A POLEMIC by marc le brun:

TILTING AT WINDMILLS, OR

WHAT'S WRONG WITH BASIC?

BASIC *vs. all those other languages*

For you FORTRAN, APL and COBOL fanatics who are upset at our s–t–r–o–n–g stand for BASIC, we owe you an explanation. We deal primarily with school age kids and prefer BASIC for any or all of these reasons:

BASIC is easily learned by large numbers of students. (We want to see many kids involved, not an elite few.)

BASIC is easily learned by people with a wide range of ability (you don't need a strong math background to learn it).

You can learn BASIC quickly. You don't need hours of instruction before you can write your own simple programs.

There are a large number of instructional materials available written for novice computer users (teach yourself style, not programmed instruction).

There is a tremendous wealth of support materials written to support BASIC in a variety of classroom situations (Huntington project, Project SOLO, Denver Calculus Project, REACT, HP and DEC materials).

The extensions to BASIC now available on nearly any disk, time sharing system make it possible to do nearly anything in BASIC that you might want to do in another language, though BASIC may not be as *efficient*.

BASIC is available in a time sharing environment on many very low cost computer systems.

Society to
Help
Abolish
Fortran
Teaching
SHAFT

P. O. Box 310
Menlo Park, Calif. 94025

Now, before you hasten to write and apply these same principles to the language of your choice, keep in mind our target – KIDS – kids ages 8 to 18 in a problem-solving, simulation and gaming environment. We're not concerned with huge number crunching applications nor do we care about super-complicated data processing schemes for which there are many languages far superior to BASIC. We just hope to introduce a large number of new people to a super, fun, learning/recreational tool and want to do it at a reasonable cost and without a great deal of fuss.

For the record, we have recently started doing some work in PILON (PYLON), a subset of PILOT, in our environment. A future issue will give details of our successes or failures. We are also very curious about LOGO and its possible use in our environment. Our friends at Lawrence Hall of Science are working on that and we hope to report on their progress as time goes on. We admit that we are BASIC BIGOTS, but we're looking at other languages as well ... but not FORTRAN, APL, or COBOL!

Leroy F.

It may come as a surprise to many educational/recreational computer users that the language they have come to know and love has serious flaws. By a flaw I mean something fundamentally wrong, as opposed to what could be called defects – problems which could be solved or somehow avoided.

Every actual computer language has its very own garden of defects, which everybody talks about but nobody does anything about. By actual computer language I refer to ones which are actually implemented on some computer somewhere, as opposed to theoretical languages, or specifications for proposed languages, which occur frequently in the literature of computer science. A defect in a language usually results in users complaining loudly about such things as "It won't let me use zero for a subscript in my arrays" or "It won't let me use the bell in PRINT statements" and so on.

Defects are annoying but not serious, they can be corrected by any reasonably competent programmer, assuming there is adequate documentation for the language, which there usually isn't, which is why manufacturers can sell "better and better" versions of the same language (and possibly additional hardware as well) to users who have become hopelessly dependent on the language. Like drug addiction, the first one is usually free.

Flaws in a language are much more dangerous, because the novice user does not notice them. They insidiously distort his views of what computers are capable of, what he is capable of doing with computers, and he is well on the way to joining those tragic figures who sit for hours, backs hunched, eyes glazed, nodding over their terminals: the "experienced" users.

I shall now treat the following topics in turn; What is right about BASIC; What is wrong about BASIC from the standpoint of education; What is wrong about BASIC from the standpoint of recreational use, and finally; What is wrong about BASIC from the standpoint of computer science.

Many of you may have noticed that this periodical has adopted BASIC as a sort of *lingua franca*. This is mainly for pragmatic reasons, and the purpose of this polemic is, in part, an attempt to temper what otherwise might be taken as an implicit whole-hearted acceptance of BASIC. As far as I can tell, the following things are more or less right about BASIC.

It's popular.
: An increasing number of people are using BASIC, and, despite problems with incompatible versions, they are able to communicate, and exchange ideas and programs with each other, even though they have extremely different orientations.

It's interactive, more or less.
: BASIC was designed to be used in a time sharing environment, so the mechanics of writing and running programs are fairly straightforward, within the limitations of the system it's implemented on, anyway. The I/O statements (INPUT and PRINT), though limited, make the writing of interactive programs relatively easy, which makes it one of the few languages suitable for recreational purposes.

It's widely available.
: Many manufacturers offer it, due to an ever growing demand for it by the educational segment of the market (as well as a not insignificant portion of the industrial). Besides, if a manufacturer provides software he also influences how much hardware the consumer buys: "But with the addition of another 32K of core memory and 16 discs you can run our WALLAPALOOZA BASIC ... " The first one is free. He can also sell you software that no self-respecting programmer would admit having written, and the consumer never suspects that he is using software which requires twice as much hardware, services half as many users, has more bugs and fewer capabilities than it could have. Anyway, BASIC is widely available, *caveat emptor*.

