Berkeley FP User's Manual, Rev. 4.1

by

Scott Baden

ABSTRACT

This manual describes the Berkeley implementation of Backus' Functional Programming Language, FP. Since this implementation differs from Backus' original description of the language, those familiar with the literature will need to read about the system commands and the local modifications.

August 12, 1987

1. Background

FP stands for a *Functional Programming* language. Functional programs deal with *functions* instead of *values*. There is no explicit representation of state, there are no assignment statments, and hence, no variables. Owing to the lack of state, FP functions are free from side-effects; so we say the FP is *applicative*.

All functions take one argument and they are evaluated using the single FP operation, *application* (the colon ':' is the apply operator). For example, we read $+ : <3\ 4>$ as ''apply the function '+' to its argument <3 4>''.

Functional programs express a functional-level combination of their components instead of describing state changes using value-oriented expressions. For example, we write the function returning the *sin* of the *cos* of its input, i.e., $sin(cos(x))$, as: sin @ cos. This is a *functional expression*, consisting of the single *combining form* called *compose* ('@' is the compose operator) and its *functional arguments sin* and *cos*.

All combining forms take functions as arguments and return functions as results; functions may either be applied, e.g., sin@ cos : 3, or used as a functional argument in another functional expression, e.g., *tan @ sin @ cos*.

As we have seen, FP's combining forms allow us to express control abstractions without the use of variables. The *apply to all* functional form (&) is another case in point. The function '& exp' exponentiates all the elements of its argument:

$$\&exp : <1.0\ 2.0> \equiv <2.718\ 7.389> \tag{1.1}$$

In (1.1) there are no induction variables, nor a loop bounds specification. Moreover, the code is useful for any size argument, so long as the sub-elements of its argument conform to the domain of the *exp* function.

We must change our view of the programming process to adapt to the functional style. Instead of writing down a set of steps that manipulate and assign values, we compose functional expressions using the higher-level functional forms. For example, the function that adds a scalar to all elements of a vector will be written in two steps. First, the function that distributes the scalar amongst each element of the vector:

$$distl : <3\ <4\ 6>> \equiv <<3\ 4>\ <3\ 6>> \tag{1.2}$$

Next, the function that adds the pairs of elements that make up a sequence:

$$\&+ : <<3\ 4>\ <3\ 6>> \equiv <7\ 9> \tag{1.3}$$

In a value-oriented programming language the computation would be expressed as:

$$\&+ : distl : <3\ <4\ 6>>, \tag{1.4}$$

which means to apply 'distl' to the input and then to apply '+' to every element of the result. In FP we write (1.4) as:

$$\&+\ @\ distl : <3\ <4\ 6>>. \tag{1.5}$$

The functional expression of (1.5) replaces the two step value expression of (1.4).

Often, functional expressions are built from the inside out, as in LISP. In the next example we derive a function that scales then shifts a vector, i.e., for scalars a, b and a vector $\vec{v}$, compute $a + b\vec{v}$. This FP function will have three arguments, namely a, b and $\vec{v}$. Of course, FP does not use formal parameter names, so they will be designated by the function symbols 1, 2, 3. The first code segment scales $\vec{v}$ by b (defintions are delimited with curly braces '{}'):

$$\{scaleVec\ \&*\ @\ distl\ @\ [2,3]\} \tag{1.6}$$

The code segment in (1.5) shifts the vector. The completed function is:

{changeVec &+ @ distl @ [1 , scaleVec]} **(1.7)**

In the derivation of the program we wrote from right to left, first doing the *distl*'s and then composing with the *apply-to-all* functional form. Using an imperative language, such as Pascal, we would write the program from the outside in, writing the loop before inserting the arithmetic operators.

Although FP encourages a recursive programming style, it provides combining forms to avoid explicit recursion. For example, the right insert combining form (!) can be used to write a function that adds up a list of numbers:

!+ : <1 2 3> ≡ 6 (1.8)

The equivalent, recursive function is much longer:

{addNumbers (null -> %0 ; + @ [1, addNumbers @ tl])} (1.9)

The generality of the combining forms encourages hierarchical program development. Unlike APL, which restricts the use of combining forms to certain builtin functions, FP allows combining forms to take any functional expression as an argument.

2. System Description

2.1. Objects

The set of objects Ω consists of the atoms and sequences $<x_1, x_2, \ldots, x_k>$ (where the $x_i \in \Omega$). (Lisp users should note the similarity to the list structure syntax, just replace the parenthesis by angle brackets and commas by blanks. There are no 'quoted' objects, i.e., 'abc). The atoms uniquely determine the set of valid objects and consist of the numbers (of the type found in FRANZ LISP [Fod80]), quoted ascii strings ("abcd"), and unquoted alphanumeric strings (abc3). There are three predefined atoms, **T** and **F**, that correspond to the logical values 'true' and 'false', and the undefined atom **?**, *bottom*. *Bottom* denotes the value returned as the result of an undefined operation, e.g., division by zero. The empty sequence, <> is also an atom. The following are examples of valid FP objects:

?	1.47	3888888888888
ab	"*CD*"	$<1,<2,3>>$
<>	**T**	$<a,<>>$

There is one restriction on object construction: no object may contain the undefined atom, such an object is itself undefined, e.g., $<1,?> \equiv ?$. This property is the so-called ''bottom preserving property'' [Ba78].

2.2. Application

This is the single FP operation and is designated by the colon (":"). For a function σ and an object x, $\sigma{:}x$ is an application and its meaning is the object that results from applying σ to x (i.e., evaluating $\sigma(x)$). We say that σ is the *operator* and that x is the *operand*. The following are examples of applications:

$+{:}<7,8>$	$\equiv$	15	$\mathbf{tl}{:}<1,2,3>$	$\equiv$	$<2,3>$
$\mathbf{1}{:}<a,b,c,d>$	$\equiv$	a	$\mathbf{2}{:}<a,b,c,d>$	$\equiv$	b

2.3. Functions

All functions (F) map objects into objects, moreover, they are *strict*:

$$\sigma : ? \equiv ?, \ \forall \ \sigma \in \mathbf{F} \tag{2.1}$$

To formally characterize the primitive functions, we use a modification of McCarthy's conditional expressions [Mc60]:

$$p_1 \rightarrow e_1 ; \cdots ; p_n \rightarrow e_n \ ; e_{n+1} \tag{2.2}$$

This statement is interpreted as follows: return function e_1 if the predicate 'p_1' is true , . . . , e_n if 'p_n' is true. If none of the predicates are satisfied then default to e_{n+1}. It is assumed that $x, x_i, y, y_i, z_i \in \Omega$.

