Professional Workstation
Research Group Technical Report #7

Illinois FP User's Manual

Arch D. Robison
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

December 27, 1989

Illinois FP 0.5 Users Manual[1]

1. Overview

Functional Programming (FP)[1] is a radically new form of programming. FP programs have neither the control flow nor variables of Von-Neumann languages. Instead programs are directly constructed from smaller programs. As a result, FP offers a new style of programming with numerous advantages, including:

Modular Programming
Program Verification
Parallel Processing
Optimization

IFP (Illinois Functional Programming)[2] [3] is an interactive functional programming implementation for UNIX and MSDOS systems. The user may interactively create and execute functional programs. In addition to Backus' FP, IFP has the following features:

Hierarchical and Modular Function Organization
Block-Structured Syntax
Error Explanations
Graphics Display List Processor[2]

The interpreter is an order of magnitude more compact and faster than previous FP implementations.

2. Prerequisites

The rest of the manual assumes the reader has read Backus' original paper on FP. [1] Other references on FP[4] [5] may be of help. Additionally, parts of the manual assume the reader understands UNIX or MSDOS[3] file structure and paths.

2.1. Organization

IFP organizes functions in a tree structure analogous to UNIX/MSDOS files. In fact each function is a file. For UNIX systems, each user specifies the root (''IFP root'') of their function tree.

---

[1]Any resemblance to the real product is purely coincidental.

[2]Once upon a time it worked. The code has since then not been maintained. So it is not implemented in most versions.

Within IFP, paths specify a path relative to the IFP root. The IFP root is set by a UNIX environment variable. For MSDOS systems, the IFP root is identical to the current drive root. (see ‘‘Environment’’ below).

Each node on the tree is either a function definition (corresponding to a file), or a module (corresponding to a directory). A function may reference another function via a path.

To avoid having to write out the entire path for a function every time, IFP has a function identifier importation feature. Functions from other modules may be imported into a module. Once imported, a function may be referenced as though it were defined in the module.

2.2. Environment (UNIX)

Before invoking IFP, two environment variables should be set. The ‘‘EDITOR’’ variable should be set to the name of your favorite editor. The ‘‘IFProot’’ variable should be set to the absolute path of your ‘‘IFP root’’.[3] The ‘‘IFPprompt’’ is optional. If set, it changes the IFP prompt. The default prompt is ‘‘ifp> ’’. Normally these variables will be set by your .login file. Below is an example of the commands which would appear in your .login file.

```
setenv  EDITOR  =  ‘‘/usr/ucb/vi’’
setenv  IFProot  =  ‘‘/mnt/bonzo/fproot’’
setenv  IFPprompt  =  ‘‘ifp>  ’’
```

2.3. Environment (MSDOS)

Before invoking IFP, two environment variables should be set. The ‘‘EDITOR’’ and ‘‘IFPDIR’’ variables should be set to the names of your favorite editor and directory listers respectively. Normally these should be set by your autoexec.bat file, e.g.:

```
set  EDITOR=C:ED.EXE
set  IFPDIR=C:SD2.COM
```

Unlike the UNIX version, there is no IFProot variable. The root of the IFP file system is the root of the current drive.

---

[3] Use the actual path, not a symbolic link. When IFP starts up, it assumes that the current directory path is a prefix of the IFP root path.

3. Using IFP

3.1. Starting IFP

To start an IFP session, change your current working directory to a directory under your IFP root. Then type "ifp". Your current working directory becomes your IFP current working module. When IFP is ready, it will respond with the prompt ‘‘ifp> ’’. To end the IFP session, type control-D or enter the command ‘‘exit’’.

3.2. Creating and Editing Definitions

To edit an IFP definition, type the command:

vi[4] foo

where foo is the name of the function to be edited. The function may be one 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. To delete an IFP definition, type the command:

rm[5] foo

3.3. Applying Functions

To apply an FP function, type the command[6]:

show *object* : *function*

The interpreter evaluates the result of applying the *function* to the *object*. The result is then pretty-printed at the terminal. Below are some example inputs and outputs.

```
show <a b c> : reverse
<c b a>
show <1 2 3> : sum
6
```

[4]If your editor is not ‘‘vi’’ (as specified by the last element of your EDITOR path), replace ‘‘vi’’ with your editor’s name. For MS-DOS, the command is always ‘‘ed’’, no matter what the editor is called.

[5]For MS-DOS, the command is ‘‘del’’.

[6]Some earlier versions (before 0.4, e.g. the BYTE BIX release) require a semicolon after the *function*.

```
show <1 2 3> : EACH [id,id]* END | sum        (* sum of squares *)

14

show <1 2 3 4 5> : EACH iota END

<
        <1>
        <1,2>
        <1,2,3>
        <1,2,3,4>
        <1,2,3,4,5>
>

exit
```

3.4. Executing UNIX Commands

If a command is not recognized by the IFP interpreter, then it is passed on to the UNIX shell ‘‘sh’’. Commands such as ‘‘ls’’ and ‘‘more’’ work as expected. Commands which change environment do not work properly, as they change their environment (within ‘‘sh’’) but not your own. For example, the ‘‘cd’’ command does not work.

