RJ 7100 (67163) 10/26/89
Computer Science

Research Report

FL LANGUAGE MANUAL,
PARTS 1 AND 2

John Backus
John H. Williams
Edward L. Wimmers
Peter Lucas
Alexander Aiken

IBM Almaden Research Center Dept. K53/803
650 Harry Road
San Jose, CA 95120

IBM Research Division
Yorktown Heights, New York ▪ San Jose, California ▪ Zurich, Switzerland

Acknowledgments

The authors would like to acknowledge numerous helpful suggestions about the design of FL from the people who are working with us on the FL compiler: Thom Linden, Brian Murphy and Paul Tucker. We would also like to thank Prof. John Hughes (Glasgow University) for his helpful suggestions during his stay with us in the Summer of 1987.

RJ 7100 (67163) 10/26/89
Computer Science

FL Language Manual†, Parts 1 and 2

John Backus
John H. Williams
Edward L. Wimmers
Peter Lucas
Alexander Aiken

IBM Almaden Research Center
San Jose, CA 95120

ABSTRACT:

FL is intended to be a programming language in which it is easy to write clear, concise, and efficient programs. FL is designed around a rich set of functionals, forms for combining existing programs to construct new ones. This emphasis on programming at the function level results in programs that have a rich mathematical structure useful in reasoning about and optimizing programs.

FL programs directly access I/O devices and the file system using primitive history-sensitive functions. An implicitly controlled history component provides this access without requiring I/O "streams" as explicit arguments of interactive programs.

FL programs can be higher-order. In addition to the primitive combining forms provided, the language provides powerful mechanisms for defining new higher-order functions. Conditional expressions and lambda expressions employ *patterns*. A pattern defines a predicate and selectors for objects having the same structure as the pattern, and it may contain arbitrary predicates for elements of that structure.

FL is statically scoped. An FL program consists of an expression together with an environment of function definitions used in evaluating that expression. The language provides a complete set of operations for combining environments, thus enhancing program reusability. For example, any environment may be modified to export only some of its definitions; this facility, together with those for building user data types, provides a powerful technique for defining abstract data types.

FL uses a simple concept of *type* in which types are identified with predicates for subsets of the value domain. The user may define new types in terms of existing ones. FL guarantees that all type errors are caught, although not all type errors are detected at compile time. For example, type errors involving stored or transmitted user defined types may be found at run time.

FL values include *exceptions* in addition to normal values (i.e., atoms, functions, sequences of values, and user defined objects). Exceptions contain information about their origin; they are generated when functions are applied to inappropriate arguments, and the user may produce them with the primitive `signal`. All functions preserve exceptions. The functional `catch` permits recovery from exceptions; it can be used with `signal` for backtracking.

The meaning of FL programs is specified by a denotational semantics that uses an unusually simple domain of values as its basis. This makes the denotational semantics a useful tool for the programmer as well as for the language analyst.

---

†Supercedes FL Language Manual (Preliminary Version), RJ5339, 11/7/86, which is now obsolete.

FL Language Manual
Parts 1 and 2

John Backus John H. Williams Edward L. Wimmers
Peter Lucas Alexander Aiken

October 25, 1989

Contents

Part I
Overview of the FL language

1 Introduction

1.1 Organization of this manual

The FL language manual consists of three parts divided into two separate documents. This is the first document, which contains Parts 1 and 2; the second document is titled *FL Language Manual, Part 3*. Part 1 gives an informal explanation of the main features of the FL language, with some short examples. The more advanced features are discussed informally in Part 2; Part 2 ends with a list of the FL primitive functions and the complete BNF syntax of FL. Parts 1 and 2 are addressed to users. Part 3 is a formal description of the denotational semantics and the primitive functions of FL; it is addressed to language experts and implementers.

1.1.1 Relationship to earlier FL Language Manual

The FL language described in this manual differs in many important respects from that described in *FL Language Manual (Preliminary Version)*, Research Report RJ 5339 (11/7/86), therefore the latter is now completely obsolete. The language described here is believed to be both simpler and more powerful than the earlier version.

1.2 Goals of the FL language

FL is a language for defining functions that map values into other values. It is a *function level* language, that is, it is based on the use of *combining forms*, operations for building new programs from existing ones. The mathematical properties of the combining forms generate many algebraic identities useful for reasoning about and optimizing FL programs. One purpose of the FL project is to test two interrelated hypotheses: (a) Can the mathematical properties of FL programs be used to compile programs that run as fast as the best Fortran and Lisp programs? and (b) Is the function level style of programming a powerful and flexible one (in spite of its unfamiliarity); does it give programs a useful structure that is helpful in understanding and reasoning about them?

1.3 General properties of the language

Briefly, an FL program is a function. FL expressions apply functions to values to obtain new values. A value may be an object or a function. Combining forms are (higher order) functions whose results are functions. Special functions yield inputs from input devices or files; others cause their arguments to be written to output devices or files. The functions used in an expression may be defined in auxilliary definitions in an appended **where** clause or they may be built from existing functions

by combining forms. Such expressions, definitions and `where` clauses comprise the basic elements of FL.

The FL language offers some enhancements to these basic elements that provide a more readable and concise shorthand for certain constructs. Patterns specify "shape" predicates *and* define functions for selecting components from values of the designated shape. Enhanced definitions employ patterns that clarify the domain of the defined function and define selector functions that name parts of the function's argument. Lambda expressions simplify the definition of higher order functions.

2 Expressions and values

Every FL *expression* has a *value*, which may be either a *normal* or *abnormal* value.

2.1 Normal values

A normal value may be an *atom*, that is, a *number*, a *char* (a character), or a *truth value* (`true` or `false`). In addition to atomic values, a normal value may be a *function*, a *sequence* of normal values, or an element of a user defined data type. A function `f` is said to be *defined* for a value `x` if the result of applying `f` to `x` is a normal value.

2.2 Abnormal values: exceptions and bottom

There are two kinds of abnormal values: *exceptions* and $\perp$, pronounced *bottom*. Exceptions result when functions are applied to inappropriate arguments and terminate, whereas $\perp$ denotes the value of a non-terminating computation. Abnormal values are persistent and cannot be elements of composite values (e.g., for any abnormal value `abnorm` and any function `f`, `f` applied to `abnorm` is `abnorm`; a sequence containing `abnorm` collapses to it leftmost abnormal value). This means that all FL functions are *strict* with respect to exceptions and $\perp$ (in contrast to *lazy* languages in which `f`($\perp$) may not be $\perp$ and a sequence may have an exception or $\perp$ as an element).

Exceptions may be system-generated, i.e., the result of applying a primitive function to an inappropriate argument, or user-generated by applying the exception-making function `signal` to a normal value. Roughly speaking, a system produced exception contains the name of the function whose application produced it, information about any special circumstances, and the offending argument of the function. User generated exceptions may contain any normal value and can duplicate system-generated exceptions. Recovery from exceptions can be effected by use of the combining form `catch` (see Section 17).

3 Expressions

The following sections describe the syntax and semantics of the basic kinds of expressions. The discussion of some more advanced expressions is postponed until Part 2.

3.1 Syntax conventions

Each informal description of a form for an expression is accompanied by a BNF description of its syntax. Italic names are non-terminals, | denotes "or", braces $\{\ldots\}$ are for grouping, x_p^* denotes zero or more x's separated by p's and x_p^+ denotes one or more x's separated by p's. For example

seqs ::= < $\{$*atom* | *name* | *seqs*$\}_,^*$ >

describes *seqs*, sequences of constants (angle brackets <...> surround sequences); these include the following expressions:

```
<>    <1,2>    <square,5,6>    <+, <4,5>, 7>
```

3.2 Atoms and function names

The simplest expressions are atoms and names of functions (Section 18 gives a list of the FL primitive functions). Thus 5.2 (a number), `false` (a truth value) and `e (a char, denoting e) are atoms and therefore expressions; function names, such as `tl` (*tail*), are also expressions.

atom	::=	*char* | *number* | *truthval*
char	::=	`*character* (backquote distinguishes chars from names)
truthval	::=	`true` | `false`
name[†]	::=	*identifier* | ∧ | ∨ | ↤ | ↦ | + | - | * | ÷ | ∘ | = | ! | ⇨ | ⇦
identifier	::=	*ident_char* $\{$*ident_char* | *digit*$\}^*$
ident_char	::=	*letter* | / | $ | % | # | _ | ? | ↑ | ↓ | ^ | ¬

Chars are formed by prefixing a character with backquote ` to distinguish them from single-character names. *Names* are function names, which are either identifiers or single-character, self-delimiting operators such as +. FL provides no mechanism for naming a value that is not a function. *Identifiers* are made of letters, digits and some special, non-self-delimiting characters; they begin with a letter or special character.