2.3.1. Structural

Selector Functions

For a nonzero integer μ,

$\mu : x \equiv$
$x = < x_1, x_2, \ldots, x_k > \chi\, 0 < \mu \le k \rightarrow x_\mu;$
$x = < x_1, x_2, \ldots, x_k > \chi\, -k \le \mu < 0 \rightarrow x_{k+\mu+1};\ ?$

pick : $< n, x > \equiv$
$x = < x_1, x_2, \ldots, x_k > \chi\, 0 < n \le k \rightarrow x_n;$
$x = < x_1, x_2, \ldots, x_k > \chi\, -k \le n < 0 \rightarrow x_{k+n+1};\ ?$

The user should note that the function symbols **1**,**2**,**3**,... are to be distinguished from the atoms 1,2,3,....

last : $x \equiv$
$x = <> \rightarrow <>;$
$x = < x_1, x_2, \ldots, x_k > \chi\, k \ge 1 \rightarrow x_k;\ ?$

first : $x \equiv$
$x = <> \rightarrow <>;$
$x = < x_1, x_2, \ldots, x_k > \chi\, k \ge 1 \rightarrow x_1;\ ?$

Tail Functions

tl : $x \equiv$
$x = < x_1 > \rightarrow <>;$
$x = < x_1, x_2, \ldots, x_k > \chi\, k \ge 2 \rightarrow < x_2, \ldots, x_k >;\ ?$

tlr : $x \equiv$
$x = < x_1 > \rightarrow <>;$
$x = < x_1, x_2, \ldots, x_k > \chi\, k \ge 2 \rightarrow < x_1, \ldots, x_{k-1} >;\ ?$

Note: There is also a function **front** that is equivalent to **tlr**.

Distribute from left and right

distl : $x \equiv$
$x = < y, <> > \rightarrow <>;$
$x = < y, < z_1, z_2, \ldots, z_k > > \rightarrow << y, z_1 >, \ldots, < y, z_k >>;\ ?$

distr : $x \equiv$

$x = << >, y > \rightarrow < >;$

$x = << y_1, y_2, \ldots, y_k >, z > \rightarrow << y_1, z >, \ldots, < y_k, z >>;$ *?*

Identity

id : $x \equiv x$

out : $x \equiv x$

Out is similar to **id**. Like **id** it returns its argument as the result, unlike **id** it prints its result on *stdout* – It is the only function with a side effect. *Out* is intended to be used for debugging only.

Append left and right

apndl : $x \equiv$

$x = < y, < >> \rightarrow < y >;$

$x = < y, < z_1, z_2, \ldots, z_k >> \rightarrow < y, z_1, z_2, \ldots, z_k >;$ *?*

apndr : $x \equiv$

$x = << >, z > \rightarrow < z >;$

$x = << y_1, y_2, \ldots, y_k >, z > \rightarrow < y_1, y_2, \ldots, y_k, z >;$ *?*

Transpose

trans : $x \equiv$

$x = << >, \ldots, < >> \rightarrow < >;$

$x = < x_1, x_2, \ldots, x_k > \rightarrow < y_1, \ldots, y_m >;$ *?*

where $x_i = < x_{i1}, \ldots, x_{im} > \chi\, y_j = < x_{1j}, \ldots, x_{kj} >,$

$1 \le i \le k\ , 1 \le j \le m.$

reverse : $x \equiv$

$x = < > \rightarrow;$

$x = < x_1, x_2, \ldots, x_k > \rightarrow < x_k, \ldots, x_1 >;$ *?*

Rotate Left and Right

rotl : $x \equiv$

$x = < > \rightarrow < >; x = < x_1 > \rightarrow < x_1 >;$

$x = < x_1, x_2, \ldots, x_k > \chi\, k \ge 2 \rightarrow < x_2, \ldots, x_k, x_1 >;$ *?*

rotr :$x \equiv$

$x = < > \rightarrow < >; x = < x_1 > \rightarrow < x_1 >;$

$x = < x_1, x_2, \ldots, x_k > \chi\, k \ge 2 \rightarrow < x_k, x_1, \ldots, x_{k-2}, x_{k-1} >;$ *?*

concat : $x \equiv$

$x = << x_{11}, \ldots, x_{1k} >, < x_{21}, \ldots, x_{2n} >, ..., < x_{m1}, \ldots, x_{mp} \gg \chi\, k, m, n, p > 0 \rightarrow$
$< x_{11}, \ldots, x_{1k}, x_{21}, \ldots, x_{2n}, \ldots, x_{m1}, \ldots, x_{mp} >; ?$

Concatenate removes all occurrences of the null sequence:

$$\textbf{concat} : <<1,3>,<>,<2,4>,<>,<5>> \equiv <1,3,2,4,5> \qquad (2.3)$$

pair : $x \equiv$

$x = < x_1, x_2, \ldots, x_k > \chi\, k > 0 \,\chi\, k \textit{ is even} \rightarrow << x_1, x_2 >, \ldots, < x_{k-1}, x_k >>;$

$x = < x_1, x_2, \ldots, x_k > \chi\, k > 0 \,\chi\, k \textit{ is odd} \rightarrow << x_1, x_2 >, \ldots, < x_k >>; ?$

split : $x \equiv$

$x = < x_1 > \rightarrow << x_1 >, <>>;$

$x = < x_1, x_2, \ldots, x_k > \chi\, k > 1 \rightarrow << x_1, \ldots, x_{\lceil k/2 \rceil} >, < x_{\lceil k/2 \rceil + 1}, \ldots, x_k >>; ?$

iota :$x \equiv$

$x = 0 \rightarrow <>;$

$x \in \mathbf{N}^{+} \rightarrow <1,2,...,x>; ?$

2.3.2. Predicate (Test) Functions

atom : $x \equiv x \in \textit{atoms} \rightarrow \mathbf{T};\ x \neq ? \rightarrow \mathbf{F};\ ?$

eq : $x \equiv x = < y, z > \chi\, y = z \rightarrow \mathbf{T};\ x = < y, z > \chi\, y \neq z \rightarrow \mathbf{F};\ ?$

Also less than (**<**), greater than (**>**), greater than or equal (**>=**), less than or equal (**<=**), not equal (**˜=**); ’=’ is a synonym for **eq**.