3.5. Executing MSDOS Commands

The only two MSDOS command that can be run from within the interpreter are ‘‘dir’’ and ‘‘del’’. Some systems seem to require ‘‘dir/’’. I don’t know why.

4. Language

IFP semantics are almost identical to Backus’ FP, though the syntax is quite different. The IFP language consists of objects, functions, and functional forms. The single operation is *apply* which maps a function and object into a new object.

4.1. Objects

Objects in FP are either atoms, sequences, or *bottom.* The latter is a special object which denotes an undefined value. Atoms are numbers, strings, or boolean values. Strings must be quoted when they look like another kind of atom or contain non-alphanumeric characters. Below is a table of some typical atoms:

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

Sequences are lists of zero or more objects surrounded by angle brackets. Sequences are written as:

$$<x_1,x_2, \cdots x_n>$$

Below is table of some typical sequences:

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

Either commas or spaces may be used to separate the elements of a sequence. The elements of the sequence may be any kind of object except ''?'', and do not have to be of the same type.

IFP sequences have the *bottom preserving*[1] property. Any sequence containing ''?'' is itself equal to ''?''.

4.2. Functions

Functions in FP always have a single argument and a single result. FP functions are analogous to UNIX programs which transform ''standard input'' into ''standard output'' without side effects.

The IFP interpreter distinguishs two kinds of functions: primitive functions and user-defined functions. Primitive functions are built into the FP interpreter; user-defined functions are created by the user. The only distinction between the two kinds of functions is that user-defined functions have definitions in terms of other IFP functions. All functions may be used in the same manner, neither primitive nor user-defined functions are privileged in any way.

IFP functions are arranged in a tree structure analogous to the way UNIX files are arranged. Each node of the tree is either a module (directory) or function (file). A function is referenced by its *pathname*, which is a sequence of node names separated by slashes. Pathnames follow the UNIX conventions. Absolute pathnames begin with a slash, which indicates that the path starts at the IFP root directory (as specified by the IFProot variable in your environment). Relative pathnames do not begin with a slash, which indicates that the path starts at the current directory. Within function definitions, the current directory is the parent node of the function. Pathnames may contain ‘‘..’’, which indicates moving up to the parent node.

For example, consider the node tree in Figure 1. The root node is ‘‘r’’. Below are some ways the function ‘‘b’’ can reference the other nodes. Note that the name of the root node is never explicitly used.

pathname	type
/sys/sum	absolute
/tmp/foo/p	absolute
foo/p	relative
../tmp/foo/p	relative
../sys/sum	relative

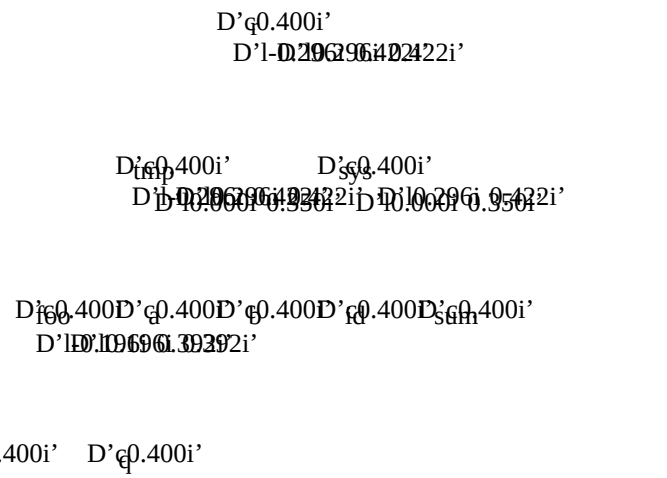

Figure 1

4.2.1. Primitive Functions

Primitive functions are built into the IFP interpreter. They have pathnames like any other function, except that there is no source code file for the function. The function descriptions are grouped into sections below. The pathname for the function's module is in parentheses at the top of each section.[7]

The following sets (types) are used in the definitions of functions:

A	atoms
B	boolean values
O	objects
R	real numbers
Z	integers
S	strings
$T*$	sequences with element type T
T^+	non-empty sequences with element type T
T^n	sequences of length n with element type T
$T^{[n,m]}$	sequences of length m with element type T^n
$[T_1,T_2]$	pair of types T_1 and T_2

A function returns ''?'' if the argument is not in its domain. The notation x_n denotes the nth element of a sequence X.

For example, the domain of the addition function is $[X,Y]\in[R,R]$. That is addition takes a pair of real numbers as its argument. We could also write this as $[X,Y]\in R^2$, since a pair is a sequence of length two.

The types may be pictured neatly with the Venn diagram in Figure 2:

4.2.1.1. Structural Functions (/sys)

Structural functions are assemble, reorganize, and select data. The primitive structural functions are listed below:

[7] NOTE: The author does not worship backward compatibility. Future versions of IFP may put primitive functions in different subdirectories.

