Arch D. Robison

Illinois Functional Programming: A Tutorial

Construct functional programs with this public domain language

Conventional computer languages are direct descendants of machine language. You can trace the ancestry of features in nearly all computer languages to a machine-level counterpart. Variables are equivalent to storage locations; GOTO and IF statements mimic jump and branch machine instructions. Though these abstractions of machine language free the programmer from machine-specific detail, they still force the programmer to work on the same word-at-a-time level as the machine.

Functional programming, however, puts programming on new foundations. Functional programs have neither the control flow nor the variables of conventional languages. Instead, programs are directly constructed from smaller programs. As a result, functional programming offers a new programming style with advantages that include modular construction, program verification, parallel processing, and optimization.

I've written an interactive functional programming language called Illinois Functional Programming (IFP) that runs under MS-DOS and UNIX. You can write and edit IFP programs with a text editor such as PC-Write and execute the programs interpretively. The IFP interpreter is reasonably compact and fast, and IFP programs execute about as quickly as their BASIC counterparts. Most of the IFP interpreter is based on John Backus's original functional programming paper. (The analogy of variables to GOTOs is from Backus.) The basic syntax is loosely borrowed from Niklaus Wirth's Modula-2.

To run IFP on an MS-DOS–based system, you'll need at least 256K bytes of RAM. Extra memory for a RAM disk is convenient but not necesssary. You'll also need three programs: the IFP interpreter (IFP.EXE), which I've placed in the public domain, a text editor, and a directory lister. You must supply the text editor and directory lister. I recommend PC-Write, which works with IFP under DOS 2.0 and 3.0, and EDLIN, which works only under DOS 3.0. All three programs must reside on one disk, and you must prepare another disk to hold your IFP functions. If you have enough memory, you'll find placing the interpreter, text editor, and directory lister in a RAM disk convenient. For more information on loading and running IFP, see the text box "Running IFP" on page 120.

Structured Data Flow

One of functional programming's advantages is that it gives you greater control of the flow of data. Let's look at the following Pascal program fragment, which computes the inner product of two vectors:

```
S := 0;
for K:=1 to N do
  S := S + A[K]*B[K];
```

At first glance, it appears to be a simply structured program. The S :=0 assignment is followed by a for loop, which controls another assignment statement. Flow of program control into the latter assignment is strictly regulated by the for loop.

Appearances can be deceiving, however. Figure 1 shows the data flow for the same program fragment. The arrows show the various information transfers. Unlike the control flow, the data flow is not well structured. The statement controlled by the for loop can receive and send data around the for loop, thus breaking the hierarchical structure. Furthermore, the variables S and K have more than one arrow going to them, which implies that their values are time-dependent.

While this program example is trivial, a large program may have global variables. Although keeping variables local to procedures eliminates some of the problem, variables are still required to communicate between procedures. Just to understand such large program listings, you must sort out variable interactions and dependencies.

A variable is to data flow what GOTO is to control flow. Just as GOTO allows control to go anywhere, a variable allows data to go anywhere. You can, of course, use scope rules to limit variable interactions, but doing so is merely the same as restricting the scope of a GOTO. The more powerful solution is structured data flow. Just as a GOTO can be replaced by a control structure such as IF...THEN... ELSE or WHILE...DO, a variable can be replaced by a data-flow structure.

Learning the IFP Language

Semantics for IFP are almost identical to Backus's Functional Programming, though the syntax is quite different. The IFP syntax reads from left to right and is block-structured in a manner similar to Pascal. Though the notation is wordy for small programs, it makes large programs easier to read by encouraging an indented style.

The IFP language consists of objects, functions, and program-forming operations (PFOs). Objects are the data structures of IFP. Functions and PFOs correspond to procedures and control structures of conventional languages; however, the

continued

Arch D. Robison holds a B.S. from Case Western University and an M.S. from the University of Illinois at Urbana-Champaign, where he is currently working on his Ph.D. He can be reached at 1304 West Springfield Ave., Urbana, IL 61801.

meanings of functions and PFOs differ radically from their conventional counterparts.

Objects

Objects in IFP are either atoms, sequences, or *bottom*. The latter represents an undefined value (e.g., the result of division by 0) and is written as a question mark (?). Atoms are numbers, strings, or Boolean values. A string must be surrounded by quotation marks if it looks like another kind of atom or contains nonalphanumeric characters. Some sample atoms and their types are shown in table 1.

