Acta Informatica 8, 97 – 112 (1977)

Language Design Methods Based on Semantic Principles

R. D. Tennent

Department of Computing and Information Science, Queen's University,
Kingston, Ontario, Canada

Summary. Two language design methods based on principles derived from the denotational approach to programming language semantics are described and illustrated by an application to the language Pascal. The principles are, firstly, the correspondence between parametric and declarative mechanisms, and secondly, a principle of abstraction for programming languages adapted from set theory. Several useful extensions and generalizations of Pascal emerge by applying these principles, including a solution to the array parameter problem, and a modularization facility.

"At first sight, the idea of any rules or principles being superimposed on the creative mind seems more likely to hinder than to help, but this is really quite untrue in practice. Disciplined thinking focusses inspiration rather than blinkers it."

G.L. Glegg: "The Design of Design"

1. Introduction

1.1. Semantic Principles

One of the motivations for the development of mathematical semantic models for practical programming languages has been to provide tools, concepts, and principles to assist designers of new languages and language features. The aim of this paper is to propose and illustrate systematic approaches to the design of two important aspects of programming languages. The methods are based on principles which derive from the approach to programming language semantics of D. Scott and C. Strachey and followers; the modern form of this theory is described in Milne and Strachey [28], but many of the concepts may be traced back to earlier work, such as McCarthy [26], Landin [21], and Strachey [34]. Tennent [36] is a tutorial introduction to the approach.

The first principle, which we term the principle of *correspondence*, was first stated and exploited by Landin [23]:

"In almost every language a user can coin names, obeying certain rules about the contexts in which the name is used and their relation to the textual segments that introduce, define, declare, or otherwise constrain its use. These rules vary considerably from one language to another, and frequently even within a single language there may be different conventions for different classes of names, with near-analogies that come irritatingly close to being exact. So rules about user-coined names is an area in which we might expect to see the history of computer applications give ground to their logic. Another such area is in specifying functional relations. In fact these two areas are closely related since any use of a user-coined name implicitly involves a functional relation; e.g. compare

$x(x+a)$ **where** $x=b+2c$ $\qquad$ $f(b+2c)$ **where** $f(x)=x(x+a)$."

The correspondence between the semantics of declarative and parametric mechanisms has significantly influenced the design of only a small number of languages [23, 9, 39, 31, 35]. In the next section, a simple and useful design discipline is proposed which is based on the correspondence principle.

The second semantic principle will be termed the principle of *abstraction.* "Abstraction" is usually understood as a process of extracting general structural properties in order to allow relatively inessential details to be disregarded, and in this sense it is familiar to every mathematician and programmer. The term also has a technical sense in certain branches of mathematics and the theory of computation. In set theory [e.g. 33], the principle of abstraction is that any property (i.e. one-place predicate formula) $P[x]$ defines a set $\{x \mid P[x]\}$. Other forms of abstraction are also possible; for $R[x, y]$ a two-place predicate, $\{\langle x, y\rangle : R[x, y]\}$ might be a binary relational abstraction [30], and for $E[x]$ a term (expression) with free variable x, $\lambda x.\, E[x]$ might be a functional abstraction [5]. In this paper, the term "abstraction" will always have this technical significance.

In set theory, the principle of abstraction is of critical importance because were it to be used in the unfettered "naive" form stated above, it would lead to inconsistencies such as the well-known "paradox" of B. Russell. What this has to do with programming language design is that Scott's theory of computation and denotational semantic models of programming languages have shown that the meanings of reasonable programming language features satisfy a continuity condition, and that within this theory "unrestricted" abstraction *is* permissible.

This theoretical result motivates the following principle of abstraction for programming languages: an abstraction facility may be provided for any semantically-meaningful category of syntactic constructs. Procedure definitions for abstraction from statements in ALGOL 60 and lambda expressions for abstraction from forms in LISP are familiar examples of such facilities.

The principles of correspondence and abstraction are independent semantic concepts; however, for their application to language design it is necessary to consider them jointly, because parametric mechanisms are always involved in abstraction and, in almost all languages, the only abstraction facilities are declarations.

1.2. Design Methodology

The principles discussed in the preceding section are not principles of design, but rather of formal semantics; however, they do suggest and legitimize systematic approaches which can help to achieve design ends. From the principle of correspondence it may be concluded that the declarative and parametric mechanisms of a language should be designed together by comparing and reconciling corresponding declarative and parametric forms. This discipline helps in both discovering and correcting unnecessary restrictions, "missing" facilities, irregularities in the syntax, semantics, or terminology, unexpected interactions or complications, and so on.

