Getting rid of labels

P.J. Landin

Published online: 5 February 2010

© The Author(s) 2010. This article is published with open access at Springerlink.com

Abstract This report presents some of the principles underlying the currently projected ISWIM programming system.

In most programming languages the user encounters situations in which he decides whether to write labels and jumps, or to use some other, less explicit way of indicating the sequence of computing steps. This paper draws attention to some of the alternatives to explicit sequencing, as offered in current languages. It explains why it is important to reduce reliance on explicit sequencing and describes new linguistic developments, not yet in current languages, that would make progress in this direction.

Keywords ISWIM · ALGOL60 · Explicit sequencing · Language design

1 Introduction

There is a game you can play with ALGOL60¹ programs—rewriting them so they don’t contain any labels. For example, the following fragment of program

```
...  
  if  $a \geq b$  then goto L;  
  y := a; goto M;  
L:  z := b;  
M:  ...
```

can be written as

¹Editor’s note: In this report, there is never a space between “ALGOL” and “60”. This was surely intended: a later paper has “ALGOL60 (*sic*)” and mentions that refereeing of his *Correspondence between ALGOL 60 and Church’s Lambda Notations* “seemed concerned only with whether a space should separate ‘algol’ and ‘60’.”

Edited by R.D. Tennent.

R.D. Tennent (corresponding author) School of Computing, Queen’s University, Kingston, Canada
e-mail: rdt@cs.queensu.ca

P.J. Landin

Univac Systems Programming Research, New York, USA

```

...
if  $a < b$  then  $y := a$  else  $z := b$ ;
...

```

More truthfully, this particular example is valid provided that, outside of the fragment displayed, the program does not refer to the labels L and M . Again, with a similar qualification,

```

...
if  $x - x' \leq E$  then goto  $L$ ;
 $x := x'$ ;
 $x' := (x + a/x)/2$ ;
 $L$ : ...

```

can be written as

```

...
if  $x - x' > E$  then begin  $x := x'$ ;
                         $x' := (x + a/x)/2$ 
end;
...

```

There is another game, having a slightly less well-defined but related goal: reducing the number of program-steps. For example, consider the following fragment:²

```

...
 $t_1 := b^2$ ;
 $t_2 := ac$ ;
 $t_3 := t_1 - t_2$ ;
 $t_4 := \sqrt{t_3}$ ;
 $t_5 := b + t_4$ ;
 $z := t_5/a$ ;
...

```

With a similar qualification, namely to the effect that outside the piece displayed the program does not refer to $t_1, t_2, \dots$, this can be rewritten

```

...
 $z := (b + \sqrt{(b^2 - ac)})/a$ ;
...

```

In each example, the re-writing has the following features:

- It makes more explicit the role of the fragment in relation to its context, namely: In each of the first two cases the fragment is a computational step that is entered and exited *in-line*; i.e., not by jumping into or out of it. In the third case the fragment changes just one variable, z , in terms of just three, a , b , and c .
- It makes more explicit the internal structure of the fragment, namely: Each of the first two is a computational step that involves selecting and performing one from two alternative substeps (of which in the second fragment, one substep is vacuous). The third fragment computes a straightforward algebraic expression.
- In the context of a long string of statements, the rewriting replaces a sequence of several items by a single item, having a more elaborate structure.

²Editor's note: Landin uses a postfix raised 2 to indicate squaring and omits an infix $\times$ when no ambiguity results.

- In writing the revised version of each fragment, the programmer was saved certain decisions that are irrelevant to the overall outcome, namely: the choice of identifiers having purely local significance; and in the third fragment, the choice of a sequence for the individual steps. While these choices are fairly wide open, there are enough mutual constraints to give opportunities for error.

In the above examples, it is fairly obvious that someone who wrote the original rather than the revised version would not be exploiting the facilities available to him. So the moral drawn here might be judged a platitude on the grounds that these days there are hardly any programmers left who fail to use these facilities. However such failure is common practice in many analogous situations, less obvious on account of being larger or involving less familiar programming techniques. This article discusses specific small examples so as to identify separately the deficiencies that force programmers to clutter up their programs with irrelevant information about how the computer is to behave, instead of reserving their attention for what outcome they require.

Two conclusions emerge. First, currently available techniques have outstripped methods of training programmers. Second, there are several ways in which currently available techniques can be pushed further.