Sequences are lists of zero or more objects surrounded by angle brackets. Below are some typical sequences.

```
<a,b,c>
<1 2 3 4 5 6>
< >
<<1 2 3> <apple banana> t>
```

Either commas or spaces can be used to separate the elements of a sequence. A sequence with no elements is simply called empty. The elements of a sequence can be any object and do not have to be of the same type. Sequences can contain other sequences as elements, so complex data structures can be expressed. Also, IFP sequences are bottom-preserving; that is, any sequence containing ? is itself equal to ?; the sequence <a,b,?,d> is equal to ?, for example.

Functions

In IFP, a function is applied to an object to yield another object. The application of function *f* to object *x* is written as *x* : *f*. In other words, *x* is the input to *f*. For example, the application of the cosine function to the number 0 is written as 0 : cos. To evaluate this statement on the IFP interpreter, we would enter show 0 : cos;, and the interpreter would answer 1. The command show indicates that we wish to evaluate an application; the semicolon marks the end of the function.

The IFP interpreter distinguishes two kinds of functions: primitive functions and defined functions. Primitive functions are built into the IFP interpreter; defined functions are created by the user. All functions can be used in the same manner; neither primitive nor defined functions are privileged in any way.

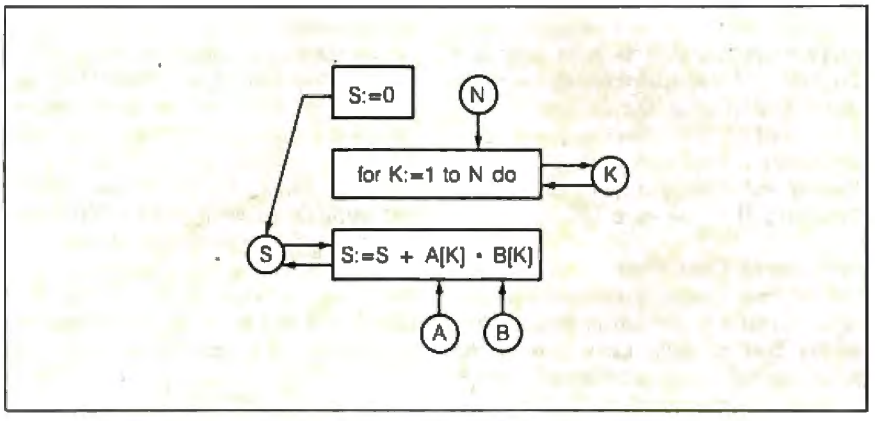

Figure 1: *The data flow of this inner product program written in Pascal is not well structured. The statement controlled by the for loop can break the hierarchical structure by sending and receiving data around the for loop statement. Compare with the data flow in figure 3.*

Table 1: *An IFP atom can be a number, a string, or a Boolean value. If a string looks like another atom or contains nonalphanumeric characters, it must be enclosed in quotation marks.*

IFP Atoms	Type
banana	string
"The cat in the hat"	string (double quotes)
'hello world'	string (single quotes)
7	number
3.1415	number
1e6	number (million)
"1.414"	string
t	Boolean true
f	Boolean false
"t"	string

Organization

IFP functions are stored in a tree structure analogous to the way in which MS-DOS files are stored. Each node of the tree is either a directory or a function, which is saved as a file. A function is referenced by its path, which is a sequence of names separated by slashes. On MS-DOS systems, the interpreter truncates each name to eight characters and converts all letters to uppercase. Paths follow MS-DOS file-path conventions, except that the separator is a forward slash (/), not a backslash (\). The use of a forward slash is the result of developing IFP on UNIX systems, which use a forward slash as the path separator. Paths may also contain two periods (..) to indicate a move up to the parent directory. Relative paths do not begin with a slash, indicating that the path starts at the current directory. Within function definitions, the current directory is the directory in which the function is defined—that is, saved to disk.

When you use a function from another directory frequently, you don't want to have to spell out the entire path every time. To avoid this, you can import a function into a directory from another directory. To import functions into the current directory, you can create an import file in the directory that is to receive the function. The file must be called %IMPORT, and it must contain declarations of the form FROM *directory* IMPORT $function_1$, $function_2$, . . . , $function_n$. In the example, *directory* is actually the directory path. For example, a typical import file might look like this:

```
FROM /sys IMPORT distl, id, iota,
length, tl;
```

After using IMPORT you can reference any imported function as though it were defined in the directory—you only have to write the function name and not the entire path.

Primitive Functions

Primitive functions (those built into the IFP interpreter) have paths like any other function, except that there is no source definition for the function. Most of these functions, such as +, or, cos, and =, are familiar to programmers. These functions behave the same as they do in most languages.

Functions always have a single input and output. However, either the input, output, or both can be a sequence of objects. For example, 2+3 is written as <2 3> : +,

continued

which gives the result 5.