[†]The character pairs /* and */ always begin and end comments, thus, in an infix expression, a name ending in / must not be immediately followed by * and a name beginning with / must not be immediately preceded by *. Inserting a space between such a name and * delimits the end or beginning of the name. See Comments, Section 16.1

Characters are Roman and Greek upper and lower case letters, the decimal digits, punctuation, brackets, special characters, plus unprintable characters that are represented by a sequence of letters and/or digits surrounded by backslashes (e.g., \SP\ denotes a space). *Numbers* are either integers or reals; 3 is an integer and 3.2 is a real. The terms *character* and *number* are described in Section 19, Syntax.

3.2.1 Truth values

Although `true` is a truth value, all the primitives of FL treat any normal value other than `false` as true; therefore in the following "is true" is synonomous with "is any normal value other than `false`".

3.2.2 Examples of function names

Here are some function names:

`tl` α $\alpha 3$ $\circ$ ρ`Abc[^3_$%` `/1` $\mapsto$

Note that `a∘b` = `a ∘ b` since $\circ$ is a self delimiting name, but, since / is not self delimiting, `a/b` is a name whereas `a / b` is an infix expression denoting the application of the function named / to `a` and `b`. Of course the characters for self-delimiting functions cannot be used in a multi-character name.

3.3 Applications

A composite expression may be built by *application* (denoted by :); if e_1 and e_2 are expressions, then the expression e_1:e_2 denotes the result of applying the value of e_1 (a function) to the value of e_2. For example, the value of the expression

`tl:<1, 2, 3>`

is the result of applying the primitive function `tl` (*tail*) to a sequence of three numbers, giving the value `<2, 3>`.

appl ::= *expr*:*expr*

3.4 Sequences

A sequence of expressions is also an expression. For example, the expression

`<tl:<1>, +:<2, 3, 4>>`

has the value

`<<>, 9>`

obtained by evaluating the two applications in the original expression from left to right. Note that there is no restriction on the "types" of the elements of a sequence. If one of the expressions in a sequence produces an abnormal value

abnorm, then evaluation stops and the whole sequence evaluates to abnorm. For example, <tl:3, tl:<>> evaluates to the exception signal:<"tl", "arg1", 3> that results from tl:3.

seq ::= < *expr*$^*_,$ >

3.4.1 Strings

A *string* is just a sequence of chars, thus all operations on sequences can be applied to strings. For example, <ˆa, ˆb, ˆc> is a string comprising the three chars ˆa, ˆb and ˆc (chars begin with ˆ to distinguish them from one-letter function names; the char ˆa denotes the character a). There is an alternative way to write strings. For example, "abc" is a string that is exactly equivalent to <ˆa, ˆb, ˆc>; thus tl:"abc" = tl:<ˆa, ˆb, ˆc> = <ˆb, ˆc> = "bc". Similarly, "a\FF\" is shorthand for <ˆa, ˆ\FF\>, where ˆ\FF\ is a char formed from the coded character \FF\; both the char and the character denote a "Form Feed" or newpage. Note that coded characters, like \FF\, can become chars by prefixing them with ˆ.

string ::= < *char*$^*_,$ > | "*character**"

4 Combining forms

FL emphasizes the function level approach to defining functions, that of building new functions by applying *combining forms* to existing functions. The result of building a new function in this way is a (function-valued) expression. Some combining forms are used so frequently that they have a special syntactic form.

4.1 Composition

Composition (∘) combines functions f and g, giving the new function f∘g, defined by

(f∘g):x = f:(g:x)

For example, (s1∘tl):x selects the second element of x since tl produces the tail of x and the primitive s1 then selects the first element of the tail. The argument x must be a sequence with two or more elements, or an exception results.

The symbol ∘ is the self-delimiting function name for composition and the expression f∘g is an instance of an *infix expression* that is equivalent to the expression ∘:<f, g>. This equivalence follows from the general infix rule:

$expr_1$ $expr_2$ $expr_3$ is an expression equivalent to: $expr_2$:<$expr_1$, $expr_3$>

4.2 Constant

The constant-valued function ~x is built from the value x and the constant combining form ~; when applied to any normal value y it yields x. Thus, for any expression *e*, ~*e* is a function-valued expression. For example, ~2:<1,5> = 2 (but ~2:abnorm = abnorm for any abnormal value abnorm).

expr ::= ... | ~*expr* | ...

4.3 Condition

Condition (→) combines three functions p, f and g, giving the new function p → f ; g defined by:

$$(p \to f\,;\, g):x \quad = \quad \begin{cases} x & \text{if } x \text{ is abnormal} \\ f:x & \text{if } p:x = \text{true (a normal value} \neq \underline{\texttt{false}}) \\ g:x & \text{if } p:x = \underline{\texttt{false}} \\ \text{exception} & \text{if } p:x = \text{exception} \\ \bot & \text{if } p:x = \bot \end{cases}$$

The condition p → f ; g produces an abnormal value abnorm at x only if x = abnorm, or p:x = abnorm, or p:x is true and f:x = abnorm, or p:x is false and g:x = abnorm. The condition (isseq → len; ~0):x computes the length of x (len:x) if x is a sequence; otherwise it yields 0 (~0:x).

The one-arm condition p → f is similar, except that it produces an exception if the predicate p is false; it is equivalent to the condition

p → f; signal∘[~"cond", ~"1arm", id]

Thus (p → f):x = f:x if p:x is true, otherwise (p → f):x is an exception containing the value <"cond", "1arm", x>.

expr ::= ... | *expr* → *expr*; *expr* | *expr* → *expr* | ...

4.3.1 Functions as predicates

Note that the definition of condition means that any function p can serve as a "predicate"; in the rest of this manual the words "function" and "predicate" are used interchangeably. The phrases "p:x is true" and "x satisfies p" indicate that p:x is any normal value other than <u>false</u>.

4.4 Construction

Construction ([...]) combines n≥0 functions $f_1, \ldots, f_n$ giving the new function $[f_1, \ldots, f_n]$ defined by:

$[f_1, \ldots, f_n]:x = \langle f_1:x, \ldots, f_n:x \rangle$

For example, [+, -]:<3, 2> = <5, 1> and [s1, t1]:<1, 2, 3> = <1, <2, 3>>. A construction produces an exception or $\perp$ if one of its functions does (the result is the first abnormal value, evaluating from left to right).

expr ::= ... | [*expr*$^*_{,}$] | ...

4.5 Combining forms without special syntax

Construction is an example of a combining form that has special syntax ([..]) to make it more readable. However, construction and all the other combining forms can be written as an application of a higher order function. For example, the function $[f_1, \ldots, f_n]$ can be written using the primitive function cons as follows:

$$[f_1, \ldots, f_n] = \texttt{cons}:\langle f_1, \ldots, f_n \rangle$$

Similarly, there are primitive functions K and cond such that

$$\begin{aligned} \text{'}x &= \texttt{K}:x \\ p \rightarrow f; g &= \texttt{cond}:\langle p, f, g \rangle \end{aligned}$$

4.6 Predicate combining forms

Predicate combining forms build (total) predicates from (total) predicates. (A total predicate is a function that yields **true** or **false** for all normal values.)

