Prof. Dr. Edsger W. Dijkstra
6602 Robbie Creek Cove
Austin, TX 78750-8138
United States of America

to Raphael L. Levien
Levien Instrument Co.
Box 31
McDowell, VA 24458

Saturday 11 November 1989

Dear Raph,

thank you very much for your letter of 12 Sep. 1989 and the enclosed "Io: a new programming notation". When I received your paper, I tried to read it twice. Both times I got stuck, so I filed it for "later", at the corner of my desk. Yesterday evening I returned to it, and still got stuck. You wrote in your covering letter "I have great difficulty in explaining [...] this notation". I can believe that for you could have done something radically new. And as it looks very simple indeed, we should try to figure out how to explain it clearly. (So clearly that a simple mind like mine can understand it!)

For a start, let me tell you the problems I encountered; in passing we can discuss how they could have been avoided, and suggest (minor) improvements.

Very minor point: if "write" is an identifier in your program, avoid starting a sentence with it, for "Write" is another identifier. On pg. 2 +4 the problem could have been solved by using a

semicolon: "... the only argument; write* displays..." (This was not confusing.)

I was distracted by the * : usually the superscript * is a postfix operator, e.g. if z is a complex number z* is its complex conjugate. It took me quite some time to realize that the Io notation has no such operator and that all would have been simpler if "write*" had been replaced by a more neutral identifier, "show" say.

I was also a little bit distracted by the -p.2 +20- "(action)", which is not used in the next example, which makes the "So" in the next line a little bit funny.

* * *

In the mean time it is more than 2 hours later. During those hours I made an effort of rewriting "Actions in Io" - page 0 & 1, included, contain my first effort -. What I tried to do was to separate semantic decisions from syntactic ones and to give the rationale for the latter ones. It may be worthwhile to point out that, in Io, the semicolon is now right-associative, and no longer fully associative as in ALGOL 60, Pascal, etc. You could add in this first paragraph a hint of the gain - not shown in the simple example - that now operators could have more actions among their arguments.

On page 3 I had two minor problems. It took me quite some time before I had realized that, while in your examples write, term (and later +, -, and =) are operators supposed to be defined elsewhere - or perhaps even part of the Io notation - ret is just an arbitrary identifier, as arbitrary as writetwice. This is one of the problems of writing keywords and predifined system identifiers in the same font as arbitrary identifiers. If both are chosen on mnemonic considerations, the reader without a language manual can get utterly confused.

The other unsolved problem was with your definitions of writetwice: has indentation a meaning or is it only a visual aid to parsing? (I hope the latter.)

I must admit that until a few minutes ago I had overlooked the periods after action (3+1) term (3+6) and ret (3+14). Presumably these periods are part of the Io notation; if so, I would like this to have been stressed. (If so, the top sentence starting with "The notation for this....." could do with another period. In this area I have had grave problems with the compositor of our latest book.)

With the last example of this section I have serious problems. What is the scope of the arbitrary identifier x ?

For instance, can I get 5 5 on the screen by writing and then performing

```
+ 2 3 → x;
write x;
write x;
term
```

?

I think so, but I also feel that scope rules need to be made more explicit. (Another thing I see only just now that on p2-3

→ variables; action

a new role for the semicolon has been introduced! I think it can still be viewed as right-associative, but this needs investigation.)

But is your own example syntactically correct? Clearly

→ x ; write x; term

is an action. If this action is supplied as last argument to + , it should be preceded by a semicolon:

+ 2 3 ; → x; write x; term

this was the place that first got me puzzled —in margine I wrote that here new definitions seemed to be hidden—. Now I have seen that you violated your own syntactic rules, I am worried. Let us suppose that this can be fixed.

* * *

Problems on page 4, apart from the possible significance of indentation. Could the first example be written as follows?

```
count: → start end ret;
    write start;
    = start end (ret);
        + start 1 → y;
        count y end ret.
```

Here, and similarly in the gcd example, the scope rules are unclear.