Structural functions, another kind of primitive function, reorganize structures. For example, the function cat catenates several sequences; <<a b> <p> <x y z>> : cat results in <a b p x y z>. Another structural function is reverse, which reverses a sequence; <2 4 6 8> : reverse results in <8 6 4 2>.

Some primitive IFP functions are subsets of other primitives. For example, the sum function adds any sequence of numbers, while the + function adds a sequence of exactly two elements. Having both makes functions easier to read—providing you choose the function that best represents your intent. Table 2 lists the commonly used IFP primitive functions.

Table 2: *The IFP structual functions assemble, reorganize, and select data. Logical functions return Boolean values. The IFP arithmetic functions work the same as those used in conventional languages.*

IFP PRIMITIVE FUNCTIONS

Structural Functions

Appending

$X, \langle y_1,y_2,\ldots y_n\rangle\rangle$: apndl → $\langle X,y_1,y_2,\ldots y_n\rangle$
$\langle\langle x_1,x_2,\ldots x_m\rangle,Y\rangle$: apndr → $\langle x_1,x_2,\ldots x_m Y\rangle$

Distributing

$\langle X, \langle y_1,y_2,\ldots y_n\rangle\rangle$: distl → $\langle\langle X,y_1\rangle\langle X,y_2\rangle\ldots\langle X,y_n\rangle\rangle$
$\langle\langle x_1,x_2,\ldots x_m\rangle,Y\rangle$: distr → $\langle\langle x_1,Y\rangle\langle x_2,Y\rangle\ldots\langle x_m,Y\rangle\rangle$

Selecting sections of a sequence

$\langle\langle x_1,x_2,\ldots x_m\rangle,K\rangle$: pick → x_K
$\langle x_1,x_2,\ldots x_m\rangle$: tl → $\langle x_2,x_3,\ldots x_m\rangle$
$\langle x_1,x_2,\ldots x_m\rangle$: tlr → $\langle x_1,x_2,\ldots x_{m-1}\rangle$
$\langle\langle x_1,x_2,\ldots x_m\rangle,K\rangle$: takel → take first K elements from left end of X
$\langle\langle x_1,x_2,\ldots x_m\rangle,K\rangle$: taker → take " " " " right " " "
$\langle\langle x_1,x_2,\ldots x_m\rangle,K\rangle$: dropl → drop " " " " left " " "
$\langle\langle x_1,x_2,\ldots x_m\rangle,K\rangle$: dropr → drop " " " " right " " "

Regrouping—*input* is a sequence of sequences

<<a b c> <1 2 3>> : trans → <<a 1> <2 b> <3 c>>
(transpose a matrix)
<<a b> <p> <x y z>> : cat → <a b p x y z>
(catenate sequences)

Miscellaneous

X : id → X
n : iota → $\langle 1,2,\ldots n\rangle$
$\langle x_1,x_2,\ldots x_m\rangle$: length → m
$\langle x_1,x_2,\ldots x_m\rangle$: reverse → $\langle x_m,x_{m-1},\ldots x_1\rangle$

Program-Forming Operations

User-defined functions combine primitive and other user-defined functions. In conventional languages, functions are combined by storing intermediate results in variables. In IFP, functions are combined by PFOs. Imagine functions as boxes with a single input and output (figure 2a). The PFOs assemble function boxes to form new function boxes. Each component function is a parameter to the PFO. Like its component functions, each PFO has a single input and output. Higher-level functions can be built from the constructed functions in the same manner.

The output of a function can be connected to the input of another function with the composition PFO (figure 2b). Composition is written using a vertical bar; the composition of *f* and *g*, for example, is *f* | *g*. Composition can be defined by the equality $x : f \mid g = g(f(x))$. Also, composition is associative, so parentheses are not necessary in expressions such as *f* | *g* | *h*.

The construction PFO ties functions together in parallel (figure 2c). The construction of functions is written as a bracketed list of the functions; for example, the construction of functions *f*, *g*, and *h* is written as [*f*,*g*,*h*].

The output is a sequence, each element of which is respectively the result of applying *f*, *g*, and *h* to the input. For example, <12 4> : [+,−,*,%] yields <16 8 48 3>. (The percent sign denotes division.)

Together, composition and construction are sufficient to define many other useful functions. For example, the cotangent function may be defined as

```
DEF cotan AS [cos,sin]|% ;
```

Another interesting function is id (for identity), which simply copies its input to its output. The identity function permits replication of the input when used in conjunction with construction. For example, you could write a function that squares its input:

```
DEF Square AS [id,id]|*;
```

Constant

Not all PFOs have function parameters; some have object parameters. One such PFO is the constant PFO, which generates constant functions. Constant functions always return the same result when applied to any object that is not ? (bottom). Constant functions are written as an object (the value to be returned) preceded by a pound sign (#). For example, *X* : #<cat in hat> will yield <cat in hat> for any value of *X*.