4.6.1 Predicate construction

Predicate construction [[..]] combines $n \geq 0$ functions $p_1, \ldots, p_n$ (regarded as predicates), giving the new predicate defined by:

$$[[p_1, \ldots, p_n]]:x = \begin{cases} p_i:x & \text{if } p_i:x \text{ is abnormal (smallest i)} \\ \textbf{true} & \text{if } x = \langle x_1, .., x_n \rangle \text{ and } p_i:x \text{ is true} \\ & \text{for each } i = 1, .., n \\ \textbf{false} & \text{otherwise} \end{cases}$$

For example, [[isnum, isseq]]:x is **true** only if x has the form <*number*, *sequence*>; otherwise it is **false**.

In addition to denoting a predicate combining form, [[..]] is unique in also denoting a *pattern combining form* (see Section 6) when one or more of its elements is a pattern.

4.6.2 Some other predicate combining forms

These are the other primitive predicate combining forms:

Combining form	Use of form	True only for
seqof	seqof:p	sequences, all of whose elements satisfy p
eqto	eqto:x	values y such that x = y
∧	p∧q	values for which both p and q are true
∨	p∨q	values for which either p or q is true
¬	¬:p	values for which p is false
↦	p↦q	non-empty sequences whose first element satisfies p and whose tail satisfies q
↤	p↤q	non-empty sequences whose last element satisfies q and whose tail-right (all but the last element) satisfies p
=	f=g	values x for which f:x = g:x

All but one of the predicate combining forms (predicate construction) are ordinary combining forms without special syntax, i.e., higher order functions that take one or two function arguments (the latter are usually written in infix form, for example, f↦g rather than ↦:<f, g>). Predicate construction has the following special syntax

$$expr ::= \ldots \mid [\![\ expr^*_{,}\ ]\!] \mid \ldots$$

but it can be expressed without its special form by using the primitive pcons:

$$[\![p_1, \ldots, p_n]\!] = \texttt{pcons:<}p_1, \ldots, p_n\texttt{>} \quad \text{for all expressions } p_i$$

5 Where clauses, environments and definitions

5.1 Where clauses

An expression may be modified by a **where** clause that defines some free function names used in it. For example, the expression

```
f:<1, 2, 3> where {def f ≡ s1∘tl}                    (1)
```

defines the function name f in the expression f:<1, 2, 3> to be the composition (∘) of the primitives s1 (selects the first element of a sequence) and tl, with the result that f:<1, 2, 3> = (s1∘tl):<1, 2, 3> = s1:(tl:<1, 2, 3>) = s1:<2, 3> = 2. The simplest **where** clause consists of the keyword **where** followed by a list of definitions (enclosed in braces {...} and separated by spaces). For example, in the expression

```
f:<1, 2, 3> where {def f ≡ g∘tl def g ≡ s1}
```

the **where** clause has a list of two definitions that give f the same definition as in (1), since g in the definition of f is defined as s1 in the same definition list. A definition list containing two definitions of the same function name is syntactically ill-formed.

5.2 Environments

A **where** clause thus provides an *environment* that associates function names with the functions. An environment may hide some of the definitions it makes.

expr	::=	... | *expr* **where** *env* | ...
env	::=	*defnlist* | **export**(*namelist*) *env* | **hide**(*namelist*) *env* | *composite_env* | **lib**(*string*) | **PF** | { *env* }
defnlist	::=	$\{defn^{+}_{blank}\}$
namelist	::=	$name^{+}_{,}$

Export and hide environments are described below. The other environments, *composite environments* (*composite_env*), *library environments* (**lib**(*string*)) and the *primitive environment* (**PF**) are discussed in Section 14.

5.3 Definitions

A definition associates a function name with a function.

defn ::= **def** *name* ≡ *expr* | **nrdef** *name* ≡ *expr* | ...

In either of the forms **def** or **nrdef** the function name *name* is assigned the function that is the value of *expr*. (Other forms of definitions are discussed in several later sections.) The keyword indicates that the definition is recursive (**def**) or non-recursive (**nrdef**). The difference is significant only if the *expr* part of a definition uses locally defined function names (those that are defined in the list of definitions containing the given definition). In a **def** definition, free function names in *expr* refer to their local definitions, if they exist; in a **nrdef** definition, all free function names in *expr* refer to definitions outside of the current definition list.

For example, the following expression

```
last where {def last ≡ isnull∘tl → g; last∘tl def g ≡ s1}
```

has as its value the recursively defined function `last` that satisfies the equation `last = isnull∘tl → s1; last∘tl` (this function is defined for non-empty sequences and computes the last element). In contrast, the following expression uses the primitive + in the redefinition of the function name +; it extends the function + to sum complex numbers (`cadd`) as well as ordinary numbers (see Section 7.2.1 for the definition of `cadd`):

```
e where {nrdef + ≡ seqof:iscomplex → cadd; +}
```

The + on the right side of the definition refers to its definition outside of the **where** clause. Uses of + in `e` are in the scope of the above definition and hence can be used in the extended sense.

5.4 Scope of definitions

Let `e` be an expression and let `ENV` be an environment; then in the expression

```
e where ENV
```

e and every subexpression of e is in the *scope* of ENV. If a function name f occurs free in e (i.e., f is not in the scope of some environment within e that defines f) and if ENV defines f, then its definition governs this occurrence of f. If ENV does not define f, then the expression e where ENV must itself be in the scope of some outer environment that defines f; otherwise the function name f denotes an exception-producing function. The outermost environment is by convention the primitive environment that defines all function names as either primitive functions or exception-producing functions.

All the names occurring in the right sides of def definitions in a definition list DL get their definitions first from definitions in DL itself, or if they are not defined there, from the surrounding environment. All the names occurring in the right sides of nrdef definitions in DL get their definitions from the surrounding environment.

5.5 Environments that hide some of their definitions

An environment can prevent some of the functions defined in its definition list from being accessible outside of the environment. The following environments

export(*namelist*) *env*
hide(*namelist*) *env*

hide some of the definitions made by *env*. Thus export hides the definitions of functions made by *env* whose names are *not* listed in *namelist* and hide hides those definitions whose names *are* listed in *namelist*. For example, the where clause in the following expression

```
e where export(f1,f2){nrdef f1 ≡ e1 def f2 ≡ e2 def f3 ≡ e3}
```

exports the definitions of f_1 and f_2, but not f_3. Thus e can see f_1 and f_2, but not f_3; e_2 and e_3 can see f_1, f_2 and f_3; e_1 can see only definitions in the surrounding environment.

The environments hide(*namelist*) *env* and export(*namelist*) *env* are ill-formed if *namelist* is not a subset of the names defined by *env*.

6 Patterns

6.1 Introduction

Patterns are constructs that denote predicates but also make definitions. For example,

```
[[x.,y.]] → [y,x]
```

is a function that reverses pairs: <X,Y> → <Y,X>. The predicate position of the condition contains a pattern [[x.,y.]]; it denotes the predicate [[tt,tt]] that is true

for pairs <X, Y>, and it defines functions x and y such that x selects X from <X, Y> and y selects Y. Thus

$$([\![\texttt{x.}, \texttt{y.}]\!] \rightarrow [\texttt{y}, \texttt{x}]):\langle \texttt{X}, \texttt{Y} \rangle = [\texttt{y}, \texttt{x}]:\langle \texttt{X}, \texttt{Y} \rangle = \langle \texttt{y}:\langle \texttt{X}, \texttt{Y} \rangle, \texttt{x}:\langle \texttt{X}, \texttt{Y} \rangle \rangle = \langle \texttt{Y}, \texttt{X} \rangle$$

since the pattern is true for pairs and defines x = s1 and y = s2.

Generally, a pattern describes structures that satisfy its predicate *and* it produces a definition list of selector functions whose names occur in the pattern. If a name definition, f., occupies a given position P in a pattern (the "." indicates that f is being defined) and this f is applied to a value of the same "shape" as the pattern, then f selects an element from position P in the value. Note that, consistent with other definitions in FL, pattern defined names do not name components of arguments but rather name functions for selecting the components.