These developments are introduced briefly in a later section. They are given an extended illustration in [4], and they are embodied in the ISWIM programming system, currently under development for the 1107.³

The techniques applied in the present report to small pieces of program are used in [4] to eliminate almost all step-by-step sequencing from a lengthy NPL⁴ program. Thus it may one day be as natural to write large expressions of the type used there, as it is today to write $(b + \sqrt{(b^2 - ac)})/a$ rather than its six decomposed steps. Some of the reasons for supposing that this development is desirable are given in a later section below.

2 Making textual structure mirror problem structure

The benefits claimed for the above examples can be stated in more general terms. If a program contains fragments that are amenable to the tricks used above, then the rewriting enables some of its important properties to be exhibited by *textual* structure, instead of being implied by the placing of jumps and labels, and the written sequence of steps. This is related to the fact that the revised version emphasizes the overall outcome rather than the steps by which the computer achieves the outcome.

The properties referred to above as important, are the things that must be discovered by someone who is trying to understand a program; e.g., to correct it, modify it, incorporate it into a larger complex, or transcribe it into a different notation. We say that a program is more “transparent” if such properties are more easily discoverable. Clearly with this meaning, “transparency” has been an important consideration in most programming innovations. The fact that it is to some extent a subjective notion has not made it less important.

³Editor’s note: The Univac 1107 was a massive computer introduced by Sperry Rand in 1962; only 36 were sold. It was the first computer programmed by Donald Knuth (at the Case Institute in Cleveland).

⁴Editor’s note: NPL was a programming-language proposal that evolved into PL/I.

Nor has the fact that it frequently conflicts with economical use of machines. For, in the first place, this is not the only economical consideration. The next point needs more stressing since it is less generally accepted. Suppose a precise notation for expressing computational requirements is transparent, albeit not backed up by a viable mechanical processor. It might still be used as the basis of a *systematic* method of program preparation. It may be worth writing ALGOL60 even if you have to hand-translate it. This is likely to pay off in itself, and the more so since systematizing an activity is a valuable and often inevitable step towards mechanizing it.

To summarize, the opposite of transparency is “opacity,” and a common source of opacity is reliance on explicit sequencing, i.e., specifying what is required from the computer by stating the steps through which the computer must proceed. The history of new programming techniques has largely been the history of proving new and more powerful alternatives to explicit sequencing. For example, the use of format directives, function declarations, and macro definitions all simplify the shape of flow charts. Among currently implemented languages, the one that goes furthest in this direction is LISP, and in a comparative assessment of program preparation [2], it came out best on two counts, both of programmer time and machine debugging time.

The last section of this article summarizes the reasons for supposing that future progress in programming will depend on progress in the elimination of explicit sequencing. This progress involves software manufacturers in two ways. They will both use and implement the new techniques.

Other sections give further specific examples of the techniques involved. These draw attention to facilities that are present in current languages but not exploited, either for lack of education or for fear of real and imagined inefficiency. They also draw attention to facilities that are provided by *some* language designers, but neglected by others, and to facilities that are not in any currently implemented languages (probably for similar reasons).

This suggests that future progress involves several related developments:

- Designing languages for mechanical processing that include the new facilities. This activity is sometimes thought to be outside the scope of mere software manufacturers, since they have languages imposed on them. But this is not so—see the later section on “Motivation.”
- Developing processors that do not penalize transparency (see [8] for an example of how someone behaves who is so scared of the way people implement ALGOL60 procedures that he uses a switch).
- Designing and using systematic methods of program preparation that use the new techniques as an intermediate result. This is a step in a systematize/mechanize cycle.
- Teaching the new techniques, and doing it first to software programmers, since it is unlikely that a technique will gain acceptance among applications programmers that has not been given a trial run among people close to the sources of new programming techniques.

3 More examples

In the first two of the examples in the introduction, the game was easy on account of the presence in ALGOL60 of two particular features; in the first case the “conditional statement,” and in particular its “two-armed” version **if** ... **then** ... **else** ... rather than the “one-armed” version (**if** ... **then** ...); and in the second case the “compound statement,” i.e., a string of statements enclosed in what are essentially brackets (though in ALGOL60 they are written

begin ... end) so as to be used like a single statement. Thus it is not possible in FORTRAN to perform rearrangements analogous to the above, since FORTRAN lacks statement brackets, and the only conditional statements it has are one-armed with a jump in the arm (as used in the first version of each fragment above). In IBM's more recent PL/I, on the other hand, both of these deficiencies have been made good.

