Singer's Comments

A REMINDER FOR LANGUAGE DESIGNERS

By

Frederic Richard *
Henry F. Ledgard *

COINS Technical Report 76-3
(Revised August 1976)

Keywords: language design, programming languages, readibility, program validation, program maintenance.
CR Categories: 4.2, 4.22.

* Computer and Information Science Department,
University Of Massachusetts,
Amherst, Massachusetts 01002, USA.

This work was supported by the U.S. Army Reseach Office.

A REMINDER FOR LANGUAGE DESIGNERS

By

Frederic Richard *
Henry F. Ledgard *

COINS Technical Report 76-3
(Revised August 1976)

Keywords: language design, programming languages, readibility, program validation, program maintenance.
CR Categories: 4.2, 4.22.

* Computer and Information Science Department,
University Of Massachusetts,
Amherst, Massachusetts 01002, USA.

This work was supported by the U.S. Army Reseach Office.

Abstract:

Current programming languages offer limited support in the development and maintenance of programs. These languages do not always account for the human limitations of their users. Notably, few languages really promote ease of readability. This paper suggests nine design principles for the development of readable high level languages. Each principle is backed up by a discussion and several examples. Among the issues discussed are the limitation of the overall complexity, the design of function and procedure facilities, the design of data type facilities, and the correspondence between syntax and semantics.

Introduction.

This paper stems from the difficulties we have had while experimenting with current programming languages. To implement real problems, no current programming language offers clean solutions. Too often, the structure of the problem must be twisted to the structure of the language.

We believe there is a need for a new general purpose, procedure oriented programming language. This UTOPIA 84 (Knuth 74) should not only be designed to enable the programmer to devise clear data structures and algorithms. It should also provide assistance to the user in the development of large programs, their verification and their maintenance. For this purpose, the readability of a language (i.e. human appreciation) is far more important that its writability (i.e. translation from precise implementation specifications).

In this paper we suggest nine language design principles for UTOPIA 84. These principles (see Table 1) are based in part on the works of Dijkstra (68), Gannon and Horning (75), Hoare (72), Knuth (67), Ledgard and Marcotty (75), Weinberg (75), Wirth (74), and Wulf and Shaw (73). No attempt is made to address the whole language design area. Little consideration is given to writability and efficiency of implementation. We believe that these goals have received too much attention in the past.

There is no formal justification for any of our principles. Each principle is supported by a short discussion and several examples borrowed from languages in widespread use: Algol 60 (Naur 63), COBOL (Murach 71), FORTRAN (X3J3 66), PL/1 (ECMA/ANSI 74), PASCAL (Jensen and Wirth 74), and SIMULA 67 (CDC 71).

1. A language should be limited in complexity and size.

2. A single concept should have a single form.

3. Simple features make simple languages.

4. Functions should emulate their mathematical analogue.

5. A clear distinction should be made between functions and procedures.

6. Multiple data types should be supported.

7. Similar features should have similar forms.

8. Distinct features should have distinct forms.

9. Remember the reader.

TABLE 1: Nine Design Principles.

1. A language should be limited in complexity and size.

Over the past few years, there has been an almost unabated tendency for languages to get larger and larger. In an effort to provide more powerful and more varied features to satisfy more users, the complexity of many languages has markedly increased. We believe this has been a mistake. Our own limitations as users, implementors, and designers call for limitations on the complexity and size of our tools.

It is easy to point out the problems of undue complexity during design and implementation. For the language designer, the evaluation of design alternatives are difficult because of the frequent interplay with other constructs within the hosting language. Formal definitions become increasingly intricate, documentation is harder to prepare and read, and inconsistencies may easily be overlooked. For compiler writers, the production of a clean, reliable, and well human-engineered implementation requires more and more work. There is no perfect language design and the more complex the language, the more difficult it is to offer the user a clean and consistent programming system.

Users pay an even higher price for undue complexity. Learning is slow, and programming often cannot proceed without constant references to the manual. Any inconsistencies take more time to learn and more energy to live with. Most of all, the user may encounter great difficulties in understanding the underlying structure of the language. Mastery and proficiency come only when the user develops a comprehensive internal model of the language. The selection of useful constructs, cleanliness of use, and understanding of error diagnostics proceed far more quickly when the user understands the language in its entirety.

Subsetting, i.e. partitioning a language into semi-independent modules, has often been presented as a practical remedy to large size. There are, however, numerous drawbacks. The user facing a new problem may wonder whether the subset he has mastered is adequate, or whether he should learn a larger subset. Programs may inadvertently activate unknown features and cause confusion. Furthermore, subsetting is of little help in reading programs written by other users, where knowledge of the whole language may be needed. Lastly, there does not seem to exist any good method for partitioning a language in a way acceptable by all users.

Admittedly, the complexity and the size of a language depend mainly on its intended application. When the size is too small, the language primitives are overloaded and the complexity in usage becomes unnecessarily high. When the size is too large, the language often offers more than is necessary, and the user is easily confused. There are few major programming languages that do not in fact suffer from undue size and complexity. The many duplicate forms and the report writer feature of COBOL are questionnable. As a teaching language, PASCAL is too complex. The case against PL/I is obvious.

In summary, programmers should not be slowed in their problem solving activities by the complexity, the size, and the unknown subtleties of their tools. Our own human limitations as users, implementers, and designers call for languages that are limited in complexity and size, and designed to be well implemented.