The exception, of course, is that if the object is undefined, the output will also be undefined. This is another aspect of the bottom-preserving property, in which undefined input always results in undefined output—in other words, garbage in, garbage out.

Constant functions introduce constants into a calculation. For instance, a function that takes the radius of a circle as input and outputs the area could be written as

```
DEF CirArea AS
    [Square,#3.141592] | *;
```

Selector

Selector functions return the *n*th element of a sequence where *n* is a positive integer. For example, <a b c d e> : 2 outputs *b*. There is also a corresponding set of select-from-right functions, written as *nr*. These select the *n*th element of a sequence, counting from the right. For example, <apple banana cherry> : 1r outputs cherry. All selectors return ? if the argument has no *n*th element or is not a sequence.

Apply to Each

The apply-to-each PFO is a distributive form that applies a function to each element of a sequence. It is written as EACH *f* END. For example, <<3 7> <8 5> <2 9>> : EACH * END multiplies each pair of numbers and results in <21 40 18>. The function within EACH may be a more complex function; for example,

continued

Arithmetic Functions

Dyadic functions—*input* ≡ <*X*,*Y*>

+ → *X*+*Y*
− → *X*−*Y*
* → *X*×*Y*
% → *X*÷*Y*
div → greatest integer ≤ *X*÷*Y*
mod → *X* mod *Y*
power → X^Y
min → min(*X*,*Y*)
max → max(*X*,*Y*)

Monadic math functions—*input* is real number

exp, ln, sqrt, sin, cos, tan, arcsin, arccos, arctan

Miscellaneous

X : add1 → *X*+1
X : sub1 → *X*−1
$\langle x_1,x_2,\ldots x_n\rangle$: sum → $\sum_i x_i$
X : minus → −*X*

Logical Functions

Comparison—*input* ≡ <*X*,*Y*>

= → equal
~= → not equal
< → less
<= → less or equal
>= → greater or equal
> → greater

Boolean functions—*A* and *B* are Boolean objects

A :~ → not *A*
<*A*,*B*> : and → $A \wedge B$
<*A*,*B*> : or → $A \vee B$
<*A*,*B*> : xor → $A \oplus B$
$\langle b_1,b_2,\ldots b_n\rangle$: all → every b_i is true
$\langle b_1,b_2,\ldots b_n\rangle$: any → at least one b_i is true

Tests—*input* ≡ X

atom → *X* is an atom
boolean → *X* is a Boolean atom
numeric → *X* is a numeric atom
null → sequence *X* is empty

