CS 65

ON CERTA IN BAS IC CONCEPTS OF PROGRAMMING LANGUAGES

BY

NIKLAUS W IRTH

TECHNICAL REPORT NO. CS 65
MAY I, 1967

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERS ITY

On Certain Basic Concepts of Programming Languages

By

Niklaus Wirth

May 1, 1967

Computer Science Department

Stanford University

Stanford, California

On Certain Basic Concepts of Programming Languages

Contents

		Page
I.	On Data Structures	2
	1. Type Definitions	3
	2. Cell Declarations	5
	3. Cell Designations	7
	4. Cells Without Explicit Names	8
	5. Blockstructure	12
	6. The "Elimination" of References	15
	7. Input -output	17
	8. Operations on Defined Types	18
	9. Summary	19
II.	On Program Structures	20
	1. Statements and Expressions	20
	2. Program Verification and Efficiency	22
	3. Subscript Ranges	23
	4. Ambiguous References	24
	5. Procedures as Data Elements	25
	6. Loops	27
	References	30

Recent developments of programming languages have led to the emergence of languages whose growth showed cancerous symptoms: the proliferation of new elements defied every control exercised by the designers, and the nature of the new cells often proved to be incompatible with the existing body. In order that a language be free from such symptoms, it is necessary that it be built upon basic concepts which are sound and mutually independent. The rules governing the language must be simple, generally applicable and consistent.

In order that simplicity and consistency can be achieved, the fundamental concepts of a language must be well-chosen and defined with utmost clarity.

In practice, it turns out that there exists an optimum in the number of basic concepts, below which not only implementability of these concepts on actual computers, but also their appeal to human intuition becomes questionable because of their high degree of generalization. The following informal notes do not abound with ready-made solutions, but it is hoped they shed some light on several related subjects and inherent difficulties. They are intended to summarize and interrelate various ideas which are partly present in existing languages, partly debated within the IFIP Working Group 2.1, and partly new.

While emphasis is put on clarification of conceptual issues, consideration of notation cannot be ignored, However, no formal or concise definitions of notation (syntax) will be given or used; the concepts will instead be illustrated by examples, using notation based on Algol as far as possible.

I. On Data Structures

The elementary concepts of computing processes are:

- There exist certain quantities, to be called "values", and elementary classes or types (possibly only one) of values among whose elements given elementary relationships hold. These relationships or mappings are represented in a computer by its operations which generate a new value (called result) which has the specified relationship to the given value(s) (called operands).
- There exist cells (usually called "variables") which are able to contain a value, and which have a name. That name serves to refer to the contained value.
- There exists an operator for the assignment of a new value to a cell.

The vocabulary used for describing processes must contain at least one denotation for each element in the universe of values, and at least one for each relationship among values in each class. While the universe of (elementary) values is usually given in a programming language, the set of cells involved in a process is particular to that process and must be defined in its description. Therefore, also names to designate those cells must be individually introduced (declared). A necessary rule is that either cell names must be distinguishable from denotations of values (and relationships), or otherwise a chosen cell name identical with a value denotation may no longer be used (directly) to denote that value,

It is important that groups of elementary values can be combined and considered as a composite or structured value. It is customary to denote such a value by listing its components, separated by a separator (e.g. comma) and delimited by brackets (e.g. parentheses). The name of the cell holding a structured value is then used to denote the entirety of the component values.

This conceptually appealing and simple solution has been realized in the language EULER [W and W]. Its practicability, however, turns out to be rather doubtful for the following reasons:

1. Since a cell may hold any value, and therefore also a composite one, its physical size in terms of computer memorycells is not fixed. Implementation of this scheme requires the use of indirect referencing and dynamic storage allocation to an extent which makes the use of such a language unattractive for many applications.

2. The very dynamicism and lack of redundancy of the language makes it difficult for the programmer to verify the correctness of a written program.

3. Assuming that individual elements of a composite value can be referenced by the name of the holding cell followed by an index, it immediately follows that the same notation should be used in assignment statements to alter elements of the structure. Since assignments can only be made to cells (and not to values), the cell holding a composite value must be considered as a structured cell. It follows that the creation of cells is a highly implicit action, since assigrnnent of an n-ary value implies the creation of n cells. The conclusion to be drawn is that a programming language should not contain the notion of a structured value, but rather the one of a structured cell. Positional relationships between values then exist only by virtue of the structure of their containing cells.