2. A single concept should have a single form.

Providing more than one form to denote a concept always increases the size of a language. The additional complexity introduced by such features should be carefully weighed against their usefulness.

Consider, for instance, the simple PL/I aggregate declaration in Figure 2.1 and the rather large number of subscripted qualified names that can be used to denote the same component of the aggregate. A similar declaration and the unique denotation of the same element, expressed in PASCAL, are given in Figure 2.2. In comparison, the complexity of multiple PL/I denotations is difficult to justify

good example

COBOL provides a further example of questionnable duplicate forms. Figure 2.3 shows two different sequences of arithmetic statements. Both perform the same computations. Further,each sequence is perfectly homogeneous to the eye. But when both notations are combined as in the third sequence of Figure 2.3, we see the problem more clearly. The symmetry of like operations is not ~~brought at~~ enhanced as in the above examples. A designer may prefer the concise, mathematical notation of the first sequence, or the English like notation of the second sequence. In any case, it would be simpler to retain a single notation in the language.

There are some situations where a duplication of forms yields great convenience without adding much to the overall complexity. For instance, fully qualified names for aggregates are often cumbersome to read and to write, especially when the same element is referenced often over a span of text. PL/I provides numerous

Declaration

```
DECLARE  1 A (10,12),
           2 B (5),
             3 C (7),
             3 D;
```

Fully qualified names

```
A(9,11)     .B(4)      .C(7)
A(9)        .B(11,4)   .C(7)
A(9)        .B(11)     .C(4,7)
A           .B(9,11,4).C(7)
A           .B(9,11)   .C(4,7)
A           .B(9)      .C(11,4,7)
A           .B         .C(9,11,4,7)
A(9,11)     .B         .C(4,7)
A(9)        .B         .C(11,4,7)
A(9,11,4,7).B          .C
```

Partially qualified names
(in some contexts only)

```
B(9,11,4)   .C(7)
             C(9,11,4,7)
B(9,11)     .C(4,7)
B(9)        .C(11,4,7)
B(9,11,4,7).C
```

Figure 2.1 : Multiple Denotations of a PL/I Structure Element.

Declaration

```
A: array [1..10,1..12] of
     record
        B: array [1..5] of
             record
                C: array [1..7] of integer;
                D: integer
             end
     end
```

Complete denotation

```
A[9,11].B[4].C[7]
```

Legal abbreviations

```
with  A[9,11]
      do    ...   B[4].C[7]  ...

with  A[9,11].B[4]
      do    ...   C[7]   ...
```

Figure 2.2 : Legal Denotations for a PASCAL Record Element.

Use of the COMPUTE verb

```
COMPUTE  TOTAL-HOURS    = OVERTIME-HOURS + REGULAR-HOURS.
COMPUTE  NUM-ON-PAYROLL = NUM-EMPLOYEES - NUM-ON-VACATION
                                 - NUM-ON-LEAVE.
COMPUTE  GROSS-PAY      = TOTAL-HOURS * WAGE.
COMPUTE  AVG-HOURS      = TOTAL-HOURS / NUM-ON-PAYROLL.
```

Use of arithmetic verbs

```
ADD  OVERTIME-HOURS TO REGULAR-HOURS
    GIVING TOTAL-HOURS.
SUBTRACT NUM-ON-VACATION, NUM-ON-LEAVE FROM NUM-EMPLOYEES
    GIVING NUM-ON-PAYROLL.
MULTIPLY TOTAL-HOURS BY WAGE
    GIVING GROSS-PAY.
DIVIDE   NUM-ON-PAYROLL INTO TOTAL-HOURS
    GIVING AVG-HOURS.
```

Mixing the two forms

```
COMPUTE  TOTAL-HOURS = OVERTIME-HOURS + REGULAR-HOURS.
SUBTRACT NUM-ON-VACATION, NUM-ON-LEAVE FROM NUM-EMPLOYEES
    GIVING NUM-ON-PAYROLL.
COMPUTE  GROSS-PAY    = TOTAL-HOURS * WAGE.
DIVIDE   NUM-ON-PAYROLL INTO TOTAL-HOURS
    GIVING AVG-HOURS.
```

Figure 2.3 : Duplicate Features in COBOL.

abbreviations (see Figure 2.1), but their legal use depends on the denotations for the other variables of the program. On the other hand, the PASCAL <u>with</u> statement (see Figure 2.2) clearly identifies abbreviated denotations over a precise span of text.

Consider also Figure 2.4, which illustrates a typical use of the PASCAL <u>case</u> statement, along with an equivalent compound <u>if</u>-statement (in fact, the <u>case</u> statement is undefined when the value of the selection expression does not fall among the alternatives specified; an <u>otherwise</u> clause would be welcome). The <u>case</u> statement avoids a clumsy nesting of <u>if</u>'s and is easier to read. Unfortunately, the PASCAl <u>case</u> statement is much too limited. A recent proposal for a more powerful <u>case</u> statement (Weinberg, Geller and Plum 75) seems promising. However, the additional complexity of this proposal remains to be investigated.

Providing multiple forms for a single concept generally makes a language more difficult to learn, use, and read. Alternate forms should be introduced only to promote readability, and only when they do so, without creating an undue increase of the complexity.

not clear to me
is a <u>CASE</u> stmt a good
example of multiple forms
or a bad one?