The design approach suggested by the principle of abstraction is based on identifying all of the semantically-meaningful syntactic categories of the language and then designing a coherent set of abstraction facilities for each of these.

The rest of this paper describes an experiment in language design illustrating these semantically-based methods. The starting point will be the language Pascal [18] which is well documented and widely used, so that we may assume familiarity with it. Analysis of the declarative, parametric, and abstraction mechanisms of Pascal along the lines described above will point out many minor problems in the language, but will also suggest revisions which would rectify these problems without requiring any major changes to the underlying concepts of the language.

2. Terminology and Meta-Variables

2.1. Syntax

(i) The left-hand side and right-hand side of a definition or comparable construction will be referred to as its *formal parameter* and *actual parameter*, respectively.
(ii) Capital Greek letters, possibly with primes or subscripts, will be used as syntactic meta-variables, as specified in Table 1.
(iii) A *sequencer* [28] is a syntactic construct whose interpretation always results in a "jump"; in Pascal, the only sequencers have the form:

Γ; **go to** N

Table 1. Syntactic categories and meta-variables

Syntactic category	Meta-variable	Comments
Numerals	N	
Identifiers	I	
Formal parameters	Φ	
Specifiers	Π	To be discussed in Section 4.2
Type expressions	T	e.g. the actual parameter of a type definition
Static expressions	Σ	Statically-evaluable
Variable expressions	Ξ	e.g. the left-hand side of an assignment
Expressions	E	e.g. the right-hand side of an assignment
Declarations	Δ	Including definitions and declaration-lists
Statements	Γ	Including blocks and statement-lists

2.2. Semantics

The conventional value of an expression will be termed an *R-value*; for a static expression, this will be qualified as being a *static* R-value. The denotation of a variable expression (without coercion) will be termed an *L-value*. The values of type expressions and labels will be termed *type values* and *label values*, respectively. Functions, procedures, and so on will be termed *abstraction values*.

3. Analysis

3.1. Correspondence

In this section the declarative and parametric forms in Pascal are systematically compared in the manner suggested by the principle of correspondence. This analysis is summarized in Table 2 and motivates the following critical comments:

(i) There are no "static" (**const** and **type**) parametric forms. A very serious consequence of this is that a programmer cannot define an abstraction to operate on array parameters with arbitrary index types or bounds; other commentators have attributed this to the lack of dynamic array bounds [11], or to constraints imposed by static type-checking [19, 24, 43, 3].

(ii) According to Jensen and Wirth [18] the control identifier of a **for** statement cannot be updated in the controlled statement, and its value is undefined on normal exit from the iteration. This is as close as Pascal gets to allowing an identifier to be bound directly to a non-static R-value. Some of the advantages of a general mechanism for R-value binding have been pointed out by Wirth and Hoare [44]:

"The intention of the programmer can be made explicit for the benefit of the reader, and the translator is capable of checking that the assumption of constancy is in fact justified. Furthermore, the translator can sometimes takes advantage of the declaration of constancy to optimize a program."

An additional advantage is that without it, a programmer cannot pass the R-value of a non-assignable structure such as a file to an abstraction and must use a

Table 2. Parameter and declaration correspondence in Pascal

Denotation	Declarative construct	Parametric construct	
		Formal parameter	Actual parameter
Static R-value	**const** $I = \Sigma$		
Type value	**type** $I = T$		
Non-static R-value	**for** $I := E \ldots$		
New L-value	**var** $I : T$	$I : I'$	E
Existing L-value	**with** Ξ **do** …	**var** $I : I'$	Ξ
Abstraction value	**function** $I(\ldots; \Phi_i; \ldots) : I'; \ldots$	**function** $I : I'$	I
Label value	**label** $N; \ldots; N : \Gamma; \ldots$		

var parameter. An earlier version of Pascal [40] had a parametric mechanism for R-value binding (inconsistently termed the **const** parameter), but it was replaced by the "value" parameter [1].

(iii) The **for** statement is not really a declarative construction because the control identifier must be declared in an enclosing procedure heading and outside of **for** loops can be used as an ordinary variable. The advantages of making the control identifier local to the **for** statement are discussed in [13].

