."

A boolean holon must contain at least one boolean expression (written in a terminal language); the value (*true* or *false*) of this evaluated expression is the value of the boolean holon.

If there is more than one boolean expression, the boolean holon value is the value of the last expression.

It will be seen in a following section that the boolean holon may have access to variables by value.

It must be clearly understood that a holon becomes a boolean holon by the context; nevertheless a holon is either a "pure" holon or boolean one. If a holon name appears in a spot such that it becomes a boolean one, all appearances of this name in places where a boolean holon is not expected is an error.

Datatest and invariant blocks

⟨datatest block⟩ ::=
    datatest (⟨simple test⟩ | ⟨test and error recovery⟩)$^+$ **end**
⟨simple test⟩ ::= ⟨boolean holon name⟩ ;
⟨test and error recovery⟩ ::=
    ⟨boolean holon name⟩ **else** ⟨holon name⟩ ;
⟨invariant block⟩ ::= **invariant** ⟨simple test⟩$^+$ **end**

A data test evaluation is performed according to the following rules: [These rules must be executed in this exact sequence]

1. The ⟨test and error recovery⟩ elements are evaluated. For each element, the boolean holon is evaluated; if its value is *false* then the

holon, the name of which follows the **else** is executed.

2. If any “**else** part” was executed during the Phase 1, we re-evaluate all the boolean holons appearing in the ⟨test and error recovery⟩ elements.
3. The boolean holons of the ⟨simple test⟩ elements are evaluated.
4. If one (or more) *false* values appear during 2 and 3, the holon execution is inhibited and a full report of the data test failure is printed out.

An invariant evaluation is performed according to the number 3 and number 4 operations of the data test evaluation rule. A data test is evaluated in front of its holon execution, an invariant is evaluated inside a holon execution, each time a connective symbol is met.

Remark: The syntax doesn’t require that the ⟨simple test⟩ elements are written after the ⟨test and error recovery⟩ ones. The sequencing—according to the evaluation rules—must be made by the compiler. The programmer must be aware of that and be careful if you use some kind of side-effect in the datatest.

Holon statements block

⟨holon statements block⟩ ::= **begin** ⟨holon statement⟩$^+$ **end**
⟨holon statement⟩ ::= ⟨sequence⟩ | ⟨ifthenelse⟩ | ⟨dowhile⟩ | ⟨repeat⟩
    | ⟨case⟩ | ⟨abort function⟩ | ⟨data structure commands⟩
⟨sequence⟩ ::= (⟨holon name⟩ ;)$^+$⟨sequence unit⟩
⟨sequence unit⟩ ::= ⟨holon name⟩ | ⟨abort function⟩
    | ⟨holon name⟩⟨abort function⟩
⟨ifthenelse⟩ ::= **if** ⟨boolean holon name⟩
        then ⟨sequence unit⟩
        [**else** ⟨sequence unit⟩]
⟨dowhile⟩ ::= **do** ⟨holon name⟩ **while** ⟨boolean holon⟩ ;
    | **while** ⟨boolean holon⟩ **do** ⟨holon name⟩ ;
⟨repeat⟩ ::= **repeat** ⟨holon name⟩ **until** ⟨boolean holon⟩ ;
⟨case⟩ ::= (⟨boolean holon name⟩ **then** ⟨sequence unit⟩ **eor**)$^+$
        else ⟨sequence unit⟩ ;
⟨abort function⟩ ::= **exit** | **stop**

(The ⟨data structure commands⟩ will be described in a following section devoted to the data structures.)

In the above syntax, any appearance of the ⟨holon name⟩ means that the named holon is executed; any appearance of a ⟨boolean holon name⟩ means that the named boolean holon is evaluated.

We shall not describe thoroughly the semantics of the holon statements, because the meanings of statements like ⟨ifthenelse⟩, ⟨dowhile⟩, ⟨repeat⟩, are well known.

An important point is: how do we look at the reserved words?

The reserved words may be understood as "flow of control switches"; that is the usual interpretation, enforced by the usual flowchart representation of a program. Another interpretation is looking to the statements as functional units and expressing their meanings using set theory.[23]

[23] Mills.

The semantics of the **case** are similar to the semantics of the conditional form in LISP. The named boolean holons are evaluated in sequence. The first boolean holon, the value of which is *true*, implies the execution of its associated ⟨sequence unit⟩. This last execution completes the **case** statement. If no boolean holon is *true*, then the last sequence unit is executed (the one which follows the **else**).

Introducing an **exit** (or **exitloop**) statement in the programming languages isn't presently well-understood. As a matter of fact, nobody has given a convincing explanation for its introduction or its rejection. We only know that it is possible to write programs without any **exit**, but that the **exit** eases the programming for some class of algorithms.

We have adopted the following policy: to admit two "functions": the **exit** and the **stop**. The **exit** means a local abortion of a holon execution. The **stop** implies a complete interruption of a program.

The syntax displays the limited use of the **exit**. It may only appear in an **if** or a **case** statement. The **exit** may be viewed as a **goto** the last semicolon of a holon body.

In a loop, the **exit** doesn't work like an "*exitloop*" statement. The syntax implies that the "repeating part" of a loop is abstracted in a holon name. The "condition part" must be abstracted in a boolean holon. If in the holon, the name of which is the "repeating part," we use an **exit**, this **exit** will bring us back to the test of the condition part. An example will be described in the appendix.

Holon terminal language statements block

```
⟨holon terminal language statements block⟩ ::=
    (⟨terminal language designator⟩⟨terminal body⟩)+
⟨terminal body⟩ ::=
    (⟨terminal language statement⟩ | ⟨data structure commands⟩)+
        ⟨end of terminal body⟩
```

The three undefined elements of this section will be defined in the section devoted to the "Internal Syntax."

The internal syntax

The external syntax describes how a holon is made of other holons. This process stops when we meet a holon written in one of the terminal languages. The terminal languages compose the internal syntax.

Holon terminal languages

These languages contain only operative statements, the meaning of which may be generalized by the following expression:

$$X \leftarrow \text{apply}(OP_1, OP_2, OP_3, \dots, OP_n) \text{ to } (Y_1, Y_2, Y_2, \dots, Y_m)$$

In this expression: X and Y_i are the names of data structures; OP_i is the symbol of an operation; "apply" is a function, the rules of which are well-defined, this function explains the order of the application of operations on the data structures.

The expression can be read like this: "the data structure X takes a new value. This value is computed by the application of a set of operations $(OP_1, OP_2, OP_3, \dots, OP_n)$ on the values of data structures $(Y_1, Y_2, Y_3, \dots, Y_m)$. The order of the operations application is defined by the function 'apply'."

In this definition, data structure and operation must be understood in a wide extent: a line on a printer is a data structure, and the term "operation" includes also functions like sine, tangent, inverse, rewind, backspace, ...

Only one thing is forbidden: a control statement. A terminal holon is a sequence of expressions to be evaluated. These expression evaluations produce results, but a terminal holon never makes a choice.

The expression previously written down isn't a syntax description. It is an abstraction of the general meaning of instructions written in a lot of languages. For instance, this expression abstracts all the following ones:

$X \leftarrow A * B + C$

print($Y * Z$)

$Z = \sin(\cos(2))$

backspace

for $X \leftarrow 1$ **step** 1 **until** 100 **do** $A[X] \leftarrow 0.0$;

(The last one isn't a control statement, it is a basic operation for an array data structure.)

We can imagine other expressions belonging to the abstract one:

List children of Node Tree (A.B.C)

$\text{sort}_{\uparrow}$(B) ($\uparrow$ meaning ascending order)

$\text{sort}_{\downarrow}$(B) ($\downarrow$ meaning descending order)

To sum up: the internal syntax is made up of a set of so-called "terminal languages," the nature of which depends on the program application areas. These languages contain only operative statements, and no control statement.

The kinds and number of terminal languages will change from one user to another, according to their specific interests. We will have terminal languages devoted, for instance, to:

- Integer Arithmetic
- Real Arithmetic
- Array Manipulation
- Character String Manipulation
- Text Edition
- Input-Output
- Graph Traversal
- Etc...

The holon system will include a list of "standard" terminal languages, plus some "special application oriented" terminal languages. A mechanical engineer would enjoy a "gear oriented" terminal language, etc...

Given the peculiar nature of the terminal languages, it is intended to define a "generating holon" which allows the programmer to define his own terminal languages. The "generating holon" will produce a small compiler for each defined terminal language.

Defining a terminal language

To define a terminal language, we must define the following elements:

1. A "name" (the designator): it is a symbol used to point out this terminal language among the available terminal languages. A terminal language name designates one and only one terminal language (for the same computer environment).

2. A set of "terminal data structure types": i.e., a finite list of names with correspondences to machine representation of the concept abstracted by the "type name." For instance: **Integer**: 4 bytes.

3. A set of "terminal language operators": i.e., a finite list of symbols which abstract some manipulations and alterations upon data structure(s). A description of the kinds of data structures upon which the operator applies and also a description of the operator application results must be furnished.

4. A description of the "apply" function for the language: this description explains the meaning of complex expressions involving many operators and data structure names. (For instance, precedence order rules.) This description implies also the definition of an "end of statement" symbol

It must be understood that a sensible design of a terminal language includes the definition of "data structure properties test functions." That means that, in order to write a powerful datatest, we must have the ability to test. For instance, the definition of operations: "lowbound" and "upbound" which we may use to determine the values (n, m) for any array.

Some syntax definitions

We return to the undefined syntax elements appearing in the "Statements block."

```
⟨terminal language designator⟩ ::= ⟨most used language designator⟩
      | ⟨special language designator⟩
⟨most used language designator⟩ ::= #
⟨special language designator⟩ ::= #⟨string⟩
⟨string⟩ ::= ⟨alphanu⟩⟨space⟩ | ⟨alphanu⟩⟨alphanu⟩⟨space⟩
      | ⟨alphanu⟩⟨alphanu⟩⟨alphanu⟩⟨space⟩
⟨alphanu⟩ ::= "an alphabet character or a digit"
⟨space⟩ ::= "space or blank"
```

(The last two syntactic definitions aren't BNF for the purpose of shortening the exposition.)

Examples of special language designator:

#GPH	standing for a graph language
#SUR	standing for a surveying language
#GR	standing for a gear computations language
#ARR	standing for an array alteration language

Etc. . .

What we call the "most used language" is in the mind of the users the terminal language which would be used the most frequently in terminal holons. This language isn't necessarily the same at all places using the Holon Programming Language.

$\langle$terminal language statement$\rangle$

We don't describe this part of the syntax, we assume that the reader can understand from the Holon terminal languages section what a terminal language syntax must look like.

$\langle$end of terminal body$\rangle$ ::= ##

Extending the character set

This section will describe a feasible extension of the concept of terminal languages designed by the programmer. This extension was suggested to me by reading some of Babbage's Paper published by Morrison.[24]

[24] P. Morrison et al. *Charles Babbage and his calculating engines: Selected writings by Charles Babbage and others*. Dover New York, 1961.

Babbage's language for technical drawing

Charles Babbage designed and used a language of his own to explain the working of the mechanical devices embedded in his calculating engines. What struck me reading Babbage's Papers was his clear distinction between a machine's static description (blueprint) and a machine's working description (statement in his language). Joining a machine's working formal description to a blueprint isn't usual.

Babbage explained that the need of this language appeared in proportion as the design of his engine progressed. It was the growing complexity of the calculating engine design which implies the need of the language describing the working of the machine elements. This may be viewed as an application of the principles expounded by Simon and Koestler.

They claim the need for a neat distinction between the world as sensed and the world as acted. This distinction ensures correct development of complex systems.

Now, looking to Babbage's Language, you may infer that with the help of a computer, it is feasible to do a lot of things with this syntax; for instance:

1. To use the working formal description for driving a moving display of the mechanical parts on a CRT-Device.

2. To expand a "macro" description of the working to the complete formal one.

But any attempt to implement systems like these is a still-born one. Why? Because the usefulness of Babbage's syntax depends on the use of a new set of small symbols, the reading of which can become as meaningful for a mechanical engineer as looking at notes of a score can be for a musician. The mechanical engineering oriented reader is urged to read Babbage's paper *On a method of Expressing by Signs the Action of Machinery*.[25]

[25] Ibid.