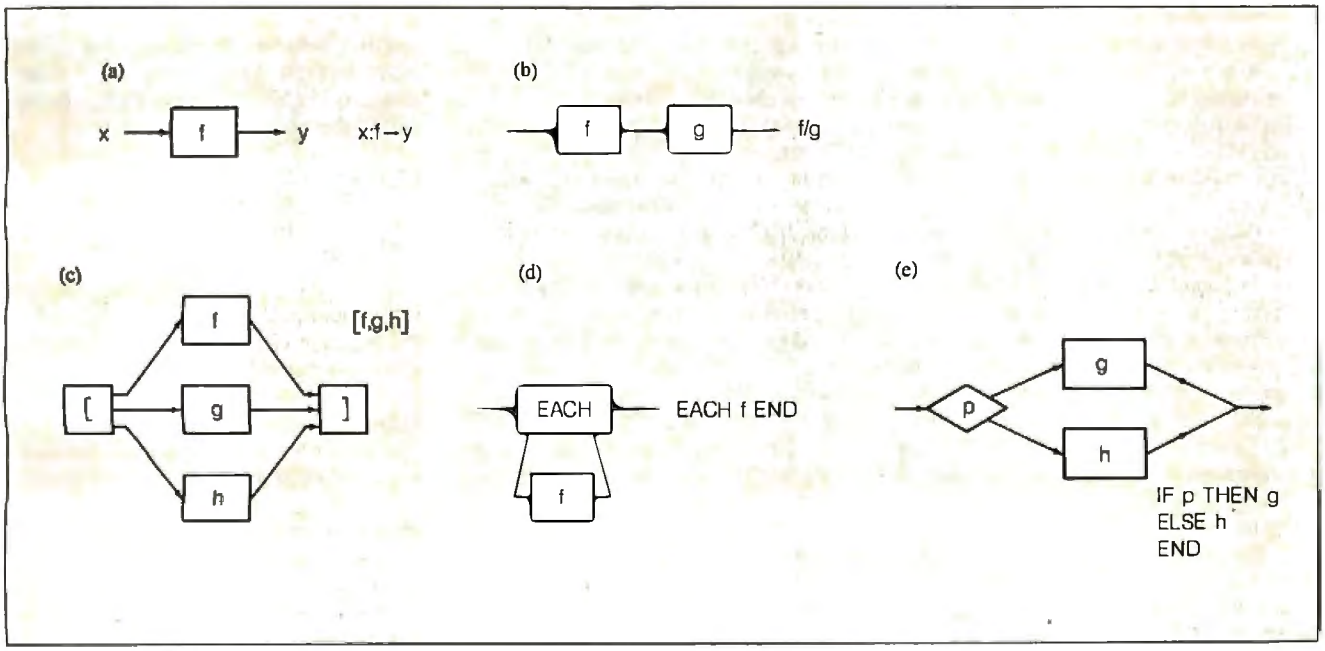

Figure 2: *Each function has a single input and output (a). The composition PFO (b) connects the output of one function to the input of another function. The construction PFO (c) ties functions in parallel to one another. The apply-to-each PFO (d) applies a function to each element of a sequence. The IF...THEN...ELSE PFO (e) lets you apply conditional functions.*

Running IFP

Before invoking IFP, you should set two environment variables. Set the EDITOR variable to the name of your favorite editor. The default editor is PC-Write (c:ed.exe). Set the IFPDIR variable to the name of your favorite directory listing program. Normally you can use the AUTOEXEC.BAT file to set these variables. Below is a sample AUTOEXEC.BAT file:

```
set EDITOR = A:edlin.com
set IFPDIR = A:sd2.com
```

To start an IFP session, change your current working directory to a directory on the IFP functions disk. Then execute the program called ifp.exe. Your current working directory becomes your current working IFP module. There is no way to change your current working directory from within IFP; you must leave the interpreter and change directories from the command level of DOS. When IFP is ready, it will respond with the prompt ifp>.

To end an IFP session, enter the command exit at the prompt. All function definitions are kept in disk files, so you shouldn't lose anything when you exit or if the computer crashes.

To edit an IFP definition file, type the command ed *name*, where *name* is the name of the function to be edited. (Since all IFP reserved words are uppercase, it is a good practice to use lowercase or mixed cases for function names.)

The function can be one that is local to the current working module or one that is imported into the current working module. If the function name is neither defined locally nor imported, then it is assumed to be a new local function. The proper syntax of a function definition file is DEF *name* AS *f*;. Definitions are in free format; line breaks are treated as spaces. As in Pascal, matching pairs of (* and *) are utilized to delimit comments.

Do not switch to another file from within the editor. You must always exit the editor to return to the IFP command interpreter first and then edit the next file. Otherwise, the interpreter won't know that its internal copy of a function is invalid.

To apply an IFP function, type the statement

show *object* : *function*;

The interpreter evaluates the result of applying the function to the object. The result is then printed at the terminal.

To list your functions, type dir at the prompt. The directory listing program specified by IFPDIR will be invoked. My directory lister won't work unless I type a trailing slash (dir/). I have not tried any other directory listing program.

To delete a function, type del *f* at the prompt. The function definition file (along with the memory copy) will be deleted. Wildcards are not permitted in the function name. Do not try to delete files with extensions (such as .bak) from within IFP; since filenames are truncated to eight characters, IFP may delete the wrong file.

Tracing Functions

Currently, IFP has a simple program trace mechanism. To trace a function, at the prompt type trace on *f1*, *f2*, . . . *fn*;, where the *f* is the function to be traced. Whenever a traced function is invoked, its argument and result are shown. Also, the argument and result of all called functions are shown. To stop tracing functions, at the prompt type trace off *f1*, *f2*, . . . *fn*;.

When tracing, the interpreter uses ellipses to abbreviate functions. You can set the depth at which ellipses occur by typing depth *n*, where n is a non-negative integer. The default depth is 2.

There is also a functional form for creating trace functions. Its form is @ *string*.

The function always returns its argument unchanged, and it prints *string:* followed by its argument. For example, <1 2 3> : EACH @banana END will print the following messages:

```
banana: 1
banana: 2
banana: 3
```

This tracing form is for debugging only, since it creates a side effect—the message.

you can square a sequence of numbers: <2 10 4 3 5> : EACH [id,id] | * END, which yields <4 100 16 9 25>. In effect, the EACH command chops the input sequence into separate objects, feeds each object to the parameter function, and assembles the corresponding outputs (figure 2d).