(Although the above tricks cannot be precisely matched in FORTRAN, this is not to say that the elimination of those labels is impossible in FORTRAN. On the contrary, there is another trick for eliminating labels, that applies to the above fragments and also to their counterparts in FORTRAN. Furthermore, any impression given here that from the present point of view ALGOL60 “dominates” FORTRAN is explicitly disclaimed. There are other design details which show their roles reversed.)

This section gives more examples of eliminating labels, i.e., in line with the first two. The point made by the third example above, namely eliminating step-by-step decomposition, will be taken up in the next section.

The following example illustrates the use of an ALGOL 60-style loop feature (as opposed to a FORTRAN-style loop feature):

<pre> L: $x' := (x + a/x)/2$; if $x - x' \leq E$ then goto M; x := x'; goto L; M: ... </pre>	<pre> for $x' := (a + a/x)/2$ while $x - x' \leq E$ do x := x'; ... </pre>
--	--

What makes this example easy game is that in the loop, the part before the test happens to fit into the **for**-clause. In a more complex situation we might eliminate one label, but not both.

For the next example (see Fig. 1), we again present the original (distilled from a published program [8]) and rewritten programs side by side. Here the re-arrangement introduces a procedure for summing a power series, in place of a piece of in-line programming that does the same job. Two points to notice here. First the familiar effect of using a closed routine, eliminating labels for the return points, $m1$, $m2$, $m3$, and $m4$. Second and more interesting, the use of a closed routine amplifies the power of another implicit sequencing device, namely the conditional statement; and the label L , which is not merely a return point, can be eliminated rather more neatly than would otherwise be possible.

4 Motivation

Two questions arise. First, what is the point of this game? Is it useful? Second, whatever the point, does it have any general application? Can *any* program in ALGOL60, NPL, FORTRAN or what you will, be rewritten without labels and jumps, or is the attempt futile except with carefully chosen examples, selected with an eye to a particular language feature?

To answer the first question briefly, the point of the game is that it draws attention to certain aspects of programming that have only recently been rescued from theoretical confusion, and which are not likely to emerge from the attendant practical haphazardness until the theoretical advances are more generally disseminated. So this article is an early chapter in an attempt at dissemination.

The dissemination is aimed at anyone who is at all concerned with the design of programming languages and notations. And, lest this be thought of as an esoteric, minority pursuit, it should be observed that it includes:

<pre> begin switch $S := m1, m2, m3, m4$; ... $c := 0$; $n := 2j + 1$; $x := p + q$; $m0: c := c + 1$; $y := 0$; for $i := n$ step -1 until 1 do $y := yx + a[i]$; goto $S[c]$; $m1: u := y$; $n := k - 1$; $x := r + s$; goto $m0$; $m2: v := 2y - 1$; if $p^2 > 2q^2$ then begin $n := j + 1$; $x := 2p + q$ end else begin $c := c + 1$; $n := k + 2$; $x := 2r - s$; end; goto $m0$; $m3: u' := y$; goto L; $m4: v' := y$; $L: \dots$ end </pre>	<pre> begin real procedure $f(n, x)$; integer n; real x; value n, x; begin $y := 0$; for $i := n$ step -1 until 1 do $y := yx + a[i]$; $f := y$ end f; ... $u := f(2j + 1, p + q)$; $y := 2f(2k - 1, r + s)$; if $p^2 > 2q^2$ then $u' := f(j + 1, 2p - q)$ else $v' := f(k + 2, 2r - s)$; ... end </pre>
--	--

Fig. 1 Example using a routine

- people who invent a notation to be used in coordinating the efforts of a programming team, whether the notation is intended for mechanical processing, or merely for inter-human communication.
- people who invent a language purely for mechanical use, as an intermediate result of some language processor; e.g., a result of a first pass.
- people who mesh two notations; e.g., an assembler language and an algebraic language, or a programming language and an operating system.
- people who choose a subset or variant of a current language, or invent some other notation, for a purely pedagogical purpose, as a tool in teaching programming, and in particular in the teaching of systems programming.
- people who invent a little private notation for their own use in a specific field, merely as an intermediate step in the hand-preparation of a program.