Considering that the previous example isn't unique, (for instance: think of the APL character set), our statement is: in many applications, computer-aided systems are cumbersome and awkward because the methods to extend the character set are cumbersome and awkward.

People can presently extend the character set only by the use of light-pen (for the input) and plotter or CRT (for the output). It is also possible to build character sets dedicated to an I/O device like the APL terminal, but this method is obviously unflexible.

In the next section, we describe a flexible device to input "personal" character sets.

A symbol generator

The device we describe will be a new combination of existing components. In spite of my lack of electronic knowledge, I think that "wiring" this device wouldn't be a nightmare.

To represent a symbol, we can use a pattern of dots. This Dots-Pattern method is already extensively used by many printing devices. We don't care about the way (impact, ink spitting, thermal, electrostatic. . .) we use to lay down the expected dots-pattern on the paper. We don't care anymore about the pattern shape (number of dots); we must have enough dots (perhaps 8×8) for generating fair copy of "continuously written" symbols.

Consider a keyboard composed of three sections:

1. A standard typewriter (or teletype) keyboard;

2. A set of keys with "undefined" characters;

3. A set of keys to manage the assignment of characters to keys of the previous set.

Each time we depress a key of the standard keyboard, a dots-pattern of the corresponding character appears somewhere (printed on a paper or lighted by LED (Light Emitting Diode)) and we assume that the binary-code of this character is recorded somewhere.

When we depress a key of the "undefined" character set, we can record a binary-code corresponding to the depressed key. This binary-code is independent of any dots-pattern. When this binary-code appears on an input or output device, this appearance can start a search for the corresponding dots-pattern in memory. When this dots-pattern is found in memory, its description is used to drive the dot-writing device.

The user depressed keys in the third key set to build up the content of the memory. For the input device, the support of the memory would be something like a magnetic card; each user introduces his own card upon which he can record his own character set.

When a user puts a recorded card into the device, the "undefined" keys will take "values" according to the memory card content. That can be displayed to the user by fitting into the hollow of each "undefined" key a matrix of LED Dots which will be turned on according to the memory card content.

The keys of the third set are used mainly to point out the coordinates of the dots which must be "activated" to form the symbol. The lighted dots would appear first on an enlarged dots-display to help the user to draw the symbol. Output devices will need memory to manage the mapping of the "undefined" characters into dot patterns. In the output character string, there will exist control commands to write content inside the memory of the output device. Commands such as: "The next 'X' bytes in this output string must write inside the memory."

Use in terminal language

The programmer defining a new terminal language can build a symbolic syntax that "tightens" the requirements of the application area. The new symbols can stand for operators or data type constants. (This use can simplify some input procedure, for instance in hotel reservation systems, the wishes of the customer can be input like the conventional ideograms used by this business.)

The set of the new symbols with their meanings—in terms of holons (procedures) and data values—will be a part of the terminal language definition.

Inner and outer syntaxes

The Holon statements are used to write the several program levels, all but the terminal ones where the programmer uses the terminal languages. The programmer can define his personal terminal languages, provided he fulfills the requirement that his new syntax corresponds only to an "operator grammar."

A program is completed when all the levels are expressed in terms of terminal languages.

As a matter of fact, there exists a clear partition between an outer syntax and several inner syntaxes.[26]

[26] M.V. Wilkes. "The outer and inner syntax of a programming language". In: *The Computer Journal* 11.3 (1968), pp. 260–263.

Data structure commands and modified syntax

This syntax description must be completed by the description of the "Data Structure Commands" and also of the "Modified Syntax." However, we delay these descriptions. The description of the semantics of these syntax elements will be more simple after the reading of the next section devoted to the representation of structured programs.

Structured Program Representation

A structured program may be represented by flowcharting,[27] but this representation does not visualize the several levels if we embed (in the flowchart) the levels in each other. A tree representation seems more adequate.[28] We have chosen a binary tree representation for a holon program.[29] This representation is used as a model of a program execution and also as a base for the implementation of the holon concept.

[27] Mills.

[28] Henderson et al.

[29] D.E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. First edition. Reading, MA: Addison-Wesley, 1968, pp. 332–333.

Binary tree representation

Figure 1 depicts a tree representation of a structured program (on the left) with its corresponding corresponding binary tree (on the right).

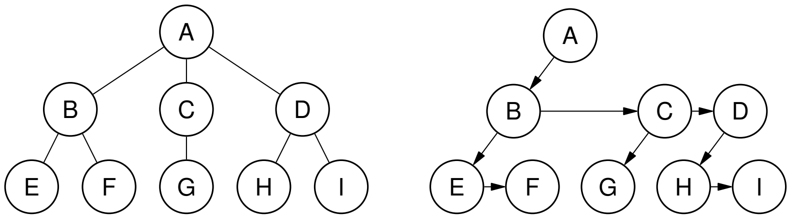


Figure 1: A structured program's tree and binary tree representations.

The contents of the nodes of these trees must be clearly understood: A node contains only a holon name, unless the holon is a terminal one (written in terminal language); the node corresponding to a terminal holon contains the holon name and the terminal instructions.

The holon tree of Figure 1 corresponds to the syntactic form:

A **begin** B; C; D; **end**

B **begin** E; F; **end**

C **begin** G; **end**

D **begin** H; I; **end**

[We temporarily use the "sequence" statement only.] Thus, for the binary tree we have the following rules:

1. A node contains a holon name of a terminal holon;

2. The left link points to the first statement of the holon body;

3. The right link points to the next statement (if any) in the holon body.

[It can be said that a left link is a graphical correspondence of **begin** and the right link is the one of a semicolon which is the connective symbol.]

Tree traversal and program execution

A program execution may be viewed as a tree traversal with respect to some order. The order must be chosen so that the execution of the terminal level instructions is performed in the correct order. For the binary tree, the traversal will be the so-called postorder one (or symmetric one).

This traversal is recursively defined by Knuth:[30]

30 Ibid.

```
if the tree is empty
  then do nothing
  else begin
    traverse the left subtree;
    visit the root;
    traverse the right subtree;
  end
```

We use the following "refinement" for the semantic level (holon) "visit the root":

```
if the root is a terminal holon
  then perform the contained instructions
  else do nothing
```

If in Figure 1, we assume that nodes E,F,G,H, and I are terminal holons, a postorder traversal will start the terminal holon executions in the correct order.

Of course, this binary tree model works only for the sequence statement of Chapter 5's syntax; but we shall develop the model in order to include the other holon statements. Nevertheless, the expansion of the model must respect the clear graphical representation of the levels that the binary tree allows.

Binary tree representation of the holon statements

We shall imagine that there exists a “textual index”[31] which is driven by the postorder traversal algorithm. When the “textual index” enters in a terminal holon, it starts the terminal holon statements.

[31] Dijkstra.

To simulate repetitive statements and conditional statements, we include in the binary tree some “switching” nodes which will be “switched” by boolean holon values. These nodes will force the “textual index” to re-traverse some subtrees or will block the way to some subtrees, according to boolean holon values.

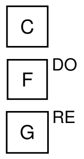
[The difference between this model and the flowchart will be the clear distinction of the levels.]

The model will be composed from the following types of nodes:

a The “empty node,” used to fix levelling inside the tree;

b The “holon name node,” representing a holon name, (The depicted symbol corresponds to a holon, the name of which is: A);

c The “boolean colon name node,” representing a boolean holon name; there are three distinct symbols according to the statement where the boolean holon name appears



d The “dispatching node,” used to implement a choice;

e The “repeating node,” used to implement a repetition.

if then else

Let the holon statement be:

if H **then** A **else** B.

The binary tree model is depicted in Figure 2.

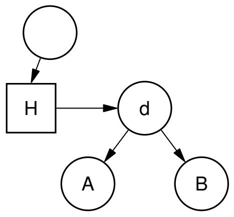


Figure 2: Binary tree model for an if-then-else statement.

In Figure 2, the holons defining the boolean holon H and the holons A and B must be linked to the corresponding nodes by left linked subtrees. H, A, and B are not necessarily terminal holons.

A postorder traversal of the tree in Figure 2 implies the following order for the visited nodes: H, A, d, B, “empty node.”

(When we say that a node is visited, that means that the left subtree rooted by this node has been traversed. During this traversal, the terminal nodes are executed in the correct order.)

The visit of H gives a value to the boolean holon H (unless there is a programming error). The value is used to modify the content of the dispatching node links.

A dispatching node may be viewed (see Figure 3) as a node with four links:

- Two structural ones which can never be modified during program execution
- Two operational ones which are modified by the boolean holon value.

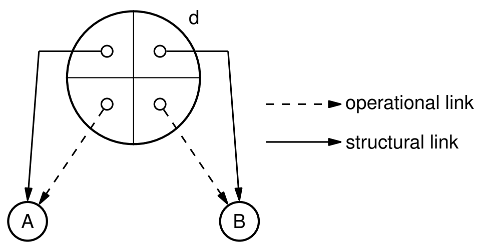


Figure 3: A dispatching node.

One of the operational links is "broken" according to the boolean holon value.

"visit the node" for a boolean holon will look like:

```
datatest has a value been given to the boolean holon? end
begin
  operational links ← structural links;
  if boolean holon value = true
    then right operational link ← nil
    else left operational link ← nil;
  erase boolean holon value;
end
```

The result of the previous operation will be the "pruning" of one of the subtrees of the labelled -d tree.

This refers to the dispatch node, i.e., one of the branches is pruned.

do while

Let the holon statement:

do A **while** C;

be modeled by the binary tree depicted in Figure 4.

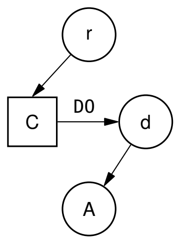


Figure 4: Binary tree model for a do-while statement.

In this case, we use a "repeating" node. This kind of node has two structural links (the right one points to the tree mapping the holon next statement; it is not depicted in Figure 4) and one left operational link modified by the boolean holon.

The symbol "do" on the side of the boolean holon is used to select the adequate "visit the node" refinement.

This refinement will be:

```
datatest has a value been given to the boolean holon? end
begin
  d-left operation link ← d-left structural link;
  if boolean holon value ≡ false
    then
      d-left operational link ← nil;
      r-left operational link ← nil;
    else do nothing;
  erase boolean holon value;
end
```

repeat until

Let the holon statement:

```
    **repeat** B **until** Y;
```

correspond to the binary tree shown in Figure 5

The refinement of the "visit the node" for the boolean holon:

```
datatest has a value been given to the boolean holon? end
begin
  if boolean holon value $\equiv$ true
    then r-left operational link $\leftarrow$ nil;
  erase boolean holon value;
end
```

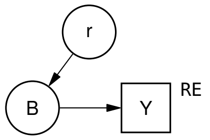


Figure 5: Binary tree model for a repeat-until statement.

case statement

Let the holon statement:

```
case M then Z
  eor N then W
  eor else T;
```

correspond to the binary tree in Figure 6.

The "visit the node" refinement used for the boolean holon in this statement is similar to the one used for the *if-then-else* statement.

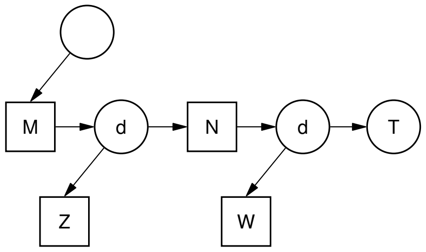


Figure 6: Binary tree model for a case statement.

Full holon representation

A full holon contains a datatest, an invariant and a sequence of holon statements. For instance:

```
N
datatest ... end
invariant ... end
begin $S_1$; $S_2$; $S_3$; end
```

The S_i are holon statements (not only holon names). The binary tree is displayed in Figure 7.