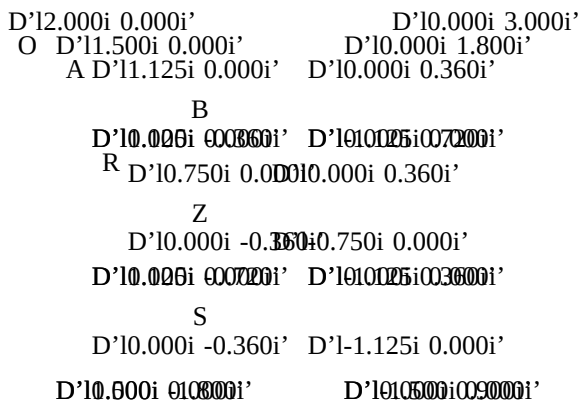

Figure 2

Name	Domain	Definition
apndl	$[X,Y]\in[O,O^n]$	$<X,y_1,y_2,\cdots y_n>$
apndr	$[X,Y]\in[O^m,O]$	$<x_1,x_2,\cdots x_m,Y>$
cat	$X\in O^{nm}$	catenate subsequences, e.g. <<a b> <x> <3 5>> → <a b x 3 5>
distl	$[X,Y]\in[O,O^n]$	$<<X,y_1><X,y_2>\cdots<X,y_n\gg$
distr	$[X,Y]\in[O^m,O]$	$<<x_1,Y><x_2,Y>\cdots<x_m,Y>>$
dropl	$[X,K]\in[O^n,0\le Z\le n]$	$<x_{K+1},x_{K+2},\cdots x_n>$
dropr	$[X,K]\in[O^n,0\le Z\le n]$	$<x_1,x_2,\cdots x_{n-k}>$
iota	$n\in Z\ge 0$	<1,2, $\cdots n$>
length	$X\in O^n$	n
pick	$[X,K]\in[O^n,0<Z\le n]$	x_K
repeat	$[X,K]\in[O,0\le Z]$	sequence $<X,X\ \cdots X>$ of length K
reverse	$X\in O^n$	$<x_n,x_{n-1},\cdots x_1>$
takel	$[X,K]\in[O^n,0\le Z\le n]$	$<x_1,x_2,\cdots x_K>$
taker	$[X,K]\in[O^n,0\le Z\le n]$	$<x_{n-K+1},x_{n-k+2},\cdots x_n>$

tl	$X \in O^{m>0}$	$<x_2, x_3, \cdots x_m>$
tlr	$X \in O^{m>0}$	$<x_1, x_2, \cdots x_{m-1}>$
trans	$X \in O^{[n,m]}$	transpose matrix, e.g. <<a 1> <b 2> <c 3>> -> <<a b c> <1 2 3>>

4.2.1.2. Arithmetic (/math/arith)

Most IFP arithmetic functions are found here. Below is a table of the existing functions. Some function's domain may be further restricted due to range limitations.

Name	Domain	Definition
+	$[X,Y]\in[R,R]$	$X+Y$
-	...	$X-Y$
*	...	$X\times Y$
%	$[X,Y]\in[R,R\neq 0]$	$X\div Y$
add1	$X\in R$	$X+1$
arcsin	$X\in R,\ -1\leq X\leq 1$	$arcsinX$
arccos	$X\in R,\ -1\leq X\leq 1$	$arccosX$
arctan	$X\in R$	$arctanX$
cos	$X\in R$	$\cos X$
div	$[X,Y]\in[R,R\neq 0]$	$\lfloor X\div Y\rfloor$
exp	$X\in R$	e^X
ln	$X\in R>0$	$\log_e X$
max	$[X,Y]\in[R,R]$	$\max(X,Y)$
min	$[X,Y]\in[R,R]$	$\min(X,Y)$
minus	$X\in R$	$-X$
mod	$[X,Y]\in[R,R]$	$X-Y\lfloor X\div Y\rfloor$ if $Y\neq 0$, 0 otherwise
power	$[X,Y]\in[R\geq 0,R]$	X^Y
sin	$X\in R$	$\sin X$
sqrt	$X\in R>0$	$\sqrt{X}$
sub1	$X\in R$	$X-1$
sum	$X\in R*$	$\sum_i X_i$
tan	$X\in R$	$\tan X$

4.2.1.3. Logic (/math/logic)

Most IFP primitive functions returning boolean values are found here. Below is a table of the existing functions:

Name	Domain	Definition
=	$[X,Y]\in[O,O]$	$X=Y$
~=	...	$X\neq Y$
<	$[X,Y]\in[R,R]\bigcup[S,S]$	$X<Y$
<=	...	$X\leq Y$
>=	...	$X\geq Y$
>	...	$X>Y$
~	$X\in B$	$\sim X$
and	$[X,Y]\in[B,B]$	$X\wedge Y$
all	$X\in B*$	$\bigwedge_k x_k$
any	$X\in B*$	$\bigvee_k x_k$
atom	$X\in O$	$X\in A$
boolean	$X\in O$	$X\in B$
false	$X\in O$	$X=\#f$
imply	$[X,Y]\in[B,B]$	$\sim X\vee Y$
longer	$[X,Y]\in[O^m,O^n]$	$m>n$
member	$[X,Y]\in[O*,O]$	$Y\in X$
numeric	$X\in O$	$X\in R$
null	$X\in O*$	$X=<>$
odd	$X\in Z$	$X \mod 2 = 1$
or	$[X,Y]\in[B,B]$	$X\vee Y$
pair	$X\in O$	$X\in[O,O]$
shorter	$[X,Y]\in[O^m,O^n]$	$m<n$
xor	$[X,Y]\in[B,B]$	$X\neq Y$

String inequalities are defined from the lexigraphical (dictionary) ordering.