It doesn't rock the boat.
: BASIC fits in real smooth with the views most people have about what computation is. It introduces them to computers without forcing them to undertake radical restructuring of their concepts. This helps make computers more acceptable, and in a very real sense more accessible to a large number of people, which is good. One of the important things about technology however, is that it forces people to adopt mental models of the world and how it works which are better aligned with reality. More about this in the next section.

Because of the increasing utilization of BASIC in the educational field the following criticisms are particularly relevant. From an educational standpoint the things wrong with BASIC are.

It doesn't rock the boat.
: As mentioned above, BASIC is not a mind-expander. It certainly expands the class of experiences and possibilities of anyone who uses it, but this is mainly due to exposure to the computer, and not to BASIC itself. Any system or structure which embodies fundamental principles in a coherent manner almost always causes significant transformations in the habits of thought of those exposed to it. For example, the language LISP, which is directly derived from the fundamental ideas of predicate calculus and recursive functions (more on this later) has a peculiar phenomenon associated with it which I call the "LISP trance." After overcoming the initial difficulties assimilating the principles behind the language the student or new user often finds himself perceiving the world in terms of these basic ideas. A tree is a recursive function applied to a seed, it's branches are a list structure, leaves become "atoms" (specialized meaning) and so on. This state can last as long as a week sometimes. The "cause" of this phenomenon is that the student *has learned patterns which are fundamental* enough to cause noticable restructuring of his internal environment. The new user "comes out of the trance" when this restructuring process has leveled off to the point where the changes involved are imperceptable in relation to the adaptations necessitated by daily life. Since BASIC contains few patterns of any significance (and even those in a muddled sort of way) it is easily used as an instructional medium. It is not necessary for teachers to retool their antiquated and inadequate concepts in order to use it to transfer their antiquated and inadequate concepts to their students. This is not to the student's ultimate benefit.

It creates false impressions.
: Besides the theoretically unsound patterns BASIC embodies, which will be discussed later, the language has two other flaws which generally cause the user to have limiting misconceptions; First, there is nothing in it which gives the user any information or facility with computer *systems* in general. Especially deficient in most implementations are the means of dealing with peripheral devices. **(Did you know that in many third-generation computers *every* device is a peripheral, including the CPU and the memory, and the "computer" is really the device which connects and coordinates all the other devices?)** Secondly, "strings" to the contrary (very few languages have real string manipulation features), it forces user to think numerically. I have nothing against numbers, or numerical analysis as such, but numerically oriented processes are but a small portion of the

entire space of computational processes, and there are many significant, useful and fun things that can be and are done without using either numbers or strings. **(Did you know that inside the computer the BASIC you use probably spends as much as 80 or 90 percent of its time performing non-numeric computations?)** For example, the fields of computational linguistics, operating systems design, and articifial intelligence are concerned almost exclusively with non-numeric processes. What, you ask, could a language be "about" besides letters or numbers? We will discuss this in the theory section. BASIC provides the interested student with very little background if he is to actually use computers in any significant way in the future, as well as causing him to adopt faulty thinking habits. It's inhibiting.

BASIC restricts exploration and experimentation, and thwarts attempts at self-directed learning, unless the learner is very highly motivated. It is not natural to express or describe processes which are not primarily numeric in nature in the BASIC language, and this implicitly restricts the kinds of uses to which it can be put. Numbers are not all that interesting to most people, and a steady diet of BASIC will never indicate to anyone the most powerful and general view of what a computer is; a *symbol processor*, that is, a tool for manipulating conceptual objects as opposed to tools which manipulate physical material. Consequently the faithful BASIC user is unlikely to discover on his own the full potentialities of the phenomenon of computation.

Many of the observations in the previous section apply with little modification to those users who are primarily concerned with the recreational use of computers. "Recreational" is a somewhat vague adjective, usually meaning "interested in games" or "interested in art." Fairly interesting games can be produced in BASIC, unless the computer is to compete on an equal basis with human players, in which case I advise you to learn some computer science, explore the literature on artificial intelligence, and forget about BASIC entirely (assuming, of course, that the games you're interested in are non-trivial). Also, many of the mathematical features in BASIC are so much dead weight as far as game playing is concerned. For the artist the situation varies depending on the media he wants to work in. There are two ways in which computers can be used in the creative process; design and experimentation, and execution of the work itself. For most users the only type of art that is even remotely possible to directly create with the machine are line drawings, and that only if you have a plotter. Design and experimentation is the most accessible area for exploration at the present time. The inadequacy of BASIC as a medium for expression of non-numeric concepts is clearly a severe limitation here. Again, if you are interested in having the machine "make up" works of art of any significance, abandon BASIC, do your computer science homework and read up on artificial intelligence. I have done a number of things in the area of computer music, one of which was a program that composed original four voice compositions. Your author now divulges a dark secret of his dismal past – *the first computer language I ever used was BASIC!* In the context of this article this is indeed a painful revelation, but it should lend some weight to my argument, since I have sat on both sides of the fence, so to speak. Anyway, the aforementioned program was written in BASIC, and one of the best available implementations at that, (I will not reveal *which* implementation firstly because I don't wish to plug *any* BASIC systems, and secondly because it was done on stolen time, only the names were changed to protect the guilty). Including debugging, it took several months to complete; had it been written in a more suitable language (such as ALGOL) it would have taken at most two weeks, even accounting for my substantial inexperience in things computational at the time. To summarize then; BASIC, while workable to a limited extent, is nearly useless to the incipient computer artist. Then again, so are most other currently available languages.