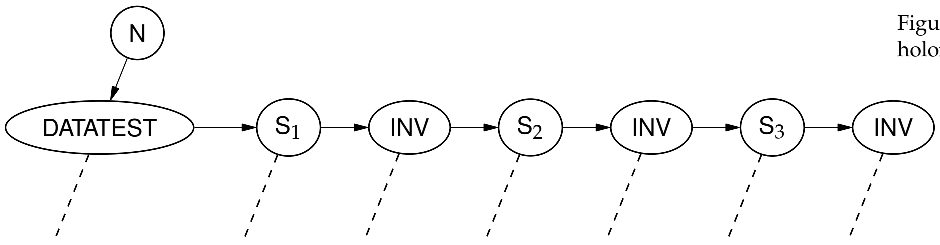


Figure 7: Binary tree model for a full holon.

The dashed lines are links to the defining holon trees for each of the holon components.

The nodes S_1, S_2, S_3 are the roots of left subtrees corresponding to holon statement trees. For instance: if S_1 is a holon name, the pending link points to the definition of this holon in terms of datatest, invariant and holon statements; if S_i is a **do while** statement, the node marked S_i in Figure 7 is the "repeating" node which "roots" a *dowhile* binary tree (see the do while section); etc.

[The description of the datatest and invariant representation is given in one of the appendices.]

The postorder traversal algorithm revisited

We can rewrite the refinement of the holon "visit the node":

```
case node is a terminal one
  then execute the contained terminal language statement
eor node is a repeating one with left operational link $\neq$ nil
  then traverse left subtree and revisit this node
eor node is a repeating one (with left operational link=nil)
  then left operational link $\leftarrow$ left structural link
eor node is a boolean one
  then operate according to the boolean type (do, re, empty)
eor else do nothing.
```

It must be understood that this postorder traversal algorithm can be used to "interpret" the program. The textual index access to and exit from a holon can be reported to display the progress and the history of the computation.

[*Remark*: In this section all the algorithms were described according to the rules of the holon syntax.]

Mapping the Binary Tree into an Executable Program

It must be clearly understood that the syntax of the holon programming language is designed in order to write well-structured and well-documented programs. A holon name can contain a full explanation of its abstracted operation.

We shall explain in this section how this well-structured and well-documented program will be mapped into a sequence of efficient machine-code instructions. In this instruction sequence, the program structure will not appear explicitly. But the code generation will be driven by the program structure.

Each holon will make use of its "tendency to integration" during the code-generation operation.

Basic integration

Let us consider a holon program with the following properties:

a. The program is complete: i.e., all the "leaves" (the pending nodes) of its binary tree representation are holons written in terminal languages.

b. The binary tree representation of the program is a minimal one: i.e., the definition of a holon appears only in one place in the tree; thus a duplication of a holon name does not imply a copy of the subtree rooted by the node with the same name.

c. The binary tree representation of the program is a left-definition one: i.e., the subtree defining a holon is "attached" to the leftmost node labelled with this holon name; this means that during a postorder traversal using the *structural* links, the textual index has access either to the subtree defining a holon or to a holon, the defining subtree of which has been already traversed.

[The traversal uses the structural links to avoid any subtree pruning or retraversal.]

We suppose also that:

a. There exist algorithms to generate machine code for each terminal language.

b. We can assign storage addresses for the data structures, and these storage addresses are used in the machine code instructions.

We have the choice to implement each holon: either as a procedure (closed routine) or an inline inclusion (open routine). The choice can be based upon a series of criteria:

1. The length of the machine-code generated for the holon.

2. The frequency of the holon name appearance in the program.

3. The relative cost of a procedure call for the used computer.

These three criteria are the expression of the computer architecture influence. They are put together in some trade-off function.

Nevertheless, the choice resulting from the trade-off function may be modified in some cases, according to the context of a holon name appearance in the program. This leads to a new criterion:

4. The kind of instruction which surrounds a holon name. (For instance, after a **do** an open routine can be used instead of a routine call.)

But this last criterion has only a local influence; i.e., the first three criteria are used to make a global decision about the implementation unit for a holon, but locally, the fourth criterion may modify the choice.

Recursive holon

The criteria of the previous section can be applied only to non-recursive holons. If a holon is a recursive one, it needs a special implementation mechanism.[32] Before applying the criteria, we must "pin" the recursive holons.

[32] E.W. Dijkstra. "Recursive programming". In: *Numerische Mathematik* 2.5 (1960), pp. 312–318.

In the conventional languages, the operation is not an easy one. Generally, the system is designed to avoid this problem either by assuming that all procedures may be recursive (ALGOL-compiler) or by assuming that the programmer will specify the recursive procedures (PL/I syntax), the latter choice assumes that the programmer is able to specify direct- and cross-recursive procedures.

The only algorithm that I know to detect the recursive procedure is the transitive closure one (see Wegner, section 4.7.6).[33] This algorithm is simple, but when you read the description with an example, you

[33] P. Wegner. *Programming Languages, Information Structures, and Machine Organization*. McGraw-Hill, 1968.

may wonder how it can be manageable if the number of procedures is large.

We shall show that if a program is a structured one, we can design a simple algorithm to detect recursive holons. [The difference between this algorithm and the transitive closure one is that it seems insensitive to the number of procedures, with respect to deciding if a procedure is potentially recursive or not.]

We suppose that the program has a binary tree representation which is a minimal one (see Basic integration). We assume that we have a list of each holon name with a list of its appearances inside the tree. Appearances are pointed out by a so-called "Dewey Binary Notation for Binary Trees."[34] Detecting recursivity will become a simple bit-strings comparison.

[34] Knuth.

Suppose a holon A define by Figure 8.

The holon names appearance list is:

A:	1, 101010
B:	101
C:	10101
D:	10, 1010

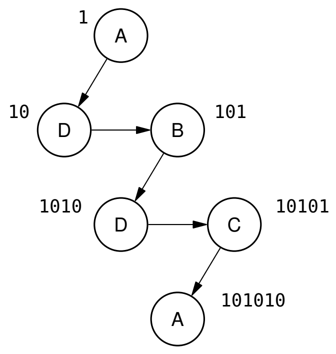


Figure 8: A binary tree labeled with Dewey binary notation.

Is "D" recursive?

"D" appears at nodes 10 and 1010; bit-string comparison (starting from the left) shows that the first dissimilar digit is a '1'. This means the second appearance of D belongs to the right subtree of the first D. Thus, D is not included in its definition (a definition is always a left subtree). Thus, D is not recursive.

Is "A" recursive?

"A" appears at nodes 1 and 101010; bit-string comparison shows that the first dissimilar digit is a '0'. Thus the second A is included in the left subtree of the first A. Thus, A is recursive. But this recursivity implies that B and C are recursive, because in a postorder traversal, when the textual index has access to the second appearance of A, the traversal of the B and C subtrees is not complete.

How do we detect cross-recursivity (like B and C)?

We take the bit-string identification of the second appearance; for A, it is: 101010. And we extract all the substrings (starting at the left), the first bits of which match the bit-string of the first appearance and the last bit of which is followed by a '0'.

For the example, this procedure will produce the set of bit-strings:

1	which corresponds to A
101	which corresponds to B
10101	which corresponds to C

[This bit-string comparison is a back-tracking from a descendant to his ancestors, in a family lineal chart.]

Remark: When we apply the last bit-strings comparison for a more complex holon tree, where there are "repeating," "empty" and "dispatching" nodes, these nodes also have Dewey labels. Of course, they must be discarded from the computed substring set.

The "tendency to integration" feature revisited

At this point, it is worthwhile to return to the holon's tendency to integration.

In the author's opinion, rejection of the structured programming principles is often based upon efficiency of the running program. People can argue that structured programs, even if they work correctly, will look like laboratory prototypes where you can discern all the individual components, but which are not daily usable. Building "integrated" products is an engineering principle as valuable as structuring the design process.

In civil engineering design, it is presently a mandatory concept (that is known as the "Shanley Design Criterion": collect several functions into one part). We may see this principle working by comparing "web" iron structured bridges, built in the eve of this century, to integrated pre-stressed concrete bridges.

Another example can be found in the rocket design: if you make a cross-section of, for instance, the German V-2, you find, going from the outer to the center of the section: external skin, structural rods, tank wall. . . ; if you cut across the Saturn-B moon rocket, you only find an external skin which is at the same time a structural component and the tank wall. [Rocketry engineers have used the "Shanley Principle" thoroughly when they use the fuel pressure inside the tank to improve the rigidity of the external skin!]

In modern engineering, the implementation of a system is a modular one when there is fear of "wear and tear" upon some parts. This kind of fear does not exist for a computer program, what happens is the need of "rewriting" some components because we understand the problem better or because the nature of the program application changes.

With the holon concept, a holon can be modified; but then, we rebuild (automatically) all the program in order to have a well-integrated program.

[In the appendix, we give another example of design integration.]

User's View and System Implementation

The holon system has two components:

1. the analyzer, and
2. the synthesizer.

The analyzer

This component collects the holons written by the programmer and builds the binary tree representation of the program. This binary tree is the "logical" support of the program and it is used to help the programmer to design the program. For instance, it permits program interpretative execution. The programmer can delete, append, change holons; his program binary tree is kept in long-term storage as long as the program design is in progress.

The analyzer checks the syntactic correctness of the programmer's holons. If there are several programmers, holons can be protected from illegal alterations.

The analyzer outputs a binary tree in a standard format which is a standard one for input to a synthesizer. The analyzer points out the recursive holons.

The synthesizer

The input of a synthesizer is a complete binary tree for a program, the terminal holons of which are all written in terminal languages. It is forbidden to synthesize a holon program where there are undefined holons.

The synthesizer produces a machine-code version of the program, using a code generator for the used terminal languages and a "synthesis strategy" to choose the implementation unit for each holon.

The synthesizer chooses the code generator and the synthesis strategy according to information given by the programmer about the computer which will run the program. (This is only a library retrieval program.)

The output of the synthesizer is a machine-code version of the program, ready for execution on a computer.

System control language

The programmer uses a language composed of short command reserve words to control the analyzer and the synthesizer. A description of the syntax of such a language is given in a following chapter.

The Initialization

The Data Structure Command Syntax is an attempt to introduce levels in the data structure as we have levels for the operations.

We shall first expound the different programming problems which have led to this syntax design; next, we shall describe the syntax in a following section.

Abstract data structures

We want to allow the programmer to use "Abstract" data structures; i.e., at some level in his program, he declares data structures belonging to an abstract type.

An "Abstract Type" is a data structure type, the name of which is meaningful in the programmer's mind; but, an abstract type doesn't automatically imply a machine implementation.

Example: Suppose we are designing a chess playing program; and we use extensively the concept of a "chess-board pattern," i.e., a pattern of the chess elements. We design a set of operations like: "Evaluate Chess-Pattern," "Strengthen Chess-Pattern," etc... At this level in the program design, the machine representation of this pattern is irrelevant but we need the concept of an "abstract" data structure: Chess-Pattern. And according to the operation which we shall perform upon a chess-pattern, we will design (refine) the abstract data structure to a machine representation. (The point is that the obvious implementation of a chess-board by an array would probably be a mistake. For instance, it seems that chess-players memorize the chess-board in the form of a list structure.[35] Another interesting example is a checker program designed by Strachey; the method used to select the checker-board representation is a revealing one.[36])

Creation and refining are allowed in the Holon Programming Language by commands *create* and *refine*. (See further).

[35] H.A. Simon. *The Sciences of the Artificial*. The MIT Press, 1969.

[36] C. Strachey. "System analysis and programming". In: *Scientific American* 215.3 (1966), pp. 112–127.

Data structure declaration and scope

We have adopted the declaration and scope used in ALGOL; nevertheless, in top-down programming the ALGOL rules must be extended.

When we are involved in the refinement of some operation (level), the need of new variables appears. But, generally the newly-introduced variables have a scope larger than the scope of the level where the need of introductions appears to the programmer.

Thus, the design of some inner levels implies the alteration of the declarations of an outer level. (An example of this programming behavior can be noted in programming by stepwise refinements.) At the International Computing Symposium '73 at Davos, Prof. Wirth used for displaying the methodology the following device: Each refinement is written in one color on a transparency, and by stacking the transparencies, we obtain the final program. But, what is revealing in the final multi-colored program is that there are some colors interspersing and not a genuine embedding of colors. I suppose that the program will be reprinted in the Symposium proceedings. Another example may be seen in Knuth's letter 1, p 7.[37]

[37] D.E. Knuth. *A Review of "Structured Programming"*. Tech. rep. STAN-CS-73-371. Stanford University, Stanford, CA: Stanford Computer Science Department, June 1973.