1. Type Definitions

These difficulties and drawbacks can be overcome by attributing to each cell a fixed structure, when the cell is introduced. For practical purposes this turns out to be hardly a restriction at all, since in most applications a program involves only a few different types of structures, while many used cells are of one and the same structure. One may consider the given elementary classes or types of values to be of elementary (degenerate) structure. A cell may then be declared to be of a given elementary type, and hence it can hold only values of that type. This is achieved by the type declarations in Algol 60. Further, more complex structures can be considered as compositions of elementary structures; and in order that a name can be attributed to that structure, also to be

called a "type", a new construct called a "type definition" has to be introduced. It may assume a form as illustrated by the following example:

type Person (Integer age; Boolean male; Real weight)

"Person" is the name of the new structure, which is composed of three elements (called "fields") which are of elementary structure; Integer, Boolean, and Real respectively. The type definition is moreover used to attribute names to the individual fields, and corresponds to the record class declaration in [W and H]. It can be assumed that the elementary types are introduced by fixed type definitions in the environment of the program. In fact, the elementary types are usually themselves composed of bits, and their substructure is dependent on particular implementations and machines,

It is sufficient to let the type definition consist of a linear list of constituents, if the constituents themselves can be of any type. Examples:

type Medicalrecord (Integer bloodtype, heart condition;
Boolean diabetic)

type Patient (Integer age; Boolean male; Medicalrecord health)

Often it is desirable to give numeric names to fields of structures, which in turn can be computed. An example is

type A (Real 1,2,3,4)

for which we immediately introduce the abbreviation

type A (Real [1:4])

without further explanation. Such a structure is called a (one-dimensional) array, and the field names are called indices; all elements are

of the same type. Multi-dimensional arrays, whose elements are designated by more than one index, could be defined as follows:

type B (Real 1,2,3|1,2,3,4) .

Above abbreviation leads to the short form

type B (Real [1:3]|[1:4]) .

The distinction of this structure and the one defined by

(type[1:3])

lies in the. fact that the 12 elements of B are of type Real, while C consists of 3 elements of type A, which in turn consists of 4 elements of type Real. If B is considered to be a matrix, then its rows and columns are not explicitly designated and appear on the same footing, while C is considered to be a linear structure of rows.

The foregoing notation has the effect of making explicit the similarity of the concepts of arrays and records [W and H]. It automatically introduces array structured fields:

type Account (Integer number; Real balance; A deposit)

2. Cell Declarations

The introduction of cells (variables, records) is required to contain an indication of the type of the cell along with the name to be associated with the new cell.

Examples:*

new (Integer) i
new (A) a, a1

*in order to facilitate reading of subsequent examples, names of cells begin with lower case letters, names of types with capital letters.

```
new (B) b, b1, b2
new (C) c
new (Person) jack, jill
new (Patient) smith
new (Account) ac
```

The symbol new is chosen to indicate that a new cell of a given type is introduced. Instead of new, cell or var might have been chosen to emphasize the creation of a cell or variable. In terms of an implementation, this declaration causes storage to be allocated,

In Algol 60

```
new (Integer) i
```

is abbreviated to

```
integer i
```

and this convention holds for all elementary types. If the language rules are such that in the place of the type identifier the type definition itself can occur, then the example

```
new (A) a
```

can also assume the form

```
new (Real [1:4]) a
```

or abbreviated

```
real [1:4] a
```

from which the analogy to Algol 60's array declaration

real array a[1:4]

becomes evident.

3. Cell Designations

Various notations are now presented to denote cells and components of structured cells:

α	β	γ	δ
a[2]	a.2	2 of a	2(a)
b[2,3]	b.2,3	2,3 of b	2,3(b)
jack[age]	jack.age	age of jack	age(jack)
smith[health]	smith.health	health of smith	health(smith)
c[2]	c.2	2 of c	2(c)
c[2[3]]	c.2.3	3 of 2 of c	3(2(c))
smith[health[diabetic	smith.health. diabetic	diabetic of health of smith	diabetic(health(smith))
ac[principal]	ac.principal	principal of ac	principal(ac)
ac[deposit[3]]	ac.deposit.3	3 of deposit of ac	3 (deposit(ac))

At this point it seems appropriate to examine the results of the previous unification of concepts, and to compare the resulting notation with constructs present in existing languages. Notation α coincides with Algal 60 in the form of "subscripted variables". β appears in PL/I and COBOL, (only applied to fixed, i.e. non-computable names). y coincides with the notation of field designators in [vW] and δ with that of [W and H], in both cases used only in connection with non-computable field names, At places where computablenames occur, expressions should be permissible, which quickly leads to syntactic abominations in all cases except α . For the use of cells with alphabetic (non-computable) field names, notation δ seems more natural because of its analogy to the conventional notation for functions and predicates, as which field names can be understood. One concludes from the foregoing that a unification of homogeneous

structures with computable field names (indices) and inhomogeneous structures with noncomputable ones (identifiers) is not desirable, mainly for reasons of notational tradition. It is even much less desirable from the standpoint of implementation, since computed indexing over an array of fields with different size is necessarily a difficult and inefficient process.