So language design is a more widespread activity than it might seem at first sight. The behaviors of all these people are likely to be affected by the considerations broached in this article.

Now for the second question: is it always possible to eliminate jumps and labels? This has two answers. First, and more important, my experiments on published programs suggest that the tricks illustrated above, and a few more given below, go a long way towards unravelling explicit sequencing *in useful programs that have physical or logical significance*, and that, along with the labels, they get rid of much obscurity. That is to say the rewritten programs are easier to understand for such purposes as modifying and hand translating. Perhaps the most dramatic evidence is the detection of such possible oversights as intermediate qualities⁵ calculated but not used, or unnecessarily recalculated.

However, it is easy to construct programs that don't yield to these tricks; e.g., hashes of randomly chosen statements strung together, not derived from a physical or logical problem. In fact, this shows that the relation between ALGOL60 and ALGOL60-without-labels is analogous to the relationship between machine-language and machine-language-generated-by-an-algebraic-processor. In each case, there is a subset that is more structured, and has independent significance. This leads to the second answer to the question: can labels always be eliminated?

Someone playing the game in ALGOL60, even using hash programs, can be assured that it is always possible. Indeed it *is* possible (demonstrated by van Wijngaarden [9]) merely by using the ALGOL60 **procedure** feature, without resorting to the tricks used in our above examples. The analogous question for NPL or FORTRAN has not (to my knowledge) been investigated, and my respective guesses are “probably” and “maybe.” However, the question is academic and irrelevant to the present purpose. For the game is put forward as an educational and research technique, not as end in itself. I hope that each hard case will suggest useful new language features, or new ways of using existing features. And that analysis of genuinely intractable examples will shed light on the question of adding imperative features to a basically non-imperative language. So far, intractability seems limited to success/failure situations and the action required on failure.

In conclusion, it is emphasized that this paper is not about how or why to transform programs containing explicit sequencing into programs that contain less or none. This disclaimer is needed since such an activity has in the past been pursued [3, 5]. That it was never very rewarding is to be expected from the considerations raised here. Programmers labour to decompose their requirements to fit into prescribed forms of “statement,” and in the process they lose important information in the forest of irrelevancies. The recovery of this information is an even more laborious task (and is one of those tasks for which it can be proved that no general rule exists). It nevertheless falls to other programmers when they use, modify, correct or transcribe. Efforts to mechanize this recovery have merely demonstrated that such efforts would be better spent on developing methods of programming that don't force programmers to throw the important information away.

Looking further ahead, there is one possible justification for mechanizing the process. The alleged embarrassment of explicit sequencing afflict both humans and machines. Consequently, even if a human fails to feel embarrassed the machine might nevertheless prefer to remove the explicit sequencing in order to facilitate further processing.

5 Effect on cost of processing

Any alleged amelioration of the lot of the computer user naturally raises the spectre of increased processing costs. There are three separate issues here.

⁵Editor's note: quantities?

1. Languages have other uses than bread and butter production runs, e.g., one-off runs, debugging, software production; see also the above list of “involved people,” and the section on “Prospects for Practical Application” in [4].
2. The language developments promoted here naturally put new burdens on implementers. (They also remove burdens imposed by outmoded bad language design.) Some of the required processing techniques exist already, some need research and development. However, not all of the proposals raise all of the problems.
3. On the other hand, there are reasons, put forward in [1], for supposing that future hardware will increasingly force the same language developments that are here promoted for users’ convenience. So perhaps the problems cannot be shirked. This coincidence of arguments *ad hominem* and *ad machinam* is not fortuitous.

6 Using fewer and bigger statements

One of the examples in the introduction was concerned with eliminating, not labels, but step-by-step decomposition. We now resume that line of development.

For example,

if $a \neq 0$ **then** $y := 1/a$ **else** $y := z$

can be rewritten as

$y :=$ **if** $a \neq 0$ **then** $1/a$ **else** z

thus replacing a statement that contains two nested statements, by one that does not.