The problem with this kind of programming behavior is how do you manage alterations of the program? Obviously, if you decide to delete a level, you must delete all the level operations and carefully scan all the "side" modifications implied by this level.

In HPL, it is possible to write declarations, the scope of which is not limited to the scope of the holon where the declaration is written. The programmer declares a variable and specifies the scope by naming an outer holon. The scope of this holon will be the scope of the declared variable. This command is the **declare at level** command.

HPL: Holon Programming Language

This command works only if the holon which contains the declaration statement in its body belongs to the subtree of the holon which defines the variable scope.

In Figure 9, if we are using a **declare at level** command inside holon G, the only levels which can appear in this command are D, C, A. To declare a variable at levels B, E, F, and H is not allowed because variables declared at these levels can't appear in statements written at level G. The analyzer checks the correct use of this command.

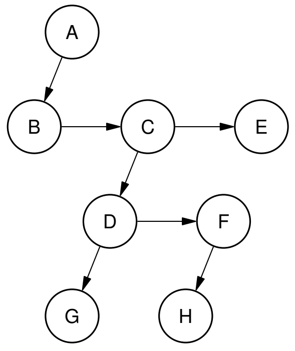


Figure 9: A holon with different scopes.

Now, let us tackle a new problem: suppose in Figure 9 that we are designing the body of the holon H. Suppose that we need to use a variable declared local to holon G (The holon G was designed before holon H. The meaning is that a "mistake" was made during the writing of holon G: We did not understand that the scope of a variable must be larger than the scope of holon G. (It is not exactly a mistake but a lack of program understanding.) We shall use inside holon H, the **same as** command, i.e., to declare a variable the same as another variable in another holon.

The analyzer will move a “virtual” declaration of the variable to a holon level which encompasses the two involved holons. (In Figure 9, this holon level would be level C.)

The declaration at level C is a “virtual” one because there is no modification of the body of holon G. The alteration of the variable scope is local to C; it appears only if H is in the G “context.” For instance, see Figure 10; in this holon, there are two appearances of holon G. The **same as** command must in this case qualify the G involving holon G (for instance, naming G as G **in** D). But the “virtual” declaration at holon level C has no effect upon the declarations of G **in** M.

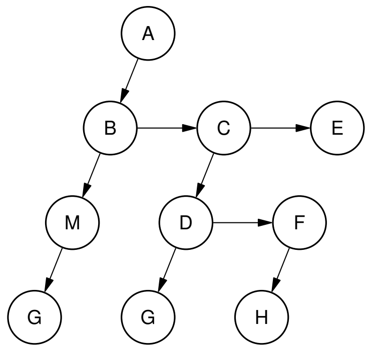


Figure 10: A holon with a “virtual” declaration.

The initialization

Watching people writing structured programs, you will notice that most of the time, they start the writing of a level like:

Level XYZ:

initialization;

do something;

do another thing:

⋮

They will agree that they write the sublevel “Initialization” because it may happen that it may be used. But really, when they lay down the level name upon the paper, they don’t know.

If you plan to read the content of this sublevel before the others, the reading will be meaningless. As a matter of fact, this initialization level conglomerates a lot of operations which must be there in order to respect the sequential execution of the program; but why they exist must be discovered by the reading of the other levels’ bodies.

In our opinion, as the programmer makes choices during the design process, parts of the “initialization” are implied by these choices. Thus, there exists a one-to-one correspondence between each part of the initialization and a design choice. Furthermore, in our opinion, the design choices which involve initialization are mainly data structure choices. The result is that in HPL, after a declaration you can write a reserved word (**initial**) followed by a holon name. The execution of this holon name is performed according to the scope assigned to the variable. This is possible even with **declare at level**.

Restricting access to data structures

We will allow the programmer to restrict access to some data structure types to a limited set of holons. Variables of the restricted access type can be used only by the holons, the names of which appear in the set.

It is also allowed to restrict access for a variable. If this variable belongs to a restricted access type, the set of the holons which can use this variable must be a subset of the holon list for the restricted access type.

Example:

create 1 file **restricted** read; write; purge **end**
declare top-secret: file **restricted** write; purge **end**

The checking of all the restrictions is made by the analyzer.

Local refinement

The refinement of an abstract data type may be a local process. For instance, we come back to the abstract concept of a "chess-board pattern": the way to refine this concept in terms of terminal data structures depends upon the functions we want applied to this abstract concept.

The programmer designing the holon which will choose the next movement will maybe use a list representation of the chess-board; but, obviously the choice will seem unacceptable for the programmer writing the holon which prints a picture of the chess-pattern, this latter programmer will be very happy to use a simple array. Thus, it seems to me to be useful to allow "local to a holon" refinements. In some holons the global refinement is switched to a local one.

(In a previous report, I had called that a "dedicated refine.")

Remark. In the first implementation of HPL, I don't plan to implement "Local Refinement," because I have not found an acceptable answer to the check involved by the local refinement. In short, if locally you change the way an abstract type is refined, that implies that variables, the scope of which encompasses the two refinements, must be "trasnlated" from one refinement to the other. You may assume that the programmer must provide the translating holon at the right place, but it seems more sensible that the analyzer requests these translating holons and decides itself where they must go. Think of two programmers writing in parallel two subtrees of a holon where they make local refinements; the analyzer would be able to output the correct "interface holon" specifications.

Data Structure Commands Syntax

We shall describe the syntax of:

1. Type descriptors;
2. Variable names;
3. The set of commands to build the data structure environment of a program.

In order to shorten the text, we don't use a full BNF description. Most of the syntactic elements will be described using plain English.

Type descriptors

The type of a variable can be:

1. either one of the terminal types, i.e., a type belonging to one of the terminal language; or
2. a structured type.

A structured type is an agglomeration of terminal types structured like a tree. In this tree, only the "leaves" are terminal types. The programmer can give names to the nodes; the names correspond to the rooted subtrees and they define new types. The several levels of the tree are pointed out by the use of numbers. Each time we step down one level in the tree type, the level number of the new level must be one unit greater than the level number of the previous level.

Example:
1 Human-Body

2 Head	3 Eyes	4 Color	**character-string**
	3 Hair	4 Number	**integer**
2 Arms	3 Number	**integer**	
	3 Length	**integer**	
2 Legs	3 Number	**integer**	
	3 Length	**integer**	
2 Trunk	3 Length	**integer**	
	3 Type	**character**;	

The names of the type can be composed of characters and the dash symbol (-).

It is possible to define an Array of Structures, i.e., in the defined type, a subtree appears several times at the same level. In this case, only one subtree is described associated with a subscript. The subscript is an integer or an integer parameter. The subscript is used in declarations to define the number of subtrees of this type; it is also used to select one subtree when we make a reference to a variable.

Remark. (1) It is said that the subscript is an integer or an integer parameter (not an integer variable). When we design a new type, we point out only the fact that some subcomponents of the tree can be repeated.

Example:

```
1   Matrix
        2   Row (%R%)
                3   element-in-row (%E%) real;
```

In a declaration, values must be given to the parameter; but the given value can be the name a pre-declared integer variable.

Example:

declare A, B, C: Matrix($R = N$, $E = M$);

declare Z: Matrix($R = S$, $E = 4$)

(2) The introduction of Arrays complicates a little bit the working of the **create** command (discussed later).

Variable names (or identifiers)

A variable name is a string of characters and dash symbols; normally space between two words is forbidden. The length is not limited.

When the type of the variable name is a structured one, the programmer can also access the subcomponents of the structured type. That is made by the use of the reserved symbol **of**. This symbol is used to link components from the leaves to the root of the structured type. Nevertheless, there is no requirement to give all the links, but the described linkage must be unambiguous.

For instance:
After **declare** Dumont, Einstein, Leon: Human_Body; we can use the following variable subcomponent references:

Number **of** Arms **of** Dumont

Type **of** Leon

Color **of** Hair **of** Einstein

These three variable references are unambiguous. It must be noticed that only the last word of a variable subcomponent reference is a variable name; the others (except of **of**s) are type descriptors. The first word stands for a terminal type, and the analyzer can check if the forthcoming operation is a legal one for this terminal type.

Commands

The following commands will be described:
⟨create⟩, ⟨refine⟩, ⟨declare⟩.

⟨create⟩

The ⟨create⟩ command is used to create new types in the form of structured types or of association of a name to a terminal type. An example of the latter is:

create color: **character-string**;

Here "color" is not a variable, it is a data type. Normally some restrictions upon the kind of character-string for "color" would be added to the create command;[38] nevertheless, at this time we have not found a suitable syntax for this domain restriction (suitable syntax for the programmer *and* the analyzer.

[38] C.A.R. Hoare. "Chapter 2: Notes on data structuring". In: *Structured programming*. Academic Press, 1972, pp. 83–174.

In the former form (creation of structured types), it is not required that the programmer writes the structured type thoroughly (from the root to the terminal type leaves). The programmer can define abstract types.

Example: **create** 1 chess_board;

The number "1" indicates that it is a structured type, but it is an abstract type.

The semantics of this command can be explained like this: when we start the writing of a program, the analyzer is aware that the legal type set is the "union" of the terminal types of the terminal languages. Using the create command, we can rename terminal types with more "convenient" type names for our program, or we can really "stretch" the type set by "creating" structured types or abstract structured ones. An important fact is that only the root of a structured type is added to the type set. If one of the Type descriptors is preceded by the reserved word **create**, it becomes an example of structured type creation, but the type set is augmented by only one element: human_body. (It will be shown that a **declare** command must associate a variable name to a type belonging to the type set.)

⟨refine⟩

The ⟨refine⟩ command is used to transform an abstract structure step-by-step into a fully-defined structure. The command syntax is:

```
refine ⟨list of abstract structures⟩ by ⟨type descriptor⟩;
```

It is possible to apply the same refinement to several abstract structures; that is the reason for the syntax component ⟨list of abstract structures⟩.

The programmer must write a level number '1' for each element appearing in the list, whatever the actual level number of the abstract structure element. The level numbers of the refining types (which follow the **by**) are level numbers relative to the level number '1'.

Example:

Suppose we have created the following abstract type:

```
create 1 chess_board;
```

Now, we made a first refinement:

```
refine 1 chess_board by 2 before_the_move
                        2 after_the_move;
```

Further in the program, we perform new refinements:

```
refine 1 before_the_move, 1 after_the_move by
            2 white_positions   2 black_positions;
```

At this time, the chess-board data type becomes:

```
1 chess_board
  2 before_the_move
    3 white_positions
    3 black_positions
  2 after_the_move
    3 white_positions
    3 black_positions;
```

The relative level numbers can be used to add components at one level; for instance, suppose we have the following command:

```
create 1 chess_board
            2 before_the_move;
```

Using the refine command, we write:

```
refine 1 chess_board by 2 after_the_move;
```

The following rules must be respected (to avoid rejection of the command by the analyzer): [we write the syntax like "**refine** A **by** B;" in order to shorten the rule descriptions.]

1. Elements of the A-list must have appeared before in the program in a **create** command or in the B-list of a **refine** command. The "before" must be understood as: in a Holon already "swallowed" by the analyzer.

2. The refinement must not be a recursive one.

A data structure is fully defined when all the refinements have led to terminal types.

Refinement of a data type can have a scope; i.e., that locally to some specified holons, a type descriptor refinement will be overlaid by a local refinement. This local refinement can be restricted to some variables. The syntax of this command looks like this:

Inside ⟨holon names list⟩
    [**for** ⟨variable names list⟩] **refine** A **by** B;

Example:

Inside print chess configuration
  refine 1 before_the_move, 1 after_the_move **by** B;
    2 **character array** [1:8, 1:8];

Outside the "print chess configuration" this refinement does not apply.

Note: In the first implementation of the holon language, we don't plan to implement the Local refinement.