A relatively appealing solution to this dilemma consists of (a.) restricting computability to numeric field names (indices), (b.) enclosing them in distinguishable brackets, and (c.) to use conventional postfix notation (α) for indices and prefix notation (6) for field designations with alphabetic names.
Examples:

```
a[2]
b[2,3]
age (jack)
health (smith)
c[2]
c[2[3]]
deposit (ac) [3]
```

4. Cells Without Explicit Names

So far, the assumption was made that in a program every cell to be involved was explicitly denoted by a name attributed to the cell by its declaration. In certain problems of data processing, however, the number of involved cells is not known a priori, nor is it necessary that all cells be available from beginning to end of the process, A facility becomes desirable to create cells at any time (i.e., dynamically).

Once a cell is created, there must be a way to refer to it. Since its name is not introduced into the program explicitly (e.g., as an identifier), it becomes necessary to consider names as objects which can be used to refer to cells. The cell creation then not only allocates a cell, but also yields the name of the allocated cell. That name is to be

called a reference, and is to be treated as an elementary value of type Reference . The dynamic cell creation can be denoted by

```
r := Person
```

which results in the assignment of the reference to the new cell to r . The form

```
r := Person [21, false, 101 51
```

can be used if the new fields are to be assigned initial values at the same time. The declaration of r is denoted by "ref r" which stands as an abbreviation of

```
new (Reference) r .
```

It turns out to be a significant advantage to implementation, if it is required that the type of object to which a reference value assigned to a given reference cell may refer, be unique. This type can be specified as follows along with the declaration of the reference cell,

```
ref [Person] r
ref [Integer] k
```

The reasons for binding reference cells to a specific class were given in [W and H].

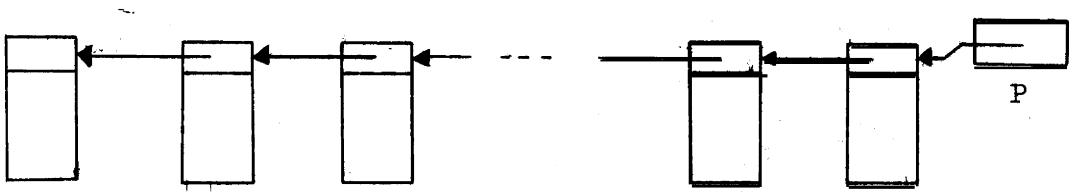
It should be noted that the type specified with a reference cell declaration does not denote a substructure of the reference cell itself, which is elementary, i.e. without substructure. It instead denotes the structure of the referenced quantity.

References can now be used to express functional relationships between the objects represented by cells. If a reference valued field f of a cell x holds a value refering to a cell y, then y is said

to have relationship f to x . The following is an example of a piece of a program using the facility of dynamic cell creation:

```
type Person(Reference [Person] son; Integer age; . ..).
ref [Person] p, q;
L: p := Person; son(p) := q;
   q := p; go, to L
```

The piece of program describes the creation of an infinite number of cells of type Person. At L, the value of q refers to the "youngest" member of the chain of descendents. Pictorially, the set of created cells may be described as follows:



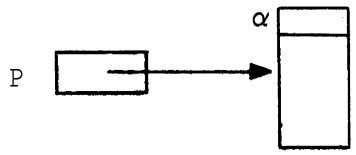
Each pointer represents a value of the class "Reference" which is held in a field of that type, called "son".

The above example also suggests a convention for denoting the value of given fields of dynamically created records, which is in conformity with the notations presented above. If in place of the name of a cell, one of whose subfields is to be designated, the name of a reference cell occurs, then it is implicitly assumed that the field of the indirectly referenced cell is denoted.
Example:

```
age (jack)          jack = name of a person
age (P)      }
son (P)      }      P = name of a reference to a person
```

This conventions seems perfectly natural and raises no problems, since

reference cells are not themselves structured and a field designation therefore undisputably must refer to a field of the referenced cell. However, a dilemma arises when the entire cell, and not one of its fields, is to be designated:



Does p now denote the reference value referring to α, or α itself? Two possible solutions are offered here:

a. p denotes the reference to α, the notation person(P) is used to denote α .

```
p := q
```

then denotes the copying of a reference, while

```
Person(p)  := Person(p)
```

denotes the copying of the values of a cell of type Person.

b. the exact meaning of p is determined by context, (e.g. correspondence of types) such that in

```
q := p
```

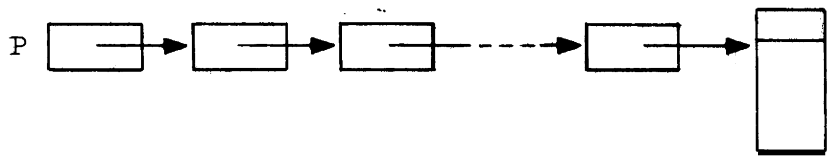
p denotes the reference value to α, while in

```
Jack := p
```

p denotes the Person cell α itself.