null :$x \equiv x = <> \rightarrow \mathbf{T};\ x \neq ? \rightarrow \mathbf{F};\ ?$

length : $x \equiv x = < x_1, x_2, \ldots, x_k > \rightarrow k;\ x = <> \rightarrow 0;\ ?$

2.3.3. Arithmetic/Logical

+ : $x \equiv x = < y, z > \chi\, y, z \textit{ are numbers} \rightarrow y + z;\ ?$ **−** : $x \equiv x = < y, z > \chi\, y, z \textit{ are numbers} \rightarrow y - z;\ ?$
***** : $x \equiv x = < y, z > \chi\, y, z \textit{ are numbers} \rightarrow y \times z;\ ?$ **/** : $x \equiv x = < y, z > \chi\, y, z \textit{ are numbers} \,\chi\, z \neq 0 \rightarrow y \div z;\ ?$

And, or, not, xor

and : $< x, y > \equiv x = \mathbf{T} \rightarrow y;\ x = \mathbf{F} \rightarrow \mathbf{F};\ ?$

or : $< x, y > \equiv x = \mathbf{F} \rightarrow y;\ x = \mathbf{T} \rightarrow \mathbf{T};\ ?$

xor : $<x, y> \equiv$

x=**T** χ y=**T** → **F**; x=**F** χ y=**F** → **F**;

x=**T** χ y=**F** → **T**; x=**F** χ y=**T** → **T**; *?*

not : $x \equiv x$=**T** → F; x=**F** → **T**; *?*

2.3.4. Library Routines

sin : $x \equiv x$ is a number → $sin(x)$; *?*

asin : $x \equiv x$ is a number χ $|x| \leq 1 \rightarrow \sin^{-1}(x)$; *?*

cos : $x \equiv x$ is a number → $cos(x)$; *?*

acos : $x \equiv x$ is a number χ $|x| \leq 1 \rightarrow \cos^{-1}(x)$; *?*

exp : $x \equiv x$ is a number → $e^{x;}$ *?*

log : $x \equiv x$ is a positive number → $ln(x)$; *?*

mod : $<x, y> \equiv x$ **and** y are numbers $\rightarrow x - y \times \left\lfloor \frac{x}{y} \right\rfloor$; *?*

2.4. Functional Forms

Functional forms define new *functions* by operating on function and object *parameters* of the form. The resultant expressions can be compared and contrasted to the *value*-oriented expressions of traditional programming languages. The distinction lies in the domain of the operators; functional forms manipulate functions, while traditional operators manipulate values.

One functional form is *composition*. For two functions φ and ψ the form φ @ ψ denotes their composition $\phi \circ \psi$:

$$(\phi\ @\ \psi) : x \equiv \phi{:}(\psi{:}x), \quad \backslash\!/ \quad x \in \Omega \tag{2.4}$$

The *constant* function takes an object parameter:

$$\%x : y \equiv y = ? \rightarrow ?\,; x, \quad \backslash\!/ \quad x, y \in \Omega \tag{2.5}$$

The function %*?* always returns ?.

In the following description of the functional forms, we assume that ξ, ξ_i, σ, σ_i, τ, and τ_i are functions and that x, x_i, y are objects.

Composition

$(\sigma\ @\ \tau){:}x \equiv \sigma{:}(\tau{:}x)$

Construction

$[\sigma_1, \ldots, \sigma_n]{:}x \equiv <\sigma_1{:}x, \ldots, \sigma_n{:}x>$

Note that construction is also bottom-preserving, e.g.,

$$[+,/] : <3,0> = <3,?> = ? \qquad \textbf{(2.6)}$$

Condition

$(\xi \text{ -> } \sigma;\ \tau){:}x \ \equiv$
$\quad (\xi{:}x) = \mathbf{T} \rightarrow \sigma{:}x\,;$
$\quad (\xi{:}x) = \mathbf{F} \rightarrow \tau{:}x\,;\ ?$

The reader should be aware of the distinction between *functional expressions*, in the variant of McCarthy’s conditional expression, and the *functional form* introduced here. In the former case the result is a *value*, while in the latter case the result is a *function*. Unlike Backus’ FP, the conditional form *must* be enclosed in parenthesis, e.g.,

(isNegative -> - @ [%0,id] ; id) (2.7)

Constant

$\%x{:}y \equiv y = ? \rightarrow ?\,;\, x, \quad \backslash\!\!/ \quad x \in \Omega$

This function returns its object parameter as its result.

Right Insert

$!\sigma{:}x \ \equiv$
$\quad x = <> \rightarrow e_f{:}x\,;$
$\quad x = <x_1> \rightarrow x_1;$
$\quad x = <x_1, x_2, \ldots, x_k> \chi\, k > 2 \rightarrow \sigma{:}<x_1, !\sigma{:}<x_2, \ldots, x_k>>;\ ?$

e.g., !+:<4,5,6>=15.

If σ has a right identity element e_f, then $!\sigma{:}<> = e_f$, e.g.,

!+ : <>=0 and !* : <>=1 (2.8)

Currently, identity functions are defined for + (0), – (0), * (1), / (1), also for **and** (T), **or** (F), **xor** (F). All other unit functions default to bottom (?).

Tree Insert

$|\,\sigma : x \ \equiv$
$\quad x = <> \rightarrow e_f{:}x\,;$
$\quad x = <x_1> \rightarrow x_1;$
$\quad x = <x_1, x_2, \ldots, x_k> \chi\, k > 1 \rightarrow$
$\quad \sigma : <|\,\sigma : <x_1, \ldots, x_{\lceil k/2 \rceil}>, |\,\sigma : <x_{\lceil k/2 \rceil + 1}, \ldots, x_k>>;?$

e.g.,

$|+{:}<4,5,6,7> \equiv +{:}<+{:}<4,5>,+{:}<6,7>> \equiv 15$ (2.9)

Tree insert uses the same identity functions as right insert.