⟨declare⟩

The semantics of a declaration are quite simple: It makes an association between a variable name and a type descriptor. But, it also implies the definition of a scope for this variable.

In conventional languages, the scope of a variable is determined by the position of the declaration of this variable amid the program text. In HPL, the rule is: A variable declaration appears where the need for this variable appears for the first time; and the declaration command is designed to explain what is the scope of the variable.

The different forms of the declare command are:

(a) The "local declare" or "simple declare." This command means that the scope of the declared variable is similar to the scope of the holon which contains the declare command.

Example:

declare Spassky, Fischer: Chess_Board;

[The type descriptor can be an abstract type; and this declaration can appear in the program even if the refinement of the abstract type to a structure of terminal types is not finished.]

The syntax does not require that the declarations are put together in front of a holon unit. They may appear anywhere between two holon language statements; the analyzer will gather the declaration in due place.

The type descriptor appearing in a "declare" statement must have appeared in a "create" earlier in the program.

(b) The "at level declare." The logical reason for this command is expounded in Data structure declaration and scope.

An example of this command:

declare Einstein: human_body **at** process_input level;

["process input" is a holon name.]

The semantics of this statement require that the holon, at the level of which the declaration is made, belongs to the "ancestors" of the holon where the command appears.

(c) The "same as declare." The logical reason for this command is expounded in Data structure declaration and scope. The semantics are also explained in that section.

Example:

declare blackboard:
  draft **same as** scratch **in** trial first solution;

This example assumes that "draft" is a type descriptor, and that "scratch" is a variable which exists in the holon called "trial first solution." The type descriptor of the "scratch" variable must be the same as the one of the declared variable; this is checked by the analyzer. [It is required that the variable exists at the level of the holon, the name of which follows the reserved word **in**. It is not required that the variable is declared at this holon level.]

Any of the three forms of declaration can contain an **initial** command. The following syntactic unit must appear before the connective symbol of the declaration:

initial ⟨holon name⟩

Example:

declare Fischer: chess_board **at** starting game **level initial**
  put chess set;

The "restricted" option

The "restricted" option is introduced before the connective symbol of either a **create** if we want to restrict access for data types or a **declare** if the restriction is for a variable.

If there is more than one holon in the restriction list, the end of the command must be specified by an **end**. See examples in Restricting access to data structures.

The ⟨value⟩ option

It was said in the Boolean holon section that a boolean holon may have access variables by "value." In a boolean holon name, the variables which we desire to be accessed by "value" are pointed out by writing the reserved word **value** before the identifier left bracket.

Example: Is matrix %A% with bounds **value** %N% **value** %M% a square one?

This concludes the informal description of the "Data Structure Commands." In the next chapter, the data structure organization of a holon program will be described.

Data Organization in Holon Programming

In this section, we shall explain how the data structures are introduced inside the holon tree. The goal is to check if each holon appearance in a tree meets a suitable data structure environment.

Data structure ordering

A holon has access to two kinds of data structures:

(a) Local data structures

(b) Non-local data structures.

The elements belonging to these sets are issued from different kinds of declarations.

We can order the local data structures from holon X:

1. Declared in the holon X body;
2. Declared from holons inner to X;
3. Induced at the holon X level by **same as** declarations.

We can order the non-local data structures for holon X:

1. Declared in an "ancestor" of X;
2. Declared non-local from holon X (by a **declare at level**);
3. Produced by a **same as** declaration.

The first holon of a program has only local data. A data structure "non-local" to a holon must appear "local" in an ancestor of this holon.

The "initial" operation

Data structures, the declaration of which is followed by the "initial" option, generate links to the initial holons corresponding to these options.

For instance, the following holon text leads to the binary tree representation in Figure 11.

```
A
begin B; declare M; C;
        declare N initial NI;
            D; declare Q initial QI;
end
```

[The test is a shortening of another one; B, C, D, NI, QI stand for holon names; the type descriptors of the declared variables are omitted.]

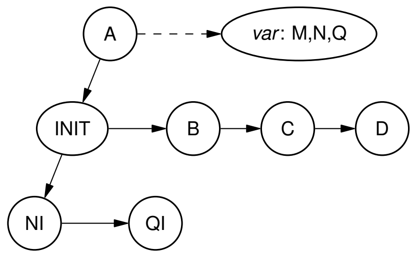


Figure 11: A holon with data structures declared after the "initial" operation.

Permanent and non-permanent parts of a holon

We call the permanent part of a holon:

1. The holon body,
2. The local data structures set.

We call the non-permanent part:

1. The non-local data structures set.

The permanent part has this name because whenever we write the name of a holon inside a holon language statement, this permanent part can be attached to the binary tree.

The non-permanent part describes the environment required by the holon; the description of the environment is a list of variable names with a type descriptor. The holon requires finding the same list (except variable names which can be different) in any place where its name appears. The name non-permanent comes from the fact that the variable names can change.

Remark. The checking of the consistency of the environment with the permanent part is really a datatest operation; but this checking can be performed completely by the analyzer (at "compile-time").

Operations on a program structured by holons

The operations that a programmer can do during program design are:

1. To define a new holon;
2. To insert in a holon statement an already existing holon which may be:

   (a) Fully defined (all its pending nodes are written in terminal languages);
   (b) Partly defined;

3. To write refinements (definitions of inner holons) of a holon, the name of which appears in more than one place in the holon tree;

4. To delete from the tree:

   (a) A holon name appearance in a holon statement;
   (b) A holon definition (without "erasing" the holon name);
   (c) A holon (name and definition).

Use of the non-permanent part of a holon

When any of the previous operations occurs during the design of a program, the analyzer uses the information gathered in the non-permanent part to check the environment state required by the operation. The analyzer reports to the programmer the reasons why the environment can preclude the completion of an operation.

Each holon name node of the binary tree is linked to a non-permanent part description which is used by the analyzer.

Running the Synthesizer

We shall expound by the use of a simple example how a synthesizer works. Suppose the holon program, depicted in Figure 12. We use a triangle to point out a holon written in terminal language.

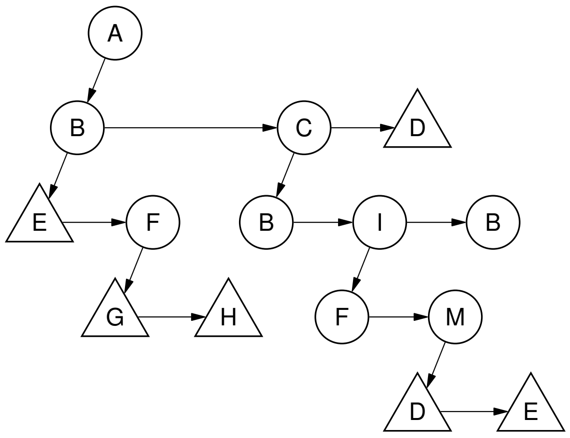


Figure 12: A holon program whose triangle-shaped nodes depict holons written in a terminal language.

The synthesizer generates the integrated program during two tree traversals. These traversals will be described by a textual index pointing out the visited nodes.

The first traversal is a postorder one. During this traversal, the synthesizer generates intermediate code for the terminal holons. Moreover, it evaluates a trade-off function, the parameters of which are the length of the intermediate code, the length of an eventual procedure call for this holon, the frequency of the holon in the program, and the kind of instructions surrounding the holon. This evaluation decides the nature of the code integration: “off-line” code (procedure) or “in-line” code (macro).

For instance, in Figure 12, we will have the following sequence of operations (the letters point out the visited nodes):

1. E : Generate intermediate code;
   Trade-off function result: off-line (we suppose that).
2. G : Generate intermediate code;
   trade-off: in-line.

3. H : Generate intermediate code;
   trade-off: in-line.

4. F : Trade-off: in-line (given F appears two times in the program, this choice is possible if the sum of the code lengths of G and H is small).

5. B : The intermediate code for B is composed of the calling sequence of procedure B and the code for F. Given B appears three times, we suppose that the trade-off result is: off-line.

6. B : No-operation because the choice is already made.

7. F : No-operation.

8. D : Generate intermediate code;
   Trade-off: in-line.

9. E : Generate a calling sequence for E.

10. M : Trade-off: in-line.

11. I : The trade-off gives immediately: in-line because “I” appears only one time, thus the length of its code is irrelevant.

12. B : No-operation.

13. C : Trade-off: in-line. (C appears only one time).

14. D : No-operation.

15. A : We can start the second traversal.

The second traversal is a preorder one. The synthesizer will generate the final machine code using the information of the permanent and non-permanent data to allocate storage and define the addresses or the references through a calling sequence.

In the example, the first visited node (after the Node A) is node B. At this point, we know that B is off-line, and we generate a calling sequence for B. After that, we traverse the left subtree of B, building the code of the procedure B. This code starts by a calling sequence of the procedure E, followed by the code for this procedure. Next, the implementation of procedure B resumes by the generation of the in-line machine code for F. Of course, code to jump around the embedded procedures in generated.

The linear sequence of code generated for the holon A will look like:

$$c(B) \quad B : [\, c(E) \quad E : [\,]GH\,] \quad c(B)\; GHD\; c(E)\; c(B)\; D$$

Meaning of the symbols:

$c(X)$ means call of procedure X;

X: Points out the position of the code of procedure X, this code is included between the two [ ];

Y: Standing alone (not followed by a colon symbol), means an in-line coding of the macro Y.

The two appearances of D in the symbol string don't mean that the generated code at these places is similar; because the addresses used in the instructions can be quite different.

Memory for the static data structures is allocated in a pseudo-stack with overlapping of the data according to the scopes of the declarations.

The dynamic data structures (for instance, data involved in the recursive procedures or data with dynamic bounds) are implemented by a stack.

The synthesizer produces (during this second traversal) the machine-code-translation of the "control switches" used by the holon statements. The behavior of the integrated program is driven by genuine machine jumps and no more by the postorder traversal model used during the analysis.

Remark. It must be noticed that the symbol string used to represent the generated code contains only terminal holon names or off-line holon names. All the other names used to indicate the program levels disappear.

The Modified Syntax

In this section, we shall describe some modifications to the syntax of the holon language. These modifications don't fundamentally change the syntax; this may be viewed as some alterations made to ease the handwriting of a holon program. The relevancy of these changes depends upon the kind of equipment used to submit programs to an analyzer. These modifications are for people using a card input system. The content of this chapter is irrelevant if you use a time-sharing system with a CRT display and some text editing program.

The reserved word "etc"

The holon syntax requires that you write the holon name followed by the definition (holon body) when you define a holon. Holon names can be very long in order to be fully explanatory text.

The reserved word **etc** permits one to (instead of writing a full holon name) write only the beginning of the name followed by the word **etc** and the parameters if there are any.

For instance:

Invert matrix %X% with bounds %N% %M%

can be written

Inv **etc** %X% %N% %M%

There must be enough letters in the abbreviation to avoid ambiguities. In the example, if the program contains several holon names beginning by "Inv," the submitted abbreviation is rejected by the analyzer. In the output document, the input abbreviation is expanded.

The text option

The programmer can introduce a "text" as a part of a holon body. This must be written:

text begin "Any text —" **end**

In this text, the programmer can explain the reason of some design decisions or anything like this. In the holon body, the holon names embedded inside the holon statements explain how the program works; in the "text," the programmer may expound why the program was written in this way. It is a "comment" to help program readers to understand the logical basement of the program.

Implicit holon names

The holon syntax requires that, in statements as "do while," "repeat until," "case"—you write the name of one holon for the repeated part of the statement.

Example:

```
A
begin B; do C while Q; Z; end
C
begin CA; CB; CC; end
```

This levelling can be omitted by writing explicitly the definition of the "repeat" part in the statement.

The previous example becomes:

```
A
begin B;
        do begin CA; CB; CC; end while Q; Z; end
```

The **begin** and **end** words are used by the analyzer to build correct linkages in the binary tree.

Holon names "bracketed" by **begin — end** can appear between the following symbols:

repeat — until

do — while

do — ; (if we have a statement **while — do** — ;)

then — else

else — ;

then — eor

Only holon names can be used between **begin — end**, not holon statements. Likewise, holon names for terminals can be omitted and the statement in terminal language is "bracketed" by the # — and ## symbols.