Again, consider the following use of non-type procedures:

```
procedure  $f(u, v, y)$ ; value  $u, v$ ; real  $u, v, y$ ;
begin ...
    (assigning to  $y$ , but not using  $y$  in right-hand sides)
end  $f$ ;
procedure  $g(u, v, w, z)$ ; value  $u, v, w$ ; real  $u, v, w, z$ ;
begin ...
    (assigning to  $z$ , but not using  $z$  in right-hand sides)
end  $g$ ;
...
 $f(a + b, c + d, t_1)$ ;
 $f(a - b, c - d, t_2)$ ;
 $g(t_1 + e, t_2 - e, e^2, t_3)$ ;
 $m[i] := a + t_3$ 
```

This can be rewritten using type-procedures as follows:

```
real procedure  $y(u, v)$ ; value  $u, v$ ; real  $u, v$ ;
begin
    ...
end  $y$ ;
real procedure  $z(u, v, w)$ ; value  $u, v, w$ ; real  $u, v, w$ ;
begin
    ...
end  $z$ ;
...
 $m[i] := a + z(y(a + b, c + d) + e, y(a - b, c - d) + e, e^2)$ 
```


More truthfully, this is valid provided the original program does not refer to t_1 , t_2 , and t_3 outside the fragments displayed above.

The features of ALGOL60 that make the above transformations possible are, respectively, conditional expressions and type-procedure declarations. Of these the latter is matched in FORTRAN and PL/I. The former could be matched by

$$y := (a \neq 0) \times 1/a + (a = 0) \times z$$

given the appropriate numerical representation of truth values (which PL/I promises), and *also* given that in multiplications by zero, the other factor is not evaluated. Such an expedient would involve a small burden of recognition that PL/I apparently renounces (see [6], p. 41). Certainly it is wrong to suggest [7] that this device necessarily provides a match for conditional expressions.

It may be observed that in eliminating labels, a reduction of the number of statements is often a byproduct. And in at least one of the above mentioned situations it was a prerequisite; the success of **for** ... **while** ... depended on there being just one assignment before the loop-termination test.

The two games, eliminating labels and using fewer statements, are related in other ways. They both tend to obviate identifiers having purely local significance. More important they both tend to give the program more parenthetical structure, in ways that are related to its overall effect. Thus out of a string of a hundred mingled assignments and jumps, we might hope to produce four or five textual units (say, blocks or procedure declarations), each having a role that, in relation to the whole, can be grasped without regard to its internal details; and each itself similarly structured.

Our examples so far have been chosen to illustrate the various tricks possible in ALGOL60. They *illustrate* our contention about the gain in transparency, but they are obviously too small and too unrepresentative to constitute serious evidence for it. So in the Appendix we include an actual ALGOL60 program, taken from *Commun. ACM*, selected fairly haphazardly. That is to say, the particular number of *Commun. ACM* happened to be at hand for independent reasons, and, of the algorithms in it, a quick scan suggested that this one uses the most elaborate explicit sequencing.

7 New developments

We have seen several situations in which ALGOL60 offers a choice between less and more explicit sequencing. We have considered exploiting these to minimize explicit sequencing in ALGOL60 programs (cf. [9], which pursues precisely the opposite aim).

We now consider what new things must be added to ALGOL60 if this game is to be taken further. That is to say we try to identify the deficiencies that force an ALGOL60 programmer into explicit sequencing.

Two remarks. First, although this discussion is phrased in terms of ALGOL60, this is more a matter of concreteness and simplicity, than for any special properties of that language. The most significant effect of having used some other language would be to meet sooner the boundary between current facilities and new developments.

Second, I repeat a point I made earlier. The features proposed here are *not* merely contributions to a discussion about ALGOL70. Their usefulness does not depend on writing a translator for some language incorporating them. They are intended as components of a systematic method of programming, regardless of whether they can be mechanically processed or need hand-translating into ALGOL60, assembly language or whatever input language is being used. Thus they can be incorporated into programming training *now*.

8 Simultaneous assignments

Consider

```
if  $a < b$  then begin  $x := a; y := b$  end
else begin  $x := 2b; y := 2a$  end
```

The arms assign to the same variables. So the example would be amenable to an earlier trick, were there just one assignment in each arm. This suggests introducing “simultaneous assignments” into ALGOL60.

```
if  $a < b$  then begin  $x, y := a, b$  end
else begin  $x, y := 2b, 2a$  end
```

and hence

```
 $x, y :=$  if  $a < b$  then  $a, b$ 
else  $2b, 2a$ 
```

Syntactically, this amounts to including the comma as another expression-building infix operator, along with $+$, $-$, etc., and making a corresponding generalization to the category variable (for the left-hand sides). It is natural to give the comma low precedence, so as to maintain the traditional passing of operand lists.