It is convenient to refer to a pattern as if it were the predicate it denotes; e.g., "the pattern [[x., y.]] is true for pairs" or "<X, Y> satisfies [[x., y.]]".

Every pattern defines at least one function and is built from elementary patterns *name.predicate* or *name.* (and from predicates) by a primitive pattern combining form ([[..]], ⇨, ⇦, pand, por and pcomp) or by a user defined pattern combining form.

6.2 Elementary patterns

The simplest patterns have the form *name.predicate*; the "." after *name* indicates that *name* is being defined. Such a pattern has the predicate *predicate* and defines def *name* ≡ id (id is the identity function). For example, x.isnum is a pattern that is satisfied by numbers and defines x = id so that x selects its entire argument, since this pattern has no substructure. Omitting the predicate is equivalent to using the predicate tt that is true for every value. For example, the pattern x. is always satisfied and defines x to be id.

6.3 Composite patterns

A composite pattern may be built from simpler patterns and predicates using *pattern combining forms*; these forms are: [[..]], ⇨, ⇦, pand, por and pcomp. Each of these pattern combining forms corresponds to a predicate combining form: [[..]], ↦, ↤, ∧, ∨ and ∘. ([[..]] does double duty as a pattern and predicate combining form.) The predicate of a composite pattern is just the combination of the predicates of its constituents combined with the corresponding predicate combining form. At least one of the components of a composite pattern must be a pattern, the others can be predicates. For example, [[x., isnum]] is a pattern that defines x = s1 whose predicate is [[tt, isnum]].

The definition resulting from a name f. in a pattern depends on its "position" in the pattern. Therefore, if a pattern is built from simpler ones with a pattern combining form, the definitions of the new pattern all come from the definitions made by its constituent patterns, but these are modified to reflect their "positions"

in the new pattern. For example, the pattern [[x., y.]] is built from x. and y. using [[..]]. Since x. occupies the s1 position in [[x., y.]], its contribution to the definitions of [[x., y.]] is its original definition, nrdef x ≡ id modified by its position, s1, in [[x., y.]], giving nrdef x ≡ id∘s1. The predicate of [[x., y.]] is just the predicate construction [[tt, tt]] of the individual predicates of x. and y., which are both tt.

In general, if $P_1, \ldots, P_n$ are patterns, $p_1, \ldots, p_n$ are their predicates, $DL_1, \ldots, DL_n$ are their definition lists, and nrdef $f_{ij} \equiv e_{ij}$ is any definition in DL_i, then the patterns built from the P_i with each of the primitive pattern combining forms have the following predicates and modified definitions (one definition for each function f_{ij} in DL_i):

Pattern	Predicate	Definitions (all nrdefs)
$[\![P_1, \ldots, P_n]\!]$	$[\![p_1, \ldots, p_n]\!]$	$f_{ij} \equiv e_{ij} \circ si$, si=s1, s2, ..
$P_1 \Rightarrow P_2$	$p_1 \mapsto p_2$	$f_{1j} \equiv e_{1j} \circ s1$, $f_{2j} \equiv e_{2j} \circ t1$
$P_1 \Leftarrow P_2$	$p_1 \mapsto p_2$	$f_{1j} \equiv e_{1j} \circ t1r$, $f_{2j} \equiv e_{2j} \circ r1$
P_1 pand P_2	$p_1 \wedge p_2$	$f_{1j} \equiv e_{1j}$, $f_{2j} \equiv e_{2j}$
P_1 por P_2	$p_1 \vee p_2$	$f_{1j} \equiv e_{1j}$, $f_{2j} \equiv e_{2j}$
P_1 pcomp P_2	$p_1 \circ p_2$	$f_{1j} \equiv e_{1j} \circ p_2$, $f_{2j} \equiv e_{2j}$

The brackets [[..]] are unique in that (a) they denote two different functions, the predicate combining form that combines predicates and the pattern combining form that combines patterns and (b) this is the only predicate or pattern combining form that has special syntax. Thus in $[\![x_1, \ldots, x_n]\!]$, [[..]] represents the pattern combining form if any x_i is a pattern, otherwise it represents the predicate combining form. All the other predicate and pattern combining forms have distinct function names.

A pattern built with primitive pattern combining forms (but not por) defines functions that never fail on any argument that satisfies the pattern.

6.3.1 Examples

Here are some patterns with their predicates, their definitions, an example of a value for which the pattern is satisfied (x_p denotes a value that satisfies p, x denotes any value) and the results of applying the pattern defined functions to this value:

Pattern Its predicate	Its definitions	Valid argument Function :: its result
[[x.p, [[y., z.q]], r]] [[p, [[tt, q]], r]]	x ≡ s1, y ≡ s1∘s2, z ≡ s2∘s2	<X_p, <Y, Z_q>, W_r> x::X_p, y::Y, z::Z_q
x.p⇻y.seqof:q p↦(seqof:q)	x ≡ s1, y ≡ t1	<X_p, $Y1_q$, $Y2_q$> x::X_p, y::<$Y1_q$, $Y2_q$>
x.seqof:isint⇦[[y.p, z.]] seqof:isint↤[[p, tt]]	x ≡ t1r, y ≡ s1∘r1, z ≡ s2∘r1	<1, 2, 3, <Y_p, Z>> x::<1, 2, 3>, y::Y_p, z::Z
x. pand [[y., z.isnum]] tt ∧ [[tt, isnum]]	x ≡ id, y ≡ s1, z ≡ s2	<Y, 3.1> x::<Y, 3.1>, y::Y, z::3.1
[[x., y.]] pcomp s1 [[tt, tt]]∘s1	x ≡ s1∘s1, y ≡ s2∘s1	<<X, Y>, 1, 2, 3> x::X , y::Y

6.4 Syntax of patterns

Patterns have the following syntax:

pat ::= *name.expr* | *name.* | [[*patlist*]] | {*expr* | *pat*} *patop pat* |
pat patop {*expr* | *pat*} | *name.pat* | pat(*expr*; *patlist*)

patlist ::= {*pat*, | *expr*,}* *pat* {, *pat* | , *expr*}*

patop ::= ⇻ | ⇦ | pand | por | pcomp | *expr*

User-defined pattern combining forms (*expr*s used as a *patop*, or *expr*s in the form pat(*expr*; *patlist*)), patterns of the form *name.pat*, and patterns built with pcomp are discussed in Section 12. Two defining occurrences of a given name (*name.*) in one pattern would create conflicting definitions for that name; therefore patterns with duplicate name definitions are syntactically ill-formed.

6.5 Use of patterns

Patterns can appear only in certain places: in the predicate position of a "syntactic" condition (i.e., in *pat* → f; g, but not in cond:<*pat*, f, g>), in the left sides of ordinary definitions, in type definitions, in expanded definitions and in lambda expressions (the use of patterns in lambda expressions and in expanded definitions is discussed in Sections 10 and 11). In each case there is a definite scope associated with the definitions made by the pattern.

6.5.1 Use of patterns in conditions

A condition with a pattern P in its predicate position is equivalent to a condition without a pattern:

P → f; g = p → (f where DL); (g where DL)
P → f = p → (f where DL)

where p is the predicate of P and DL is its definition list. The scope of the definitions made by a pattern in a condition is the arm or arms of the condition.

Although a pattern may appear as the predicate of a two-armed condition, for example, as in *pat* → *expr1* ; *expr2*, care must be taken when using a function f in *expr2* that is defined by *pat*, since *expr2*, and hence f, see arguments for which *pat* is false, and these arguments may not have the shape needed for f to have its intuitive meaning as a positional selector. In general, good practice is to apply pattern-defined functions only to arguments for which the pattern is true. For example, a misleading use of the pattern-defined function x occurs in

```
([[x., y.]] → x∘tl):<X, Y>
```

The intuitive meaning of x is a selector for X from pairs <X, Y>, but here x does not see <X, Y>. Instead it sees tl:<X, Y> = <Y>; therefore, since x = s1, it selects Y: (x∘tl):<X, Y> = Y. Care must also be exercised in using a pattern built with por, since its defined functions may fail even when applied to arguments that satisfy the pattern.