(iv) There is a misleading notational irregularity in the use of **var** for binding an identifier to a *new* L-value in declarations and to an *existing* L-value as a parameter; furthermore, the terminology of "value" parameter is inconsistent with the corresponding **var** declaration.

(v) Any data type may be specified for a **var** declaration, but the type of a "value" parameter must be assignable.

(vi) The restriction to a type identifier (I') rather than a general type expression (T) at four places in Table 2 is unnecessary; this restriction apparently originates in an earlier implementation of Pascal in which **type** definitions were interpreted somewhat differently.

(vii) There is no explicit actual parameter in a **var** declaration, so that there is no mechanism for initializing the new L-value at the time of allocation.

(viii) There is no general declarative mechanism for binding identifiers to existing L-values.

(ix) The **with** statement in Pascal allows abbreviated references to the fields of a record; however, the "formal parameters" (the field names for the record type) are implicit, and this seems to impair program readability in non-trivial applications.

(x) The types of parameters of abstractional parameters cannot be specified, preventing complete static type-checking.

(xi) There is no parametric mechanism for label values.

This list of irregularities, missing or overly-specialized facilities, unnecessary restrictions, and so on is fairly lengthy, but the problems are quite minor and easily rectified. Many have been pointed out before in commentaries on Pascal [11, 19, 24, 43, 10, 4]; the significance of this analysis is that it exemplifies a *systematic* approach to discovering imperfections and flaws in a design of declarative and parametric mechanisms which does not require a complete semantic definition or implementation. For further evidence of the usefulness of this approach, the reader is invited to carry out a similar analysis of ALGOL 60, and compare the results with the trouble-spots, mistakes, and features that could have been in ALGOL 60 discussed by Wichmann [38].

3.2. Abstraction

Identification of the semantically-meaningful syntactic categories of a language is not as straightforward as in may seem. Many distinctions necessary in a concrete

Table 3. Abstraction in Pascal

Syntactic category	Abstraction values
Statements	Procedures
Expressions	Functions
Type expressions	
Static expressions	
Variable expressions	
Declarations	
Sequencers	

syntactic specification (such as among expression, factor, term, etc.) have only syntactic significance; on the other hand, a syntactic specification may not need to make a subtle distinction with semantic significance. For example, the **go to** construct is usually termed a statement, but from a semantic point of view it is more properly classified as a sequencer because of the nature of its interpretation.

As a last example, we mention the guarded command concept introduced by Dijkstra [8]; this might seem to be a semantically-meaningful category, but the paper does not give a semantic interpretation for guarded commands themselves. The non-deterministic **if** ... **fi** and **do** ... **od** constructs are defined by breaking up the constituent guarded commands into guards and statement-lists. If semantic meanings were assigned to guarded commands per se, this would suggest an abstraction facility with guarded commands as bodies.

The semantically-meaningful categories in Pascal and the associated abstraction facilities are summarized in Table 3. In addition to the obvious observation that five out of seven possibilities do not exist in Pascal, the only remark that needs to be made is that the body of a function is actually a statement rather than an expression, and its value is obtained by using the name of the function as a write-only variable (in L-value contexts).

4. Synthesis

4.1. Introduction

In this section the use of the proposed methods in language design will be illustrated by outlining one possible revision of Pascal which avoids the difficulties discussed in the preceding analysis, yet does not make any significant change in the underlying conceptual framework. We have tried to keep the notation as close as possible to that of Pascal; the following simple extensions and modifications will be assumed in the examples:

(i) A static expression is allowed to be any statically-evaluable expression.

(ii) A block (qualified statement) has the general form:

with Δ
do Γ

and can appear in any appropriate context, and not just as a procedure body.

(iii) Abstraction definitions will have the forms

procedure $I(\ldots;\Phi_i;\ldots)=\Gamma$
or: **function** $I(\ldots;\Phi_i;\ldots):T=E$

The use of expressions rather than statements as function bodies is more consistent with the principle of abstraction and avoids many semantic complications [29, 25]. Although our examples will not require it, a block expression construction (as in [44]) would be needed to allow imperative computation in function bodies.

(iv) A qualified declaration construction adapted from [23] will be used to provide a set of abstractions with "own" variables:

private Δ
within Δ'

The effect is that the scope of Δ is limited to Δ'; however, the dynamic lifetimes of L-values allocated in Δ continue until exit of the block in which the construction appears.