An expression with a comma for its principal operator is called here a “listing.” Its value is a list (e.g., of numbers). To put lists on the same footing as numbers there must be single identifiers that denote lists, e.g., introduced by declarations like

```
(real, real)  $u$ ;
(real, integer)  $x, y, z$ ;
```

This would open the possibility of such assignments as $u := a + b, a - b$. But more important, we must have type-procedures that produce lists, e.g.,

```
(real, real) procedure complex mult( $a, b$ );
  value  $a, b$ ; (real, real)  $a, b$ ;
  begin real  $ra, ia, rb, ib$ ;
     $ra, ia := a$ ;
     $rb, ib := b$ ;
    complex mult :=  $ra \times rb - ia \times ib$ ,
     $ra \times ib + ia \times rb$ 
  end
```

If a listing can occur as the right-hand side of an assignment or definition, as an arm of a conditional expression, and as the “result-expression” of a function, then all the previous tricks for eliminating explicit sequencing can be used not only for scalar operations but also in cases of composite information structures.

In particular, the shift from real arithmetic to complex arithmetic no longer means a radically different program structure (as it now does in ALGOL60). Notice that the point is not met merely by providing complex arithmetic (as in, say, NPL). This leaves the same deficiency to recur when a user wishes to manipulate bundles of information (e.g., **real, real, integer**) having some private significance.

9 Auxiliary definitions

One of the ways in which program steps proliferate is the need to abbreviate such a step as

$$y := (a + b + c) \times (a + b + c + 1)$$

To avoid the repetition, both textual and dynamic, this is usually written

$$\begin{aligned} t &:= a + b + c; \\ y &:= t(t + 1) \end{aligned}$$

But the abbreviation carries a penalty, both for the writer, and for anyone who reads the program. If t is newly coined for this purpose it needs “declaring”; if not, it needs careful choice to avoid colliding with its other roles. And for a reader seeking an overall grasp of the program’s logical structure (e.g., to modify it), there are now *two* variables whose relationship with the context has to be traced.

The latter point is partly met by writing

```
begin real  $t$ ;
 $t := a + b + c$ ;
 $y := t(t + 1)$ 
end
```

but not very satisfactorily. For, the overall effect of this step, namely resetting t to something involving a , b , c , is less evident here than in the original “unabbreviated” formulation. We require some kind of phrase bearing the same relation to a right-hand side that a block bears to a statement. This is met, for instance, by either of these proposed notations:

$$\begin{array}{ll} y := t(t + 1) & y := \mathbf{let} \ t = a + b + c; \\ \mathbf{where} \ t = a + b + c & t(t + 1) \end{array}$$

Formally, this amounts to the following:

1. Introduce a new category of phrase called (using the notation of the ALGOL60 report)

$\langle \text{definition} \rangle$

and having, for its simplest manifestation, the following composition:

$\langle \text{definition} \rangle ::= \langle \text{identifier} \rangle = \langle \text{expression} \rangle$

(Later we shall introduce other more elaborate forms of definition).

2. Introduce a generalized kind of expression

$\langle \text{long expression} \rangle$

for use in the contexts previously filled by expressions (especially right-hand sides of assignments, and perhaps also of definitions themselves—but this depends on one’s taste in “precedence”).

$\langle \text{long expression} \rangle ::= \langle \text{long expression} \rangle \mathbf{where} \ \langle \text{definition} \rangle$
 $\quad \mid \mathbf{let} \ \langle \text{definition} \rangle; \langle \text{long expression} \rangle$
 $\quad \mid \langle \text{expression} \rangle$

For the sake of definiteness this proposal of “definitions” has been described in terms of ALGOL60, but any language is susceptible to similar extension. This includes not only algebraic, number-processing languages, but also pidgin-English languages, and languages for processing character-strings, formulae, documents, programs, etc. Indeed, they usually stand in greater need of the extension.