6.5.2 Patterns and predicates on the left side of definitions

Patterns or predicates may appear on the left side of definitions to indicate the domain of the defined function and (in the case of patterns) to define functions for accessing parts of its argument. For example, the definition

```
def sum_prod ← [[x.isnum, y.isnum, z.isnum]] ≡ (x*z) + (y*z)
```

indicates that sum_prod takes an argument of three numbers X, Y and Z in the form <X, Y, Z> and returns the result X*Z + Y*Z. This definition is equivalent to the following one with the pattern on the right:

```
def sum_prod ≡ [[x.isnum, y.isnum, z.isnum]] → (x*z) + (y*z);
                                    signal∘[~"sum_prod", ~"arg1", id]
```

The advantages of putting a domain pattern or predicate on the left sides of definitions are (a) uniform visibility of the domains of defined functions, and (b) automatic generation of the informative exception-making function that identifies the defined function name. If def f ≡ P → e is used (instead of f ← P ≡ e), then f:x produces an exception (when x does not satisfy P) that identifies a one-arm "cond" as the mis-applied function, rather than "f".

The above equivalence holds in general; the following two definitions define the same function

```
def f ← pattern   ≡   expression      is a definition equivalent to
      def f       ≡   pattern → expression; signal∘[~"f", ~"arg1", id]
```

The same equivalence holds with an expression (i.e., a predicate) on the left:

```
def f ← expr   ≡   expression      is a definition equivalent to
     def f     ≡   expr → expression; signal∘[~"f", ~"arg1", id]
```

7 User defined types

7.1 FL types

The notion of *type* in FL is based on predicates. The primitive FL types are sets that correspond to certain primitive total (i.e., defined for all normal values) predicates. For example, `isnum` is true for all numbers and false for all other values, `isint` is true for integers, `ischar` for chars, and `isseq` for sequences. Using predicate combining forms, the user can write total predicates for other classes of values. For example, `seqof:isint` is true for sequences of integers; `[[ischar, seqof:isint]]` is true for pairs whose first element is a char and whose second is a sequence of integers; and `isint↦isseq` is true for non-empty sequences whose first element is an integer.

7.1.1 Partial ordering of types

Primitive types are partially ordered by inclusion. E.g., integer (or `isint`) and real (or `isreal`) are disjoint types; both are subtypes of number (`isnum`) since `isint:x` or `isreal:x` implies `isnum:x`. Composite types, built by predicate combining forms from primitive types, inherit a partial order from the primitive types. For example, the type `[[eqto:"a", seqof:isint]]` is a subtype of `[[isstring, seqof:isnum]]`.

7.2 Type definitions

Users may introduce new kinds of objects and operations on such objects by making *type definitions*. For example, a new data type of complex numbers, represented by pairs of numbers, can be defined by

```
type complex ≡ [[isnum, isnum]]
```

This type definition defines a new function `mkcomplex` (for `makecomplex`) that transforms a pair of numbers `<x, y>` into a complex number `c`. Although `mkcomplex:<x, y>` = `c` is represented by the pair of numbers `<x, y>`, it is not this pair itself; it is a value of type complex. For example, `s1:c` produces an exception, not `x`. Here `mkcomplex` is the *constructor* for the type `complex`. This type definition also defines two other functions for the type, its *destructor*, `uncomplex` (for un-make `complex`), and its *predicate*, `iscomplex`. The constructor `mkcomplex` is defined only for values that satisfy the representation predicate on the right of the `type` definition (i.e., only for pairs of numbers); the type predicate `iscomplex` is defined for all values but is true only for complex numbers; and the destructor `uncomplex` is defined only for complex numbers and yields their representation `<x, y>`. Of course, many environments that define a type will use its destructor internally but hide it externally.

7.2.1 Defining functions on a user defined type

Consider the following definition of complex addition, `cadd`:

```
def cadd ← seqof:iscomplex ≡ mkcomplex ∘ pairadd ∘ α:uncomplex
    where {def pairadd ← seqof:[[isnum, isnum]] ≡ α:+ ∘ trans}
```

The left side of this definition assures that cadd is defined only on sequences of complex numbers. The right side first turns a sequence of complex numbers into a sequence of pairs of numbers by applying uncomplex to each complex number (α:uncomplex), then adds the sequence of pairs elementwise (pairadd), and finally turns the resulting pair of numbers into a complex number (mkcomplex). The argument of pairadd here is a sequence of real-imaginary pairs; trans (transpose) turns this into a pair of sequences, the first containing all the real parts, the second containing all the imaginary parts; α:+ then adds each of these two sequences, yielding the real and imaginary parts of the resulting complex number.

(The earlier Section 5.3 on Definitions, used cadd to extend the operator + to sum complex numbers as well as ordinary numbers. The predicate isnum can be similarly extended to include complex numbers.)

7.2.2 Patterns in type definitions

It might be desirable to define selector functions real and imag on complex numbers, where real:c is the real part of the complex number c and imag:c is its imaginary part. For example, one could write

```
def real ≡ s1∘uncomplex
def imag ≡ s2∘uncomplex
```

Since it is common to want such selectors, FL allows a pattern (instead of a predicate) on the right side of a type definition to accomplish the same goal. In the type definition

```
type complex ≡ [[real.isnum, imag.isnum]]
```

mkcomplex, iscomplex and uncomplex are defined as before, and real and imag are defined as above — the functions defined by the pattern are composed with the destructor of the type, uncomplex. The representation predicate is the pattern's predicate, [[isnum, isnum]].

7.3 Local and global types

Two type definitions in different scopes can define two types with identical names. Although the scope rules prevent any ambiguity in the use of the identically named type functions that result, the scope rules do not prevent functions of one type from being applied to objects of the other type. To prevent type errors from arising in this circumstance, the constructors defined by two identical occurrences, T_1 and T_2, of the type definition type t ≡ p, must produce *different* objects, even when these are built from the same base object x. This means, for example, that applying the destructor unt_1 of T_1 to an object mkt_2:x (built with the constructor of T_2) will produce an exception, not x.

However, when type definitions occur in library environments (see Section 14.2) they become *global* type definitions and multiple copies of such environments result

in copies of these definitions that produce *identical* type functions. To summarize: type definitions that are not in library environments are *local* type definitions; the constructors defined by two identical local type definitions (in different scopes) produce different objects from identical arguments. Type definitions in library environments are *global* type definitions; the constructors defined by two copies of a global type definition produce identical objects from identical arguments. For more discussion of local and global types and of type safety, see Section 13.

7.4 Syntax of type definitions; functions defined by them

The syntax of type definitions, *typedef*, is

typedef ::= **type** *name* ≡ { *expr* | *pat* }

A type definition of this form defines mk*name*, un*name*, is*name*, plus all the functions defined by the pattern *pat*, if the right side is a pattern. The function mk*name* is defined for any x for which the representation predicate on the right (the expression or the predicate of the pattern) is true and yields an element of type *name* . The function is*name* is defined for all normal values and is true only for values of type *name*. The function un*name* is defined only for values of type *name* and yields values for which the representation predicate is true. Any function name defined by *pat* is defined for any value of type *name* and yields the corresponding element of its representation.

7.5 Example of a complete type definition: Stack

This section presents an annotated example of the definition and use of a new type, stack, along with definitions of some functions on stacks. One possible type definition for stack is

```
type stack ≡ isseq
```

in which stacks are represented by sequences. The constructor mkstack converts any sequence into a stack; the type predicate isstack is true for all stacks and false for all other values; the destructor unstack converts any stack into the sequence that represents it.