Before leaving the subject of computer art I would like to make a few brief technical comments on making "teletype pictures:" First, the best ways of doing it are fairly complicated; Second, find out if your BASIC system will let you PRINT a thing called a "bare carriage return," which lets you type characters on top of each other (sometimes you can do it with TAB, assuming of course you *have* TAB on your system, . . .); Third, even the best teletype pictures look terrible, and if you keep doing it hair will grow on your palms and; Fourth, if you persist on this road to esthetic nitwitdom, you will be aided and abetted by the author in an article on Teletype Plotting in a future issue of this amoral periodical.

It is now my intention to examine the "rotten apple which spoiled the bunch," the things that are wrong about BASIC from the standpoint of computer science. It is in this area that the worst flaws exist; hopefully the following comments will also give some indication of what better languages are like.

BASIC has a regular rat's nest of flaws in the area called *control structures* in the language of computer science. What's worse, these problems interact with each other, so it is somewhat difficult subject to untangle. Nevertheless, in the interests of clean living, we shall attempt to thread our way through them.

Imagine your favorite program is RUNning merrily along. At any moment the computer is executing some particular line. Call that line the *active line* and pretend all the other lines are just hanging around waiting to become active when the computer gets to them. The computer does whatever you told it to do in that line. Now if we look at a smaller segment of time we find that the computer is not doing everything on that line all at once, instead it's executing some part of the active line, such as adding two numbers together. We could look at even smaller segments of time, but we would wind up looking at machine language or logic circuits or electronics or quantum mechanics, depending on how small we got, and none of those things concern us here.

We can also look at larger segments of time, and we see that whole big chunks of our program become active together, that is, when one of the lines in the chunk becomes active then you know that all the other lines in the chunk are going to become active pretty soon. This happens to the lines inside a FOR-NEXT loop for instance. These "chunks" have a special name in computer science, they are called *blocks*. Now we can sit and just watch the blocks in our program become active, *without paying any attention to what the lines in the block are telling the computer to do.* If we watch the blocks become active for a while, we will see a couple of different things happen: a block will be active and then suddenly another block somewhere else will be active (the computer did a GO TO), a block will become active over and over (that's a FOR-NEXT loop), sometimes one block will become active after a particular block and sometimes a different one will (IF-THEN) etc. **(What do you imagine a GOSUB looks like?)**

If we watch the program for a long time, maybe for several RUNs, we will observe certain regularities in which blocks become active before or after other blocks. These regularities form a pattern or structure, and that is what the control structure of the program is. (Flowcharts are a primitive way to draw pictures of control structures.)

Now if we look at a LISTing of our program we can see that the places or lines in the program where the computer goes to another block are always one of a small set of statements; GO TO, IF-THEN and so on. Statements of this sort are sometimes called *control statements.*

At this point I can show you one of the flaws in BASIC. Why it's a flaw I will explain a little later. Take the program LISTing and look at it. It is possible to figure out what the control structure is, for instance we could take the LISTing and cut it up into blocks and paste them on a big piece of paper and draw arrows between them, to show what can come after what.

Now take another LISTing and cut it into blocks like before, *and then cut off all the line numbers* and mix up all the blocks and then try to make another chart of the control structure like before, Can't do it, right?

Now do exactly what you did in the last paragraph *except don't cut off the very first line number in each block.* Now you can make the chart. Obviously there is something special about those line numbers which makes them different from all the other line numbers in the program. (So what are all those other line numbers for anyway? We'll discuss that pretty soon.)

Now for one last experiment. Cut all the line numbers off a LISTing and save them, and throw the rest of the program away. Now assume you have amnesia, and don't remember all this stuff we just did. Put a mark of some kind next to all the line numbers in the program which are special. Can't tell them apart, right?

Now clean up all those scraps of paper and I will tell you what this cut and paste exercise was all about.

The first computer science flaw we have found in BASIC is this:

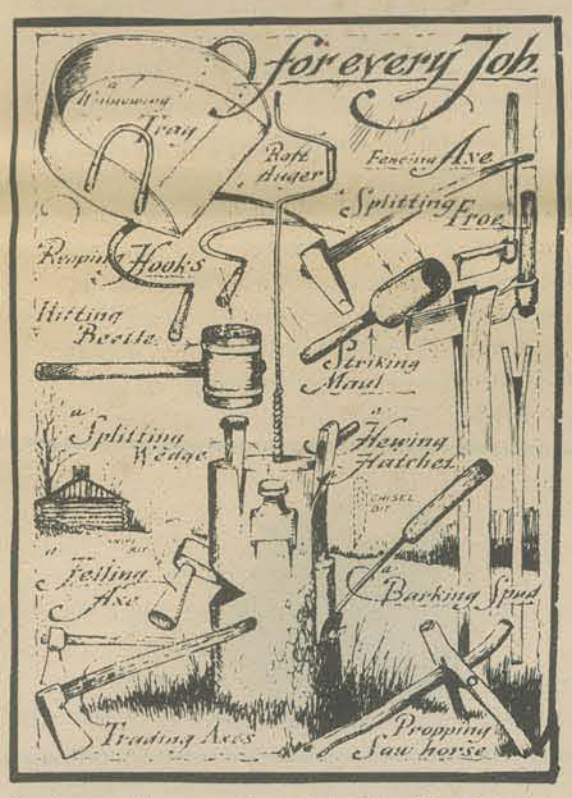
EVERYTHING YOU ALWAYS WANTED TO KNOW ABOUT BASIC BUT WERE AFRAID THAT YOU COULDN'T UNDERSTAND . . .

The written form of the program gives very little indication of its control structure. Since the control structure reflects how the program operates it is next to impossible to figure out what goes on inside the computer by looking at the program. Since the purpose of a program is to describe what goes on inside the computer, this is clearly a flaw.

In fact, without those special line numbers, there wouldn't be *any* indication at all of the control structure. The only way we could find out is by looking at the "activations" inside the machine, and that is usually just not practical. Those special line numbers have a name in computer science – they are called *labels.* In languages that have labels they are usually written as words, like HERE, HITHER or FOO. Poor old FORTRAN has both line numbers and also labels which are numbers. Never mind, no use beating a dead horse.

Why does BASIC have all those useless line numbers? You are told that line numbers exist so the computer knows what order to do things in, and also so you can insert lines and move them around and stuff. As a matter of fact, line numbers are one of many different means for accomplishing these ends. It is far from the best in terms of user convenience, *but* it is among the easiest to implement when you are designing a language. I do not feel, however, that the needs of the user should at any time be ignored just to make implementation easier. After all, to the user, "a language is forever." Since lines with labels are the only ones the program cares about, all you have to do is remember where they are when you type them in. Then you call all the lines that come between it and the next labeled line things like FOO.1, FOO.2 and so on. If you want to switch them around or edit them later, the EDIT command should be able to tell which is which. But the only time you need those names is when you are typing it in or editing it, they don't have anything to do with the program itself. If you're wondering how you keep all the blocks in the right order, don't worry – *you don't have to.* The only thing that connects them are control statements, and they're already in the blocks. You can think of them as being separated, just like in the chart you made.

Languages that use blocks and labels instead of line numbers are called *block-structured languages.* A very good example of a block-structured language is ALGOL. The idea behind block-structured languages is that the form of the program "on paper" should in some way reflect it's control structure. There is, in addition, another useful thing that can



be done when you have blocks. The blocks can have different "flavors" that is, the language can have different types of blocks to indicate different kinds of processes. BASIC has only one sort of block (in the strict sense) and that is the block defined by FOR and NEXT statements. In languages like ALGOL however, there are many different "flavors" which do what BASIC does with the somewhat messy IF-THEN-ELSE and ON-GO TO instructions, as well as a whole lot of other neat things. So blocks are a means by which the program itself can contain useful information about what it is supposed to do.

You may have noticed that the FOR-NEXT blocks in the program you were examining did not need labels. In fact, you do not, in general, need labels for blocks if their boundries and "flavor" are well defined. You may be wondering at this point if your humble author does not have a few screws loose somewhere, first he goes to great pains to explain all about labels, and then he turns around and says that they're hardly ever necessary. If you've guessed it's because I am about to reveal another flaw, you're correct. So let's get down to business.

Let's make a list of the various control statements that still can't be done by some flavor of block, or else need labels for some reason:

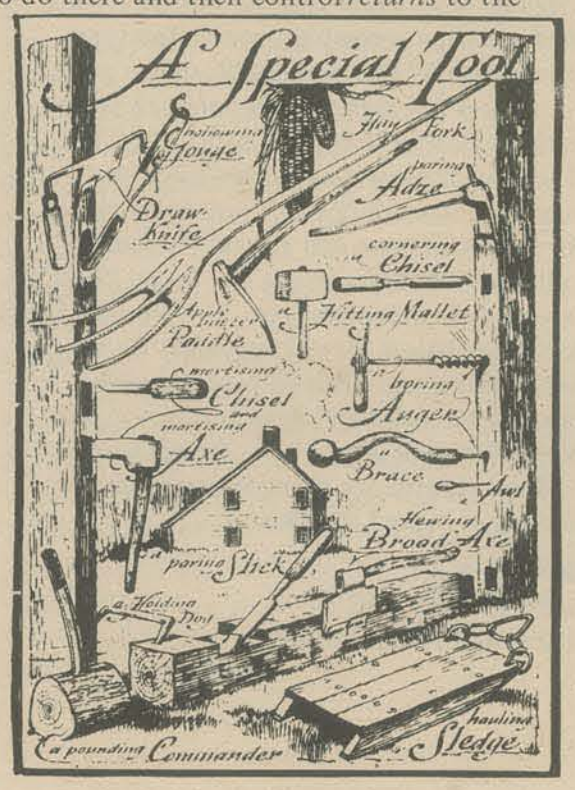
GOSUB (and his sidekick RETURN)
DEF and DIM and
GO TO