IF. . . THEN. . . ELSE

The IF. . . THEN. . . ELSE PFO lets you apply conditional functions. It is written as *x* : IF *p* THEN *g* ELSE *h* END. Interpreted, that function would read, "If $p(x)$ is true, then evaluate $g(x)$; if $p(x)$ is false, then evaluate $h(x)$" (figure 2e).

The level of nesting of conditional forms can be reduced by using ELSIF clauses. For example:

```
IF p1 THEN f1
ELSE
   IF p2 THEN f2
   ELSE
      IF p3 THEN f3
      ELSE g
      END
   END
END
```

can be rewritten as

```
If p1 THEN f1
ELSIF p2 THEN f2
ELSIF p3 THEN f3
ELSE g
END;
```

The ELSE statement can never be omitted, however. If you want to pass the input straight through the ELSE statement, use ELSE id. For example, you can define the absolute value function as

```
DEF Abs AS
      IF [id,#0]|< THEN minus
      ELSE id
      END;
```

If the input is less than 0, then Abs takes the negative of the input; otherwise, the input is returned unchanged.

Filter

The filter PFO filters through elements of a sequence that meet a criterion. It is written as FILTER *p* END, where *p* is the criterion expressed as a Boolean function. For example, to filter a sequence for all pairs of equal elements, you could write <<*a a*> <*c d*> <*x x*> <*y y*> <*r g*>> : FILTER = END, which would result in <<*a a*> <*x x*> <*y y*>>. The FILTER function is an IFP extension to Backus's Functional Programming. Though FILTER can be done in terms of

continued

In IFP, the notion of correspondence is implemented directly as the structural function trans.

other functions and PFOs, it occurs sufficiently often to merit its inclusion as a PFO.

Right Insert

The insert PFO is used to reduce a sequence. It is written as $<x_1,x_2,\ldots x_n>$: INSERT *f* END. For example, INSERT + END returns the sum of a nonempty sequence. The PFO is called insert because if *f* were written in infix notation, it would be inserted between the elements of the sequence. For example, the summation function mentioned above expands as $<x_1,x_2,x_3>$: INSERT + END to yield $(x_1+(x_2+(x_3+(\ldots))))$. The INSERT PFO is much the same as EACH, except that it sends pairs of values to its parameter function.

Functions formed with INSERT PFO are always undefined for empty sequences. This differs from Backus's Functional Programming, in which such functions returned the right identity element of the parameter function. In theory, this is a convenient feature (e.g., summing an empty sequence would yield 0), but it is impractical for the interpreter to know the identity element of user-defined functions. There are so few cases in which the interpreter could know the identity element that we might as well define special functions for those cases, such as:

```
DEF sum AS
    IF null THEN #0
    ELSE INSERT + END
    END;
```

Alternatively, we can append the identity element to the end of the sequence before inserting

```
DEF sum AS
    [id,#0] | apndr | INSERT +
    END;
```

Note that INSERT starts at the right of the sequence. Currently there is no "left insert" that would reduce a sequence starting from the left. We can get the same effect, however, with INSERT and the sequence reversal function: reverse | INSERT reverse | *f* END.

WHILE...DO

The last PFO is WHILE...DO, which allows indefinite composition. It is written as WHILE *p* DO *f* END;. That is, apply the fewest *f*s required such that $x:f:f:f:\ldots:p$ is true. The WHILE PFO lets us rewrite some recursive programs nonrecursively. For example, consider the recursive function Last, which finds the last element of a sequence by recursively shortening the input sequence:

```
DEF Last AS
    IF t1 | null THEN 1  (* case
      of one-element sequence *)
    ELSE
      t1 | Last  (* recursive
        case *)
    END;
```

(The t1 function takes all but the first element from a sequence; the null function checks if a sequence is empty.) We can rewrite Last nonrecursively as

```
DEF Last AS
```

```
WHILE t1 | null |~ DO
      t1
   END |
   1;
```

(The tilde is the "not" function.) This version of Last keeps taking the tail of the input while there is more than one element left. The WHILE function will then output a one-element sequence for which we take the first element. In the case of Last, we could have also used the insert PFO:

```
DEF Last AS
    INSERT 2 END;
```

Note that the INSERT version of Last works from the back to the front of the input sequence; the WHILE version works from front to back.

Replacing Subscripts and Loops

The von Neumann languages deal with scalar items directly. To handle larger structures such as arrays, loops, and subscripts, you indicate the necessary steps and connections within a computation. Consider adding two vectors *A* and *B* in a Pascal program; you would write

```
for K:=1 to N do
    C[K] :=A[K] + B[K];
```

In English, the algorithm would be stated, "Each element of the result is the sum of the corresponding elements of the arguments." The key word here is "corresponding." In Pascal, the loop and subscripts accomplish the correspondence. In IFP, the notion of correspondence is implemented directly as the structural function trans, which groups corresponding elements. For example, <<*a b c d*> <1 2 3 4>> : trans results in <<*a* 1> <*b* 2> <*c* 3> <*d* 4>>. (The name trans comes from "transpose." If each subsequence represents a row of a matrix, trans transposes the matrix.)