The latter solution, which is adopted in [vW], obviously leads to further

problems when p has values referring to cells which are themselves of the class Reference.



It is only possible to denote either the reference value held by p, or the person cell which constitutes the end of the reference chain, but no intermediate reference values. Apart from the conceptual intricacies which would make a program using such constructs rather difficult to understand;, certain well-founded doubts about their practical usefulness suggest that the dynamic creation of cells of type Reference (and elementary types in general) should not be included in a programming language. Another aspect of this topic is presented in the next section.

5. Blockstructure

Blockstructure was introduced into Algol 60 to delimit the scope of names (identifiers). Since names are attached to quantities by their declaration and are not themselves manipulatable values, a cell itself becomes **unaccessible** as soon as the scope of its name is left. The storage space allocated for a cell can therefore be released at the same time.

Dynamically created objects do not have a name which appears in the program, but can only be reached via internally created references whose "lifetime" is not bound to any scope limitations (in the same sense as constants do not have a limited scope). Release of storage space reserved for a dynamically created cell can therefore not be initiated on exit of control from a given block, but only by unspecified events at a time when no references to that cell can be made either directly or indirectly from cells which have a given explicit name (in [vW] called "appellation"). However, because of the convention that reference cell declarations must

be accompanied by a specification of the type of referenced cells, such a declaration cannot be made outside the scope of the pertinent type definition. Consequently, the existence of accessible reference values is restricted to the scope of that type definition, and all cells of that given type become unaccessible upon exit from its scope, at which time a storage release can be initiated,

So far, references (i.e. reference values) could only enter into a process through the dynamic creation of cells. In particular, references pointing to explicitly named quantities have so far not been considered, However, they are an integral part of the language EULER, as well as the Algol successor proposedin[vW] and they call for further investigation.

In EULER, a reference value referring to a quantity named x is denoted as @x . In [vW] the reference to a quantity x is denoted simply by x; context decides whether the value of quantity x is meant, or the reference to that quantity, much in the same way as context determines the meaning of "age(Jack)" and "age(p)" in the example above. This is possible, because unlike in EULER fixed types are associated with all named quantities. At this point, however, a contradiction is introduced, if all assumptions given in this paragraph are retained: while reference values are not subject to limited scopes, the explicit name which occurs in the program text (the appellation) does have a fixed scope. This fact leads to calamitous situations unless the meaning of the Algol block structure is revised, as the following example shows:

```
        begin ref [Integer] k; integer j;
              begin integer i; i := 1;
α:                  k := i
β:            end;
γ:            j := k
        end
```

At α, according to the fact that k is of type ref, the reference pointing to the cell i is assigned to k . At β, the scope of i ends, and according to Algol tradition, the storage space occupied by

i is released. At γ, the value of the cell referenced by k (still i ?) is assigned to j . It becomes necessary to revise the definition of Algol 60 to the effect that the rules of scope apply to names (appellations) only, but not to the named quantities themselves. This seems to defeat the very aim of blockstructure, In fact, the postulate is equivalent to requiring that all storage be allocated in the same way as for dynamically allocated cells described above.

The only plausible solution seems to be to disallow the declaration of reference cells bound to explicitly named quantities.

Before the consequences of such a restriction are discussed, a few considerations of implementation are appropriate. As noted above, there exist cells of elementary type9 and those of composite structure. Most computers are capable to copy and assign any elementary value equally efficiently as a reference value (address). It is therefore advisable always to deal with the considered value itself, and not with a reference to it. After all, the ultimate access to a value will always be more elaborate, if it has to be made via an indirect reference. It is felt that a language should do its best to discourage the use of indirect addresses in such cases.

Composite structures, on the other hand, are not as easily manipulated as references. Moreover, since the size of composite structures can usually be computed (arrays in Algol 60), their allocation must be made dynamically, and their access must then necessarily be indirect. It follows that composite structures implicitly use a reference cell as described above, whether it is requested by the programmer or not. An Algol 60 array declaration is indeed more precisely described by the explicit steps

```
type α (Real 1:n);
ref [α] a;
a := α
```

than merely by

real array a[1:n],

and the occurrence of a in a program should be understood as the denotation of the reference value pointing to the dynamically allocated array cells. It follows that composite cells should not be explicitly named, or if this is allowed in the language, it should be understood to be an abbreviation in the above sense. In cases where the type definition is given in the same block as the reference declaration (or in the abbreviated version even together), the effect on storage allocation is the same as that of Algol 60 array declarations: storage can be released on exit from the block.