For GOSUB the situation seems fairly obvious – we make up a new flavor of block, called say a SUBROUTINE block or a PROCEDURE and we give it a name (label) and we end the block with a RETURN statement. Then if somewhere else in the program we want to call the block we write GOSUB MUMBLE or whatever the name of that block happens to be. OK, that's pretty straightforward, but let me ask you this: Have your programs ever had problems because BASIC sort of "fell" into a subroutine from the statements preceding that section of your program? Not nice is it? Here's computer science flaw number two:

BASIC does not adequately differentiate those sections of the program which are self-contained and meanfully separate and distinct. As a consequence, the "flow" of control into and out of subroutines may deviate radically from its intended and logical "course."

Subroutines are *fundamentally different* from other parts of the program and the language should provide a definite mechanism for delineating them. In addition, control should *absolutely never* pass into a subroutine unless that subroutine has been *explicitly called* from some other part of the program.

You may be surprised at the inclusion of DEF and DIM in my list. You probably don't think of them as control statements at all, but as some other sort of beast entirely. Let's go back to where we made up the SUBROUTINE flavored block. You'll remember that at that point we gave it a name (label). Think for a minute about what subroutines are for; they are special "chunks" of the program that we want the computer to "activate" at various and possibly widely separated places in the program. They perform some useful function and then pass control back to whatever part of the program called them. We indicate the subroutine that we wish to activate by means of its label (name). Now consider what happens when we put one of the FN_ functions in a statement somewhere. The computer goes down the line and various parts of it become active (like when it finds a + or something) and then it gets to our FNX (or whatever we name it). At that point control passes to the line on which we DEFined FNX, the computer does whatever we told it to do there and then control *returns* to the



line that was active at first. As far as control goes, this is *exactly like a subroutine.* This distinction that BASIC makes is completely unnecessary, and exists only because it was convenient for the people that wrote BASIC, certainly not because there is anything good about it as far as the user is concerned. We have "discovered" that BASIC had that flavor block all along; the DEF block. Unfortunately most implementations of BASIC only allow DEF blocks that are one line long.

We have not by any means exhausted DEF as a source of flaws. There is an important difference between calling a subroutine with a GOSUB and calling the one line subroutine attached to a DEF statement using an FN_ type call. GOSUB is just a sort of glorified GO TO in that control and *only* control gets passed to the subroutine and RETURNed to the main body of the program. With DEF, the situation is different – *information as well as control* gets passed between the various parts of the program. Information is passed to the DEF "block" by putting it inside the parentheses that follow the FN _ which calls it. Information is passed back to the part of the program which did the calling in the form of the *value* which the "function" *returns.* The things in the parentheses after an FN_ are called *arguments*, and are said to be *passed* in the same sense as control is said to be passed. Argument passing is an extremely important and fundamental idea in computer science. A value returning subroutine (usually called *procedure*) is also a concept with a similar degree of significance. All of the built in functions in BASIC (such as SIN, RND, etc) are value returning procedures. . . which brings us to computer science flaw number three:

There is no consistent or even adequate mechanism is BASIC for passing control and arguments or for returning values. As a consequence the user must bear the burden of "bookkeeping" which, in fact, should be shouldered by the machine.

Pardon the "poetry" please. There's more – another fundamental element found in computer science, the *recursive function* is next to impossible to produce in any form whatsoever in BASIC. A recursive function is a function which is defined in terms of itself. Another way of putting it would be to say a recursive procedure is a procedure which calls itself. There is nothing at all "funny" about this idea. If subroutines can call other subroutines, why shouldn't they be able to call themselves as well? Recent results in computer science indicate that the Theory of Computation itself may have very deep roots indeed in the Theory of Recursive Functions. The idea seems to be so important that I will diverge a moment to give a simple example. The factorial function for integers may be defined as follows:

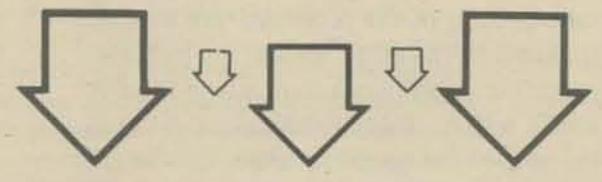
FACTORIAL (X) = 1 if X is zero

X*FACTORIAL(X–1) if X is not zero

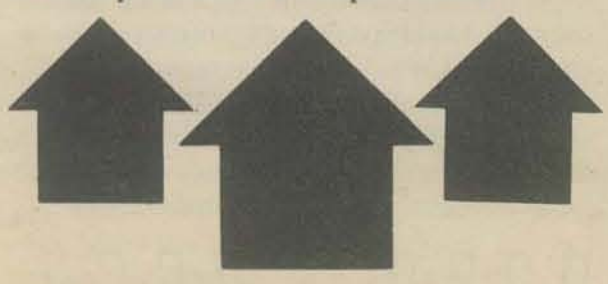
You can see what novice LISP users have to cope with. After you feel you understand this way of defining factorial (as opposed to using a FOR-NEXT loop). Try writing it in BASIC using either DEF or GOSUB. If you can actually get BASIC to swallow it in the first place, try RUNning it to calculate 20! or 30!. If it doesn't blow up consider yourself a natural born programmer, assuming of course, you get the right answer within the limits of accuracy of your system. This is an extremely trivial example for which there exist other and possibly better definitions or calculation methods.