The "proc" and "mac" options

The programmer can supersede the decision of the synthesizer by the use of the **proc** or **mac** options.

A holon name followed by the reserved word **proc** (**mac**) means that the programmer wants the implementation unit for this holon to be a procedure (macro), regardless of the synthesizer's choice.

If this reserved word appears in the holon definition, the choice is valid for all appearances of this holon in the program. If the reserved word follows a holon name inside a holon statement, the choice is valid only at this place.

However, a **mac** option for a recursive holon will always be rejected by the synthesizer.

This option allows the programmer to "contest" the trade-off function of the synthesizer. It may also be used during a new synthesizer design to test the trade-off function.

Holon System Control Language

We shall describe a holon system control language. This language is an "interface" between the programmer and the implemented version of the analyzer and synthesizer which he is using.

Analyzer commands

The majority of the control language commands deal with the analyzer.

(A) To start the design of a new program, the programmer uses the following command:

> **new program** ⟨holon name⟩
>   **by** ⟨programmer identification⟩ **end**

The result of this command will be the opening of a file to store the future holons of the program and the introduction of a new item in the file which lists the undertaken programs.

(B) To access a program, the programmer issues the **program** command:

> **program** ⟨holon name⟩
>   **by** ⟨programmer identification⟩ **end**

If the holon name is recorded as a "program" and if the programmer identification is a legal one, the analyzer is ready to obey the programmer requests.

(C) The **append** command is used to introduce the definitions of new holons:

> **append** ⟨holon⟩ . . . **end append**;

If holons in the "append" command have names which don't appear in holon statements of other holons, the analyzer keeps these holons in a "wait list" for future use. Each time the analyzer finishes an append session for a program, it tries to append the holons which are in the waiting list.

After that, the programmer receives the following information:

1. An edited text of the appended holons;
2. A report on syntactic errors;
3. The status of the waiting list;
4. A list of the undefined holons.

(D) The programmer can used the **undefine** command to erase the last definition given for a holon. If the erased definition contains names of already-defined holons, these latter definitions are not destroyed if these holon names appear elsewhere in the program. The command is:

undefine ⟨holon name⟩ . . . **end undefine**;

After an "undefine" session, the analyzer reports the list of the undefined holons for the program.

The appearance of the name of a holon existing in the wait list results in the complete deletion of this holon: the definition and the name of this holon are deleted.

(E) The **change** command is a combination of the two previous ones:

change ⟨holon⟩ . . . **end change**;

For each holon appearing in the "change" list, the operations are the following ones:

1. The previous definition of this holon becomes "undefined." But this operation is not conducted like the regular **undefine** command: if the erased definition contains names of already-defined holons, these latter definitions are not destroyed if these holon names appear elsewhere in the program or if they appear in the new definition.
2. The new definition is appended.

The use of this command is mainly the "remodelling" of the holon statements of a holon.

(F) The following command:

output analyzed program ⟨holon name⟩ **upon**
  ⟨device description⟩ **by** ⟨programmer identification⟩;

is used to obtain the output (according to a standard format) of a program (partly or completely finished). The device description points out the support of the output: cards, tape, disk, etc.

(G) The reverse operation of the previous one is made by:

program ⟨holon name⟩ **from** ⟨device description⟩
  by ⟨programmer identification⟩;

Synthesizer commands

There are two commands:

(A) The first one is for requesting a program synthesis. The programmer gives the name of the requested integration strategy:

> **synthesize** ⟨program name⟩ **with** ⟨strategy name⟩
>   **by** ⟨programmer identification⟩;

(B) The second one requests the execution of a synthesized program:

> **execute** ⟨program name⟩
>   **by** ⟨programmer identification⟩;

The ⟨programmer identification⟩ is used in all these commands to protect the program from illegal users.

Program Example

In this section, we shall show an example of a program written in the holon programming language.

Problem description

This problem appeared in Dijkstra, Section 16.[39] We quote here his problem description:

[39] Dijkstra.

> We consider a character set consisting of letters, a space and a point. Words consists of one or more, but at most twenty letters. An input text consists of one or more words, separated from each other by one or more spaces and terminated by zero or more spaces followed by a point. With the character valued function RNC (Read Next Character) the input text should be read from and including the first letter of the first words up to and including the terminating point. An output text has to be produced using the primitive PNC(x) (i.e., Print Next Character) with a character valued parameter.. . . The text is to be subjected to the following transformation:
>
> 1. In the output text, successive words have to be separated by a single space
> 2. In the output text, the last word has to be followed by a single point
> 3. When we number the words 0, 1, 2, 3, . . . in the order from left to right (i.e., in which they are scanned by repeated evaluation of RNC), the words with an even ordinal number have to be copied, while the letters of the words with an odd ordinal number have to be printed in the reverse order.
>
> For instance (using "␣" to represent a space) the input text
>
> ‘this␣␣␣is␣␣a␣silly␣␣program␣␣.’
>
> has to be transformed into
>
> ‘this␣si␣a␣yllis␣program.’

The holon program

The following pages give the holon program written using the modified syntax. The holons are written in the order of their design.

```
odd inversion program
  begin
    find first word starting character;
    repeat find a word and print correctly until end of useful file;
  end

find first word etc
  begin
    read first symbol;
    while last read symbol is a space do read next symbol;
  end

find a word etc
  begin
    buffer useful characters;
    print left to right or right to left according to even or odd;
    find word terminator and print it and set end of file if full stop;
  end

buffer useful etc
  begin
    repeat begin buffer last read symbol;
      read next symbol;
      end
    until last read symbol is a space or a full stop;
  end

find word terminator etc
  begin
    if last read symbol is a space then skip other spaces;
    if last read symbol is a full stop
      then print it and set end of file;
      else print a space;
  end

print left etc
  begin
    if even word
      then print left-right;
      else print right-left;
    increase word count;
  end
```

```
even word
  begin
    declare even: boolean at odd inversion program level
    initial # even ← true ##;
    # even ← true ##;
  end

increase word count
  begin
    # even ← not even ##;
  end

skip other spaces
  begin
    repeat read next symbol until last read symbol is not a space;
  end

read first symbol
  begin
    declare lrs: character at odd inversion program level;
    # lrs ← RNC ##;
  end

last read symbol is a space
  # lrs ≡ '␣' ##;

read next symbol
  # lrs ← RNC ##;

end of useful file
  begin
    declare EOF: boolean at odd etc level;
    initial # EOF ← false ##;
    # EOF ≡ true ##;
  end

buffer last read symbol
  datatest number of symbols in the buffer less than 20; end
  begin
    declare buffer: character (20) at find a word etc level;
    declare pointer: integer at find a word etc level;
    initial # pointer ← 0 ##;
    # pointer ← pointer + 1; buffer(pointer) ← lrs ##;
  end
```

```
last read symbol is a space or a full stop
  # (lrs ≡ '␣') **or** (lrs ≡ '.') ##

last read symbol is not a space
  # lrs ≠ '␣' ##

last read symbol is a full stop
  # lrs ≡ '.' ##

print it and set **etc**
  # PNC('.'); EOF ← *true* ##

print a space
  # PNC('␣') ##

print left-right
  **begin**
    **declare** point2: **integer**;
    **initial** # point2 ← 1 ##;
    **repeat** # PNC(buffer(point)); point2 ← point2 + 1 ##
    **until** # point2 > pointer ##;
  **end**

print right-left
  **begin**
    **while** # pointer ≠ 0 ## **do**
      **begin**
        # PNC(buffer(pointer)); pointer ← pointer − 1 ##;
      **end**
  **end**
```

The integrated program

The synthesizer will produce code for a program equivalent to the one written below. In this code, the block structure will be disentangled and the storage for all the variables will be allocated on the pseudo-stack.

[This intermediate ALGOL-program form doesn't exist in the system; it is only said that the generated code will be equivalent to the code generated for the following program by an ALGOL compiler.]

```
begin
  declare even: boolean, lrs: character, EOF: boolean;
  even ← true; EOF ← false;
  lrs ← RNC; while lrs ≡ '␣' do lrs ← RNC;
  repeat
    begin
      declare buffer: character(20), pointer: integer;
      pointer ← 0;
      repeat
        if pointer ≥ 20 then (datatest exit);
        pointer ← pointer + 1;
        buffer(pointer) ← lrs;
        lrs ← RNC;
      until (lrs ≡ '␣') or (lrs ≡ '.');
      if even ≡ true
        then
          begin declare point2: integer; point2 ← 1;
          repeat PNC(buffer(point2)); point2 ← point2 + 1;
          until point2 > pointer;
          end
        else
          begin while pointer ≠ 0 do
            begin PNC(buffer(pointer)); pointer ← pointer − 1; end
          end
      even ← not even;
      if lrs ≡ '␣'
        then begin repeat lrs ← RNC until lrs ≠ '␣'; end
      if lrs ≡ '.'
        then begin PNC('.'); EOF ← true; end
        else PNC('␣');
    end
  until EOF ≡ true;
end
```

Comments

A. I confess that I made a change after reading Dijkstra's students' answer: the holons "even word" and "increase word count" were written:

```
even word
  begin
    declare wordcount: integer at odd inversion program level
    initial # wordcount ← 0 ##;
    # even(wordcount) ← true ##;
  end
```

```
increase word count
  begin # wordcount ← wordcount + 1 ## end
```

I had some trouble after writing this because I was afraid of an overflow of "wordcount." That would require the introduction of a "datatest" with error recovery in "increase word count." The error recovery would be to "reset" the counter. But with this datatest, the holon "increase wordcount" seems to be a very cumbersome one. It is not the overflow that really bothers me, but that there may be an "overflow interrupt." So I solve all of these problems by writing this (retrospectively) incredible holon:

```
increase word count
  begin # wordcount ← [wordcount + 1] mod 2 ## end
```

It was the wordcount **integer** declaration that hindered me from thinking about a boolean variable.

B. The reader may think that the result of the work is a very verbose program. That is true, but the holon program explains the process in a different way from a "normal" program:

1. The logic of the program answer to the problem is contained in the program text. You don't need to disentangle this logic from the instruction sequence.
2. The program text is self-explanatory.

C. Normally, with a real holon system, the programmer would submit his first-written holons and receive (for instance, on punch cards which he shall put in front of the definition) the list of the undefined holons.

D. Advice on the usefulness of the system must not be established by the fast that for so small a problem, the generation of the holon answer seems longer than with a normal language. It is better to ask oneself: "... and when faced with a program a thousand times as large, should I compose it in the same way?"[40]

[40] Ibid.

E. The use of the "datatest" must be noticed. It is put at its place as a "fire-wall" for the program.

Reading Knuth's version of the program suggests that it would be more sensible to write a "datatest" with the error-recovery option.[41] The processing of the string will continue even after a "too long word." For instance:

[41] Knuth.

```
datatest
  number of symbols in the buffer less than 20
  else begin
    print a quoted buffer;
    reset pointer to zero;
  end
end
```

With this datatest the following odd word:

thiswordislongerthantwentycharacters

will print as:

"thiswordislongerthan"sretcarahcytnewt

[I forgot in the program to write the refinement of the datatest. Normally, this must be written.]

Macro-processing of Holon Name Parameters

The previous-described syntax allows only the use of parameters for variables. Parameters can be used for holon names provided we macro-process holon definitions if there is a holon name parameter.

For instance, the postorder traversal of a binary tree is an algorithm which appears frequently in this survey; the semantics of "visit the node" is the only change. One may imagine a super-holon: "traverse binary tree in postorder using %holon% for visiting the node."

Each appearance of this holon will produce an expansion of its body, in order to build the whole program tree structure.

For the time being, we don't plan to implement this feature in the first version of the holon language.

Holon Parallel Programming

Reading the book of Brinch Hansen[42] and Hoare's paper,[43] it appears to me that it would be interesting to study if, with some modifications, the holon language can be used to program parallel processes. As a matter of fact, the language was not designed with this purpose in mind.

[42] P.B. Hansen. *Operating system principles*. Prentice-Hall, 1973.