Apply to All

$\mathbf{\&}\,\sigma : x \equiv$

$x = <> \rightarrow <>;$

$x = <x_1, x_2, \ldots, x_k> \rightarrow <\sigma{:}x_1, \ldots, \sigma{:}x_k>;\ ?$

While

$(\mathbf{while}\ \xi\ \sigma){:}x \equiv$

$\xi{:}x = \mathbf{T} \rightarrow (\mathbf{while}\ \xi\ \sigma){:}(\sigma{:}x);$

$\xi{:}x = \mathbf{F} \rightarrow x;\ ?$

2.5. User Defined Functions

An FP definition is entered as follows:

$$\{fn\text{-}name\ fn\text{-}form\}, \tag{2.10}$$

where *fn-name* is an ascii string consisting of letters, numbers and the underline symbol, and *fn-form* is any valid functional form, including a single primitive or defined function. For example the function

$$\{factorial\ !{*}\ @\ iota\} \tag{2.11}$$

is the non-recursive definition of the factorial function. Since FP systems are applicative it is permissible to substitute the actual definition of a function for any reference to it in a functional form: if $f \equiv 1@2$ then $f : x \equiv 1@2 : x,\ \ \forall\ x \in \Omega$.

References to undefined functions bottom out:

$$f{:}x \equiv\ ?\ \forall\ x \in \Omega,\ f \notin \mathbf{F} \tag{2.12}$$

3. Getting on and off the System

Startup FP from the shell by entering the command:

/usr/local/fp.

The system will prompt you for input by indenting over six character positions. Exit from FP (back to the shell) with a control/D (**ˆD**).

3.1. Comments

A user may end any line (including a command) with a comment; the comment character is '#'. The interpreter will ignore any character after the '#' until it encounters a newline character or end-of-file, whichever comes first.

3.2. Breaks

Breaks interrupt any work in progress causing the system to do a FRANZ reset before returning control back to the user.

3.3. Non-Termination

LISP's namestack may, on occasion, overflow. FP responds by printing ''non-terminating'' and returning bottom as the result of the application. It does a FRANZ reset before returning control to the user.

4. System Commands

System commands start with a right parenthesis and they are followed by the command-name and possibly one or more arguments. All this information *must be typed on a single line*, and any number of spaces or tabs may be used to separate the components.

4.1. Load

Redirect the standard input to the file named by the command's argument. If the file doesn't exist then FP appends '.fp' to the file-name and retries the open (error if the file doesn't exist). This command allows the user to read in FP function definitions from a file. The user can also read in applications, but such operation is of little utility since none of the input is echoed at the terminal. Normally, FP returns control to the user on an end-of-file. It will also do so whenever it does a FRANZ reset, e.g., whenever the user issues a break, or whenever the system encounters a non-terminating application.

4.2. Save

Output the source text for all user-defined functions to the file named by the argument.

4.3. Csave and Fsave

These commands output the lisp code for all the user-defined functions, including the original source-code, to the file named by the argument. Csave pretty prints the code, Fsave does not. Unless the user wishes to examine the code, he should use 'fsave'; it is about ten times faster than 'csave', and the resulting file will be about three times smaller.

These commands are intended to be used with the liszt compiler and the 'cload' command, as explained below.

4.4. Cload

This command loads or fasls in the file shown by the argument. First, FP appends a '.o' to the file-name, and attempts a load. Failing that, it tries to load the file named by the argument. If the user outputs his function definitions using fsave or csave, and then compiles them using liszt, then he may fasl in the compiled code and speed up the execution of his defined functions by a factor of 5 to 10.

4.5. Pfn

Print the source text(s) (at the terminal) for the user-defined function(s) named by the argument(s) (error if the function doesn't exist).

4.6. Delete

Delete the user-defined function(s) named by the argument (error if the function doesn't exist).

4.7. Fns

List the names of all user-defined functions in alphabetical order. Traced functions are labeled by a trailing '@' (see § 4.7 for sample output).

4.8. Stats

The ''stats'' command has several options that help the user manage the collection of dynamic statistics for functions[1] and functional forms. Option names follow the keyword ''stats'', e.g., '')stats reset''.

The statistic package records the frequency of usage for each function and functional form; also the size[2] of all the arguments for all functions and functional expressions. These two measures allow the user to derive the average argument size per call. For functional forms the package tallies the frequency of each functional argument. Construction has an additional statistic that tells the number of functional arguments involved in the construction.

Statistics are gathered whenever the mode is on, except for applications that ''bottom out'' (i.e., return bottom – ?). Statistic collection slows the system down by ×2 to ×4. The following printout illustrates the use of the statistic package (user input is emboldened):

---

[1] Measurement of user-defined functions is done with the aid of the trace package, discussed in § 4.9.

[2] ''Size'' is the top-level length of the argument, for most functions. Exceptions are: *apndl*, *distl* (top-level length of the second element), *apndr*, *distr* (top-level length of the first element), and *transpose* (top level length of each top level element).

```
            )stats on

Stats collection turned on.

            +:<3 4>
7
            !* @ iota :3
6
            )stats print

plus:       times    1

times:      times    2

iota:       times    1

insert:     times    1              size  3

                     Functional Args
                Name                Times
                times               1


compos:     times    1              size  1

                     Functional Args
                Name                Times
                insert              1
                iota                          1
```

4.8.1. On

Enable statistics collection.

4.8.2. Off

Disable statistics collection. The user may selectively collect statistics using the on and off commands.

4.8.3. Print

Print the dynamic statistics at the terminal, or, output them to a file. The latter option requires an additional argument, e.g., ‘‘)stats print fooBar’’ prints the stats to the file ‘‘fooBar’’.

4.8.4. Reset

Reset the dynamic statistics counters. To prevent accidental loss of collected statistics, the system will query the user if he tries to reset the counters without first outputting the data (the system will also query the user if he tries to log out without outputting the data).

4.9. Trace

Enable or disable the tracing and the dynamic measurement of the user defined functions named by the argument(s). The first argument tells whether to turn tracing off or on and the others give the name of the functions affected. The tracing and untracing commands are independent of the dynamic statistics commands. This command is cumulative e.g., ')trace on f1', followed by ')trace on f2' is equivalent to ')trace on f1 f2'.

FP tracer output is similar to the FRANZ tracer output: function entries and exits, call level, the functional argument (remember that FP functions have only one argument!), and the result, are printed at the terminal:

```
                )pfn fact

{fact (eq0 -> %1 ; * @ [id, fact @ s1])}
                )fns

eq0             fact  s1

                )trace on fact
                )fns

eq0             fact@      s1

                fact : 2

1 >Enter> fact [2]
|2 >Enter> fact [1]
| 3 >Enter> fact [0]
| 3 <EXIT<  fact  1
|2 <EXIT<  fact  1
1 <EXIT<  fact  2

2
```

4.10. Timer

FP provides a simple timing facility to time top-level applications. The command ‘‘)timer on’’ puts the system in timing mode, ‘‘)timer off’’ turns the mode off (the mode is initially off). While in timing mode, the system reports CPU time, garbage collection time, and elapsed time, in seconds. The timing output follows the printout of the result of the application.