(v) The statement

loop Γ **end**

is executed by repeatedly executing Γ, presumably until an appropriate jump out of the construct.

4.2. Correspondence

The "theoretical" ideal suggested by the principle of correspondence may be summed up as follows: for any formal parameter Φ and compatible actual parameter A, the effect on a block body of the qualifying declaration $\Phi=A$ should be identical to the effect on an abstraction body having Φ in its formal parameter list when A is the corresponding argument. The proposed declarative and parametric forms summarized in Table 4 (which should be compared with Table 2) achieve this ideal almost completely. The fine points of this proposal are explained in the following remarks, but the reader may want to skip ahead to the examples in Figures 1 to 5.

(i) A **type** or **const** parameter form binds occurrences of its identifier in subsequent formal parameters of the declaration list or parameter list, as well as in the body of the block or abstraction.

Table 4. Proposed parameter and declaration correspondence

Denotation	Formal parameter	Actual parameter
Static R-value	**const** $I:T$	Σ
Type value	**type** I	T
Non-static R-value	**val** $I:T$	E
New L-value	**new** $I:T$	E
Existing L-value	**var** $I:T$	Ξ
Abstraction value	**function** $I(\ldots;\Pi_i;\ldots):T$	I

(ii) Static parameters can be implemented by generating a separate code segment for every distinct actual parameter. This approach would allow efficient Pascal-like storage access and complete static type-checking. The code produced would not be any more space-consuming than for a comparable Pascal program; in Pascal, the multiple copies of the abstraction would have to appear in the source program as well as the object code. Of course, this is only the simplest possible implementation approach; in many situations an optimizing compiler would be able to merge together the code segments. For an obvious reason, the use of static parameters in recursive abstractions would have to be restricted, for example by requiring that the actual parameter in a recursive call be identical to the corresponding formal parameter.

(iii) The actual parameter corresponding to a **new** formal parameter form whose data type is *not* assignable is a special kind of procedure call. The procedure must be one whose last formal parameter is a **var** parameter of that type. The call supplies arguments for all but the last of the procedure's parameters; the newly-allocated *L*-value becomes the last argument of the initializing procedure. Wang [37] termed this mechanism "call-by-initialize". A simple example is:

new *f*: **file of** *char* = *rewrite*.

For **new** declarations, the actual parameter may be omitted as in Pascal.

(iv) Restrictions on **var** declarations analogous to those on **var** parameters in Pascal would be necessary to preserve disjointness of variables [41, 42, 17, 15, 25].

(v) If the type of a **val** formal parameter form is not assignable, then the actual parameter must not be selectively updated in the scope of the binding. This would allow a call-by-reference implementation and is similar to a restriction suggested by Hoare [12].

(vi) Specifiers (meta-variable Π) are provided to allow complete type specification of abstractional parameters. A specifier has the form of a formal parameter without an identifier; for example, **val**:*T*. There seems to be no reason not to allow **var** specifiers in procedural parameters, but other possible specifier forms such as **const**:*T*, **type**, and higher-order abstractions might be excluded to simplify implementation.

(vii) An ALGOL 60-like **label** parameter would be possible, but it would not correspond to Pascal's **label** declarations; furthermore, a corresponding **label** declaration would be a useless facility without more extensive label-valued expressions. We have chosen to avoid the well-known problems that would arise by following this line of thought; instead, we adopt the long-advocated [22, 6] and currently fashionable approach of getting rid of labels and **go to**s entirely, in favour of sequencers whose targets are determined by the structure of the program text. There are several possibilities: **stop** (exit program), **return** (exit procedure body), **break** (exit loop), and so on. We adopt just one: **exit**, which exits the current block; an abstraction facility to be discussed later will permit parameterization and "labelling" of exits.

(viii) The **for** statement is easily modified to be a declarative construct (using R-value binding) as follows:

for I:T=E_1 **to** E_2 **do** Γ

There would also be an analogous **downto** form and we will also use the simplified form:

for I: T **do** Γ

for the very common case that the identifier is to be bound to every value in a type T.

(ix) **val** and **var** parameters should not be identified with call-by-value and call-by-reference implementations, respectively; these would always be correct, but other implementations may be more convenient or efficient in some cases [12]. Since the proof rules for Pascal [17] require disjointness of variables, call-by-value/result will also be a correct implementation for **var** parameters; also, if the compiler knows that an actual **val** parameter cannot be updated in the scope of the binding (for example, in a function, which according to the proof rules must be free of side effects), then call-by-reference would also be correct.