[43] C.A.R. Hoare. "Towards a theory of parallel programming". In: *International Seminar on Operating System Techniques, APIC Studies in Data Processing*. Vol. 9. Academic Press. 1972, pp. 61–71.

I have not studied thoroughly this extension, at the present time. Nevertheless, some of Hoare's requirements seem to be fulfilled more or less easily:

1. The **cobegin** and **coend** notation for disjoint processes. Moreover, the non-permanent parts of the co-processed holons can be used to check the disjointness at analyze-time.
2. The **exit** statement as it is defined (**goto** the end of a holon) fulfills the ban of jump out of a parallel statement (the parallel statement is a co-holon).
3. The "when B" condition would be a datatest with the error recovery option: **datatest** B **else** waiting **end**.
4. The "array remapping" may be interpreted as a local refinement.

This section is only a hint to an eventual extension of the holon language.

Holon Operating System

Introduction

We want to use the fact that we know the structure of a program in order to manage a virtual memory system. Since we have a structured program, we have a better knowledge of the behavior of the program. We can determine exactly the future behavior of a program; we want to use this information to build a system furnishing its working set beforehand to a program. Among the instructions of a program, there will be some page transfer requests; but, those requests are written automatically in due places in order to avoid page faults; the Holon Operating System will prepare a suitable working set for each executed program.

Paging of a holon program

The terminal levels, linked by some suitable code corresponding to the machine code for the holon language statements (**if** . . . **then** . . . **else** . . . **do** . . . **while** . . . **etc** . . .) are "paged." These pages correspond to the operational (or functional) part of the program; now, we must also "page" the data structure, for that purpose we manage two different memory zones: a paged stack and a paged pseudo-stack.

In the pseudo-stack, we will put all the variables—the dimensions and the number of which may be known at compile time (this includes all the variables except the ones appearing in arrays with dynamic bounds and in recursive procedures). The variables in the pseudo-stack are "overlaid" if it is possible; for instance the two component variables of an if-then-else clause are overlaid. But the maximum length of this pseudo-stack may be known at compile time. And at any point in the program, we may specify what is the length of the used part of the pseudo-stack.

Now, in the stack we put all the data, the number or the dimensions of which depend on the computation (dynamic arrays, data local to recursive procedures).

The difference between the two stacks is: we take two computation points in the program (two different positions of a textual index), for instance p_1 and p_2, with the following condition: p_2 is to the right of p_1 in the holon tree (the meaning of which is: there are some computations to do for moving the textual index from point p_1 to p_2); now we call $\psi(p_1)$ and $\psi(p_2)$ the extents of the pseudo-stack at point p_1 and p_2, and in the same way S1 (p_1) and S1(p_2); $\psi(\pi)$ is a fixed value not correlated to the input data string; but S1(π) depends on the input string.

Now, look at Figure 13—it represents a holon program with its code, and the two "paged" stacks. (Remark: we suppose temporarily that there exists only in-line code.)

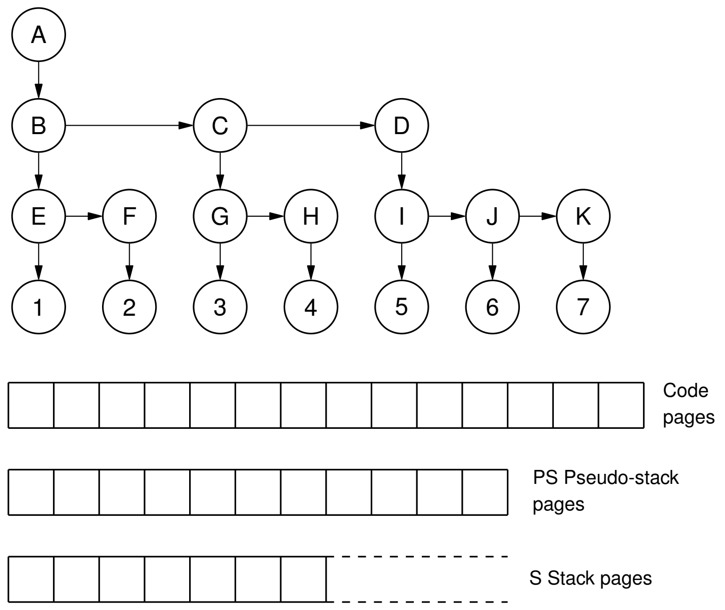


Figure 13: A holon program with its corresponding pages and "paged" stacks.

The pagination of the seven terminal holons is made like this: the code of the terminal holons composes a string of machine code and we break the string, putting the first x words in the first page, the next x words in the second page, etc.; i.e., the paging isn't related to the structure of the program.

Now, in the next section, we shall see how a Holon Operating System would work.

Monitoring the page transfers

We shall call the "working set for a holon" the code and data pages needed by a holon to avoid any disruption during the computation. The important feature of the holon tree is that the "working set" is almost—because of S—well-defined. For instance, in Figure 13, the working set for holon A is composed of all the code and PS pages, plus an amount of S pages (this amount is not known, but the holon's need of some space (some pages) in the S stack may be known). Now, the working set of the holon B is composed of the pages needed to perform the computation involved in the holon B, but also the first pages needed by holon C to start its computation: we need the last ones because we must avoid disruption not only during the holon computation, but also between two holons.

As long as a holon computation progresses, the holon may also indicate the pages it doesn't need anymore.

With these two operations in mind (a holon may make page requests in advance; a holon may also indicate the pages to be dropped), it may be understood that we may build a holon operating system where, as much as possible, the program's behavior is managing the page requests.

Structure of the computer

We need a computer with an internal paged store, and (probably) with a two-level back store. [The matter of the number of levels will appear further on, because in the Holon Operating System it will appear that the page size will be determined by the transfer speed between the levels of the store hierarchy.]

We need also a "monitor," this will be either a program or a small hardware computer. The purpose of the monitor is: to receive the page requests from the running programs and to manage the transfer of the pages.

Fundamental structure of the paging algorithm

[The following description is a sketch—it is the departure point of a study.]

We first suppose that the processor is allocated to the different processes according to a scheduling policy. Each holon will request its working set before starting execution. Nevertheless, the monitor will discard requests from "overly ambitious" holons. The definition of an "overly ambitious" holon is up to a computer center manager—it will depend on the average job profile; and, for instance, the "overly ambitious" barrier may be changed during the day.

If a holon is discarded (because it is overambitious), the hierarchic levels of this holon are stepped down until the first "moderate" holon is found; its page requests are obeyed as soon as possible and this holon starts running. Now, during its execution, it will request pages in advance and the monitor will try to bring these pages in time; it is also the holon program that points out the pages which may be overlaid by others, because it doesn't need them anymore. It is only in case of emergency (deadlock) that the monitor decides to remove a page not released by a holon.

Working set for the fundamental instructions

If-then-else statement

Suppose the statement '**if** Q **then** T **else** F; next;' ('next' is not necessarily a holon name; it may be another holon statement).

Suppose that we have the following paginations: (We consider only "code pages")

Q: $Q_1\ Q_2\ Q_3 \cdots Q_N$ pages

T: $T_1\ T_2\ T_3 \cdots T_N$ pages

F: $F_1\ F_2\ F_3 \cdots F_N$ pages

next: $N_1\ N_2 \cdots N_X$ pages

The working set for the if-then-else statement is: ($Q_1\ Q_2 \cdots T_1\ F_1$); like this, the statement will be executed without disruption. The test is made and the first page of both parts of the alternative are included; in the beginning of the T_1 page, for instance, there will be requests to the monitor:

1. Page from F_1 may be overlaid,
2. Please bring the following pages into the internal core: $T_2, T_3, T_4, \ldots T_M, N_1$.

Without waiting for the answer to the request, we start the execution of the T_1 page instructions.

Remark. The requested pages depend on the nature of the holon T; if this one is a new if-then-else statement, then the requested pages will correspond to the if-then-else working set. In this case, we have brought into memory and unused page (in the example: F_1), but that is not a problem, as we don't want to reduce its minimum number of page transfers during a program execution—we want to limit to a minimum the number of page faults.

Case statement

It appears that the case statement may be managed in the same way as the if-then-else statement.

Repeat and while statements

Suppose statement '**repeat** P **until** Q; next;' with the pagination:

P : $P_1\ P_2\ P_3 \cdots P_N$

Q : $Q_1\ Q_2\ Q_3 \cdots Q_N$

next : $N_1\ N_2\ N_3 \cdots N_X$

The working set is: $(P_1\ P_2 \cdots P_N\ Q_1 \cdots Q_N\ N_1)$. In fact the selected P_i will depend on the P statement type; but normally (except tying up the internal memory) the deletion of pages will be delayed until reaching the beginning of the page N_1.

Example: In the if-then-else statement, when we reach Q_2, we may request deletion of Q_1; but if this statement is embedded in a repeat statement, this deletion is not sensible, because it will overload the page transfer mechanism. In the same way, embedded **repeat** statements lead to fixed pages.

Maximum number of pages fixed

In the previous section we presented a memory policy management that is program-oriented, because it is the program itself that requests pages and also deletes them according to its needs (the monitor preempts pages only if the pages fixed by the processes tie up the internal memory.) There exists another policy: to fix a maximum for the number of code pages allocated to a process. Because we have structured programs, in front of each page we may, at compile-time, write the list of the $(n-1)$ pages requested to work (n is the maximum number allowed) and the list of page modifications for the pseudo-stack and the stack. With this policy, we have a deletion each time we leave a page, followed by a page transfer (except in the case where we are in a **repeat** statement, the extent of which is less than n; see the following example.

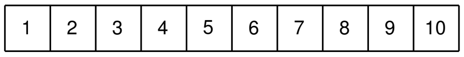


Figure 14: A program with 10 pages.

In Figure 14, pages 1, 2, and 3 compose a **repeat** statement. We list the page requests for $n = 5$ on entry in:

1: $1, 2, 3, 4, 5$

2: $2, 3, 1, 4, 5$

3: $3, 1, 2, 4, 5$

4: $4, 5, 6, 7, 8$

5: $5, 6, 7, 8, 9$

This example shows that there is no transfer and deletion until we reach the head of page 4: but at that time, we have to transfer 3 pages; and if by chance the **repeat** statement covers exactly 5 pages, then there will be a page fault.

Priorities in the resources used by a program

The page-management policy, in our opinion, consider that the pages used for the pseudo-stack and the stack are never preempted, for several reasons:

1. The cost of a page removal is higher for a data page than for a code page because removal of the former implies a copy in the back store; the latter allows immediate overlay;
2. If we delete some data pages of a running process, we will have data-originated fault pages.

On the other hand, we may use the number of data pages and of code pages requested by a program to establish dynamic priorities between the programs: for instance, the program time slice may be proportional to its number of code pages, i.e., we give more time to the programs involved in loop computations; or we may sort the program waiting queue according to the number of data pages, i.e., we give priority to the "memory monopolizing" programs.

Page size

In order to respect the philosophy of the system, the page size will be chosen according to the performance of the page transfer device and of the monitor program: the transfer time for a page must be less than the average execution time of the instructions contained in one page (in order to avoid disruption of a process computation).

This requirement may limit the application of this kind of operating system to computers using "slave memory" (like the IBM 360/85 cache store) or "distributive memory" (for instance, one level of standard core memory, the next level being large capacity store LCS, the last one a drum). The previous discussion explains the kind of policy that would be used to move the pages between the different levels of the memory hierarchy.

Deadlock

The approach presented in previous sections must be studied to understand how it may lead to a deadlock; this deadlock is owing to requests for data pages when there is no free page frame. But it seems that in this case, preemption of a job would solve the problem. (Deadlocks due to peripherals are another problem.)

Conclusion

This is only an outlook of what may be done about a holon operating system; the profit of the use of the structured programming methodology would be enhanced by the improved performance of computer systems.

Some additional comments about the holon operating system

Hoare and McKeag reject the possibility of prepaging.[44] The first reason is the inability to have a significant knowledge of the future behavior of the programs. The second one is that the rate of a program execution in a multiprogramming environment is unknown.

[44] C.A.R. Hoare et al. "A survey of store management techniques". In: *Operating Systems Techniques*. Academic Press, 1972.