Obvious developments of this idea make the following rewriting possible:

$$\begin{array}{ll}
 s := p + 2q; & y := (as + bt)/(cs - dt) \\
 t := 2p - q; & \textbf{where } s = p + 2q \\
 y := (as + bt)/(cs - dt) & \textbf{and } t = 2p - q
 \end{array}$$

An alternative, given now because it leads up to a later point, is to use a single definition with multiple “definee” and right-hand side thus:

$$\begin{array}{l}
 y := (as + bt)/(cs - dt) \\
 \textbf{where } s, t = p + 2q, 2p - q
 \end{array}$$

Again consider

$$\begin{array}{ll}
 \textbf{real procedure } f(x); & y := f(p + 2a)/f(2p - q) \\
 \textbf{value } x; \textbf{ real } x; & \textbf{where real procedure } f(x); \\
 f := ax^2 + bx + c; & \textbf{value } x; \textbf{ real } x; \\
 y := f(p + 2a)/f(2p - q) & f := ax^2 + bx + c;
 \end{array}$$

or better (using FORTRAN style function declarations and default conventions about types):

$$\begin{array}{l}
 y := f(p + 2a)/f(2p - q) \\
 \textbf{where } f(x) = ax^2 + bx + c
 \end{array}$$

10 The case against explicit sequencing

This report has used small specific examples to focus attention on the explicit/implicit sequencing choice. There are also general considerations suggesting that this choice is a matter of some importance. These are listed briefly here and will be developed elsewhere.

1. Historically the development of algebraic languages has largely consisted of “the growth of the right-hand side.” Each loosening of the constraints on what can be written as a right-hand side has meant fewer and bigger computational steps, fewer decisions about their sequence, fewer names to be invented for intermediate results, and fewer labels and jumps.
2. In the past the issues were not understood; for example, in ALGOL60, structural considerations relating to the kind of entity being referred to, are interrelated and mutually constrained in a rather unsystematic way. For example, a user finds himself in a different situation as regards explicit sequencing, according as he is using real or complex numbers. The solution to this is *not* an extension to complex arithmetic: the problem would merely recur in some other form. A system for using computers must provide a framework wherein the addition of new concepts, such as complex arithmetic, is isolated from the choices between implicit and explicit sequencing.
3. There is an absolute limit on this growth of the right-hand side (i.e., of expressions built up by functional composition), which each innovation brings marginally nearer, and which is precisely delineated by an area of mathematical study called “combinatory logic.” This limit can be built in to a programming language entire, rather than approached by piecemeal additions.
4. This development tends to close gaps between languages that have in the past been falsely attributed to the different needs of different “problem orientations.” Many of these differences can now be seen to be of two kinds:
 - (a) different subsets of the above mentioned limit;
 - (b) different notations for writing the same features.

So we might hope to implement and teach “language free programming,” with varying problem orientations met by adding decorations to a single basic system.

5. There is no theoretical impediment to expressing each requirement from a computer as a single right-hand side; i.e., writing every program as a function definition:

$$f(\dots) = \text{right-hand side expression}$$

and every run as a single *Print* statement:

$$\text{Print}(\text{right-hand side expression})$$

So the “withering away of the statement” may be no vain hope.

6. “The withering away of the statement” is a phenomenon we may live to see. But it will come from further investigation of the activity of communicating with computers, not from applying some mechanical rule to conventional programming methods. Thus there is a fairly straightforward rule for eliminating labels from ALGOL60 programs [4], but its interest is merely that it assures us the task is not impossible. Unlike our intuitive methods of eliminating labels, it merely transmits and even amplifies the opacity of the original program.

Thus the arguments given here do not support a ban on explicit sequencing. They support discovering and making available more attractive alternatives that will progressively erode its use.

7. In conventional programming it is possible to distinguish roughly between those features of a program that contribute to specifying the machine’s step-by-step behavior, and those features that merely contribute to specifying the desired outcome. The former tend to obscure the latter.
8. A great deal of program construction consists of modifying, correcting, and combining already existing programs, rather than starting from scratch. This suggests an emphasis on techniques for describing things in terms of other things; i.e., for expressing functional relationships with great freedom concerning the kinds of entities in the domain and range.
9. With greater use of multiply-sequenced hardware, it will become more important that the user does not arbitrarily constrain the sequencing of computational steps. Hence the need for notations that permit him to avoid explicit sequencing.
10. In current developments (JOSS, MAC, QUICKTRAN, etc.) new operating techniques are being superimposed on language techniques that were not designed for them. The on-line construction of programs is likely to involve a great deal of “describing things in terms of things.” The last refuge of imperatives will probably be

- the lowest level of analysis;
- the highest level; i.e., the *user’s* (mental) step-by-step activity.