(continued page 8)

There exist however many important useful and fun things which cannot be done any other way. For instance the BASIC you use "figures out" what your lines "mean" and untangles complicated arithmetic expressions by means of a method known as *recursive descent.* I certainly hope so anyway, if it doesn't it's either ten years behind or ten years ahead of other commercially available implementations. There are other, more complicated forms control structures can take, which have mysterious names like *co-routines, parallel n-control streams* and *mutually recursive procedures,* but I won't go into all that since it can all be summed up as computer science flaw-number four:



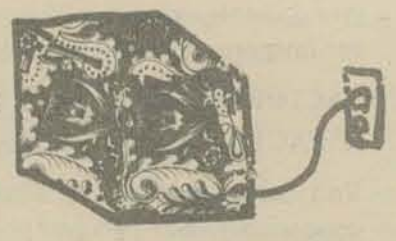
BASIC has no provisions at all for complex control structures, most importantly there is no means of recursion. The result being that it is a very poor language in which to describe processes of a complex nature.



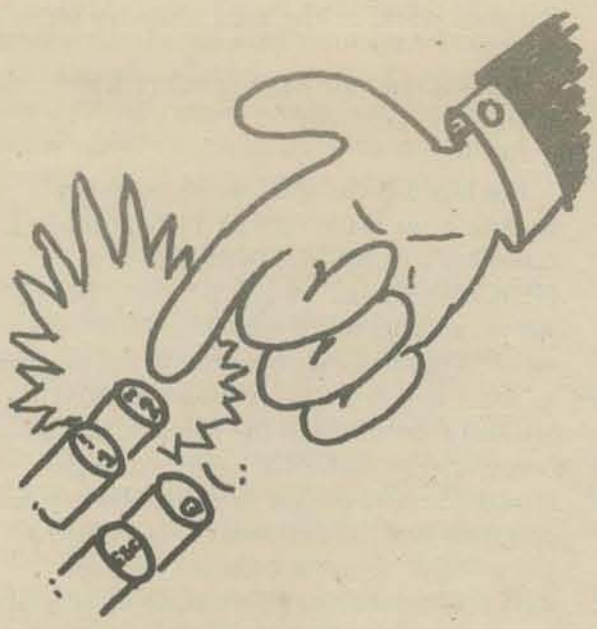
There is still much that computer science has to learn about control structures, and much fascinating and controversial work is being done in this field. BASIC sidesteps the issue entirely by being as bad as possible. Heaven protect me for saying this but even FORTRAN had argument passing. I'm not quite finished with DEF and we haven't even started on DIM but we're almost done with control structures so I will dispense with the last item on our list – GO TO.

Fairly recently it has been demonstrated that in languages with adequate control mechanisms GO TO's are unnecessary. In addition they have been shown to have various nasty side effects, so they have been blacklisted in computer science. They certainly can make a horrible mess out of an otherwise reasonable program. Here are two things you can do in your spare time to earn big money and learn why GO TO's are bad news. First, RUN the following program:

```
10 REM *** LAFF RIOT PROGRAM ***
20 LET N=1
30 FOR I=1 TO 2
40 GOTO 60
50 NEXT I
60 PRINT N;
70 LET N=N+1
80 GOTO 30
90 END
```



Either BASIC will blow up or go on PRINTing numbers forever. If your BASIC system blew up I will explain why a littler later. I also respectfully advise you not to RUN it if there are other users on the system as it might interfere with what they are doing if it blows up - - depending on the implementation. Unless of course, you're some kind of revolutionary or are in a nasty mood.



The second thing is a game you can play with your friends – I call it GO TO NUTS. Each player takes a simple program, say ten lines or so. Then rearrange the lines so their order makes no sense at all. Next insert a whole lot of GO TO's so that the program will RUN. Try to at least triple the length of the original program. Be sneaky, make long chains of GO TO's and put in groups of GO TO's that don't have anything to do with the program. Have more than one GO TO going to the non-GO TO lines. Players then trade programs. Winner is first person to figure out what the program he has is supposed to do. Booby prize goes to first player to go stark staring mad. Players caught in infinite loops are eliminated – RUNning the program is considered cheating. A bug in a program disqualifies the player who wrote it. Try it, you'll hate it, if you survive. Advanced users can no doubt invent more sophisticated variations.

Anyway, you get my point – keep the number of GO TO's in your program to an absolute minimum. BASIC sometimes forces you to use them, but avoid them whenever possible. Large programs with lots of GO TO's very often develop horrible bugs. The same sorts of problems can result from IF-THEN statements as well, because of computer science flaw number two.