7.5.1 Definitions of isemptystack, top, pop, push, mkemptystack

```
def isemptystack←isval ≡ isstack → isnull∘unstack; FF
```

Defined for all values, since isval, the domain predicate, is always true and isstack is total. True only for the empty stack.

```
def top←isstack ∧ ¬:isemptystack ≡ s1∘unstack
```

Defined only on non-empty stacks; produces the top of a stack. The domain predicate on the left side is redundant since unstack fails on non-stacks and s1∘unstack fails for empty stacks; however, its use guarantees a "top" exception rather than a "cond" exception if top is mis-applied (although compile-time type checking will normally indicate potential errors); see Section 6.5.2. It is good practice to use the left side of definitions to indicate the domains of defined functions in a standard way.

def pop ← isstack ∧ ¬:isemptystack ≡ mkstack∘tl∘unstack

Deletes the top of a non-empty stack.

def push ← [[a.isval, stack.isstack]] ≡ mkstack∘al∘[a, unstack∘stack]

Argument = x = <*value*, *stack*>. Appends a:x = s1:x = *value* on the left of the sequence that represents stack:x = s2:x = *stack* (*append left*, al, is the primitive such that al:<1, <2, 3, 4>> = <1, 2, 3, 4>). The resulting sequence is converted to a stack.

def mkemptystack ← isval ≡ mkstack ∘ []

Produces an empty stack when applied to any argument.

In normal practice the definitions of type stack might be enclosed in a where clause that hides the definitions of unstack and mkstack and exports the definitions of isstack, mkemptystack, isemptystack, push, top and pop (see Section 5.5).

8 Input, output and persistent files

8.1 The (Value, History) semantics of FL

FL treats operations on external (I/O) devices and persistent files as real events that take place in a definite chronological sequence. To treat these operations this way and at the same time to require that FL programs be functions means that FL programs must have a different functionality (i.e., domain and range) than has been indicated in the discussion up to this point.

So far, FL functions have been presented as if they simply mapped values into values, but this is not the whole truth. Actually, every FL function f maps a pair (*value*, *history*) into another such pair; however, the history component is implicit and not accessible to the user, thus at each point in the evaluation of an expression there is a unique history. The history component models the state of I/O devices and the file system. However, most FL functions f do not involve I/O and hence are independent of the history component, i.e., for any value x there is a y such that, for all histories h:

$$f(x, h) = (y, h)$$

In that case, one can think of f as mapping values into values, as if f:x = y, even

though this equation is really shorthand for: for all h, f:(x,h) = (y,h). (The notation in this equation is not part of the language; in the following, all notation involving a history h is not part of the FL language.) Even when a function f does I/O,

$$f(x, h_1) = (y, h_2)$$

one may informally think of f as mapping x, with the history h_1, into y with the side effect of changing h_1 into the result history h_2 that records the new I/O status. Every function f that depends on the history also changes it by adding a record of the event.

8.1.1 Order of evaluation of expressions

Since the evaluation of an application may have a side effect, evaluation must take place in a definite order. All expressions are evaluated from left to right; in non-I/O expressions that do not fail this order is not important, but in expressions that do I/O or produce an exception the order is important. For example, in evaluating

```
[in, in, in]:"kbd"
```

the first step yields <in:"kbd", in:"kbd", in:"kbd">; this is evaluated from left to right, giving <'c_1, 'c_2, 'c_3>, where 'c_1 is the first char from the keyboard, 'c_2 the second and 'c_3 the third (thus the result is a string and could be written "$c_1c_2c_3$"). More precisely, the evaluation of <in:"kbd", in:"kbd", in:"kbd"> starting with the history h takes place by evaluating in("kbd",h) = ('c_1,h_1) and then in("kbd",h_1) = ('c_2,h_2) and finally in("kbd",h_2) = ('c_3,h_3).

The order of evaluation and the nature of the I/O primitives assure that there can be only one history at any point in the course of evaluation.

8.2 Primitive functions for I/O and file access: in, out, get, put

There are four primitive functions concerned with I/O. The first two provide input and output to external devices: in:*devicename* returns the next input from the named device, out:<*devicename*, x> returns <*devicename*, x> as its value and transmits x to the named device. The last two functions are similar but are concerned with accessing and storing files in a persistent file system; the value of get:*filename* is the named file (which can be any value), put:<*filename*, x> returns <*filename*, x> as its result and makes x the value of the named file. *Devicename* and *filename* are strings. Files are just FL values, e.g., a text file might be a sequence of strings or a value of type text.

8.2.1 Example of a program with I/O

Consider a program, prompt, where the value of prompt:*message* is the character typed at the keyboard in response to the appearance of *message*, which prompt places on the screen.

```
def prompt ← msg.isstring ≡ in ∘ ~"kbd" ∘ out ∘ [~"scr", msg]
```

The pattern on the left side assures that prompt fails unless it gets a string; it defines msg = id. Here is a sample evaluation:

```
prompt:"type a char"
        = (in∘~"kbd"∘out∘[~"scr", msg]):"type a char"
        = (in∘~"kbd"∘out):<"scr", "type a char">
                since msg = id
        = (in∘~"kbd"):<"scr", "type a char">
                and "type a char" appears on screen
        = in:"kbd"
        = `c
                the char for the character typed at the keyboard
```

Part II
Other features of FL

Part 2 describes features of the language that have not been fully discussed up to this point. None is essential; some provide important conveniences and notation for writing programs in a more readable and type-robust form; others provide certain technical features that will be of interest only to more experienced FL programmers.

9 Additional notation for expressions

9.1 Infix notation

As an aid to readability the FL language provides a uniform infix notation. For example 3+4 is a valid expression whose value is +:<3, 4> = 7. In fact, 3+4 is just shorthand for +:<3, 4>. Similarly f∘g is an infix expression that is shorthand for ∘:<f, g>. In general, for any three expressions, $expr_1$, $expr_2$, $expr_3$, the following is an *infix expression*:

$expr_1$ $expr_2$ $expr_3$

and has the same meaning as the expression

$expr_2$:<$expr_1$, $expr_3$>

The infix rule is overloaded to deal with infix patterns (written with binary pattern combining forms):

P_1 *expr* P_2

denotes a pattern if P_1 and P_2 are patterns or expressions and at least one is a pattern. (Recall that patterns are syntactically distinct from expressions.) The

infix rule asserts that this form is equivalent to the following pattern (whose meaning is fully explained in Section 12.1)

pat(*expr*; P_1, P_2)

This pattern denotes the result of combining P_1 and P_2 with the pattern combining form denoted by *expr*.

9.2 Prime and lift

It is convenient to be able to transform a function f that operates on a sequence of objects into a function f′ that operates on a sequence of functions, where

$$f':\langle f_1,\ldots,f_n\rangle = f\circ[f_1,\ldots,f_n]$$

The arguments of f′ are then the functions that produce the arguments of f. *Priming* a function expression e accomplishes this transformation: e′ abbreviates lift:e, where lift is a primitive combining form that has the following effect:

$$f':\langle f_1,\ldots,f_n\rangle = \texttt{lift:}f:\langle f_1,\ldots,f_n\rangle = f\circ[f_1,\ldots,f_n]$$

Thus lift:f:x = f ∘ cons:x. Priming provides convenient function level infix expressions based on the standard interpretation of infix:

(f op′ g):x = f:x op g:x

This property of primed infix operators follows from the standard interpretation of infix expressions:

$$(g\ f'\ h):x = (f':\langle g,h\rangle):x = (f\circ[g,h]):x = f:\langle g:x,\ h:x\rangle = (g:x\ f\ h:x)$$

In general, if e is an expression, then e′ is an expression that abbreviates lift:e. And if x, y and z are expressions, then x y′ z is an expression equivalent to y∘[x, z]. Priming of expressions is also useful in abstracting variables from an expression (this is discussed in Sections 10.1 and 10.3 on lambda expressions).