Not clear

Sample PASCAL IF Statement

```
     if command = insert then insertlines(currentposition)
else if command = delete then deletelines(currentposition,
                                          linecount)
else if command = print  then printlines (currentposition,
                                          linecount)
else if command = search then
                                begin
                                  searchstring(currentposition,
                                     string,stringfound,newposition);
                                  if stringfound
                                     then currentposition := newposition
                                end
```

Sample PASCAL CASE Statement

```
case command of

     insert:  insertlines(currentposition);
     delete:  deletelines(currentposition,linecount);
     print:   printlines (currentposition,linecount);
     search:  begin
                searchstring(currentposition,string,
                             stringfound,newposition);
                if stringfound
                   then currentposition := newposition
              end
end
```

Figure 2.4: Alternate PASCAL Control Structures.

3. Simple features make simple languages.

It would be too simplistic to characterize the complexity of a language only by its size. Each construct has an inherent complexity as well as an interplay with other features.

A designer should be especially careful of features with a highly dynamic behavior. Consider the Algol 60 call-by-name feature: it is a powerful feature, not too difficult to learn (in the following discussion, we will ignore a possible clash of identifiers with call-by-name parameters. A call-by-name parameter can have a complex run-time behavior not reflected by its written representation. For example, "Jensen's device" (Figure 3) has been used to promote call-by-name parameters (Knuth 67). When considered alone, the declaration of the procedure SIGMA looks innocent indeed. The invocation of SIGMA seems natural because of its analogy with a classical mathematical notation. However, when the procedure declaration and its invocation are examined together, it takes some effort to realize that SIGMA is activated N+1 times to compute the double sum of the elements of an N*L array. Note that neither the declaration or the invocation of SIGMA 'explains' Jensen's device. Furthermore, if more descriptive names had replaced L, N, and A, the similarity with mathematics would no longer appear. This is a sufficient reason to question the usefulness of call-by-name parameters. A language designer should be very cautious of clever examples. They usually promote features of greater complexity than the eye can meet.

```
begin
     integer array  A [1:N, 1:L];
     integer  I, J;
     integer  GRANDTOTAL;
                 .
                 .
                 .
integer procedure SIGMA (K, LOW, HIGH, TERM );
             value LOW, HIGH;
             integer K, LOW, HIGH, TERM;
             begin
                  integer SUM;
                  SUM := 0 ;
                  for K := LOW step 1 until HIGH
                      do SUM := SUM + TERM ;
                  SIGMA := SUM
             end
        .
        .
        .

      GRANDTOTAL := SIGMA(I, 1, N, SIGMA(J,1,L,A[I,J]) );

        .
        .
        .

end
```

Figure 3 : Jensen's Device is used to sum the elements of an N x L array.

A further illustration is provided by our friend the goto statement. Its basic mechanism is simple to explain, but its interplay with other features leads to significant problems. Arbitrary branching usually requires that some variables be given definite values on entry or exit. These associations, however, are not explicit in the program text. A cleaner solution is offered by the basic one-in one-out control structures (see Ledgard and Marcotty 75). The advantage of one-in, one-out control structures is not only the explicit mention of the conditions upon which the control flow is modified, but also a clean behavior when combined together or with other features of the language.

A similar issue concerns the introduction of pointers in a high level language. Recursive data structures (Hoare 75) are an adequate substitute in most cases. They simplify program reading and specification by replacing pointer manipulations with logical operations on structures (note that PL/I provides a similar hiding mechanism).

In summary, the simplicity of a language relies as much in the number and the simplicity of basic features as in the simplicity of their interaction. The art of language design is to achieve a tolerable balance.

BUT...
They do not provide for dynamic manipulation of data structures. Note that even pascal has pointers.

Don't rule them out altogether

People use them indiscriminately

4. Functions should emulate their mathematical analogue.

Function and procedure facilities are the basic tools for program decomposition. They provide the operational abstractions necessary to manage complex problems. The usefulness of these abstraction tools is so important that they demand a careful design.

In most procedural languages, an analogy is made with conventional mathematics. Expressions in programming languages are meant to be read as expressions in mathematics. The invocation of functions within expressions hides irrelevant computational details and, most importantly, facilitates the use of new operational abstractions. Accordingly, our understanding of function facilities in programming languages is based on our mathematical background. In mathematics, a function is a mapping from a set of values to a set of values. In programming languages, a function is understood as an algorithmic transformation from input values to a single output value.

In most programming languages, there appear a number of discrepancies from the simple mathematical analogue. In particular, assignments in function declarations may cause side-effects. For example, consider the well-known Algol 60 program (Knuth 67) of Figure 4.1. Since the variable GLOBAL is modified within the body of the function SUCCESSOR, this program will print <u>false</u> rather than <u>true</u> (the Algol 60 Report leaves the order of evaluation of expressions undefined; however, the Report does not forbid modifications of globals in functions; consequently, the output of Figure 4.1 will be <u>false</u> or <u>true</u> depending on the implementation).

```
begin
     integer GLOBAL;

     integer procedure SUCCESSOR (FORMALPARM);
                       value FORMALPARM;
                       integer FORMALPARM;
                       begin
                         SUCCESSOR := FORMALPARM + 1;
                         GLOBAL := SUCCESSOR
                       end;

     GLOBAL := 0;
     print( (GLOBAL + SUCCESSOR(GLOBAL))
               = (SUCCESSOR(GLOBAL) + GLOBAL)   )
end
```

Figure 4.1: Modification of a global variable in an Algol 60 function.