In the gcd example it would have helped me if you had stressed that no new rules or conventions are needed for functions (like gcd) that deliver a value (or a number of values). My first (erroneous) impression was that the line "ret x." represented something new, only explained by example. Is my impression correct that in order to show gcd of (2+4) and (4+11), I have to program

```
+ 2 4 → x;
  + 4 11 → y;
    gcd x y → z;
      write z;
      term
```

?

And that all intermediate results have to be named? If so, the section "Taking arguments" should/could perhaps be extended by pointing out the following.

The third argument of + is an action that expects a single (integer) argument. We wish to supply that integer argument to write, which requires two arguments (integer and action). The → x mechanism creates the possibility to name the integer argument supplied and to offer it in combination with term -say- to write. Would -your terminology-

+ 2 3 write*

have been semantically equivalent with your last example of "Taking arguments"? If so, that would have clarified → x. (It would almost make →x into an action, viz. that of naming its arguments; in that case the semicolon would no longer be overloaded!)

~~The example on top of p.5 does not make sense to me: I don't see what the line~~

Am I correct that the last example on p.4 should have been

writepair (twonums) ;term ?

I do not understand makepair on top of p.5 and would like to see an example of its use. I tried to write 242 338 on the screen using writepair and makepair but did not succeed: writepair needs two action arguments, makepair needs besides x and y, one action argument, and term is the only further action available.

In the section Data Structures I got lost. I tried to convince myself that

```
append-to-head 1 empty → list;
writelist list;
term
```

shows 1 on the screen. I applied the definition of append-to-head, doing the substitutions for num, list, and ret: I got

(→ list; writelist list; term)(→ null full; full 1 empty) *)

in which I interpreted list, null & full as dummies. I could not eliminate them. Did I something wrong?

I think that, if it can all be made unambiguous and free of contradictions, you did something very ingeneous that indeed may give a delightful combination of simplicity and flexibility. A formal syntax would help. Then you would not have fallen into the trap of the missing parentheses. Also it should be possible to state the reduction rules. Would the next step after *) have been the elimination of list? Like

writelist (→null full; full 1 empty); term ?

Now I am worried about the semicolon in front of term! I give up.

If you can clarify matters, please let me know: your endeavour seems intriguing enough.

With my greetings and best wishes,

yours ever,

Edsger W. Dijkstra

encl. 0-1

The main mechanism in Io is the action, which consists syntactically of an operator followed by the appropriate number of arguments of the appropriate types. For instance, let show be an operator accepting one argument and let its application consist of displaying its argument on the screen. Then

```
show 5
```

is an action, and when that action is performed, the screen shows 5 .

[From original text: "This type of mechanism..... has written the number"]

To display 5 6 7 we could now adopt the convention of programming

```
write 5 (write 6 ( show 7))
```

which shows explicitly that the first write is applied to two arguments, viz., the number 5 and the action (write 6 (show 7)) . This syntactic convention, however, would lead to unacceptably many closing parentheses at the end of the program.

Alternatively we could adopt the convention of omitting all parentheses

```
write 5 write 6 show 7
```

because with the knowledge of the number of

arguments that each operator is applied to, our program without parentheses admits only the previous parse. Experience with heavily context-dependent parsing has been sufficiently unfavourable not to adopt this convention for Io.

. In Io, a convention has been adopted that seems to give us the best of two worlds. For an action that occurs in an argument list but is not the last argument of this list, the parentheses are maintained, i.e., we program

(action) .

Since these actions are very much in the minority, not too many parentheses are introduced this way. An action that occurs as last argument on a parameter list, however, is not surrounded by a parenthesis pair but is preceded by a semicolon instead, i.e. we program

; action .

Since such actions are the vast majority, we have eliminated most parentheses without sacrificing context-free parsing. In Io, we write our above example

write 5 ; write 6 ; show 7 .

[etc.]