4.11. Script

Open or close a script file. The first argument gives the option, the second the optional script filename. The ‘‘open’’ option causes a new script-file to be opened and any currently open script file to be closed. If the file cannot be opened, FP sends and error message and, if a script file was already opened, it remains open. The command ’’)script close’’ closes an open script file. The user may elect to append script output to the script-file with the append mode.

4.12. Help

Print a short summary of all the system commands:

```
    )help
Commands are:

load <file>                      Redirect input from <file>
save <file>                      Save defined fns in <file>
pfn <fn1> ...                    Print source text of <fn1> ...
delete <fn1> ...                 Delete <fn1> ...
fns                              List all functions
stats on/off/reset/print [file]  Collect and print dynamic stats
trace on/off <fn1> ...           Start/Stop exec trace of <fn1> ...
timer on/of                      Turn timer on/off
script open/close/append         Open or close a script-file
lisp                             Exit to the lisp system (return with ’ˆD’)
debug on/off                     Turn debugger output on/off
csave <file>                     Output Lisp code for all user-defined fns
cload <file>                     Load Lisp code from a file (may be compiled)
fsave <file>                     Same as csave except without pretty-printing
```

4.13. Special System Functions

There are two system functions that are not generally meant to be used by average users.

4.13.1. Lisp

This exits to the lisp system. Use "ˆD" to return to FP.

4.13.2. Debug

Turns the ’debug’ flag on or off. The command ‘‘)debug on’’ turns the flag on, ‘‘)debug off’’ turns the flag off. The main purpose of the command is to print out the parse tree.

5. Programming Examples

We will start off by developing a larger FP program, *mergeSort*. We measure *mergeSort* using the trace package, and then we comment on the measurements. Following *mergeSort* we show an actual session at the terminal.

5.1. MergeSort

The source code for *mergeSort* is:

```
# Use a divide and conquer strategy
{mergeSort | merge}

{merge atEnd @ mergeHelp @ [[], fixLists]}

# Must convert atomic arguments into sequences
# Atomic arguments occur at the leaves of the execution tree
{fixLists &(atom -> [id] ; id)}

# Merge until one or both input lists are empty
{mergeHelp (while and @ &(not@null) @ 2
                (firstIsSmaller -> takeFirst ;
                                   takeSecond))}

# Find the list with the smaller first element
{firstIsSmaller < @ [1@1@2, 1@2@2]}

# Take the first element of the first list
{takeFirst [apndr@[1,1@1@2], [tl@1@2, 2@2]]}

# Take the first element of the second list
{takeSecond [apndr@[1,1@2@2], [1@2, tl@2@2]]}

# If one list isn't null, then append it to the
# end of the merged list
{atEnd (firstIsNull -> concat@[1,2@2] ;
                       concat@[1,1@2])}

{firstIsNull null@1@2}
```

The merge sort algorithm uses a divide and conquer strategy; it splits the input in half, recursively sorts each half, and then merges the sorted lists. Of course, all these sub-sorts can execute in parallel, and the tree-insert (|) functional form expresses this concurrency. *Merge* removes successively larger elements from the heads of the two lists (either *takeFirst* or *takeSecond*) and appends these elements to the end of the merged sequence. *Merge* terminates when one sequence is empty, and then *atEnd* appends any remaining non-empty sequence to the end of the merged one.

On the next page we give the trace of the function *merge*, which information we can use to determine the structure of *merge*'s execution tree. Since the tree is well-balanced, many of the *merge*'s could be executed in parallel. Using this trace we can also calculate the average length of the arguments passed to *merge*, or a distribution of argument lengths. This information is useful for determining communication costs.

```
            )trace on merge

            mergeSort : <0 3 -2 1 11 8 -22 -33>
| 3 >Enter> merge [<0 3>]
| 3 <EXIT<  merge  <0 3>
| 3 >Enter> merge [<-2 1>]
| 3 <EXIT<  merge  <-2 1>
|2 >Enter> merge [<<0 3> <-2 1>>]
|2 <EXIT<  merge  <-2 0 1 3>
| 3 >Enter> merge [<11 8>]
| 3 <EXIT<  merge  <8 11>
| 3 >Enter> merge [<-22 -33>]
| 3 <EXIT<  merge  <-33 -22>
|2 >Enter> merge [<<8 11> <-33 -22>>]
|2 <EXIT<  merge  <-33 -22 8 11>
1 >Enter> merge [<<-2 0 1 3> <-33 -22 8 11>>]
1 <EXIT<  merge  <-33 -22 -2 0 1 3 8 11>

<-33 -22 -2 0 1 3 8 11>
```

5.2. FP Session

User input is **emboldened**, terminal output in Roman script.

fp

FP, v. 4.1 11/31/82

)load ex_man

{all_le}

{sort}

{abs_val}

{find}

{ip}

{mm}

{eq0}

{fact}

{sub1}

{alt_fnd}

{alt_fact}

)fns

abs_val	all_le	alt_fact	alt_fnd	eq0	fact	find
ip	mm	sort	sub1			

abs_val : 3

3

abs_val : -3

3

abs_val : 0

0

abs_val : <-5 0 66>

?

&abs_val : <-5 0 66>

<5 0 66>

)pfn abs_val

{abs_val ((> @ [id,%0]) -> id ; (- @ [%0,id]))}

[id,%0] : -3

<-3 0>

[%0,id] : -3

<0 -3>

```
        - @ [%0,id] : -3
3
        all_le : <1 3 5 7>
T
        all_le : <1 0 5 7>
F
        )pfn all_le
{all_le ! and @ &<= @ distl @ [1,tl]}
        distl @ [1,tl] : <1 2 3 4>
<<1 2> <1 3> <1 4>>
        &<= @ distl @ [1,tl] : <1 2 3 4>
<T T T>
        ! and : <F T T>
F
        ! and : <T T T>
T
        sort : <3 1 2 4>
<1 2 3 4>
        sort : <1>
<1>
        sort : <>
?
        sort : 4
?
        )pfn sort
{sort (null @ tl -> [1] ; (all_le -> apndl @ [1,sort@tl]; sort@rotl))}
        fact : 3
6
        )pfn fact sub1 eq0
```