Even the access to a global variable within a function declaration may cause a loss of transparency in an expression. In the example of Figure 4.2, the global variable INCREMENT is modified between two invocations of INCREASE. The meaning of INCREASE is thus dynamically modified and, although the two invocations are identical, different results will be produced.

Another discrepancy occurs when parameters of a function are modified within the function declaration. In the well-known example (Weil 65) of Figure 4.3, the function INCREMENT BY NAME is evaluated twice during the invocation of ADD BY NAME. Since INCREMENT BY NAME modifies its parameter, successive evaluations do not yield the same result.

Many other languages also allow side-effects in function invocations. For easier validation and better readability, we recommend that functions be implemented according to the simple model discussed earlier. In particular, all parameters should be considered as input values that are "evaluated" upon invocation. No assignment should be performed on parameters within functions. If references to global are allowed, the function declaration should at least contain mention of this fact in its header.

good

Designing functions from a simple mathematical model implies strong restrictions on their use. However, the very nature of these restrictions forces the programmer to devise clear solutions and enables the program reader to rely on a transparent notation for expressions.

The ability to use a global in a function is often necessary & desireable and I believe it would be a mistake to rule it out entirely.

```
begin
     integer INCREMENT;
     integer procedure INCREASE (BASE);
                        integer BASE; value BASE;
                        INCREASE := BASE + INCREMENT;


     INCREMENT := 1;
     print( INCREASE(1) );

     INCREMENT := 100;
     print( INCREASE(1) )
end
```

Figure 4.2: Modification of a function through a Global Variable in Algol 60.

```
begin
     integer innocent;

     integer procedure INCREMENT BY NAME (corrupt);
                       integer corrupt;
                       begin
                            corrupt := corrupt + 1;
                            INCREMENT BY NAME := corrupt
                       end;

     integer procedure ADD BY NAME (evil);
                       integer evil;
                       ADD BY NAME := evil + evil;

     innocent := 1;
     print( ADD BY NAME( INCREMENT BY NAME(innocent) ) );
     print( innocent )
end
```

Figure 4.3: Algol 60 call-by-name parameters.

5. A clear distinction should be made between functions and procedures.

Many abstractions encountered in programming cannot be programmed with functions. An operation may contain inherent side effects, invoke input-output, create or update a structure, or modify the run-time environment. It would be misleading to extend the simple model of functions to these abstractions for, unlike the analogue of function invocations with mathematical expressions, the procedure invocation is the analogue of a statement.

The main conceptual difference between procedures and functions is that modifications of the execution environment are allowed in procedures. In most languages, global variables may be referenced and modified in procedures. Before further discussing the issue of global variables, it must be pointed out that, in some cases, the use of globals results from poor language design. Consider a state transition table, a keyword mapping table, or any kind of unvarying information whose lookup is limited to one module. To represent such a constant object in some languages (e.g. PASCAL), a variable must be declared and initialized outside of the module where it is used, i.e. it must be global. A more natural solution would be to have local, stuctured constants.

But what if it is not constant??

there is some use of globals

Since modification of the execution environment is the essence of a procedure, problems of poor readability and difficult validation that were eliminated for functions must be re-examined. The design of a procedure facility should minimize these problems (see Gannon and Horning 75). In the first place, a complete specification of interfaces should be required (Wulf and Shaw 73, Deremer and Kron 76). The procedure header should indicate which parameters are input values, output results, and updated variables, as shown in Figure 5.1. The language processor should make sure that each parameter is used properly according to the header specification. Thus, efficient parameter passing modes can be generated by the compiler. The procedure invocation (CALL statement or procedure statement) should contain similar information as illustrated in Figure 5.2.

As to global variables accessed in procedures, they should be regarded as "implicit" parameters. Their use may increase the conciseness of procedure invocations and thus improve readability. The procedure header, however, should explicitly mention all global variables that are referenced or updated (see Figure 5.1).

The basic design of function and procedure facilities we have presented may appear very restricted. Indeed, there are attractive extensions like polymorphic procedures or procedures with functional parameters whose arguments are variable in number and type (e.g. see Gries and Gehani 76). However, the effect of such extensions on readability and ease of validation should be carefully assessed before their introduction in a language.

```
procedure SWAP (input I, J: integer);

     updated var
                    A: array [1..MAX_ELEMENTS] of integer;
     var
                    TEMP: integer;

     begin
          TEMP := A[I];
          A[I] := A[J];
          A[J] := TEMP;
     end
```

Figure 5.1: Complete Specification of Interfaces in Procedure Declaration.

```
                .
                .
                .
parse_if_statement( input current_pos,
                    output parse_error, subtree, new_pos);

if parse_error = serious
    then recover_statement(update current_pos,
                           output fatal_error);
                .
                .
                .
```

Figure 5.2: Specification of Actual Parameters in Procedure Invocations.

6. Multiple data types should be supported.

A data type is usually defined as a distinguished set of values and associated operators. Since all programming languages are designed to manipulate some kind of data, they all provide one or more data types.