These considerations of usage and implementation of elementary and composite cells also apply to their treatment as procedure parameters, in spite of the fact that proponents of references to named quantities use the parameter mechanism as their motivation.

The quintessence of the foregoing three paragraphs then is that

- Cells of elementary type are always declared and thus have explicit names (appellations);
- Cells of composition are always created dynamically, and their structure is known through explicit type definition;
- Reference cells are of elementary type and their values are always bound to refer to quantities of a given type. As a consequence, references can refer to composite cells only.

The postulated restriction does in fact not limit the power of a language, since it is always possible to define a structure (type) consisting of a single field only.

6. The "Elimination" of References

In a language where it is understood that composite records are always referenced indirectly, the role of the symbol ref reduces to that of a reminder of this convention. Effectively, it could be omitted, i.e.

the notation

```
ref [T] a,b
```

could be replaced by the shorter"

```
T a,b
```

where T is an identifier introduced by a type definition. This has been done in the AED language [R]. It must then be clearly understood that

```
a := b
```

denotes the copying of a reference, and not of the referenced structures themselves. A somewhat confusing consequence is illustrated by the piece of program below whose last statement does not only alter the age of p, but also that of q !

```
new (Person)p, q;
p := Person; age(p) := 10;
q := p;
age (p) := age(p) + 1
```

Whether that abbreviation is used may be a matter of taste, but the point of view that the coexistence of both

```
new (T) r
```

and

```
ref [T] r
```

may contribute more to the conceptual complexity of a language than to its usefulness, is justified.

7. Input - Output

Input - output operations are assignments of values (usually composite structures) held on one storage medium to cells allocated on another medium. If input - output handling is to be an integral part of a language, then the rules governing input - output activities must be consistent with the rules governing other activities. The simplest way of choosing consistent rules is taking the same rules. This implies that data to be input or output must be declared on the "external" media as they are on the "internal" one. As a matter of fact, the specification of the storage medium together with the declaration of cells may be considered as an implementation dependent comment.
Example:

```
ref [T] a,b [disk]
ref [T] c,d [tape,]
ref [T] x,y [core]
```

Assignments such as

```
x := a      c := y
```

can then be understood as denoting input and output operations respectively. Note that in this case it does not suffice to copy the reference only, since the references are supposed to point to cells in the specified storage medium.

The reduction of input - output operations to mere assignments invites for heavy misues of the I-O capabilities of presently known secondary storage media, unless certain natural restrictions on the kind of structures are introduced, which take the inherent nature of such devices into account. In order to express a proposal on some such restrictions, a structure classification is introduced:

- A structure whose number of fields is fixed by the type definition is called a static structure.

- A structure whose fields are all of the same type is called an array structure. (Its field names are usually computed indices.)
- A structure without fields which hold (references to) other structures is called a basic structure.

If assignments are made which necessitate a transfer of information between different storage media, and if these transfers are to be achievable with a minimum of administrative overhead, then they ought to be restricted to arrays of static basic structures. Such a restriction does indeed not affect all applications which make use of what are usually called record files, i.e. linear sequences of records of data which contain no cross-references among each other. The restriction can be somewhat relaxed by merely requiring that possible reference fields contain the value null upon assignments.

The notation of a file is here introduced in the sense of what is more specifically called a serial file or a tape, and it is defined as a linear array of static structures (as above). The file differs, however, from the more general array in the restrictive manner in which access can be made to its elements: With a serial file is associated an implicit index which designates the one currently accessible element. Each assignment to or from the file automatically increments this index by unity, Moreover, certain standard operations on files are introduced which make it possible to change that index.

This notion of a file seems to be necessary and sufficient to include in a satisfactory manner the handling of storage devices with an inherently serial access mode, such as tapes, line printers, card readers and punches.

8. Operations on Defined Types

Algol 60 specifies only operations on what are here called values of elementary type. These are the operations present in the hardware of computers. Operations on values of structured type are usually expressed in terms of sequences of operations on components. A facility for conveniently abbreviating such sequences is the procedure in Algol 60. A

modification of this concept which makes the usual infix notation applicable is obtained by extending the meaning of elementary operations to structured ones by declaring that the operations apply elementwise to the constituents of the structure. This modification is called "overloading" (cf. also [H]) and applies to array structures. In the previous chapters, this principle has already been applied to the assignment operation.