{fact (eq0 -> %1 ; *@[id , fact@sub1])}

{sub1 -@[id,%1]}

{eq0 = @ [id,%0]}

&fact : <1 2 3 4 5>

<1 2 6 24 120>

eq0 : 3

F

eq0 : <>

F

eq0 : 0

T

sub1 : 3

2

%1 : 3

1

alt_fact : 3

6

)pfn alt_fact

{alt_fact !* @ iota}

iota : 3

<1 2 3>

!* @ iota : 3

6

!+ : <1 2 3>

6

find : <3 <3 4 5>>

T

find : <<> <3 4 <>>>

T

find : <3 <4 5>>

F

)pfn find

{find (null@2 -> %F ; (=@[1,1@2] -> %T ; find@[1,tl@2]))}

[1,tl@2] : <3 <3 4 5>>

<3 <4 5>>

[1,1@2] : <3 <3 4 5>>

<3 3>

alt_fnd : <3 <3 4 5>>

T

)pfn alt_fnd

{alt_fnd ! or @ &eq @ distl }

distl : <3 <3 4 5>>

<<3 3> <3 4> <3 5>>

&eq @ distl : <3 <3 4 5>>

<T F F>

!or : <T F T>

T

!or : <F F F>

F

)delete alt_fnd

)fns

abs_val all_le alt_fact eq0 fact find ip
mm sort sub1

alt_fnd : <3 <3 4 5>>

alt_fnd not defined

?
{g g}

{g}
g : 3

non-terminating

?

[Return to top level]

FP, v. 4.0 10/8/82

[+,*] : <3 4>

<7 12>

[+,* : <3 4>

syntax error:

```
[+,* : <3 4>
     ^
```

ip : <<3 4 5> <5 6 7>>

74

)pfn ip

{ip !+ @ &* @ trans}

trans : <<3 4 5> <5 6 7>>

<<3 5> <4 6> <5 7>>

&* @ trans : <<3 4 5> <5 6 7>>

<15 24 35>

mm : <<<1 0> <0 1>> <<3 4> <5 6>>>

<<3 4> <5 6>>

)pfn mm

{mm &&ip @ &distl @ distr @[1,trans@2]}

[1,trans@2] : <<<1 0> <0 1>> <<3 4> <5 6>>>

<<<1 0> <0 1>> <<3 4> <5 6>>>

distr : <<<1 0> <0 1>> <<3 4> <5 6>>>

<<<1 0> <<3 4> <5 6>>> <<0 1> <<3 4> <5 6>>>>

&distl : <<<1 0> <<3 4> <5 6>>> <<0 1> <<3 4> <5 6>>>>

<<<<1 0> <3 4>> <<1 0> <5 6>>> <<<0 1> <3 4>> <<0 1> <5 6>>>>

```
          &ip @ &dist & distr @ [1,trans @ 2] : <<<1 0> <0 1>> <<3 4> <5 6>>>

   syntax error:

   [+,* : <3 4>
        ^
   &ip @ &distl & distr @ [1,trans @ 2] : <<<1 0> <0 1>> <<3 4> <5 6>>>
                ^
   &ip @ &distl @ distr @ [1,trans@2] : <<<1 0> <0 1>> <<3 4> <5 6>>>

?
```

6. Implementation Notes

FP was written in 3000 lines of FRANZ LISP [Fod 80]. Table 1 breaks down the distribution of the code by functionality.

Functionality	% (bytes)
compiler	34
user interface	32
dynamic stats	16
primitives	14
miscellaneous	3

Table 1

6.1. The Top Level

The top-level function *runFp* starts up the subsystem by calling the routine *fpMain*, that takes three arguments:

(1) A boolean argument that says whether debugging output will be enabled.

(2) A Font identifier. Currently the only one is supported **’asc** (ASCII).

(3) A boolean argument that identifies whether the interpreter was invoked from the shell. If so then all exits from FP return the user back to the shell.

The compiler converts the FP functions into LISP equivalents in two stages: first it forms the parse tree, and then it does the code generation.

6.2. The Scanner

The scanner consists of a main routine, *get_tkn*, and a set of action functions. There exists one set of action functions for each character font (currently only ASCII is supported). All the action functions are named *scan$<font>*, where *<font>* is the specified font, and each is keyed on a particular character (or sometimes a particular character-type – e.g., a letter or a number). *get_tkn* returns the token type, and any ancillary information, e.g., for the token "name" the name itself will also be provided. (See Appendix C for the font-token name correspondences). When a character has been read the scanner finds the action function by doing a

(*get ′scan$ <font> <char>*)

A syntax error message will be generated if no action exists for the particular character read.

6.3. The Parser

The main parsing function, *parse*, accepts a single argument, that identifies the parsing context, or type of construct being handled. Table 2 shows the valid parsing contexts.

id	**construct**
top_lev	initial call
constr*$$*	construction
compos*$$*	composition
alpha*$$*	apply-to-all
insert*$$*	insert
ti*$$*	tree insert
arrow*$$*	affirmative clause of conditional
semi*$$*	negative clause of conditional
lparen*$$*	parenthetic expr.
while*$$*	while

Table 2, Valid Parsing Contexts

For each type of token there exists a set of parse action functions, of the name *p$<tkn-name>*. Each parse-action function is keyed on a valid context, and it is looked up in the same manner as scan action functions are looked up. If an action function cannot be found, then there is a syntax error in the source code. Parsing proceeds as follows: initially *parse* is called from the top-level, with the context argument set to ''*top_lev*''. Certain tokens cause parse to be recursively invoked using that token as a context. The result is the parse tree.

6.4. The Code Generator

The system compiles FP source into LISP source. Normally, this code is interpreted by the FRANZ LISP system. To speed up the implementation, there is an option to compile into machine code using the *liszt* compiler [Joy 79]. This feature improves performance tenfold, for some programs.

The compiler expands all functional forms into their LISP equivalents instead of inserting calls to functions that generate the code at run-time. Otherwise, *liszt* would be ineffective in speeding up execution since all the functional forms would be executed interpretively. Although the amount of code generated by an expanding compiler is 3 or 4 times greater than would be generated by a non-expanding compiler, even in interpreted mode the code runs twice as quickly as unexpanded code. With *liszt* compilation this performance advantage increases to more than tenfold.

A parse tree is either an atom or a hunk of parse trees. An atomic parse-tree identifies either an fp built-in function or a user defined function. The hunk-type parse tree represents functional forms, e.g., compose or construct. The first element identifies the functional form and the other elements are its functional parameters (they may in turn be functional forms). Table 3 shows the parse-tree formats.