9.3 Raised operators

It is convenient to have some operators behave either as unlifted or lifted functions, depending on whether their arguments are objects or functions. For example, it is desirable to have both 3+4 and (s1+s2):<3, 4> produce 7. However, since s1+s2 is just shorthand for +:<s1, s2>, + needs to be defined to lift its arguments when applied to functions. This is accomplished by defining + to be

def + ≡ seqof:isfunc ∧ ¬:isnull → lift:add; add

where add sums sequences of numbers.

Functions f that can be applied either to a sequence of objects or (in its lifted form) to a sequence of functions, such as +, are called *raised functions* and can be expressed as f = raise:g where g is some function that is defined only on sequences of objects (i.e., non-function values). The functional raise satisfies

$$\texttt{raise:f} = \texttt{seqof:isfunc} \wedge \neg\texttt{:isnull} \rightarrow \texttt{lift:f; f}$$

See Section 10.3 for the definition of `raise`. The primitive raised functions of FL are the following:

`+` `-` `*` `÷` `le` `ge` `lt` `gt`

where `le`, `ge`, `lt` and `gt` are, respectively, *less than or equal, greater than or equal, less than,* and *greater than.*

9.4 Special primed forms; multiple primes

A few combining forms `F` may be written using special forms in addition to the standard form $\texttt{F:<f}_1,\ldots,\texttt{f}_n\texttt{>}$. These are construction: $\texttt{[..]} = \texttt{cons:<..>}$, predicate construction: $[\![..,\texttt{p}_i,..]\!] = \texttt{pcons:<..,p}_i\texttt{,..>}$, condition: $\texttt{p} \rightarrow \texttt{f; g} = \texttt{cond:<p,f,g>}$, and constant: $\bar{}\texttt{x} = \texttt{K:x}$. The use of prime with these special forms is permitted for clarity and uniformity. (See Sections 10.1 and 10.3 for examples of their use.) Thus, for example, $\texttt{cons}'\texttt{:<f,g>}$ may be written as $\texttt{['f,g]}$, $\texttt{pcons}'\texttt{:<f,g>}$ as $[\![\texttt{'f,g}]\!]$, $\texttt{cond}'\texttt{:<p,f,g>}$ as $\texttt{p} \rightarrow' \texttt{f; g}$, and $\texttt{K}'\texttt{:x}$ as $\bar{}'\texttt{x}$ (although $\texttt{K}'\texttt{:x} = \texttt{lift:K:x}$ is meaningful only if `x` is a sequence of functions: $\texttt{K}'\texttt{:<f}_1,\ldots,\texttt{f}_n\texttt{>} = \texttt{K}\circ\texttt{[f}_1,\ldots,\texttt{f}_n\texttt{]}$).

The application symbol : may be primed. Colon : may be thought of as an abbreviation for `apply`, since $\texttt{f:x} = \texttt{apply:<f,x>} = \texttt{f apply x}$. Therefore

$$(\texttt{f}:'\texttt{g}):\texttt{x} = (\texttt{f:x}):(\texttt{g:x})$$

since $(\texttt{f}:'\texttt{g}):\texttt{x} = (\texttt{apply}'\texttt{:<f,g>}):\texttt{x} = (\texttt{apply}\circ\texttt{[f,g]}):\texttt{x} = \texttt{apply:<f:x, g:x>}$.

If `P` is a pattern, then $\texttt{P} \rightarrow' \texttt{f; g}$ is ill-formed, since `<P,f,g>` is not an expression, hence $\texttt{cond}'\texttt{:<P,f,g>}$ is not either. Similarly, $[\!['..,\texttt{P},..]\!]$ is ill-formed if `P` is a pattern.

Multiple primes are possible both for the standard and non-standard forms of functions: if `e` is any expression, so is $\texttt{e}' = \texttt{lift:e}$, thus $\texttt{f}'' = (\texttt{f}')' = \texttt{lift:(lift:f)}$. Therefore $\texttt{["f,g]} = \texttt{cons}''\texttt{:<f,g>}$, $\texttt{p} \rightarrow''' \texttt{f; g} = \texttt{cond}'''\texttt{:<p,f,g>}$, and $\bar{}''\texttt{x} = \texttt{K}''\texttt{:x}$.

10 Lambda expressions

10.1 Why does FL have both combining forms and lambda expressions?

The function level style (using patterns and condition) is often hard to use for defining new combining forms (higher order functions). Of course the object of this style is to provide an initial set of combining forms that is suitable for ordinary programming. Defining new combining forms can therefore be thought of as meta-programming, or extending the basic FL language. But it will often be desirable to define new combining forms for new data types.

It is hard to define higher order functions in FL because one must abstract arguments. For example, consider the question of defining the combining form α,

apply-to-all (α:f applies f to every element of a sequence argument). For any function f the following equation should hold

(1) α:f = isnull → []; al∘[f∘s1, (α:f)∘tl]

The goal of abstraction is to eliminate f and obtain an expression for α alone. Recall that al (*append left*) appends its first argument onto the left of the sequence that is its second argument. Here is the corresponding abstracted equivalent of (1):

(2) α = ~isnull →' ~[]; ~al ∘' [' id ∘' ~s1, α ∘' ~tl]

The reader can check that α:f as given by (2) reduces to the right side of (1). (Recall the effect of prime (') and constant (~): (p →' q; r):f = p:f → q:f; r:f and ['.., g, ..]:f = [.., g:f, ..] and (g ∘' h):f = g:f ∘ h:f and ~c:f = c.)

Although it is possible for the user to perform this abstraction, the resulting definition of α is not as clear and simple as one would like. (Furthermore, (2) needs a delay operator to be correct.) Lambda expressions are provided for defining higher order functions so that the user does not need to do abstraction himself. Compare (2) with the definition of α using a lambda expression:

(3) def α = λ(f.)(isnull → []; al∘[f∘s1, delay:α:f∘tl])

The effect of applying the lambda expression on the right to an argument g is to replace occurrences of f by g. (The full meaning of lambda expressions is given below.) The definition of α in (3) also deals with the strictness issue (equations (1) and (2) ignore that issue). The function delay has the effect of preventing the evaluation of α:f until it gets an argument, i.e., delay:α:f is a normal form (no reduction rule applies to it) and delay:α:f:x = α:f:x. (Without the use of delay, (2) defines α to be ~⊥.)

10.2 The general form and meaning of lambda expressions in FL

A lambda expression has the form:

λ(*pattern*)*expression*

The value of a lambda expression is a function. It is always in normal form, even if *expression* is not.

If x satisfies the pattern P, the result of (λ(P)E):x is $E_{f \to f:x}$, which is E with each occurrence of a name f defined by P replaced by f:x, i.e., by the part of x that f selects. If x does not satisfy P, the exception signal:<"lambda", "arg", x> is the result; but if P's predicate p produces an exception p:x, then that is the result. Since a lambda expression, λ(P)E, fails when applied to an argument that does not satisfy its pattern P, the functions that P defines never fail in computing $E_{f \to f:x}$ (provided that P is built with primitive pattern combining forms other than por).

10.3 Examples

Suppose one wishes to define map2 such that map2:<f, g, h> = [f∘g, f∘h]. A possible definition using a lambda expression is:

```
def map2 ≡ λ([[f., g., h.]])[f∘g, f∘h]
```

Here is the evaluation of map2:<r, s, t>:

```
map2:<r, s, t>  =  (λ([[f., g., h.]])[f∘g, f∘h]):<r, s, t>
                =  [f:<r, s, t> ∘ g:<r, s, t>, f:<r, s, t> ∘ h:<r, s, t>]
                =  [r∘s, r∘t]
```

The pattern [[f., g., h.]] defines f = s1, g = s2, h = s3, and it is satisfied by <r, s, t>.