So called "typeless" languages are indeed a contradiction in terms. In LISP (Weissman 67) and GEDANKEN (Reynolds 70), values may be atoms, integers, reals or booleans. However, no declaration can restrict the range of values taken by identifiers. A true "unitype" language is BLISS (Wulf, Russel, and Habermann 71). BLISS provides only one basic type, namely bit patterns, to represent all quantities.

Although the above languages have been widely accepted, we find them difficult to read, mainly because the interpretation of identifiers cannot be derived from their declaration or from the context in which they are used. We believe that the association of a name with a specific data type should be made explicit. At the same time, a language should offer a sufficient number of basic data types (e.g. boolean, character, integer, real) and structuring mechanisms (e.g. array, string, record) to avoid obscure programming.

Another problem with many current programming languages is implicit type coercion. Implicit type coercion often makes program validation and modification hazardous. We believe that there should be no automatic type conversion in a language, except, perhaps, from integer to real or from subrange to scalar. Other conversions should

why specify integer or real at all?? Let that be compiler decided from usage. like BASIC be specified by the programmer using built-in functions.

Providing multiple basic data types and structuring facilities may appear sufficient. However, we believe that the programmer should be allowed to define his own data types to adapt the language to an application. There are two separate aspects to the notion of a data type "extension": abstraction and implementation.

From the abstraction point of view, the programmer defines a new type by naming a set of objects and operators relevant to the application. For instance, the (limited) type definition facility of PASCAL offers the possibility to declare and name "new" classes of objects(Figure 6.1). Such a declaration helps clarifying the meaning of values that a variable of this type can assume.

The implementation aspect of a new data type consists in programming the representation and operators of this new type. The implementation is usually performed in terms of previously defined types and operators. For instance, Figure 6.2 shows the definition of the type "stack of integers" using the class facility of SIMULA 67.

What constitutes a good mechanism for a full data type facility is still being explored (e.g. see Conference On Data Abstraction 76). Some combination of the PASCAL and SIMULA facilities, where the exchange of information between a data type definition and its use would be tightly controlled, would provide great convenience (see Koster 76).

There are advantages to multiple data types other than abstraction and readability. First, a strict notion of type allows an extensive type checking to be performed at compile time. Being able to put more confidence in a syntactically correct program is

```
Type
     commandtype = (insert,delete,search,invalidcommand);

     tokentype   = (keyword,identifier,constant,
                    specialsymbol, unrecognizable);

     constanttype = (integerconst,realconst,string);
```

Figure 6.1: Sample PASCAL scalar type declarations.

```
class   stack ( maximumsize );
        integer maximumsize;

        comment This class defines the type stack of integers;


        begin
           integer array store [1:maximumsize];

           integer topindex, maxstorage;

           boolean procedure empty;
                     empty := (topindex ¼ 1);

           boolean procedure full;
                     full := (topindex = maxstorage);

           integer procedure top;
                     top := store[topindex];

           procedure push (token);
                     integer token;
                     begin
                        topindex := topindex + 1;
                        store[topindex] := token;
                     end;

           procedure pop (token);
                     name token;
                     integer token;
                     begin
                        token := store[topindex];
                        topindex := topindex - 1;
                     end;


           comment stack initialization at creation time;

           topindex := 0;
           maxstorage := maximumsize

        end class stack;
```

Figure 6.2: Declaration of the Class "Stack of Integers" in SIMULA 67.

important when maintaining it. Second, since axiomatic definitions of types can be produced, validation of programs can be accomplished more rigorously (see Guttag 76).

Third, a good data type facility makes a variety of implementation economies possible because more information about the data elements is available at compile. Thus, it is due, in part, to its data type facility (an extensible one at that) that PASCAL programs run as efficiently as they do.

NFC Type, but leave it out

7. Similar features should have similar forms.

Syntax has often been compared to the icing that covers a cake. Of course, if the cake is stale, the icing will little improve it. But if the cake is fine, the taster will soon associate its flavor with its appearance. In programming languages, a concept and its external representation are often taken synonymously. For example, we often use the terms "if-statement" and "while-statement" rather than the terms "selection control structure" and "iteration control structure". The association between concepts and their representation is an important human factor in the design of a programming language. To benefit from such associations and promote readability, similar syntactic forms should be used for similar features.

Our first example deals with the concept of declarations and their syntactic forms. A sample of possible PL/I declarations appears in Figure 7.1a. The syntax of these declarations is somewhat confusing. The variable declarations and procedure declarations do not follow a similar scheme. In the variable declarations, the LIKE attribute provides the aggregate PURCHASE with the same structure as SALE, although this is not so obvious at first glance. A structure itself is indicated only by an integer before the major component name. The amount of information provided by each declaration is not identical, mainly because of default attributes. In the procedure header, the declaration of formal parameters takes two steps.

In comparison, the PASCAL declarations of Figure 7.1b. are

(a) PL/I

```
DECLARE  INDEX FIXED;

DCL  1 SALE,
       2 DATE,
           3 YEAR CHAR(2),
           3 MONTH CHARACTER(3),
           3 DAY CHAR(2),
       2 TRANSACTION,
           3 (ITEM,QUANTITY) FIXED (7,0),
           3 PRICE,
           3 TAX FIXED;

DECLARE 1 PURCHASE LIKE SALE;

UPDATE_STOCKS : PROC (ARTICLE,AMOUNT);
                DCL  (ARTICLE,AMOUNT) FIXED (7,0);
```

(b) PASCAL

```
type operation =
        record
           date: record
                    year:  array [1..2] of char;
                    month: array [1..3] of char;
                    day:   array [1..2] of char
                 end;
           transaction:
                 record
                    item,
                    quantity: integer;
                    price:    integer;
                    tax:      integer
                 end
        end;

var  index:    integer;
     sale:     operation;
     purchase: operation;

procedure updatestock (article, amount: integer);
```

Figure 7.1: PL/I and PASCAL declarations.