Form	Format
user-defined	<atom>
fp builtin	<atom>
apply-to-all	*{alpha$$* Φ*}*
insert	*{insert $$* Φ*}*
tree insert	*{ti $$* Φ*}*
select	*{select $$* μ*}*
constant	*{constant $$* μ*}*
conditional	*{condit $$* Φ_1 Φ_2 Φ_3*}*
while	*{while $$* Φ_1 Φ_2*}*
compose	*{compos $$* Φ_1 Φ_2*}*
construct	*{constr $$* Φ_1 Φ_2 , . . . , Φ_n *nil }*

Note: Φ and the Φ_k are parse-trees and μ is an optionally signed integer constant.

Table 3, Parse-Tree Formats

6.5. Function Definition and Application

Once the code has been generated, then the system defines the function via *putd*. The source code is placed onto a property list, *′sources*, to permit later access by the system commands.

For an application, the indicated function is compiled and then defined, only temporarily, as *tmp $$*. The single argument is read and *tmp $$* is applied to it.

6.6. Function Naming Conventions

When the parser detects a named primitive function, it returns the name *<name >$fp*, where *<name>* is the name that was parsed (all primitive function-names end in *$fp*). See Appendix D for the symbolic (e.g., compose, +) function names.

Any name that isn't found in the list of builtin functions is assumed to represent a user-defined function; hence, it isn't possible to redefine FP primitive functions. FP protects itself from accidental or malicious internal destruction by appending the suffix ''*_fp*'' to all user-defined function names, before they are defined.

6.7. Measurement Impelementation

This work was done by Dorab Patel at UCLA. Most of the measurement code is in the file 'fpMeasures.l'. Many of the remaining changes were effected in 'primFp.l', to add calls on the measurement package at run-time; to 'codeGen.l', to add tracing of user defined functions; to 'utils.l', to add the new system commands; and to 'fpMain.l', to protect the user from forgetting to output statistics when he leaves FP.

6.7.1. Data Structures

All the statistics are in the property list of the global symbol *Measures*. Associated with each each function (primitive or user-defined, or functional form) is an indicator; the statistics gathered for each function are the corresponding values. The names corresponding to primitive functions and functional forms end in '$fp' and the names corresponding to user-defined functions end in '_fp'. Each of the property values is an association list:

```
(get 'Measures 'rotl$fp) ==> ((times . 0) (size . 0))
```

The car of the pair is the name of the statistic (i.e., times, size) and the cdr is the value. There is one exception. Functional forms have a statistic called funargtyp. Instead of being a dotted pair, it is a list of two elements:

```
(get 'Measures 'compose$fp) ==>
((times . 2) (size . 4) (funargtyp ((select$fp . 2) (sub$fp . 2))))
```

The car is the atom 'funargtyp' and the cdr is an alist. Each element of the alist consists of a functional argument-frequency dotted pair.

The statistic packages uses two other global symbols. The symbol DynTraceFlg is non-nil if dynamic statistics are being collected and is nil otherwise. The symbol TracedFns is a list (initially nil) of the names of the user functions being traced.

6.7.2. Interpretation of Data Structures

6.7.2.1. Times

The number of times this function has been called. All functions and functional forms have this statistic.

6.7.2.2. Size

The sum of the sizes of the arguments passed to this function. This could be divided by the times statistic to give the average size of argument this function was passed. With few exceptions, the size of an object is its top-level length (note: version 4.0 defined the size as the total number of *atoms* in the object); the empty sequence, ‘‘<>’’, has a size of 0 and all other atoms have size of one. The exceptions are: *apndl, distl* (top-level length of the second element), *apndr, distr* (top-level length of the first element), and *transpose* (top level length of each top level element).

This statistic is not collected for some primitive functions (mainly binary operators like +,-,*).

6.7.2.3. Funargno

The number of functional arguments supplied to a functional form.

Currently this statistic is gatherered only for the construction functional form.

6.7.2.4. Funargtyp

How many times the named function was used as a functional parameter to the particular functional form.

6.8. Trace Information

The level number of a call shows the number of steps required to execute the function on an ideal machine (i.e., one with unbounded resources). The level number is calculated under an assumption of infinite resources, and the system evaluates the condition of a conditional before evaluating either of its clauses. The number of functions executed at each level can give an idea of the distribution of parallelism in the given FP program.

7. Acknowledgements

Steve Muchnick proposed the initial construction of this system. Bob Ballance added some of is own insights, and John Foderaro provided helpful hints regarding effective use of the FRANZ LISP system [Fod80]. Dorab Patel [Pat81] wrote the dynamic trace and statistics package and made general improvements to the user interface. Portions of this manual were excerpted from the *COMPCON-83 Digest of*

Papers[3].

8. References

[Bac78]
John Backus, ‘‘Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs,’’ *CACM*, Turing Award Lecture, 21, 8 (August 1978), 613-641.

[Fod80]
John K. Foderaro, ‘‘The FRANZ LISP Manual,’’ University of California, Berkeley, California, 1980.

[Joy79]
W.N. Joy, O. Babaoglu, ‘‘UNIX Programmer’s Manual,’’ November 7, 1979, Computer Science Division, University of California, Berkeley, California.

[Mc60]
J. McCarthy, ‘‘Recursive Functions of Symbolic expressions and their Computation by Machine,’’ Part I, *CACM* 3, 4 (April 1960), 184-195.

[Pat80]
Dorab Ratan Patel, ‘‘A System Organization for Applicative Programming,’’ M.S Thesis, University of California, Los Angeles, California, 1980.

[Pat81]
Dorab Patel, ‘‘Functional Language Interpreter User Manual,’’ University of California, Los Angeles, California, 1981.

[3] Scott B. Baden and Dorab R. Patel, ‘‘Berkeley FP – Experiences With a Functional Programming Language’’, © 1982, IEEE.

Appendix A: Local Modifications

1. Character Set Changes

Backus [Ba78] used some characters that do not appear on our ASCII terminals, so we have made the following substitutions:

constant	$\bar{x}$	%*x*
insert	/	!
apply-to-all	α	&
composition	$\circ$	@
arrow	$\rightarrow$	->
empty set	ϕ	<>
bottom	$\perp$	?
divide	$\div$	/
multiply	$\times$	*

2. Syntactic Modifications

2.1. While and Conditional

While and conditional functional expressions *must* be enclosed in parenthesis, e.g.,

(**while** *f g*)

(*p* **->** *f* **;** *g*)

2.2. Function Definitions

Function definitions are enclosed by curly braces; they consist of a name-definition pair, separated by blanks. For example:

{**fact** !* @ iota}

defines the function **fact** (the reader should recognize this as the non-recursive factorial function).