9. Summary

- There exists a given set of elementary data types without substructure. This set includes the type reference.
- A type definition introduces a structured data type and associates an identifier with it. The structure of this type is specified as either a sequence of fields, each designated by a field identifier and each being of a fixed type, or of a single or multidimensional array of elements of homogeneous type which are designated by computable indices.
- Variables, here called cells, have a fixed type, i.e. can store only values of that given type.
- Cells of elementary type are introduced by declarations. The scope of their name (appellation) and the lifetime of the cells themselves, is determined by blockstructure.
- Cells of structured type are introduced "dynamically". They have no appellation; instead, they are accessed indirectly via a reference which is the value of a cell of type reference.
- Declarations of cells of type reference always specify the type of the cells to which the reference may refer.
- If a reference r points to a cell C of type T, the notations r and T(r) are used to designate the reference itself and the referenced cell C respectively. The meaning of cell designators is context independent.
- In declarations of reference cells, it is possible to specify the storage medium to be used for storing the referenced cells.

II. On Program Structures

1. Statements and Expressions

The fundamental notion in program structures is that ofan(assignment) statement. It indicates a "closed action", by which is meant that after its execution the effect of the performed actions are entirely represented by the values of the cells participating in the process. Executing a program with paper and pencil, one can dispose of any intermediate results (scratch paper) after each statement. This is a conceptually most appealing situation which is appreciated in particular when one is confronted with the task of verifying a program. In Algol, the execution of an (assignment) statement consists of the evaluation of an expression, followed by the assignment of the obtained result to one or several cells. (Note that all statements in Algol, except the go to "non-statement", can essentially be reduced to the assignment statement or sequences thereof.) The expression is the part which is evaluated with the possible aid of scratch paper, and the fact that the scratch paper can be discarded after each statement is contained in the syntax, where (expression) can be a constituent of (statement), but not vice versa. This scheme is destroyed in Algol by the fact that it is possible to use a function procedure (whose body consists of statements) as a constituent of an expression. What is not visible in the syntax is achieved by application of the "copy rule": (statement) becomes a constituent of (expression) . The consequences of this situation have been hotly debated on many occasions and are collected under the subject "side-effects". They are as undesirable as much as perspicuity of programs is desirable; it turns out;, however, that in certain mild and disguised forms they can be quite useful. And if a facility is useful in some instances, it becomes most difficult to dispose of it just for the sake of sound principles.

Nevertheless, the question arises whether side-effects should be embraced as an integral part of a language, or whether they should be exterminated entirely.

The former solution is realized by eliminating the distinction between statements and expressions, and by recognizing the assignment expression

```
v := e
```

as the identity operation on e with the side-effect of assigning the value of e to v . This philosophy has been adopted in EULER [W and W], and in [vW]. Constructs such as

```
a := b + (c := d × e) -f
```

are now as legal as Algol 60's

```
real procedure g ; g := c := d × e;
a := b + g - f .
```

As a consequence, every (formerly called) statement has now a value, and the execution of a sequence of n statements results in the piling up of n values (on scratch paper). To remedy this, the statement separator ";", which in Algol 60 has merely syntactical functions, becomes an active operator charged with the duty to discard the value of the last computed value. Consequently, blocks have a value, and so do proper procedures: E.g.

```
begin a := 1; b := a+1; c := b+1 end
```

must be attributed the value 3. It becomes necessary to introduce the notion of partial functions, since no value can be attributed to the dummy statement.

The latter solution, namely the elimination of side-effects, is realized by redefining the body of type procedures to be an expression instead of a statement (which includes at least one peculiar kind of assignment to the procedure identifier in Algol 60). This solution is

as radical as the former, and its consequences are also far-reaching. Because of the conceptual importance of the pure statement and its role in facilitating program verification, it should not be ignored.

2. Program Verification and Efficiency

When developing a program, one automatically constructs a verification of its correctness. The fact that (all too often) lapses occur in the design is due to the lack of a systematic (I do not say "formal") verification method. Only recently have attempts been made at establishing more rigid guidelines for such a method [N], and the fact that they are not widely used is partly due to the lack of languages whose designers have recognized this problem sufficiently clearly. Verification methods are simplified, if a language has an appropriate structure, and if certain constructs are amenable to fixed verification rules.

Here verification means the deduction of the truth of certain assertions about a program strictly on the basis of the program text, i.e. without its evaluation. A verification must therefore depend on information which is just as well available to the compiler, and which indeed may be used by the compiler to perform certain (partial) verifications automatically. Conditions which can be verified in this way do not have to be checked at execution time, which contributes to the efficient execution of a program. In this light, the interests of efficiency and program perspicuity emerge as identical.