Nevertheless, it seems that prepaging through the monitor control can be feasible with some kind of memory, the latency time of which is small; moreover, if it appears that a program generates page faults under prepaging, the monitor may try to avoid this by some modification of the scheduling policy (for instance, if we have a round-robin policy, we may shorten the time-slice of the faulting program.)

In the holon system, the synthesizer can use the description of the computer architecture to choose the place and the extent of the page requests. The avoidance of page faults is a design goal, not a guarantee.

The Atlas system,[45] uses an elaborated algorithm to detect pages involved in loop computations. The Burroughs B6700 tries to define a "working set" for each program and adjusts it in order to provide a fixed rate of page faulting.[46] The prepaging policy that we have described should be compared to these kinds of designs.

[45] M.H.J. Baylis et al. "Paging studies made on the ICT Atlas computer". In: *Information Processing* 68 (1968), pp. 831–837.

[46] D.P. Fenton. "B6700 Working set memory allocation". In: *Operating Systems Techniques*. Academic Press, 1972, pp. 321–327.

Conclusion

In this survey we have described the main features of the holon concept and its use in programming languages.

As "the proof of the pudding is in the eating," we presently start an implementation of the holon programming language. With it, it will be possible to evaluate practically the usefulness of the language.

Pierre-Arnoul de Marneffe
Université de Liège
Liège Belgium

Appendix

In this appendix, we comment on a few peculiar points.

The use of the exit statement

The **exit** statement is only a **goto** the end of a holon; this restriction forbids the **exit** statement to be used to get out of a loop. Using the **exit** statement, you first reach the end of the repetitive part of the loop; but next, you must pass through the test part of the loop.

Example: The searching of a file for the appearance of a given key. There are two exits of the searching process: either an item matching the key is found, or the end of the file is reached without finding any matching item.

A holon program for this may be written as follows:

```
search for a matching %KEY%
 begin do try to match next item in the file to %KEY%
  while the end of file is not reached or the item is not found;
 end
the end of file etc
 begin
  declare unfound, NOEOF: boolean at search etc level
  initial # unfound ← true; NOEOF ← true ##;
  (unfound ≡ true) and (NOEOF ≡ true) ##
 end
try to match etc
 begin
  declare F: file at search etc level
  declare index: integer at search etc level
  initial # index ← 1 ##;
  if # F(index) ≡ 'EOF' ## then # NOEOF ← false ## exit;
  if # F(index) ≡ %KEY% ##
   then # unfound ← false ## exit
   else # index ← index + 1 ##;
 end
```

%KEY% is not declared because it must appear in an outer holon requesting this search.

As a matter of fact, the example is too simple to show the use of **exit**. We can code the two **if**s like a **case** statement because there is nothing to do after the two tests. An obvious way to code the program is something like:

```
index ← 1;
while (F(index) ≠ %KEY%) and NOEOF
do index ← index + 1;
```

But the matter here was to show how people must use the **exit** statement, if one wants to use it.

Understanding the holon concept

The author really doesn't know to what extent the reader can grasp the holon concept without reading Koestler's book.[47]

[47] Koestler.

Nevertheless, we shall give two other examples of using the holon model. The first one, quoted from "The Revolutionist's Handbook" by George Bernard Shaw, is a case for the necessity of environment-checking. The second one is a personal interpretation of some early American building designs; it will expound the tendency to integration.

The self-controlled glutton

This description was used by Bernard Shaw to expound his statement that: "The survival of the fittest means finally the survival of the self-controlled, because they alone can adapt themselves to the perpetual shifting of conditions . . . ."

"The Glutton, as the man with the strongest motive for nourishing himself, will always take more pains than his fellows to get food. When food is so difficult to get that only great exertions can secure a sufficient supply of it, the Glutton's appetite develops his cunning and enterprise to the utmost; and he becomes not only the best fed but the ablest man in the community. But in more hospitable climates, or where the social organization of the food supply makes it easy for a man to overeat, then the Glutton eats himself out of health and finally out of existence."[48]

[48] G.B. Shaw. "The Revolutionist's Handbook". In: *The Bodley Head Bernard Shaw: Collected Plays with Their Prefaces*. Vol. 2. 1903, pp. 739–80.

The integration design at work

An example of two ways (an unintegrated one and an integrated one) of implementing the same function can be seen in two early American buildings: George Washington's Mount Vernon and Thomas Jefferson's Monticello.

It seems that a plantation of that time was composed by a lot of "units": the mansion, the laundry, the smoke house, the kitchen, the dairy, the stable, the weaving house, etc....

Mount Vernon looks like a small town, each of the "units" implemented like a house.

At Monticello, Jefferson put together all the units by locating them beneath two long terraces forming aisles of the mansion. All the "units" are connected by an all-weather passageway. This way to "integrate" the buildings renders all the "units," except the mansion, as inconspicuous as possible.

The final results, as a visitor can see, are completely different; nevertheless, they both have fulfilled the same function.

Chronology

The first report on holon programming circulated privately in December '72.[49]

[49] Marneffe.

In February '73, a submitted draft was accepted by the selection committee of the International Computing Symposium '73. The ICS was held in Davos, Switzerland, the 4th through the 7th of September '73.

The paper "Holon Programming" will be published in the ICS '73 proceedings by North-Holland.[50]

[50] Marneffe et al.

In September '73, P. Henderson (University of Newcastle upon Tyne) made us aware of a series of three articles on the CADES system by David Pearson (ICL) published in Computer Weekly (7/26/73, 8/2/73, 8/19/73). The CADES system is using a top-down approach called "structural modelling." The designers of this system call a "holon" a unit of software design. The CADES system is a data-oriented one: "...The emphasis is on code being designed to manipulate defined data structures rather than data structures being invented to support the codification of abstract functions..." That is working in the opposite direction of our approach. Calling their units "holons" is correct because these units have the two features of a holon: (1) "...A highly resilient, modular design results with each holon only manipulating a strictly-defined small subset of the universal set of available data"; and (2) "...It is to be emphasized that one level only will appear finally in machine-executable codified form..."

I am very happy that other people are tempted to use the "holon concept" to master program complexity.

Acknowledgments

First of all, I want to thank Prof. D. Ribbens (University of Liège, Belgium), who is my teacher in computer science, especially for the many discussions and comments about this work.

All our gratitude for Prof. E.W. Dijkstra for his many helpful comments he made on previous reports.

I want to thank Prof. H. Leroy (Institut d'Informatique, Namur, Belgium), Prof. E. Milgrom (University of Louvain-la-Neuve, Belgium), and Mr. W.H. Feijen (T.H.E. Eindhoven) for the useful comments and criticisms they made.

I thank the Belgian National Fund for Scientific Research (F.N.R.S.) and IBM–Belgium to have granted our stay at the IBM Federal System Center (Gaithersburg, Maryland).

I am indebted to Dr. Harlan Mills of IBM Gaithersburg, who so kindly accepted my stay at IBM Gaithersburg. I shall never forget how the people there and especially Mrs. Carole Hall, Mr. Jay Arnold, Jayadev Misra, Jim Whitlock, and Bill Patterson welcomed me so friendly.

I must thank Mrs. Barbara Heflin, who managed to type this text so efficiently, and the Word Processing and Art Department for the work they have done.

Transcriber's Endnote

The phrasings used by Prof. de Marneffe have been preserved except in a few cases where a gallicism obscured the meaning. Please send any bugs or suggestions to `holon.programming@gmail.com`.

Thanks to Don Knuth for directing me to the only extant hard copy of this survey, for sending a copy of his 1974 letter to the author, and for allowing the letter to be reproduced here. Thanks to the librarians at the German National Library of Science and Technology in Hanover, Germany and those at Love Library in Lincoln, Nebraska who helped me get a hold of this rare book. Thanks to the creators of the `tufte-book` document class for creating such a lovely template.

And, of course, for TEX and literate programming!

And many thanks to the late Prof. Pierre-Arnoul Frédéric Guy Donat de Marneffe (1946–2023) for his work and his kind answers to my inquiries.

Mitch Gerrard
April 2024

Bibliography

Baker, F.T. “Chief programmer team management of production programming”. In: *IBM Systems journal* 11.1 (1972), pp. 56–73.

Baker, F.T. “System quality through structured programming”. In: *Proceedings of the December 5–7, 1972, Fall Joint Computer Conference, part I*. 1972, pp. 339–343.

Baylis, M.H.J., D.G. Fletcher, and D.J. Howarth. “Paging studies made on the ICT Atlas computer”. In: *Information Processing* 68 (1968), pp. 831–837.

Dijkstra, E.W. *EWD249: Notes on Structured Programming*. Technical University Eindhoven, 1970.

Dijkstra, E.W. *EWD316: A Short Introduction to the Art of Programming*. Technical University Eindhoven, Aug. 1971.

Dijkstra, E.W. “Go to statement considered harmful”. In: *Communications of the ACM* 11.3 (1968), pp. 147–148.

Dijkstra, E.W. “Recursive programming”. In: *Numerische Mathematik* 2.5 (1960), pp. 312–318.

Dijkstra, E.W. “The humble programmer”. In: *Communications of the ACM* 15.10 (Oct. 1972), pp. 859–866.

Fenton, D.P. “B6700 Working set memory allocation”. In: *Operating Systems Techniques*. Academic Press, 1972, pp. 321–327.

Hansen, P.B. *Operating system principles*. Prentice-Hall, 1973.

Henderson, P. and R. Snowdon. “An experiment in structured programming”. In: *BIT Numerical Mathematics* 12.1 (1972), pp. 38–53.

Hoare, C.A.R. “Chapter 2: Notes on data structuring”. In: *Structured programming*. Academic Press, 1972, pp. 83–174.

Hoare, C.A.R. “Towards a theory of parallel programming”. In: *International Seminar on Operating System Techniques, APIC Studies in Data Processing*. Vol. 9. Academic Press. 1972, pp. 61–71.

Hoare, C.A.R. and R.M. McKeag. “A survey of store management techniques”. In: *Operating Systems Techniques*. Academic Press, 1972.

Horning, J. and B. Randell. “Process structuring”. In: *ACM Computing Surveys (CSUR)* 5.1 (1973), p. 26.

Knuth, D.E. *A Review of “Structured Programming”*. Tech. rep. STAN-CS-73-371. Stanford University, Stanford, CA: Stanford Computer Science Department, June 1973.

Knuth, D.E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. First edition. Reading, MA: Addison-Wesley, 1968, pp. 332–333.

Koestler, A. *The Ghost in the Machine*. Macmillan, 1968.

Marneffe, P.A. de. *Holon Programming: definition du langage*. Privately circulated.

Marneffe, P.A. de and D. Ribbens. “Holon programming”. In: *International Computing Symposium*. Ed. by A. Günther et al. Amsterdam, North Holland, 1973.

Mills, H.D. *Mathematical foundations for structured programming*. Tech. rep. FSC-72-6012. Gaithersburg, MD: IBM Federal Systems Division, Feb. 1972.

Morrison, P. and E. Morrison. *Charles Babbage and his calculating engines: Selected writings by Charles Babbage and others*. Dover New York, 1961.

Naur, P. and B. Randell, eds. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. 1969.

Shaw, G.B. “The Revolutionist’s Handbook”. In: *The Bodley Head Bernard Shaw: Collected Plays with Their Prefaces*. Vol. 2. 1903, pp. 739–80.

Simon, H.A. “The architecture of complexity”. In: *Proceedings of the American philosophical society* 106.6 (1962), pp. 467–482.

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

Strachey, C. “System analysis and programming”. In: *Scientific American* 215.3 (1966), pp. 112–127.

Wegner, P. *Programming Languages, Information Structures, and Machine Organization*. McGraw-Hill, 1968.

Wilkes, M.V. “The outer and inner syntax of a programming language”. In: *The Computer Journal* 11.3 (1968), pp. 260–263.

Wirth, N. “Program development by stepwise refinement”. In: *Communications of the ACM* 14.4 (Apr. 1971), pp. 221–227.

Woodger, M. “On semantic levels in programming”. In: *IFIP Congress (1)*. 1971, pp. 402–407.