2.3. Sequence Construction

It is not necessary to separate elements of a sequences with a comma; a blank will suffice:

<1,2,3> ≡ <1 2 3>

For nested sequences, the terminating right angle bracket acts as the delimiter:

<<1,2,3>,<4,5,6>> ≡ <<1 2 3><4 5 6>>

3. User Interface

We have provided a rich set of commands that allow the user to catalog, print, and delete functions, to load them from a file and to save them away. The user may generate script files, dynamically trace and measure functional expression execution, generate debugging output, and, temporarily exit to the FRANZ LISP system. A command must begin with a right parenthesis. Consult Appendix C for a complete description of the command syntax.

Debugging in FP is difficult; all undefined results map to a single atom – *bottom* (‘‘?’’). To pinpoint the cause of an error the user can use the special debugging output function, **out**, or the tracer.

4. Additions and Ommissions

Many relational functions have been added: $<$, $>$, $=$, $\neq$, $\leq$, $\geq$; their syntax is: <, >, =, ~=, <=, >=. Also added are the **iota** function (This is the APL iota function an n-element sequence of natural numbers) and the exclusive OR ($\oplus$) function.

Several new structural functions have been added: **pair** pairs up successive elements of a sequence, **split** splits a sequence into two (roughly) equal halves, **last** returns the last element of the sequence (<> if the sequence is empty), **first** returns the first element of the sequence (<> if it is empty), and **concat** concatenates all subsequences of a sequence, squeezing out null sequences (<>). **Front** is equivalent to **tlr**. **Pick** is a parameterized form of the selector function; the first component of the argument selects a single element from the second component. **Out** is the only side-effect function; it is equivalent to the **id** function but it also prints its argument out at the terminal. This function is intended to be used only for debugging.

One new functional form has been added, tree insert. This functional form breaks up the the argument into two roughly equal pieces applying itself recursively to the two halves. The functional parameter is applied to the result.

The binary-to-unary functions (’**bu**’) has been omitted.

Seven mathematical library functions have been added: sin, cos, asin ($\sin^{-1}$), acos ($\cos^{-1}$), log, exp, and mod (the remainder function)

Appendix B: FP Grammar

I. BNF Syntax

fpInput → (fnDef | application | fpCmd[a])∗ | ’ˆD’

fnDef → ’{’ name funForm ’}’

application → funForm ’:’ object

name → letter (letter | digit | ’_’)∗

nameList → (name)∗

object → atom | fpSequence | ’?’

fpSequence → ’<’ (ε | object ((’,’ | ’ ’) object)∗) ’>’

atom → ’T’ | ’F’ | ’<>’ | ’"’ (ascii-char)∗ ’"’ | (letter | digit)∗ | number

funForm → simpFn | composition | construction | conditional |
constantFn | insertion | alpha | while | ’(’ funForm ’)’

simpFn → fpDefined | fpBuiltin

fpDefined → name

fpBuiltin → selectFn | ’tl’ | ’id’ | ’atom’ | ’not’ | ’eq’ | relFn | ’null’ | ’reverse’ |
’distl’ | ’distr’ | ’length’ | binaryFn | ’trans’ | ’apndl’ | ’apndr’ |
’tlr’ | ’rotl’ | ’rotr’ | ’iota’ | ’pair’ | ’split’ | ’concat’ | ’last’ | ’libFn’

selectFn → (ε | ’+’ | ’-’) unsignedInteger

relFn → ’<=’ | ’<’ | ’=’ | ’˜=’ | ’>’ | ’>=’

binaryFn → ’+’ | ’-’ | ’∗’ | ’/’ | ’or’ | ’and’ | ’xor’

libFn → ’sin’ | ’cos’ | ’asin’ | ’acos’ | ’log’ | ’exp’ | ’mod’

composition → funForm ’@’ funForm

construction → ’[’ formList ’]’

formList → ε | funForm (’,’ funForm)∗

conditional → ’(’ funForm ’->’ funForm ’;’ funForm ’)’

constantFn → ’%’ object

insertion → ’!’ funForm | ’|’ funForm

alpha → ’&’ funForm

while → ’(’ ’while’ funForm funForm ’)’

II. Precedences

1.	%, !, &	(highest)
2.	@	
3.	[· · ·]	
4.	-> · · · ; · · ·	
5.	while	(least)

[a] Command Syntax is listed in Appendix C.

Appendix C: Command Syntax

All commands begin with a right parenthesis (‘‘)’’).

)fns
)pfn <nameList>
)load <UNIX file name>
)cload <UNIX file name>
)save <UNIX file name>
)csave <UNIX file name>
)fsave <UNIX file name>
)delete <nameList>
)stats on
)stats off
)stats reset
)stats print [UNIX file name]
)trace on <nameList>
)trace off <nameList>
)timer on
)timer off
)debug on
)debug off
)script open <UNIX file name>
)script close
)script append <UNIX file name>
)help
)lisp

Appendix D: Token-Name Correspondences

Token	Name
[	lbrack$$
]	rbrack$$
{	lbrace$$
}	rbrace$$
(	lparen$$
)	rparen$$
@	compos$$
!	insert$$
|	ti$$
&	alpha$$
;	semi$$
:	colon$$
,	comma$$
+	builtin$$
+ µ[a]	select$$
*	builtin$$
/	builtin$$
=	builtin$$
-	builtin$$
->	arrow$$
- µ	select$$
>	builtin$$
>=	builtin$$
<	builtin$$
<=	builtin$$
˜=	builtin$$
%o[b]	constant$$

[a] µ is an optionally signed integer constant.

[b] o is any FP object.

Appendix E: Symbolic Primitive Function Names

The scanner assigns names to the alphabetic primitive functions by appending the string ‘‘$fp’’ to the end of the function name. The following table designates the naming assignments to the non-alphabetic primitive function names.

Function	Name
+	plus$fp
-	minus$fp
*	times$fp
/	div$fp
=	eq$fp
>	gt$fp
>=	ge$fp
<	lt$fp
<=	le$fp
˜=	ne$fp

Table of Contents