Examples of the proposed parametric and declarative forms are given in Figures 1 to 5. Figure 1 is the heading of a procedural realization of the prime-finding algorithm in Dijkstra [7, Section 9]. This should be compared to [41, p. 141]; *primes* is a general procedure rather than a specialized program. Note the use of the generalized forms of static expressions and parameter specifications, and the similarity between the bindings of n in the parameter list and m in the declaration list.

```
procedure primes (const n: integer; var p: array [1..n] of integer)
=begin
   with const m: integer = round (sqrt(n));
        new mult: array [1..m] of integer
             ⋮
   do
             ⋮
  end
```

Fig. 1

Procedure *order* in Figure 2 takes advantage of the polymorphism of the operator $\geqq$; some legal calls are:

order (*integer*, *i*, *j*)
order (*colour*, *c*, *d*)
order (*real*, *x*, *y*)
order (**set of** *colour*, *s*, *t*)
but: *order* (↑*person*, *p*, *q*)

would be a compiler-detectable error.

```
procedure order (type t; var x, y:t);
=if x≧y then
   begin
      with val z:t=x
      do x:=y; y:=z
   end
```

Fig. 2

Procedure *maparray* in Figure 3 illustrates the use of a specifier in an abstractional parameter. In Figure 4 a single stack is represented by private array A and depth index p; the scope of these identifiers is limited to the bodies of the abstractions. Procedure *matmply* in Figure 5 is for multiplying matrices and illustrates the use of **val** and **var** parameters, **var** declarations, the simplified **for** statement, and generalized parameter specifications.

```
procedure maparray (type ind, d;
                    var A: array [ind] of d;
                    procedure p(var: d))
=for i: ind do p(A[i])
```

Fig. 3

```
with
   private
      new A: array [1..n] of d;
      new p:0..n=0
   within
      procedure push (val x:d)
      =begin p:=p+1; A[p]:=x end;
      procedure pop (var x:d)
      =begin x:=A[p]; p:=p-1 end;
      function empty: Boolean=(p=0)
do
   ... push( ) ... pop( ) ... empty ...
```

Fig. 4

```
procedure matmply  (type ind1, ind2, ind3;
                    var A: array [ind1, ind2] of real;
                    val B: array [ind1, ind3] of real;
                    val C: array [ind3, ind2] of real)
=for i: ind1 do
   for j: ind2 do
   begin
      with var Aij: real=A[i,j]
      do Aij:=0;
         for k: ind3 do
            Aij:=Aij+B[i,k]* C[k,j]
   end
```

Fig. 5

4.3. Abstraction

In this section we will describe a "full complement" of abstraction facilities for the syntactic categories in Pascal; these are summarized in Table 5. Abstractions from expressions and statements are, of course, functions and procedures; for the other syntactic categories, the following proposals are made:

Table 5. Proposed abstractions

Syntactic category	Abstraction values
Expressions	Functions
Statements	Procedures
Static expressions	Static functions
Type expressions	Type functions
Variable expressions	Selectors
Declarations	Modules
Sequencers	Sequels

(i) Static and Type Expressions
Incorporating a facility for abstraction from statically-evaluable forms such as static and type expressions is particularly straightforward; there are no implicit dynamic parameters so that there is no need to make a distinction analogous to the one between:

function $I{:}T=E$

and: **val** $I{:}T=E$

Hence, **const** and **type** definitions can be generalized to have optional parameters; we call the abstracted entities *static functions* and *type functions*, respectively. Examples are given in Figure 6, which also shows how such functions would be invoked by supplying actual parameters.

```
const sum (const n: integer): integer = n*(n+1) div 2;
type vector (const n: integer; type d) = array [1..n] of d;
:
new V: vector (sum (m), real)
```

Fig. 6

Static function and type function parameters are conceivable, but would not be possible if **const** and **type** specifiers were not allowed.

(ii) Variable Expressions
An abstraction from a variable expression will be termed a *selector*, after Hoare [14]. To preserve disjointness and stack implementability, the L-value returned by a selector must be a component of either a **var** parameter of the selector or a private variable accessible to it; furthermore, a selector should have no side effects. An example of a (parameter-less) selector is given in Figure 7; *top* allows the top of the stack to be accessed and updated, without any pushing or popping. Selectors can be composed with other selectors, including built-in mechanisms such as array indexing and record field selection. The selector facility is a safe way of providing some of the power of computed references, as recommended by Hoare [16].