4.2.1.4. String Functions (/sys)

The string functions are:

Name	Domain	Definition
explode	$X \in S$	sequence of characters in X
implode	$X \in S*$	string made by catenating strings in X
patom	$X \in A$	string representation of X, e.g. 123 : patom $\rightarrow$ "123"

4.2.1.5. Miscellaneous Functions (/sys)

The miscellaneous functions are listed below. Each function description is preceded by a title line of the form:

---

function domain definition

---

apply $[X,F] \in [O,S*]$ apply F to X

F is a sequence of strings representing a pathname to a defined function. The result is the function referenced by F applied to X. Example:

<<3 4> <math arith "+">> : apply $\rightarrow$ 7

F may also be an anonymous function. Anonymous functions are objects that are enclosed by parentheses. For instance, the previous example could be written as

<<3 4> (+)> : apply $\rightarrow$ 7

Functions built from functional forms may also be objects, for example:

<<<1 2 3> <4 5 6>> (trans|EACH * END|sum) $\rightarrow$ 32

Function objects are considered to be atomic. Functions that act on sequences will not behave properly when applied to a function object. The ‘‘apply’’ function combined with function objects lets us define our own functional forms. For example, we can define a functional form Twice which applies a function twice as:

DEF Twice AS [apply,2]|apply;

Then we can write:

3 : [id,([id,id]*)] ! Twice → 81

---

assoc $[X,Y] \in [(O^+)^*,O]$ associative lookup

X is an association sequence, which is a sequence of non-empty subsequences. The first element of each subsequence is the *key* of the subsequence. The result of assoc is the first subsequence of X with a key equal to Y. If no matching key is found, f is returned. The key may be any type of object. Examples:

<<<a b c> <w x y z> <i j>> w> → <w x y z>
<<<a b c> <w x y z> <i j>> U> → f

---

def $X \in S^+$ definition

The definition function returns the object representation of its argument. The representation of a function is a sequence of strings denoting its absolute pathname. The representation of a functional form is a sequence. The first element of the sequence is a pathname to the functional form. The remaining elements of the sequence are parameters of the functional form. Suppose, for example, we define the inner product function:

DEF Inner AS trans ! EACH * END ! INSERT + END

and ''Inner'' is defined with a module with pathname ''/math/linear''. Then ''<math linear Inner> : def'' will result in:

```
<
        <sys compose>
        <sys trans>
        <<sys each> <math arith *>>
        <<sys insertr> <math arith +>>
>
```

Currently, the representations of functional forms are:

$\#c$	<<sys constant> $\#c$ >
$\#?$	<<sys constant>>
n	<<sys select> n >
n r	<<sys select> $-n$ >
$f_1 \mid f_2 \mid \cdots f_n$	<<sys compose>, $f_1, f_2, \cdots f_n$
$[f_1, f_2, \cdots f_n]$	<<sys construct>, $f_1, f_2, \cdots f_n$
$\hat{}c$	<<sys fetch> c >
EACH f END	<<sys each> f >
FILTER p END	<<sys filter> p >
INSERT f END	<<sys insertr> f >
IF p THEN g ELSE h END	<<sys if> p g h >
WHILE p DO f END	<<sys while> p f >

ELSIF clauses are always expanded into equivalent nested IF-THEN-ELSE constructions. Note the special case for $\#?$, the representation <<sys constant> ?> would be useless due to the bottom-preserving property.

---

id $X \in O$ identity

The identity function returns its argument. It is useful as a place holder in functional forms. For example, the ''square'' function can be written as:

DEF Square AS [id,id] | *;

4.2.2. User Defined Functions

The user may define functions by creating definition files (source code). The definition in the file is of the form:

DEF *foo* AS *bar*;

where *foo* is the name of the function and *bar* is the definition. The name of the file must also be *foo*. The definition may be any IFP function. For example, you can define the square function as:

DEF Square AS [/sys/id,/sys/id] | /math/arith/*;

Writing out the entire pathname of functions is not necessary. If the function is defined in the same subdirectory, then just its name is necessary. If the function is defined in another subdirectory, then you can ''import'' it. An imported function can be referenced as though it were defined in the directory into which it is imported. To import functions into a subdirectory, you create an ''import file'' with the name %IMPORT with the editor. The form of the %IMPORT file is:

FROM $directory_1$ IMPORT $f_1, f_2, \ldots f_n$;

FROM *directory*$_2$ IMPORT g_1,g_2, ... g_m;
...