---

longer, but clearer. A similar scheme is used for type declarations, variable declarations, and procedure declarations. Notably, the declarations of a structures variable and of an integer variable follow the same scheme.

As a second example, consider the syntax of PASCAL control structures (without the goto). Some disparity in the form of control stuctures can be noticed. The case and end keywords of a case statement (see Figure 2.5) clearly delimit this construct in the program text; conversely, the if statement is not bracketted in a similar fashion (Figure 7.2). A more important discrepancy also appears. Whereas a list of statements can be used in a repeat...until construct, the if, case, and while constructs may only accomodate a single statement. To include a sequence of instructions in an if or a case statement,a clumsy begin...end bracketting pair must be added. Since control structures form a class of features, the same syntactic scheme should apply for all of them. Accordingly, examples of a modified PASCAL syntax are shown in Figure 7.3.

A discussion of statement lists cannot omit the "missing semicolon" problem . The use of a separator in statement lists needlessly singles out the last statement, which does not have an ending punctuation mark. This rule is difficult to learn and remember (see Gannon and Horning 75). Conversely, the use of a statement terminator provides a more natural rule for all statements (see Figure 7.3).

Similar forms for similar features can greatly reduce the conceptual complexity of a programming language. The likeness of forms

```
if  (linecount = maxlineperpage)
      then
            begin
                  pagecount := pagecount + 1;
                  newpage(printfile);
                  printheader(printfile, pagecount, date);
                  linecount := 1;
             end
      else
            linecount := linecount + 1
```

Figure 7.2: A sample PASCAL if statement.

```
(a)   if (line_count = max_line_per_page)
            then
                  page_count := page_count + 1;
                  new_page(print_file);
                  print_header(print_file, page_count, date);
                  line_count := 1;
            else
                  line_count := line_count + 1;
      endif;


(b)   while  (input_char in digits)  do

            number := number*10 + int_value(input_char);
            read(input_char);

      endwhile;


(c)   repeat

          digit := digit + 1;
          one_tenth := number div 10;
          decimal_digit := number - 10*one_tenth;
          number := one_tenth;

      until  (number = 0);


(d)   case  command  of

            insert: insertlines(current_position);

            delete: deletelines(current_position, line_count);

            print:  printlines(current_position, line_count);

            search: searchstring(current_position, string,
                                 string_found, new_position);
                    if string_found
                        then current_position := new_position;
                    endif;
      endcase;
```

maybe

but...

Figure 7.3: Control Structures with Full Bracketing.

indicates to the user the likeness of contents. These associations should be carefully designed, for even a single anomaly can confuse the user.

8. Distinct features should have distinct forms.

The association between concepts and their representation supports the use of similar forms for similar features. Reciprocally, it is important not to give rise to misleading associations. Distinct concepts should be emphasized by distinct syntactic forms.

The formal parameters and the local variables of a procedure form distinct conceptual categories. In FORTRAN and PL/I (see Figure 8.1), formal parameters appear in the procedure heading, but their declaration is made along with the declarations of local variables. On the other hand, this distinction is well made in ALGOL 60. The declaration of formal parameters are located in the module header. However, some confusion remains because a formal parameter may occur up to three times in the header (e.g. LOWBOUND and UPPERBOUND in Figure 8.1). A better solution is offered in PASCAL where the declarations of formal parameters are grouped in the procedure header.

The declarations of variables and the sequence of operations performed upon these variables represent distinct concepts. In COBOL, this distinction is made by using DATA and PROCEDURE DIVISION's. On the other hand, PL/I allows declarations to be located anywhere in a procedure. A similar objection can be made against the FORMAT statement in FORTRAN. FORMAT statements are not executable and they seriously slow down program reading when located among executable statements.

In the previous section, we proposed a full bracketting for control structures. Of course, these control structures differ in some manner, for they are not duplicate features. Unfortunately, this difference is not generally emphasized enough. In PASCAL, the end keyword is the closing bracket of too many constructs, e.g. blocks, compound statements, and case statements. Readers can easily be confused by the "matching end" problem. Distinct constructs should have distinct pairs of brackets. Preferably, the two brackets should be short and have the same length; but most of all, they should be readable. For this reason, fi, esac, elihw, or nigeb are not acceptable. In this paper (see Figure 7.3), we used endif, endcase, endwhile, and end, but we are not fully satisfied with them. great!!

Similarly, some languages use the "+" symbol to denote addition, set union, and boolean OR. This can lead to obscure constructs, because the exact interpretation of a single "+" must be derived from the type of its operands. Using "+", "U", and "OR" surely adds to readability. The programmer, not the designer should be responsible for any possible operator overloading. However, the character sets used in current programming languages are limited, and it might still be necessary to associate two different meanings with a single token. This should only be allowed where the the contexts for each interpretation are so different that no confusion arises. why should they be?