Back to DEF. Information is passed to the one line "subroutine" in the form of arguments. Let's make up a "function" FNR so we will have something to point at.

```
10 DEF FNR(X,Y) = X – INT(X/Y)*Y
```

Now suppose that somewhere else in the program we were to write:

```
500 PRINT FNR(15,7)
```

When BASIC got to that line it would PRINT the number 1 (the remainder of 15 divided by 7). Now let's look a little more closely at what BASIC did. First it ignored the 500. Then it "saw" the word PRINT and got all ready to type something on the Teletype. Then it "saw" the label, FNR, of our subroutine. Then it passed the *value* 15 to the variable X, that is, it did a sort of "LET X=15" The same thing was also done for Y and 7. In computer science X and Y would be called *formal parameters* and 15 and 7 would be called *actual parameters.* The funny "LET" operation is called *binding.* The whole process looks like this in formal language: *Passing of arguments to subroutines is accomplished by binding the values of the actual parameters to the variables given as the formal parameters.*

Now suppose we changed our program a little so it looked like this:

```
500 LET X = 2
510 LET Y = 3
520 PRINT FNR(15,7),X,Y
```

When the computer got to that part of the program it would type out 1 2 3 (with a little more space between the numbers). But wait a minute; if BASIC *really* was binding 15 to X and 7 to Y it should type 1 15 7. Aha! We have caught the computer doing something "funny." The X in the DEF "block" is somehow different than the other X. They look the same, but variables given as formal parameters get special treatment *inside* the "block" they are used in. They are said to be *local* to that block. They are also sometimes called "dummy variables" but that is not a nice thing to call anybody. The idea of the same variable having different meanings or values in different parts of the program is called *scoping.* Now suppose we changed FNR so that instead of being defined in terms of X and Y we define it in terms of, say, P and Q. It would work just as well. So we can see that scoping is a very useful thing to have, since it allows us to define things without having to worry about what the actual parameters will be, in exactly the same way that variables themselves are useful because we can define things without having to worry about what their actual values are going to be. Now suppose we were to define another function, FNZ:

```
20 DEF FNZ(X,Y) = X + Y + Z
```

And changed the program so it included the following lines:

```
600 LET Z = 3
610 PRINT FNZ(1,2)
```

When it got to that part the computer would type a 6. Now let's change *just* Line 600 so that it looks like this:

```
600 LET Z = 4
```

Now the computer would type a 7. It is easy to see why this happens even though we did not change Line 610 – it is because we changed the value of Z. The computer does not care about the values we gave X and Y earlier in the program because they are local to FNZ. But it *did* matter what the value of Z was at that point because it was "outside" of the scope of FNZ. These "outside" variables

are said to be *global* (in this case Z is global with respect to FNZ). Now we can see another useful property of scoping; the various parts of our program do not have to "worry" about what "names" they give the variables they use unless they want to refer to *the same thing,* in which case that thing is global with respect to them. Exactly the same thing is true of people – it does not matter a bit if both of us have a secret name for something or other, such as gizmo, which we use when we are thinking or muttering to ourselves, but if we are out in the forest together and you say the word *bear* because one is standing behind me, I had better not think you are talking about the weather, because if I do I will get eaten up! So pardon the anthropomorphism of the last paragraph. To get back to the subject – BASIC only performs scoping inside of DEF blocks. Everywhere else, variables are *always* global. If people were like that it would be like hearing what everybody else was thinking, which would get pretty noisy. Another thing that happens because of this is that you have to be very careful that various parts of

your program do not clobber the value of a variable that another part of the program is using. Again, if people were like that and you were thinking about something and *anybody* anywhere else in the world (globe) used the word for it and changed its meaning the meaning of that word would also all of a sudden change inside *your* head, and what's worse, you'd never notice it until you started thinking nonsense, and maybe not even then. So here is computer science flaw number five:

BASIC does not provide the user with scoping. As a result he has to do a lot of bookkeeping. In addition, programs written in BASIC are liable to have bugs caused by conflicts in variable usage in different parts of the program.

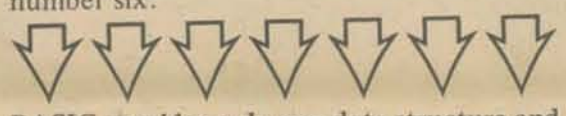
Now what in the world does DIM have to do with all this? In answering this question I will fulfill the promise I made way back when, to explain what a language can be "about" besides numbers. To do this we will have to explore the idea of *value* a little more deeply.

When I used the word value in the previous sections chances are that you just thought "number" when you read it, and maybe thought I was trying to be a smarty pants by not just saying number right out. An example will straighten this out: If I asked you "what is the value of learning about computer science" and you said "two thousand eight hundred fourteen and a half" I would start to worry that you might get violent. I am not just pulling a semantic trick here – variables are just *names* for things, not necessarily the *same* thing all the time, but at any given instant they are the name of some-*thing*, just like the word "that" doesn't always refer to the same thing, but when I use it you know what I'm talking about, or you say "what are you talking about?" if you don't. The "thing" that the variable happens to be the "name for" at that moment is its value. Special names which always mean the same thing (have the same value) are called *constants*. For instance, 3 is the name for the "idea of three-ness" if people use it, but to a computer it is the name for a certain pattern of on and off bits.