Selector parameters would make disjointness checking much more difficult, and should probably not be allowed.

(iii) Declarations
In our "stack" example (Figs. 4 and 7), the definition of the concept is tied to its actual creation and use. A separation can be achieved by means of a facility for

```
with
   private
      new A: array [1..n] of d;
      new p:0..n=0
   within
      procedure push (val x:d)=...
      procedure pop (var x:d)=...
      function empty: Boolean=...
      selector top: d=A[p]
do
   :
   top := y
   :
```

Fig. 7

```
with
   module stack (const n: integer; type d)
   =(private
         new A: array [1..n] of d;
         new p:0..n=0
      within
         procedure push ...
         procedure pop ...
         function empty ...
         selector top ...);
   :
do
   :
   begin
      with stack (100, real) do
                          :
   end
```

Fig. 8

abstracting from the declarations to yield an entity we term a *module*, after Schuman [32]; the result is shown in Figure 8.

The module *stack* can be invoked in the heading of any contained block and this would bind free occurrences of the public identifiers *push*, *pop*, etc. in the body of that block; a separate "stack" would be allocated for each such invocation (necessarily in distinct blocks). A useful compiler action would be to make explicit in the output listing the identifiers which become bound by a module invocation, and their specifications. The module facility is similar to the block prefixing mechanism in SIMULA [2].

Module parameters would allow a kind of dynamic scope convention and should probably not be allowed.

(iv) Sequencers

An abstraction from a sequencer will be termed a *sequel*[1]. After getting rid of **goto** *s*, the only sequencer construct that remains has the general form: Γ; **exit**. We may therefore adopt the convention that in the body of a **sequel** definition the terminating **exit** may be left implicit; that is, the body is a statement, and after its execution control transfers to the end of the block in which the sequel is

[1] These are not related to the sequels of Milne [27]

```
begin
   with sequel found (val j:1..m) = B[j] := B[j] + 1
   do
      with new i:0..m = hash(x);
         sequel present = found(i);
         sequel absent = begin A[i] := x; found (i) end
      do loop
            if A[i] = x then present;
            if A[i] = 0 then absent;
            i := i - 1; if i = 0 then i := m
         end
end
```

Fig. 9

declared. The resulting language facility is a special case of Landin's [23] "program-points" and essentially similar to Zahn [45] and Knuth's [20] "events".

Figure 9 is an example adapted from Knuth [20]. The hash table A is to be searched using an index computed from the search argument x by the function *hash*; the sequels *present* and *absent* represent success or failure of the search, and the sequel *found* represents the action to be taken when an index for x is finally established.

Sequel parameters would be similar to label parameters in ALGOL 60 and would be useful though somewhat difficult to implement on some machines.

4.4. Discussion

As an illustration of the proposed design methods the presentation of the revisions and extensions has emphasized the objectives of better correspondence between parameters and declarations, and completeness of the abstraction facilities. In practice these must not be regarded as design ends, but merely as means; although detailed description of a proposed revision of Pascal is not the main purpose of this paper, it is appropriate at this point to evaluate the proposal in order to emphasize that while a systematic method can be used "as a reliable and inspiring guide" (Dijkstra), it does not relieve a designer of the responsibility to critically evaluate a design. We use the design goals expounded in [15, 42, 43].

(i) Flexibility
Notwithstanding the conservative opinions expressed by its designer [43], Pascal has been shown to be susceptible to some useful and clean extensions and generalizations. The features of the proposal that seem to contribute most to expressivity as compared to Pascal are static parameters, which provide a simple, secure, efficient, and general solution to the notorious array parameter problem, and the new abstractions, especially the **module** concept. None of Pascal's capabilities have been taken away except the **go to** sequencer, which few will lament, and the original form of the **with** statement, which might be advantageously replaced by a record constructor notation.

(ii) Security
No aspect of the proposal would compromise security in any way; indeed, features such as **val** parameters, specifiers, and **private** declarations would significantly improve it.

(iii) Readability and (iv) Efficiency
All aspects of the proposal are at least as good as Pascal in these respects.