The vector addition program in IFP makes the correspondence and then adds each pair:

```
DEF VectorAdd AS
DEF trans|EACH + END;
```

The input to the IFP program would be a pair of vectors. For example, to add the vectors <1,10,100> and <4,3,2>, we would write <<1 10 100> <4 3 2>> : VectorAdd, which would yield <5 13 102>.

Now you should know enough to write the IFP version of the inner product program listed at the beginning of this article. First, make the correspondence, then take each product, and finally sum the products:

```
DEF Inner AS
    trans |
    EACH * END |
    INSERT + END;
```

The corresponding data-flow graph is shown in figure 3. Compare its simplicity with the graph for the Pascal program.

Scaling a vector demonstrates another common structural function. In Pascal, you would scale vector *A* by scale factor *S* with the following program fragment:

```
for K:=1 to N do
  B[K] := S * A[K];
```

Each element of the input vector is multiplied by the scale factor. Structurally, the scale factor is being distributed over each vector element. In IFP, this distribution is done directly by the distl function. The name distl stands for "distribute from left." It distributes the left argument over each element of the right argument. The

continued

statement <a <1 2 3 4>> : distl yields <<a 1> <a 2> <a 3> <a 4>>.

You can apply a function to each pair formed by distl with the EACH PFO. The vector scaling function could be written as DEF VectorScale AS distl | EACH * END;. Each resulting pair has the distributed argument on the left side. The mirror image of distl is distr, which distributes the right argument over the left argument. For example, <<1 2 3 4> z> : distr results in <<1 z> <2 z> <3 z> <4 z>>.

Another important structural function is the outer product. Suppose we want to generate a Boolean matrix that is the result of comparing all elements in vector *X* against all elements in vector *Y*. In Pascal, we would write

```
for J:=1 to N do
   for K:=1 to M do
      C[J,K] := X[J] = Y[K];
```

To write the routine in IFP, you need a structural function that takes all possible pairs from two sequences. Such a function can be defined from functions already discussed. In the Pascal program above, *X*[J] does not change within the inner loop—the *X*[J] distributes across each *Y*[K]. In IFP you can compute all possible pairs by combining distl and distr; for example, you could write <<a b c> <1 2 3>> : distr | EACH distl END.

To make the comparisons, you could use two levels of EACH and apply the resulting function to the pair <*X Y*>. Of course, when you have two levels of EACH composed, you can merge. Therefore, you could write the Compare function as the following:

```
DEF Compare AS
      distr |
      EACH
            distl | EACH = END
      END;
```

The outer product function occurs quite frequently, so you might want to define it:

```
DEF Outer AS
      distr | EACH distl END;
```

The simplicity of that definition is another advantage of structural functions. In Pascal, the outer product would be difficult if not impossible to make into a separate procedure. So whenever an outer product is necessary in Pascal, the programmer must rewrite it from for loops. In IFP, you define it only once.

At first glance you may think structural functions are grossly inefficient compared to loops and subscripts. Actually, for an interpreter structural functions are quite efficient for two reasons. First, the sequences are represented internally as shared linked lists, so when the interpreter appears to copy large chunks of data, it really just copies the top levels of the lists. Second, the structural functions are interpreted only once. An equivalent BASIC interpreter must interpret the subscript expressions every time through the loop.

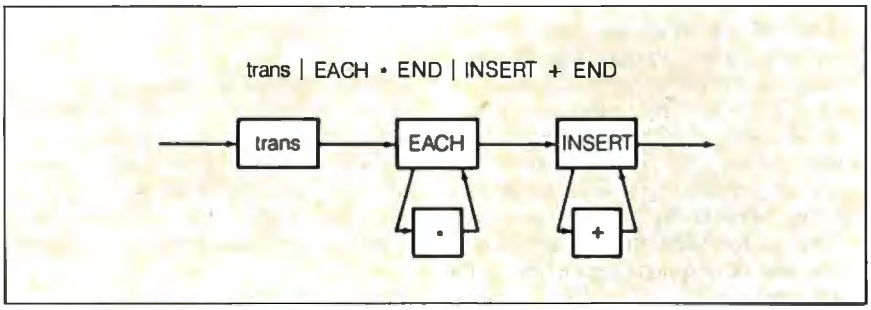

Figure 3: *The data flow of the inner product program is greatly simplified when written in IFP. Compare with the data flow in figure 1.*