The *directory* is a pathname to a directory. For example, typical import file might be:

```
FROM /sys IMPORT apndr,distl,id,iota,takel;
FROM /math/arith IMPORT +,-,*,%;
```

Since the function ''id'' is imported, the square function can be defined as:

DEF Square AS [id,id] | *;

4.2.3. Functional Variables

Functional variables[6] are locally defined functions with special scope rules. A functional variable definition is written:

{*lhs* := *function*}

where *lhs* (left hand side) is either a function name or construction of *lhs*'s. All function names in the *lhs* become functional variables within their scope. The scope is *boundary structured* as opposed to block structured, which means that the variables may be seen only through certain boundaries. The scope rules can be defined by the following transformations:

{V := h} v → h | V_v^{-1}

{V := h} [f_1,f_2,...] → [{V := h} f_1, {V := h} f_2, ...]

```
{V := h} IF p THEN x          IF {V := h} p THEN {V := h} x
           ELSE y        →       ELSE {V := h} y
           END                   END
```

where → indicates that the left side may be simplified to the right side. ''V'' denotes a left-hand side containing the functional variable ''v''. V_v^{-1} is the inversion function of ''V'' for ''v''. For example, if $V = [a,b,c]$, then $V_c^{-1} = 3$. Variables must be defined before use. Note that the vertical bar of composition cuts off the scope, e.g. in

{[*x*,*y*] := id} *g* | *h*

the function *g* can ''see'' *x* and *y*, but *h* can not.

An example of a definition with functional variables appears below:

```
(*
  * InsertSort
  *
```

```
 * This function sorts a sequence of numbers or strings into ascending order
 * using insertion sort.
 *
 * Examples:
 *
 *         <3 1 4 1 5 9 2> : InsertSort == <1 1 2 3 4 5 9>
 *
 *         <all work and no play> : InsertSort == <all and no play work>
 *
 * The sequence may not mix strings and numbers.
 *)
DEF InsertSort AS
    IF null THEN id        (* Check for trivial case *)
    ELSE
        [tl,[1]] ! apndr !
        INSERT
            {[Element,Seq] := id}
            {[Left,Right] := [Seq, distl ! FILTER > END ! length] ! [takel,dropl]}
            [Left,[Element],Right] ! cat
        END
    END;
```

In this example, *Element*, *Seq*, *Left*, and *Right* are functional variables.

4.3. Functional Forms

Functional forms combine functions and objects to create new functions.

4.3.1. Constant

Constant functions always return the same result when applied to any value which is not ‘‘?’’. Constant functions are written as:

$$\#c$$

where c is the constant value to be returned. A constant function applied to ‘‘?’’ results in ‘‘?’’. Note that the function ‘‘#?’’ always returns ‘?’. Examples:

```
923 : #<cat in hat> → <cat in hat>
<a b c d e f> : #427 → 427
? : #<q w er t y> → ?
5 : #? → ?
```

4.3.2. Selection

Selector functions return the nth element of a sequence and are written as n, where n is a positive integer. Note the distinction between # 5, which returns the value 5, and 5, which returns the fifth

element of its argument. There are also a corresponding set of select-from-right functions, written as nr. These select the nth element of a sequence, counting from the right. All selectors return ‘‘?’’ if the argument has no nth element or is not a sequence. Below are some examples of applying selector functions:

```
<a b c d e> : 1 → a
<a b c d e> : 2 → b
<apple banana cherry> : 1r → cherry
<apple banana cherry> : 4 → ?
hello : 1 → ?
```

4.3.3. Composition

The function composition of two functions is written as:

$$f \mid g$$

Applying the result function is the same as applying f and then g. E.g.: Function composition is defined by the equality:

$$x : (f \mid g) \equiv (x : f) : g$$

Since function composition is associative, the composition of more than two functions does not require parentheses. The composition of $f_1, f_2, \cdots f_n$ is written:

$$f_1 \mid f_2 \mid \cdots f_n$$

Composition syntax is identical to UNIX’s pipe notation for a reason: function composition is isomorphic to a pipe between processes without side effects.

4.3.4. Construction

The construction of functions is written as bracketed list of the functions. For example, the construction of functions f_i is written:

$$[f_1, f_2, \cdots f_n]$$

Function construction is defined by the equality:

$$x : [f_1, f_2, \cdots f_n] \equiv \langle x:f_1, x:f_2, \ldots x:f_n \rangle$$

4.3.5. Apply to Each

The EACH functional form applies a function to each element of a sequence. It is written as

EACH f END

It is defined by the equality:

$$\langle x_1, x_2, \cdots x_n \rangle : \text{EACH } f \text{ END} \equiv \langle x_1{:}f, x_2{:}f, \cdots x_n{:}f \rangle$$

4.3.6. If-Then-Else

The IF functional form allows conditional function application. It is written as

IF p THEN g ELSE h END

and is defined by the equality:

$$x : \text{IF } p \text{ THEN } g \text{ ELSE } h \text{ END} \rightarrow \begin{cases} g(x) & \text{if } p(x)=\text{t} \\ h(x) & \text{if } p(x)=\text{f} \\ ? & \text{otherwise} \end{cases}$$