(v) Simplicity
This objective is of primary importance, yet it is the most difficult to evaluate objectively; many of the issues have been discussed by Hoare [15] and Wirth [42, 43]. The proposed revision is certainly a "larger" language than Pascal, in that there are more facilities and a compiler for it would be somewhat more complicated; all the same, we do not regard the revised language as significantly more complex. Firstly, it is evident that exactly the same semantic and syntactic concepts underlie both, as well as the same approaches to type checking, coercions, storage management, data structuring, and control structuring.

Secondly, the total number of parametric *and* declarative forms in the revision is essentially the same as in Pascal. Or course, it might be desirable to have fewer such mechanisms, but it seems that provision of an explicit choice among a variety of simple and essentially different forms is as important to transparence, efficiency, and security for binding mechanisms as it is for control structures, data structures, and coercion conventions, and we believe the approach described to be consistent with Pascal's general design philosophy.

Finally, the regularity and conceptual coherence of the abstraction facilities, as well as their usefulness, compensate for the increased volume of the language.

In summary, the proposed changes and extensions would seem to be worthy of consideration for future versions of Pascal or new Pascal-like languages. It should also be noted that many of the facilities could even be accomodated into current Pascal implementations as local extensions, since no significant conceptual change would be involved.

5. Concluding Remarks

Recent interest in programming style and methodology has had the effects of, firstly, confirming the considerable practical importance of language design issues, and, secondly, demonstrating the usefulness of systematic and disciplined approaches to design problems. In this paper, two language design methods have been shown to be helpful both in analyzing language features in order to discover irregularities, unnecessary restrictions, and missing facilities, and also in synthesizing coherent, consistent and complete language features. There has been space to consider only one linguistic framework and a single design approach, but the generality of the semantic principles of correspondence and abstraction on which the methods are based suggests that they would be equally useful in other frameworks.

Perhaps the most important contribution of the methodological approach is that while mainly intended to help a designer to cope with the detailed problems of achieving consistency, completeness, and regularity in the design of specific language features, it also has the effect of drawing his attention to deeper structural issues, such as the nature of the denotable values and semantically-meaningful syntactic categories of the language under study. Significant progress in language design will require new concepts and the methods discussed in this paper should contribute to the future development of such concepts by focussing attention on structural aspects of programming languages.

Acknowledgements. The suggestions and encouragement of Robert Milne and other members of the Oxford University Programming Research Group are gratefully acknowledged. Comments by Mike Jenkins and Mike Symes on an earlier version were very helpful in improving the presentation. This research was supported by the National Research Council of Canada, grant A8990.

References

1. Ammann, U., Wirth, N.: Advantages of the value parameter over the constant parameter. Unpublished memo, Eidgenössische Technische Hochschule, Zürich, 1972
2. Birtwistle, G.M., Dahl, O.J., Myrhaug, B., Nygaard, K.: SIMULA BEGIN. Philadelphia: Auerbach 1973. Also Lund: Studentlitteratur 1974
3. Brinch Hansen, P.: Universal types in Concurrent Pascal. Information Processing Letters **3**, 165-166 (1975)
4. Bron, C., de Vries, W.: A Pascal compiler for PDP-11 mini-computers. Software Practice and Experience **6**, 109-116 (1976)
5. Church, A.: The calculi of lambda conversion. Princeton: Princeton University Press 1941
6. Dijkstra, E.W.: *Goto* statement considered harmful. Comm. ACM **11**, 147-148, 538, 541 (1968)
7. Dijkstra, E.W.: Notes on structured programming. In: Structured programming (O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare, eds.). London: Academic Press 1972
8. Dijkstra, E.W.: Guarded commands, non-determinacy, and formal derivation of programs. Comm. ACM **18**, 453-457 (1975)
9. Evans, A.: PAL – a language for teaching programming linguistics. Proc. 23rd ACM National Conference, Princeton: Brandin Systems Press 1968
10. Grosse-Lindemann, C.O., Nagel, H.H.: Postlude to a Pascal compiler bootstrap on a DEC system 10. Software Practice and Experience **6**, 29-42 (1976)
11. Habermann, A.N.: Critical comments on the programming language Pascal. Acta Informatica **3**, 47-57 (1973)
12. Hoare, C.A.R.: Procedures and parameters: an axiomatic approach. In: Symposium on Semantics of Algorithmic Languages (E. Engeler, ed.), Lecture Notes in Mathematics, Vol. 188. Berlin-Heidelberg-New York: Springer 1971
13. Hoare, C.A.R.: A note on the *for* statement. BIT **12**, 334-341 (1972)
14. Hoare, C.A.R.: Notes on data structuring. In: Structured programming (O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare, eds.). London: Academic Press 1972
15. Hoare, C.A.R.: Hints on programming language design. Computer Science Department, Stanford University, CS-403, 1973
16. Hoare, C.A.R.: Recursive data structures. International J. Computer and Systems Sciences **4**, 105-132 (1975)
17. Hoare, C.A.R., Wirth, N.: An axiomatic definition of the programming language Pascal. Acta Informatica **2**, 335-355 (1973)
18. Jensen, K., Wirth, N.: Pascal: User manual and report. Lecture Notes in Computer Science, Vol. 18. Berlin-Heidelberg-New York: Springer 1974
19. Knobe, B., Yuval, G.: Towards Pascal II. The Hebrew University of Jerusalem, 1974
20. Knuth, D.E.: Structured programming with *goto* statements. Computing Surveys **6**, 261-301 (1974)
21. Landin, P.J.: The mechanical evaluation of expressions. Computer J. **6**, 308-320 (1964)
22. Landin, P.J.: Getting rid of labels. Univac Systems Programming Research Report, New York, 1965
23. Landin, P.J.: The next 700 programming languages. Comm. ACM **9**, 157-164 (1966)
24. Lecarme, O., Desjardins, P.: More comments on the programming language Pascal. Acta Informatica **4**, 231-243 (1975)
25. Ligler, G.T.: A mathematical approach to language design. Conference Record of the Second ACM Symposium on Principles of Programming Languages, Palo Alto, 1975