Listing 1: *The quicksort algorithm in Pascal has complicated interactions between loops and subscripts that obscure the basis of the algorithm.*

```
(*
 * QuickSort
 *
 * Sorts a sequence of numbers into ascending order using
 * quicksort algorithm.
 *
 * Parameters:
 *      A = array to be sorted
 *      L = index of first element in A
 *      U = index of last element in A
 *
 *)
type Sequence = array [1..100] of real;

procedure QuickSort (var A: Sequence; L,U: integer);

   var J,K: integer;

   procedure Swap (var X,Y: real);
      var T: real;
      begin
         T:=X;
         X:=Y;
         Y:=T;
      end;

   begin
      if L<U then
         begin
            J:=L;          (* A[L] is the partition key *)
            K:=U;
            repeat
               while (J < U) and (A[J] <= A[L]) do J:=J+1;
               while (L < K) and (A[K] >= A[L]) do K:=K-1;
               if J<K then Swap (A[J],A[K]);
            until J>=K;
            Swap (A[L],A[K]);
            QuickSort (A,L,K-1);  (* Sort lower partition *)
            QuickSort (A,K+1,U);  (* Sort upper partition *)
         end;
   end;
```

A Larger Program

All the examples so far have been trivial. To show the real power of functional programming, I'll show the quicksort algorithm in IFP. QuickSort sorts a sequence of keys, which in this example are real

Listing 2: *In IFP, the basis of the quicksort algorithm can be identified easily. (Although the algorithms in listing 1 and 2 are not exactly the same, each implements the quicksort idea as simply as possible in the respective language.)*

```
(*
 * QuickSort
 *
 * This function sorts a sequence of numbers or strings
 * into ascending order using the Quicksort algorithm.
 *
 * Examples:
 *
 *  <3 1 4 1 5 9 2> : QuickSort == <1 1 2 3 4 5 9>
 *
 *  <all work no play> : QuickSort == <all no play work>
 *
 * The sequence may not mix strings and numbers.
 *)

DEF QuickSort AS
   IF [length,#2] | < THEN id
   ELSE
      [id,1] | distr |
      [
         FILTER < END | EACH 1 END | QuickSort,
         FILTER = END | EACH 1 END,
         FILTER > END | EACH 1 END | QuickSort
      ] | cat
   END;
```

numbers to be sorted into ascending order. If the sequence contains fewer than two elements, no sorting is required. If the sequence contains at least two keys, then the first element is chosen as the partition key. All elements less than the partition key are picked out and sorted recursively. Likewise, all elements greater than the partition key are also picked out and sorted recursively. The sorted subsequences are then catenated with all the elements that were equal to the partition key.

Listing 1 shows QuickSort written in Pascal. The loops and subscripts obscure the basis of the algorithm—that of restructuring the sequence into subsequences. Listing 2 shows QuickSort written in IFP. Not only is the restructuring basis of the algorithm more obvious, but the fact that all the partition-key comparisons (in the FILTER) can be done in parallel is clear. (Admittedly, the algorithms in the two listings are not quite the same; each implements the basic quicksort idea as simply as possible in its respective language.)

Future Directions

A lot of work is still required to make functional programming more practical than conventional languages. Input and output, graphics, abstract data typing, and efficient compilation require research and development. Backus has described how inputs to functions can be named without losing the structured flow graphs. But despite its current primitive form, functional programming is worth trying. It will give you a new perspective on programming.

Editor's note: *IFP.TXT (the IFP reference manual) is available on disk, in print, and on BIX. IFP.EXE is available on BIX for users with IBM PCs. See the insert card following page 328 for details. Listings are also available on BYTEnet. See page 4.* ■

BIBLIOGRAPHY

Backus, John. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs." *CACM*, vol. 21, 8, pages 613–641, August 1978.

Backus, John. "The Algebra of Functional Programs: Functional Level Reasoning, Linear Equations, and Extended Definitions." *Formalization of Programming Concepts.* New York: Springer-Verlag, 1981.

Baden, Scott. "Berkeley FP User's Manual, revision 4.1." *UNIX Programmers Manual,* July 27, 1983.

Darlington, J., et al. *Functional Programming and Its Applications.* New York: Cambridge University Press, 1982.

Harrison, Peter G., and Hessam Khoshnevisan. "Functional Programming Using FP." BYTE, August 1985.

Robison, Arch D. "A Functional Programming Interpreter." Thesis, University of Illinois, Urbana-Champaign, 1987.

Robison, Arch D. "IFP User's Manual." Professional Workstation Research Group Technical Report #7, University of Illinois, Urbana-Champaign, 1985.