Thus even if very large functional expressions are not written, there is an implication that they might have been, had the users’ intellect been more powerful. Hence the interest in notations that can express arbitrarily large programs without explicit sequencing.

11 Conclusion

Our overall aim is better man-machine communication. This means better language design, and better use of the linguistic facilities available. Our subsidiary aim is the explanation of

particular techniques directed to this end. The game of getting rid of labels is a picturesque means of explanation.

It also happens, in the present state of computer education, to be an effective tactic for someone who is trying to understand the details of an existing program, in order, say, to modify it or hand-translate it into another language.

Someone who experiments with the game presented in this article, taking existing programs from his own past work and from, say, the *Commun. ACM*, is likely to reach the following conclusions:

1. Jumps and labels—and, more generally, all forms of explicit sequencing—are used more widely than they need to be, and for the following reasons:
 - a) The language lacks the appropriate facilities for implicit sequencing.
 - b) Such facilities as are offered are not fully exploited, often because current teaching breeds a certain timidity towards “advanced” are reputedly “inefficient” techniques.
2. Explicit sequencing, whether forced by the language used, or merely bred by traditional methods of teaching, is often at the expense of “transparency.” That is to day, roughly, it tends to occur most in programs that are least illuminating about their outcome, and hence, usually, most difficult to modify or transcribe into another language.
3. ALGOL60 made some progress in reducing the compulsions to provide explicit sequencing. But it did not go far enough.
4. For a programmer immersed in current attitudes the trick of getting rid of labels is probably the only way of understanding these issues. But it must be done with some intelligence. The trick is not foolproof; it is merely a means to understanding. It suggests a rather more radical approach in which not only are jumps eliminated, but the actual computational steps are reduced in number, and increased in power and written size. And this makes language improvements even more pressing.

The above conclusions are likely to follow from experimentation on the lines indicated here. But there are, independent of experiment, good reasons for not being surprised by them.

Open Access This article is distributed under the terms of the Creative Commons Attribution Noncommercial License which permits any noncommercial use, distribution, and reproduction in any medium, provided the original author(s) and source are credited.

Appendix

The methods presented above are here⁶ applied to Algorithm 196 from the *Commun. ACM*. This ALGOL60 program uses complex arithmetic and starts with a procedure *complex* to fill the gap left here by ALGOL60. The biggest cheat in our rendering is that it presumes permission to infix the complex operators, and indeed to use the same symbols. Without this permission we would have been forced to use *prefixed* operators. This issue is not very significant for the present contention. A facility that would legitimize it is described by McCarthy in *Algol Bulletin* 19.

⁶Editor’s note: The handwritten notations in Landin’s re-written program are indecipherable in the photocopy available to me. Both the original program, which was reproduced from [8], and Landin’s functional reformulation have been omitted here.

References

1. Brown, G.W.: A new concept in programming. In: Greenberger, M. (ed.) *Computers and the World of the Future*, pp. 251–271. MIT Press, Cambridge (1962)
2. *Software Advances 1964–73*, volume II, Technical supplement. Diedold Group Inc., December (1964)
3. Duncan, F.G., Hawkins, E.N.: Pseudo-code translation on multi-level storage machines. In: *Information Processing, Proceedings of the International Conference on Information Processing*. UNESCO, Paris (1960)
4. Landin, P.J.: Programming without imperatives—an example. *Univac Systems Programming Research*, February (1965)
5. Marill, T.: Computational chains and the simplification of computer programs. *IRE Comput. Trans.* **EC-11**(2) (1962)
6. NPL technical report: IBM, December (1964)
7. Radin, C., Rogoway, H.P.: NPL: highlights of a new programming language. *Commun. ACM* **8**(1), 9–17 (1965)
8. Rodman, R.D.: Algorithm 196. Müller’s method for finding roots of an arbitrary function. *Commun. ACM* **6**(8), 442–443 (1963)
9. van Wijngaarden, A.: Recursive definition of syntax and semantics. In: Steel, T.B. Jr. (ed.) *Formal Language Description Languages for Computer Programming*, pp. 13–24. *Proceedings of the IFIP Working Conference, Baden bei Wien, Austria, September 1964*. North-Holland, Amsterdam (1966)