26. McCarthy, J.: A basis for a mathematical theory of computation. In: Computer programming and formal systems (P. Braffort, D. Hirschberg, eds.). Amsterdam: North-Holland 1963
27. Milne, R.E.: The formal semantics of computer languages and their implementations. Oxford University Computing Laboratory, Programming Research Group, technical microfiche TCF-2, 1974
28. Milne, R.E., Strachey, C.: A theory of programming language semantics. London: Chapman and Hall. Also New York: Wiley 1976
29. Mosses, P.: The mathematical semantics of ALGOL 60. Oxford University Computing Laboratory, Programming Research Group, technical monograph PRG-12, 1974
30. Quine, W.O.: Set theory and its logic. Cambridge (Mass.): Harvard University Press 1963
31. Reynolds, J.C.: Gedanken – a simple typeless language based on the principle of completeness and the reference concept. Comm. ACM **13**, 308–319 (1970)
32. Schuman, S.A.: Towards modular programming in high-level languages. Algol Bulletin **37**, 12–23 (1974)
33. Stoll, R.R.: Set theory and logic. San Francisco: Freeman 1963
34. Strachey, C.: Towards a formal semantics. In: Formal language description languages (T. Steel, ed.). Amsterdam: North-Holland 1966
35. Tennent, R.D.: Mathematical semantics and design of programming languages. University of Toronto, Ontario, Canada, Ph. D. thesis, 1973
36. Tennent, R.D.: The denotational semantics of programming languages. Comm. ACM **19**, 437–453 (1976)
37. Wang, A.: Generalized types in high-level programming languages. Institute of Mathematics, University of Oslo, Norway, Research Reports in Informatics, No. **1**, 1975
38. Wichmann, B.A.: ALGOL 60: Compilation and assessment. London: Academic Press 1973
39. van Wijngaarden, A., et al.: Report on the algorithmic language ALGOL 68. Numer. Math. **14**, 79–218 (1969)
40. Wirth, N.: The programming language Pascal. Acta Informatica **1**, 35–63 (1971)
41. Wirth, N.: Systematic programming – an introduction. Englewood Cliffs (N. J.): Prentice-Hall 1973
42. Wirth, N.: On the design of programming languages. In: Proc. IFIP Congress 74 (J.L. Rosenfeld, ed.), Stockholm. Amsterdam: North-Holland 1974
43. Wirth, N.: An assessment of the programming language Pascal. IEEE Trans. Software Engineering 1, pp. 192–198 (1975)
44. Wirth, N., Hoare, C.A.R.: A contribution to the development of ALGOL. Comm.ACM **9**, 413–431 (1966)
45. Zahn, C.J.: A control statement for natural top-down structured programming. In: Programming Symposium Proceedings, Colloque sur la Programmation, Paris (1974). Lecture Notes in Computer Science, Vol. 19. Berlin-Heidelberg-New York: Springer 1974

Received June 1, 1976