The level of nesting of conditional forms may be reduced by using ELSIF clauses:

```
IF p1 THEN g1
ELSE
    IF p2 THEN g2
    ELSE
        IF p3 THEN g3
        ELSE h
        END
    END
END
```

4.3.7. Filter

The FILTER functional form filters through elements of a sequence satisfying a predicate. It is written as:

FILTER p END

where p is the predicate. It is defined by the functional equality:

```
EACH
    IF p THEN [id] ELSE [] END
END | cat
```

For example, if you wish to find all numeric elements in a sequence, you could write:

FILTER numeric END

The FILTER functional form is an IFP extension to Backus' FP.

4.3.8. Right Insert

The INSERT functional form is defined by the recursion:

INSERT *f* END ≡ IF tlnull THEN 1 ELSE [1,tl | INSERT *f* END] | *f* END

Typically it is used for crunching a sequence down. For example,

INSERT + END

returns the sum of a sequence.

Unlike Backus' FP, functions formed with INSERT are always undefined for empty sequences. The reason is that it is impractical for the interpreter to know the identity element of user-defined functions. The number of cases where the interpreter could know the identity element are so few that you might as well define special functions for those cases, e.g:

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

Alternatively, you can append the identity element to the end of the sequence before inserting, e.g.:

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

Currently there is no ''left insert'' form. The left insertion of *f* can be written as:

reverse | INSERT reverseIf END

4.3.9. While

The WHILE functional form allows indefinite composition. It is written as:

WHILE *p* DO *f* END;

and is defined by the recursive functional equality:

```
WHILE p DO f END ≡ IF p THEN
                     f | WHILE p DO f END
                   ELSE id
                   END
```

That is the *WHILE* PFO applies the fewest *f*'s such that *x:f:f:f...:p* is true.

4.3.10. Fetch[8]

The fetch functional form allows easy access to association sequences (see function /sys/assoc in section 4.2.1.5 for a description of association sequences.) A fetch is written as ^*c*, where *c* is an object. The fetch form is defined by the functional equality:

```
^c ≡ IF EACH pair END | all THEN [id,#c]assoc2
     ELSE #?
     END;
```

Note that the input is restricted to a sequence of pairs. For example,

```
<<a 1> <b 2> <c 3>> : ^b -> 2
```

4.4. Comments

Comments are delimited by matching pairs of ‘‘(*’’ and ‘‘*)’’. Comments may be inserted anywhere not adjacent to a token. For example:

```
DEF foo AS bar; (* This is a comment.  DEF foo AS bar is not a comment *)
```

4.5. Syntax Summary

Below is an EBNF grammar for IFP:

[8]The fetch functional form is being deimplemented. It may or may not exist on your IFP interpreter.

Def →	'DEF String 'AS' Comp ';'
Comp →	Simple { '|' Simple }
Simple →	Conditional | Constant | Construction | Each | Filter | Insert | Path | While | Fetch | Debug | FunVar
Conditional →	'IF' Comp 'THEN' Comp { 'ELSIF' Comp 'THEN' Comp } 'ELSE' Comp 'END'
While →	'WHILE' Comp 'DO' Comp 'end'
Insert →	'INSERT' Comp 'END'
Each →	'EACH' Comp 'END'
Filter →	'FILTER' Comp 'END'
Fetch →	'^' String
Constant →	'#' Object
Debug →	'@' Object
FunVar →	'{' Lhs ':=' Comp '}'
Lhs →	String | '[' [Lhs { ',' Lhs }] ']'
Construction →	'[' [Comp {',' Comp}] ']'
Path →	['/'] String {'/' String}
Object →	Bottom | Atom | '<' [Atom {','Atom}] '>'
Bottom →	'?'
Atom →	Number | String | Boolean
Boolean →	't' | 'f'

Strings may be in single or double quotes. The strings ''t'' and ''f'' must be quoted to distinguish them from boolean atoms. Strings of digits must also be quoted to distinguish them from numeric atoms.

5. IFP Graphics (optional)[9]

There are no graphics primitives in IFP itself, rather IFP is used to calculate a display list. A display-list processor then draws the picture specified by the display list. To send an IFP result to the display-list processor instead of the screen, use the command:

graph *object* : *function*

instead of the ''show'' command.

5.1. Coordinate System

Points on the graphics display are referenced by <X,Y> coordinate pairs. <0,0> is the lower left corner, <1,1> is the upper left corner. Currently there is no clipping. Lines outside the display are wrap around in weird and not-so-wonderful ways.

[9]This option is currently not implemented. If this section inspires you, get the source code and fix it (see G_*.c).