Transparency can be obtained by combining similar forms for similar features and distinct forms for distinct features. Thus, the similarities and differences of basic concepts are easily apparent to the user, who can rapidly learn to recognize the various forms in programs.

FORTRAN

```
SUBROUTINE PLOT(LOW, UPPER, CURVE)
       REAL LOW
       DIMENSION LINE(120)
       ...
```

PL/I

```
PLOT_CURVE: PROCEDURE(LOW_BOUND, UPPER_BOUND, CURVE);
       DCL (LOW_BOUND, UPPER_BOUND) FLOAT;
       DCL CURVE  ENTRY (FLOAT) RETURNS FLOAT;
       DCL LINE (120)
       ...
```

Algol 60

```
procedure PLOTCURVE( LOWBOUND, UPPERBOUND, CURVE);
                     value LOWBOUND, UPPERBOUND;
                     real  LOWBOUND, UPPERBOUND;
                     real procedure CURVE;
   begin
        integer array LINE [1:120];
        ...
```

PASCAL

```
procedure PLOTCURVE( LOWBOUND, UPPERBOUND: real;
                     function CURVE: real       );

       var LINE: array [1..120] of integer;
           ...
```

Figure 8.1: Formal Parameters and Local
Variables for a plotting routine.

9. Remember the reader.

Once a program written, it will be read many times by its author or other programmers. It is thus important that the program listing clearly convey all information necessary to the reader.

The overall structure of a program and the basic organization of modules are the outlines on which a program reader establishes his understanding. Consider the task of reading a PASCAL or Algol 60 program you have never seen before. First, you will probably inspect the global declarations. Then, you will turn the pages to the end of the listing to find the body of a program. However, further examination of the variable declarations is needed to grasp important details of the program body. Much back and forth page flipping will occur before the first level of the program is understood. For each successive level, the same process is repeated, but with more difficulty, because the boundaries of each module are less apparent.

In general, the top-down development of a program exhibits the overall structure of a tree. Reading and understanding such a program is simplified if the program were presented in top-down fashion. To achieve top-down readability, the program listing should represent a breadth first traversal of the program tree. Thus, the program reader is led step by step through the successive levels of the program with minimum effort. As mentionned above, PASCAL and Algol 60 do not allow such a presentation. In FORTRAN, the programmer can choose the textual order of his modules, but no relative position is enforced. PL/I allows any combination of the Algol 60 and FORTRAN schemes.

The program code alone is usually inadequate to explain all of a program. Additional information must be provided by the programmer, e.g. the meaning of important variables, the description of algorithms, the peculiarities of a run-time environment, and references to existing documentation, etc. To promote this pratice, a languages should offer easy and secure documentation tools.

Provision for long names, along with a "break" character (e.g. the "_" in "CURRENT_POSITION"), represents an incentive to imbed documentation in the code. Possible break characters are the hyphen (COBOL) and the underscore (PL/I). The Algol 60 and FORTRAN convention where blanks may be interspersed arbitrarily in identifiers (e.g. ADD BY NAME in Figure 5.1) is not recommended, for various occurences of an identifier may look quite different.

More comprehensive documentation is usually given in comments. The following kinds of information are provided in comments:

a) General information: e.g. program purpose, modification record, references to external documentation, and run-time requirements.

b) Module summary: e.g. specification of the local computations, input and output domains, and algorithm used.

c) Statement grouping: e.g. identification or paraphrase of a group of statements to highlight their logical content as a unit.

d) Statement support: e.g. emphasis of a crucial step, assertions, and precise meaning of constant and variables.

Unfortunately, most languages do not provide adequate comment facilities. In COBOL, general information is given in the

IDENTIFICATION DIVISION and in the ENVIRONMENT DIVISION, but the remaining types of comments are not distinguished and must be made on a line by line basis. PL/I and PASCAL offer a single parenthetic scheme which does not distinguish between the various types of documentation. There is little need to mention the highly complex rules for Algol 60 comments and their mediocre readability.

In our opinion, a single comment scheme is rarely sufficient to encompass all possibilities of the above classification and, at the same time, to emphasize their differences. On one hand, general information and module summaries appear usually in dense blocks at the beginning of programs and modules: a simple parenthetic scheme allowed only in module headers is needed to accomodate the type of documentation. On the other hand, statement grouping and statement support comments are usually short. A line oriented comment scheme would be more appropriate for this type of comment. One such scheme might be the use of a distinguished token, say "/*", to begin the comment anywhere on a line; a comment would be implicitly terminated by the end of the line. Specific designs could introduce additional schemes, e.g. assertion comment.

In summary, a programming language should offer easy and secure documentation tools to help the programmer produce readable listings. Indeed, the top-down listing feature and viable comment schemes do not appear easy to devise and require further study. But their usefulness makes it an important topic for careful design.

Parting Comments.

UTOPIA 84 is still a long way off. The selection of the primitives of a language and the elaboration of data type facilities are important issues that we barely touched upon. Moreover, the design of a comfortable operating environment, including input/output primitives, and the quality of an implementation have a serious effect on the acceptance of a language.

Through the design principles presented in this paper we have tried to emphasize that all consequences of a design decision should be evaluated. Each design decision should promote ease of learning, program validation, and program maintenance. We cannot underestimate the use of formal definitions in the language design cycle, for they should provide useful indications on the simplicity and clarity of the result. Above all, the designer should strive to keep a language small, consistent, and readable.