A first example of a language facility aimed at these two goals is the association of a fixed type to all variables in Algol 60. The relaxation of this rule for formal parameters contributes heavily to inefficiencies of executed programs. A similar step is the binding of reference variables to a specific record class in [W and H], which contributes to both clarity and efficiency of programs in a way that without it the entire record handling feature would appear as unattractive. Another example, also in [W and H], is the for-statement which is defined differently from that in Algol 609 to the effect that the control value depends on the for clause alone, and cannot be altered through "side-effects"

from the iterated statements. This example illustrates clearly how certain language structures with appropriate definitions can facilitate verification methods.

Two other instances of similar facilities with the same aim are to be outlined below. They both refer to the for statement as defined in [W and H], as opposed to that of Algol 60.

3. Subscript Ranges

If one considers it essential that at least during execution of a program undefined situations be detected--and anyone concerned with the question of reliability of computed results should--then it becomes necessary to test whether computed indices lie within the declared subscript range. This testing, which in general can only be performed at run time, is costly and makes the use of arrays unattractive compared with the use of records in [W and H], where access to fields does not require any checking, since it can be verified from the program text alone. It is thus highly desirable to introduce a notation for certain common situations where the subscript checking can be performed by the compiler. The for statement appears as most appropriate: for an index being a control value, run time checking can be omitted if the compiler can deduce that the range of the control value does not exceed the range of the subscript.

Example:

```
real array a[1:n];
for i := 1 step 1 until n do s := A[i] + s
```

For such a verification in this example the compiler (and the human verifier) must have the ability to compare symbolic quantities (here n) and to establish the fact that no assignment to n occurred between declaration and for statement. This task could be drastically simplified by a facility which makes it possible to associate a name (identifier) with a range.

Example:

```
range R = 1:n
real array a[R];
for i :=R step 1 do s := A[i] + s
```

More generally, the facility to specify a given range with the declaration of a variable could be introduced and each assignment to this variable would include a range check:

```
integer (R) i
```

4. Ambiguous References

Each -facility designed toward compile-time verifyability introduces some sort of restriction. It is essential to assert that the restriction is not a handicap but rather an aid to the programmer. The nature of the rule that every reference be bound to a certain type ("record class" in [W and H]) is in that sense ambiguous. It is often desirable that reference fields of structures be able to point to structures of several types. This problem is discussed in [H], and the very plausible concept of record sub-classes is presented, which here might be called "categories" (of a certain "type"). A type definition may now assume the form:

```
type Person (integer age; ref (Person) father, mother;
category Man (integer draftcardnumber;
                ref (Person) youngest child, spouse),
          Woman (Boolean pregnant; ref (Person) spouse),
            Child)
```

Fields common to all categories are listed first;then the categories are introduced, each followed by a (possibly empty) list of "private" fields.

A reference assigned to a cell declared as

```
ref (Person) r
```

can now point at either a cell of category Man, Woman, or Child (which are all said to belong to the type Person). Whether a field designator such as

```
pregnant (r)
```

is valid can only be determined at run time. Of course, a programmer uses this field designator only where he (maybe mistakenly) assumes that r always points to a Woman. Usually, he uses a test to predetermine this fact explicitly, such as

```
if r is Woman then ... pregnant (r) . . .
```

A language should express this common situation in a way that an implicit check connected with the field designator can be avoided. A construction where this is possible must necessarily resemble the one used for avoiding subscript range checking and use the concept of a quantity to which no assignment can occur within a certain scope. The following notation is adopted from [H] in a slightly modified form:

```
for t := r when Man do S1
          when Woman do S2
          when Child do S3
```

Here r is a reference expression, S1, S2, S3 are statements, and t is a local quantity implicitly declared like i in the previous example.

Implementations on multi-register computers can take advantage of this construction by holding the quantity t, which is most likely often to be used in S1, S2, S3, in a register.

5. Procedures as Data Elements

In Algol, procedures are static in nature and distinct from data. Procedures cannot 'be manipulated; they can only be executed. The phenomena.

of "remote activation" by procedure statement, and of "passing on" of a procedure as a name parameter, are explained through textual substitution (copy rule).

With EULER the notation of-a manipulatable procedure was implemented so that procedures can be referenced indirectly through references which in turn can be assigned to variables. This solution unifies in a most appealing way the two concepts of procedure and name parameter of Algol 60. The denotation of a literal procedure consists of the procedure text enclosed in quote marks and is therefore called a quotation. In a language like Algol, an elementary value type procedure has to be defined to express this situation as follows:

```
A :     procedure p;
B :     p := 'x := x+1';
C :     p
```

The meaning of this piece of text is the following: at A, a variable (cell) p is introduced, at B, a quotation is assigned to p, and at C the occurrence of p denotes the execution of that quotation. A notation which is more consistent with the one of the previous chapter is the following:

```
A :     ref [Procedure] p , q;
B :     p := Procedure (x := x+1);
C :     ex p;
D :     q := p
```

This notation makes possible the assignment of procedure references to variables of the appropriate type without implying the execution of the denoted procedure.