5.2. Display List Structure

Below is an EBNF grammar for the display list.

```
display-list →   < {display-list} > | polyline | color | transform | text
polyline →       <'line' { < x y > } >
color →          <'color' color-index display-list >
text →           <'text' atom size ['center']>
transform →      <'trans' t-matrix display-list >
t-matrix →       <<Txx  Txy  Tx0> <Tyx  Tyy  Ty0≫
```

5.2.1. Polyline

The polyline structure specifies a sequence of points. It is of the form:

$$\text{<line } \langle x_1,y_1\rangle \langle x_2,y_2\rangle \cdots \langle x_n,y_n\rangle\text{>>}$$

where each $\langle x_i,y_i\rangle$ is a point coordinate. Adjacent points in the sequence are connected with line segments. For example, the sequence:

<line <0 0> <0 5> <1 5> <1 0> <0 0>>

draws a box 1 unit wide and 5 units high.

5.2.2. Color

The color structure draws the display-list in the color specified by the color index (0..15). The color applies to all parts of the subordinate display-list which are not subordinate to a color structure within. In other words, a structure is colored by its inner-most bounding ‘‘color’’ structure.

5.2.3. Transform

The transform structure draws the display-list as transformed by the t-matrix. Transforms may be nested. Transforming a display-list converts coordinates $\langle x,y\rangle$ into coordinates $\langle x',y'\rangle$ via the formula:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} T_{xx} & T_{xy} & T_{x0} \\ T_{yx} & T_{yy} & T_{y0} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

5.2.4. Text

The text structure draws the atom with the lower-left corner at (0,0). Each character is drawn in a *size* by *size* box (including spacing) The lower-left corner of the text is at <0 0> by default. Including the *center* option centers the text on <0 0>.

6. Debugging

Currently, IFP has simple program trace mechanism.[10] To trace a function, respond to the IFP prompt with:

trace on $f_1, f_2, \cdots f_n$;

where the f's are functions 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, respond to the IFP prompt with:

trace off $f_1, f_2, \cdots f_n$;

When tracing, the interpreter ellipses are used to abbreviate functions. You can set the depth at which ellipses occur with the *depth* command:

depth *n*

where *n* is a non-negative integer. The default depth is two.

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

@message

The function formed always returns its argument unchanged, and it prints ‘‘message: ’’ followed by its argument. For example,

<1 3 5> : EACH @banana END

will print the messages:

banana: 1
banana: 3
banana: 5

This tracing functional form is for debugging only, since it creates a side effect (the message!), it is not

[10]This will be replaced by a much better trace mechanism as soon as the author as time to design one.

truly functional.

7. Differences between IFP and Backus' FP

7.1. Domain

Backus' FP has two types of atom, the string and empty sequence. IFP atoms do not include the empty sequence. IFP include numbers and truth values as atoms distinct from strings.

7.2. Functions

There are many new primitives. See the section on ''Primitives'' elsewhere. Of particular interest are the functions *cat, dropl, takel, taker,* and *iota.*

7.3. Functional Forms

Backus' FP defines the INSERT form for empty sequences as returning u_f, the right identity element of f. IFP does not define INSERT for empty sequences. If necessary, use one of the following functions:

IF null THEN u_f ELSE INSERT f END END

[id,u_f] | apndr | INSERT f END

IFP has a new functional form, FILTER, which filters a sequence according to a predicate. It is written as:

FILTER p END

When applied to a sequence X, it returns a sequence of x_i for which p is true.

7.4. Syntax

The IFP syntax is designed to facilitate indentation and comments. All functional forms bracket their parameters, so no parentheses are necessary. The differences between Backus' FP and IFP syntax are shown below.

Backus	IFP
CBA	A ⌈ B ⌈ C
[F,G,H]	[F,G,H]
p→f;g	IF p THEN f ELSE g END
p→f; q→g; h	IF p THEN f ELSIF q THEN g ELSE h
αf	EACH f END
/f	INSERT f END
(while p f)	WHILE p DO f END
(bu f x)	[id,#x] ⌈ *f*
f̄	#*f*
Def f ≡ x	DEF f AS x;
ϕ	<>
⌊	?

Also, parentheses are neither necessary nor allowed in IFP function definitions.

Finally, IFP functions are arranged in a tree structure and referenced by pathnames. All pathnames are expanded into absolute pathnames when read by the interpreter, so the meaning of a pathname is static. This is an important point, otherwise IFP would have significantly different (and more complex) semantics than Backus' FP.

8. Functional Programming Techniques

8.1. Functional Programming Identities

Functional programs can be improved by algebraic substitutions. Below is a table of some IFP identities. The notation ''$f \equiv g$'' indicates f and g are equal and have the same domain. The notation ''$f \subset g$'' indicates that g is equal to f over f's domain, but may have a larger domain than f.

EACH f END ⌈ EACH g END	≡	EACH f ⌈ g END
[#*c* ,id] ⌈ distl	≡	EACH [#*c* ,id] END
[takel,dropl] ⌈ cat	⊂	1
apndl ⌈ length	⊂	2 ⌈ length ⌈ add1
apndr ⌈ length	⊂	1 ⌈ length ⌈ add1
iota ⌈ length	⊂	id
reverse ⌈ length	≡	length
tl ⌈ length	⊂	length ⌈ sub1
tlr ⌈ length	⊂	length ⌈ sub1
apndl ⌈ reverse	≡	[2 ⌈ reverse,1] ⌈ apndr
apndr ⌈ reverse	≡	[2,1 ⌈ reverse] ⌈ apndl
reverse ⌈ reverse	⊂	id
trans ⌈ trans	⊂	id

8.2. Common Subfunctions

The interpreter is not very smart about common subfunctions, it reevaluates a function every time its encountered. Use functional variables (section 4.2.3) to factor out common subfunctions.