So variables are general purpose names, just like "that." Now names can be used for whatever things we want to talk about, they're not picky. In computer languages what we want to talk about are three kinds of things: particular things or objects we are interested in for which we use *symbols* (because computers are dopes and don't know about anything else); patterns or systems that these things can form, which are called *structures;* and the ways in which these patterns can change, which are called *processes*, and are represented by programs.

Up to this point we have talked about everything except structures. True, we did discuss control structures, but they have more to do with processes than things or symbols. If you like you may call "thing patterns" *data structures* which is what is done in computer science. There is also a formal theoretical relationship between structures and processes, but I am not going to confuse the issue by going into all that. If you're interested, think of programs as patterns of symbols and see where it leads, or look into LISP which does away with the distinction almost entirely.

In BASIC, we have symbols for three things: numbers, strings of characters and variables. It is very useful to have symbols that are just plain symbols, but I would call this a defect not a flaw. BASIC provides only one type of structure for these objects – arrays. That is where the DIM statement comes in. The DIM statement is a member of a class of things computer scientists call *declarations.* A declaration is a statement which says "within the current scope these symbols represent objects which have this particular structure." In BASIC the particular structure indicated by the DIM statement is that of an "n-dimensional dense array with orthogonal euclidean connectivity." I give it's full name to show just how special a case it is. There are a whole lot of useful structures besides array thought, such as trees, directed graphs, sparse arrays, and sets. I will not go into what all these things are used for, I will only mention that inside the computer BASIC spends most of it's time doing things with data structures and only a small portion doing things with numbers. We are now ready to state computer science flaw number six:



BASIC provides only one data structure and contains absolutely no facilities for manipulating data structures. Since structures are one of the three main components of computation this flaw causes BASIC to be severely limited.

Besides the DIM statement there are three "hidden" or implicit declarations in BASIC – string variables are declared by the $ in their name, formal parameters are declared by their appearance inside the DEF statement, and the variable used in a FOR-NEXT loop is implicitly declared to be what is called *index variable.* In general, when a declaration is executed, a copy of the given data structure is created. These copies are sometimes called *incarnations.* That is why you can use expressions like

```
SIN(SIN(SIN(X)))
```

Each time the SIN routine in BASIC is called, a new copy of the formal parameter it uses is created. The LAUGH RIOT PROGRAM is intended to test whether the BASIC you're using creates copies of index variables. If it does, the program will fill up the computer's memory with copies of I (the index variable) and then blow up when the memory is full. The last number the program PRINTS will be a rough indication of how much memory you are given by your system. If memory is shared by the other users, the program will blow them up too (which isn't very nice). If your BASIC does not create copies of index variables, the program will just RUN forever, and so much for that. Incarnation is a sort of "time scoping" and produces the same sort of advantages as regular scoping. Things like recursive functions are not possible without creating incarnations.

In conclusion –

BASIC has its uses, and can often be made to perform many of the functions I have outlined, but only if the user is willing to expend the necessary time and effort. It's not the worst introduction to computing, but you can only go so far before it's time to move on to better languages.

BASIC is workable but it is a whole lot less than what the user should be satisfied with. Manufacturers will continue to push BASIC until YOU, the consumer, DEMAND more and better quality languages and software. Any computer large enough for BASIC is also large enough for an ALGOL or LISP of equivalent complexity, and the software *may already exist*, but is not being pushed because, at the present time, BASIC is the big money maker.

Even ALGOL and LISP were long ago superceded by better languages, but commercial availibility lags so far behind development that there would be little point in my even mentioning them in this article. If you're interested, look into the PLANAR language, developed at MIT, which is itself becoming obsolete.

Another factor is that the educational/recreational market was really "cracked open" by BASIC, and the manufacturers will milk it for all it's worth before they introduce something else. And, they are not about to produce good instructional materials for other languages until then either. I'm not criticizing, that's just how it is right now. And the only way it's going to be any different is if YOU stop accepting whatever you're given and start DEMANDING what you need and want.

Furthermore, until now, there has never been any reason to develop languages and systems specifically for educational/recreational use. Here at PCC we are constructing a model of the "ideal" educational/recreational language, and you can help by sending us ideas and suggestions. Until the community of users gets a clear idea of what it wants and needs, we will not be able to supply manufacturers with constructive criticisms. The model is to help us do this.

Finally, the community of users needs to be better informed about the entire field of computation. Since nobody is going to do this for us, we have to help each other get it together. That is, in part, what this magazine is all about.

This article was written under extreme time pressure, and is not as good as it should be. Become an associate editor of PCC – rip out pages, scribble corrections and comments on them, or send us your questions, confusions and gripes. Next issue we will use your feedback to provide further inputs to the educational community. Maybe even a "User's Bill of Rights." Now it's YOUR turn. . .