As in the case of references in the previous chapter, the concept of an assignable program text in coexistence with blockstructure unfortunately causes irreconcilable difficulties. This is here even more apparent,

because a program text may obviously contain names (identifiers) with limited scopes, and hence be assigned to quantities outside that scope. Example:

```
begin procedure p; integer i,k;
      i := 100;
      begin integer i,j;
            p := 'k := i+j';
            i := j := 10; p
      end;
      p
end
```

While the first occurrence of p appears as entirely legitimate, the second is highly problematic.

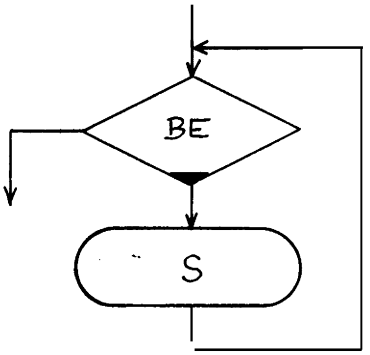
6. LOOPS

A loop is an extremely frequently occurring program structure, and it should be representable by a simple, yet flexible notation. Programmers which are used to conceive their algorithms in terms of flowcharts, and then transliterate flowcharts into sequential language, are apt to frequently employ the go to statement. It is contended here that loop structures should be expressible through program structures rather than the cumbersome use of labels which not only tend to overly increase the amount of needed identifiers, but also constitute a strong temptation to construct puzzling mazes of program paths.

In [W and H] the following simple construct for a simple loop is suggested:

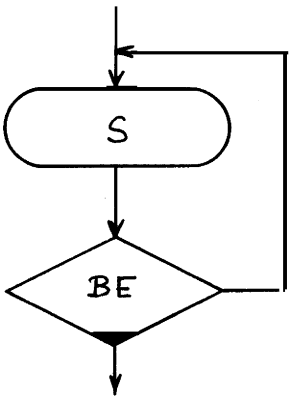
while (Boolean expression) do (statement)

corresponding to the flowchart



This construct is considered to be superior to Algol 60's for statement with while element, because of its simplicity and obviousness. However, it is somewhat rigid in that it requires that the test for loop termination is made before the statement is ever executed. Burroughs Extended Algol (B5500) has remedied this situation by offering a similar construct, which implies that the test is made after the first execution of the statement!--,

do (statement) until (Boolean expression)



It appears that this is only a partial remedy., In fact, one usually ends up writing program pieces of the following 'kind

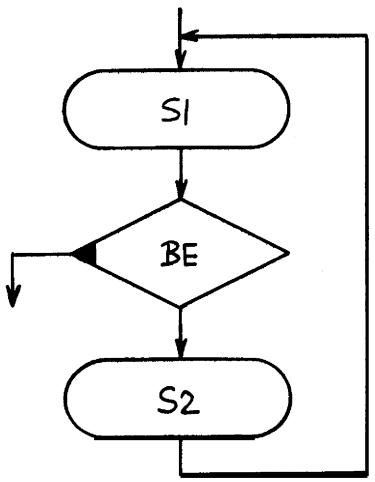
```
S1; while BE do begin S2; S3 end
go to L; do begin S2; L:S3 end until BE
```

where S1 and S3 are identical sequences of statements. The topological similarity of these constructs suggests that a ternary construction is needed*, consisting of two statements and a test for loop termination,

*The idea was raised by Don Kunth in connection with PL 360. It is also present in the JUMP OUT statement of B5500 Stream Procedures.

separated by adequately chosen delimiters. The following is suggested here.

```
repeat in
    S1;
when BE exit;
    s2
end
```



The former while and until statements can be represented as repeat statements with either S2 or S1 being a dummy statement,

An alternative selection of delimiters results in:

```
turn on begin
    S1;
when BE drop out
    s2
end
```

References

[H] C. A,. R. Hoare, "Record Handling," lectures given at Nato Summer School on Programming, 1966.

[N] P. Naur, Proof of Algorithms by General Snapshots BIT 6, 4 (1966) p p. 310 - 316.

[R] D. T. Ross, "AED Language," Electronic Systems Lab., MIT.

[vW] A. van Wijngaarden, Proposal for a Successor to Algol, Working Document "Warsaw 2", IFIP WG 2.1.

[W and H] N. Wirth and C. A. R. Hoare, "A Contribution to the Development of ALGOL," Comm. ACM 9/6 (June 1966).

[W and W] N. Wirth and H. Weber, "EULER, A Generalization of ALGOL," Comm. ACM 9/1-2 (Jan./Feb. 1966).