8.3. State Machines

You can simulate a state machine in IFP by defining the state transition function D, which maps an input and state into another state:

$$[\text{input,state}] \; : \; D \; \rightarrow \; \text{state}$$

You then run the state machine with the function

$$\text{apndl} \; | \; \text{reverse} \; | \; \text{INSERT} \; D \; \text{END}$$

which yields the final state when applied to the initial conditions <initial-state,tape>.

8.4. Tail Recursion

Regrettably, the IFP interpreter currently does not recognize tail recursions as iterations. Thus near-infinite recursions will cause a stack overflow. If this is a problem, rewrite the function with the WHILE functional form to remove the tail recursion.

For example, consider the tail recursive function:

```
DEF f AS
      IF p THEN g
      ELSE h | f              (* tail recursion *)
      END;
```

We can rewrite is as:

```
DEF f AS
      WHILE p~ DO h END | g;
```

9. Installation Notes

9.1. Machine Dependence

The IFP interpreter is machine independent, as long as your machine has 32-bit two's complement integers and IEEE floating point. If not, you should take a look at the *struct.h* and *F_arith.c* source files. The *struct.h* file defines all the principle types and limit definitions (e.g. MaxInt,

MAXFLOAT). The *F_arith.c* contains the arithmetic functions. See the comments in the code for details.

9.2. Compiling Options

Look in the Makefile and "struct.h" for possible compiling options. Not all options are available in all releases. Normally, the release version comes ready to compile on UNIX boxes. For MSDOS, you will have to modify the Makefile and change the OPSYS variable in ‘‘struct.h’’. The graphics interface is extremely machine dependent, though should not be difficult to modify it for other machines.

Table of Contents

1. Overview 1
2. Prerequisites 1
2.1. Organization 1
2.2. Environment (UNIX) 2
2.3. Environment (MSDOS) 2
3. Using IFP 3
3.1. Starting IFP 3
3.2. Creating and Editing Definitions 3
3.3. Applying Functions 3
3.4. Executing UNIX Commands 4
3.5. Executing MSDOS Commands 4
4. Language 4
4.1. Objects 4
4.2. Functions 5
4.2.1. Primitive Functions 7
4.2.1.1. Structural Functions (/sys) 7
4.2.1.2. Arithmetic (/math/arith) 9
4.2.1.3. Logic (/math/logic) 10
4.2.1.4. String Functions (/sys) 12
4.2.1.5. Miscellaneous Functions (/sys) 12
4.2.2. User Defined Functions 14
4.2.3. Functional Variables 15
4.3. Functional Forms 16
4.3.1. Constant 16
4.3.2. Selection 16
4.3.3. Composition 17
4.3.4. Construction 17
4.3.5. Apply to Each 18
4.3.6. If-Then-Else 18
4.3.7. Filter 18
4.3.8. Right Insert 19
4.3.9. While 19
4.3.10. Fetch[8] 20
4.4. Comments 20
4.5. Syntax Summary 20
5. IFP Graphics (optional)[9] 21
5.1. Coordinate System 21
5.2. Display List Structure 22
5.2.1. Polyline 22
5.2.2. Color 22
5.2.3. Transform 22
5.2.4. Text 23
6. Debugging 23

7. Differences between IFP and Backus' FP 24
7.1. Domain 24
7.2. Functions 24
7.3. Functional Forms 24
7.4. Syntax 24
8. Functional Programming Techniques 25
8.1. Functional Programming Identities 25
8.2. Common Subfunctions 26
8.3. State Machines 26
8.4. Tail Recursion 26
9. Installation Notes 26
9.1. Machine Dependence 26
9.2. Compiling Options 27

2. Backus, John. *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*. **CACM** (August 1978) vol. 21,8, pp. 613-641.

32. Robison, Arch D. *A Functional Programming Interpreter*. **THESIS** (January 1987).

31. ----. *Illinois Functional Programming: A Tutorial*. **BYTE** (February 1987) vol. 12,2, pp. 115-125.

5. Baden, Scott. *Berkeley FP User's Manual, Rev. 4.1*. **UNIX Programmers Manual** (July 27,1983).

10. Darlington, J., J. V. Guttag, P. Henderson, J. H. Morris, J. E. Stoy, G. J. Sussman, P. C. Treleaven, D. A. Turner, J. H. Williams and D. S. Wise. *Functional Programming and its Applications*. (1982).

4. Backus, John. *The Algebra of Functional Programs: Functional Level Reasoning, Linear Equations, and Extended Definitions*. **Formalization of Programming Concepts** (1981).