A note on implementation must be made. Although we have given little consideration to efficiency of implementation, we doubt that any of our recommendations would lead to high inefficiency. Even so, if one considers the actual cost of software development and maintenance, a sensible gain in readability justifies some loss of efficiency.

In parting, we must admit that some notions used in this paper, like readability, remain purely subjective. Language designers may be easily misled if they keep to their own notions. They must listen to the users and interpret their complaints. After all, users remain the ultimate judges in language design.

Acknowledgments.

We are grateful to Michael Marcotty for his helpful comments on the drafts of this paper. We also would like to thank Andrew Singer, Louis Chmura, Caxton Foster, and Amos Gileadi for fruitful discussions.

REFERENCES:

CDC 71
Control Data Corporation. Simula Reference Manual. Publication No 602348000 (1971).

CONFERENCE ON DATA ABSTRACTION 76
Conference on Data: Abstraction, Definition, and Structure. March 22-24, Salt Lake City, Utah. Sigplan Notices, Vol 11, Special Issue (April 1976),pp.1-190.

DEREMER AND KRON 76
DeRemer, F., and Kron, H. Programming-in-the-large versus Programming-in-the-small. IEEE Transactions on Software Engineering, Vol SE-2, No 2 (June 1976), pp.80-86.

DIJKSTRA 68
Dijkstra, E.W. Goto Statement Considered Harmful. Comm. of the ACM, Vol 11, No 3 (March 1968) pp.147-148.

ECMA/ANSI 74
European Computer Manufacturers and American National Standards Institute. PL/I. ECMA/TC10/ANSI.X3J3. BASIS 1-12 (July 1974).

GANNON AND HORNING 75
Gannon, J.D., and Horning, J.J. Language Design for Programming Reliability. IEEE Transactions on Software Engineering Vol SE-1, No 2 (June 1975) pp.179-191.

GRIES AND GEHANI 76
Gries,D., and Gehani, N. Some Ideas on Data Types in High Level Languages. Conference on Data: Abstraction, Definition and Structure. Sigplan Notices, Vol 11, Special Issue (April 1976), p.120.

GUTTAG 76
Guttag, J. Abstract Data Types and the Development of Data Structures. Conference on Data: Abstraction, Definition and Structure. Siplan Notices, Vol 11, Special Issue (April 1976), p.72.

HOARE 72
Hoare, C.A.R. Hints on Programming Language Design. Computer Science Department. Stanford University. Tech. Rep. STAN-CS-74-403 (December 1973) pp.1-32.

HOARE 75
Hoare, C.A.R. Recursive Data Structures. International Journal of Computer and Information Sciences, Vol 4, No 2 (1975) pp.105-132.

JENSEN AND WIRTH 74
Jensen, K., and Wirth, N. PASCAL User Manual and Report. Lectures Notes in Computer Science NO 18, Springer Verlag (1974).

KNUTH 67
Knuth, D.E. The Remaining Trouble Spots in Algol 60. Comm. of the ACM, Vol 10, No 10 (October 1967) pp.611-618.

KNUTH 74
Knuth, D.E. Stuctured Programming with Go To Statements. Computing Surveys, Vol 6, No 4 (December 1974) pp.261-302.

KOSTER 76
Koster,C.H.A. Visibility and Types. Conference on Data: Abstraction, Definition, and Structure. Sigplan Notices, Vol 11, Special Issue (April 1976), pp.179-190.

LEDGARD AND MARCOTTY 75
Ledgard, H.F., and Marcotty, M. A Genealogy of Control Structures. Comm. of the ACM, Vol 18, No 11 (November 1975) pp.629-639.

MURACH 71
Murach, M. Standard COBOL. SRA (1971)

NAUR 63
Naur, P. (Editor) Revised Report on the Algorithmic Language Algol 60. Comm. of the ACM, vol 6, No 1 (January 1963) pp.1-17.

REYNOLDS 70
Reynolds, J.C. GEDANKEN: A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept. Comm. of the ACM, Vol 13, No 5 (May 1970) pp.308-319.

WEIL 65
Weil, R.L. Jr. Testing the Understanding of the Difference between Call by Name and Call by Value in Algol 60. Comm. of the ACM, Vol 8, No 6 (June 1965) p378.

WEINBERG, GELLER AND PLUM 75
Weinberg, G.M., Geller, D.P., and Plum, T.W-S. IF-THEN-ELSE Considered Harmful. Sigplan Notices, Vol 10, No 8 (August 1975) pp.34-44.

WEISSMAN 67
Weissman, C. Lisp 1.5 Primer. Dickenson Publishing Company (1967).

WIRTH 74
Wirth, N. On the Design of Programming Languages. Information Processing 74. North Holland Publishing Company (1974) pp.386-393.

WULF, RUSSEL AND HABERMANN 71
Wulf, W.A., Russel, D.B., and Habermann, A.N. BLISS: a Language for Systems Programming. Comm. of the ACM, Vol 14, No 12 (December 1971) pp.780-790.

WULF AND SHAW 73
Wulf, W., and Shaw, M. Global Variables Considered Harmful. Sigplan Notices, Vol 8, No 2 (February 1973) pp.28-34.

X3J3 76
American National Standards Committee X3J3. Draft proposed ANS FORTRAN. Sigplan Notices, Vol 11, No 3 (March 1976).