As another example, consider the definition of raise, the function that lifts its argument function when its second argument is a sequence of functions:

```
def raise ≡ λ(f.isfunc)(seqof:isfunc ∧ ¬:isnull) → lift:f; f
```

A final example: suppose one wishes to define the primitive function C ("Curry") such that for any function f and values x and y:

```
C:f:x:y = f:<x, y>
```

(Application associates to the left.) Without lambda expressions it is necessary to abstract away all three arguments, yielding

```
C = K ∘″ [″ ~K, ~(~id)]
```

This combinatorial abstraction can be avoided by using lambda expressions:

```
def C ≡ λ(f.isfunc)λ(x.)λ(y.)f:<x, y>
```

This definition requires C's first argument to be a function. The evaluation of C:g:3:4, where g is a function, proceeds as

```
C:g:3:4  =  ((λ(f.isfunc)λ(x.)λ(y.)f:<x, y>):g):3:4
         =  (λ(x.)λ(y.)g:<x, y>):3:4      since g is a function
         =  (λ(y.)g:<3, y>):4
         =  g:<3, 4>
```

The patterns in the lambda expressions of C define f = x = y = id; thus f is replaced in λ(x.)λ(y.)f:<x, y> by f:g = id:g = g. If g is not a function it fails to satisfy the first pattern, and the first step produces an exception (which persists). The last two patterns are satisfied by any value.

10.4 Lambda patterns on the left side of definitions

In a style similar to the use of patterns on the left side of definitions, there is an alternative form for the above definition of map2 which puts the lambda information on the left side (but without the λ):

```
def map2([[f.,g.,h.]]) ≡ [f∘g, f∘h]
```

Similarly, using the alternative form for the definition of C:

```
def C(f.isfunc)(x.)(y.) ≡ f:<x,y>
```

10.5 General form of definitions

In general, a definition can have the form

$$\underline{\texttt{def}}\ \texttt{f}(\texttt{P}_1)(\texttt{P}_2)\ldots(\texttt{P}_n) \leftarrow \texttt{P}_{n+1} \equiv \texttt{E}$$

for $n \geq 0$, where f is a function name (or a pattern — see the next section), each $\texttt{P}_i$ is a pattern (but $\texttt{P}_{n+1}$ may be a predicate) and E is an expression. All the following definitions are equivalent:

$$\underline{\texttt{def}}\ \texttt{f}(\texttt{P}_1)\ldots(\texttt{P}_n) \leftarrow \texttt{P}_{n+1} \equiv \texttt{E}$$
$$\underline{\texttt{def}}\ \texttt{f}(\texttt{P}_1)\ldots(\texttt{P}_n) \equiv \texttt{P}_{n+1} \rightarrow \texttt{E};\ \texttt{err}_{n+1}$$
$$\underline{\texttt{def}}\ \texttt{f}(\texttt{P}_1)\ldots(\texttt{P}_{n-1}) \equiv \texttt{P}_n \rightarrow \lambda_u(\texttt{P}_n)(\texttt{P}_{n+1} \rightarrow \texttt{E};\ \texttt{err}_{n+1});\ \texttt{err}_n$$
$$\vdots$$
$$\underline{\texttt{def}}\ \texttt{f} \equiv \texttt{P}_1 \rightarrow \lambda_u(\texttt{P}_1)(\ldots(\texttt{P}_n \rightarrow \lambda_u(\texttt{P}_n)(\texttt{P}_{n+1} \rightarrow \texttt{E};\ \texttt{err}_{n+1});\ \texttt{err}_n)\ldots);\ \texttt{err}_1$$

where $\lambda_u(\texttt{P})\texttt{E}$ is the same as $\lambda(\texttt{P})\texttt{E}$ except that no test is made that the argument satifies P, and the exception-producing functions $\texttt{err}_{n+1}, \ldots, \texttt{err}_1$ are the following for each i=1,..,n+1:

$$\texttt{err}_i \quad = \quad \texttt{signal}\circ[\text{`"f"}, \text{`"argi"}, \texttt{id}]$$

11 Expanded definitions

The definitions discussed up to this point provide for left sides that (a) indicate the domain of the defined function (**def** f ← {*pattern* | *expr*} ≡ *expr*) and (b) indicate the domain with lambda abstraction of one or more variables (**def** f(*pattern*) ≡ *expr*). Expanded definitions provide for left sides that include these facilities and also allow the user to: (c) indicate the range of the defined function (a predicate that must hold for its results) and (d) make parallel definitions of several functions.

The syntax of an expanded definition is

exdef ::= **exdef** *pat argexp* ≡ *expr*
argexp ::= *empty* | $(pat)^+$ | $(pat)^* \leftarrow \{pat \mid expr\}$

Here *pat* is a pattern; *argexp* is the same as for ordinary definitions and may be empty, or have zero or more lambda patterns (*pat*) possibly followed by ← *pat*. The expanded definition

exdef *pat argexp* ≡ *expr*

is equivalent to a list of definitions, one for each function name f defined by *pat*:

$$\underline{\texttt{def}}\ \texttt{f}\ \textit{argexp} \equiv (\textit{pat} \rightarrow \texttt{f};\ \texttt{signal}\circ[\tilde{}\texttt{"f"},\ \tilde{}\texttt{"range"},\ \texttt{id}])\circ \textit{expr}$$

Each of these definitions checks that the result of *expr* satisfies *pat* and if so it selects the part of *expr*'s result from the position that `f.` has in *pat*; this is the value of `f` for this definition (note that the `f` on the right is defined by *pat*; it selects the appropriate part of the result of *expr* and is not the function `f` being defined on the left). If the result of *expr* does not satisfy *pat*, the result is an exception indicating a range error for `f`. For example, the following defines $\texttt{f} = \texttt{s1}\circ[\texttt{r},\texttt{s}]$ and $\texttt{g} = \texttt{s2}\circ[\texttt{r},\texttt{s}]$; if `r` and `s` are always defined and have no side effects, then $\texttt{f} = \texttt{r}$ and $\texttt{g} = \texttt{s}$.

$$\underline{\texttt{exdef}}\ [\![\texttt{f.},\texttt{g.}]\!] \equiv [\texttt{r},\texttt{s}]$$

Since the pattern defines `f` and `g`, this expanded definition is equivalent to the two following definitions:

$$\underline{\texttt{def}}\ \texttt{f} \equiv ([\![\texttt{f.},\texttt{g.}]\!] \rightarrow \texttt{f};\ \texttt{signal}\circ[\tilde{}\texttt{"f"},\ \tilde{}\texttt{"range"},\ \texttt{id}])\circ[\texttt{r},\texttt{s}]$$
$$\underline{\texttt{def}}\ \texttt{g} \equiv ([\![\texttt{f.},\texttt{g.}]\!] \rightarrow \texttt{g};\ \texttt{signal}\circ[\tilde{}\texttt{"g"},\ \tilde{}\texttt{"range"},\ \texttt{id}])\circ[\texttt{r},\texttt{s}]$$

These are not recursive definitions; e.g., the `f` on the right side of the definition of `f` is defined to be `s1` by the pattern $[\![\texttt{f.},\texttt{g.}]\!]$. After eliminating the patterns and replacing the functions they define by their definitions, these become

$$\begin{aligned}
\underline{\texttt{def}}\ \texttt{f} &\equiv ([\![\texttt{tt},\texttt{tt}]\!] \rightarrow \texttt{s1};\ \texttt{signal}\circ[\tilde{}\texttt{"f"},\ \tilde{}\texttt{"range"},\ \texttt{id}])\circ[\texttt{r},\texttt{s}] \\
&\equiv \texttt{s1}\circ[\texttt{r},\texttt{s}] \qquad \text{since } [\![\texttt{tt},\texttt{tt}]\!] \text{ always true if } [\texttt{r},\texttt{s}] \text{ defined} \\
\underline{\texttt{def}}\ \texttt{g} &\equiv ([\![\texttt{tt},\texttt{tt}]\!] \rightarrow \texttt{s2};\ \texttt{signal}\circ[\tilde{}\texttt{"g"},\ \tilde{}\texttt{"range"},\ \texttt{id}])\circ[\texttt{r},\texttt{s}] \\
&\equiv \texttt{s2}\circ[\texttt{r},\texttt{s}]
\end{aligned}$$

(The primitive `tt` always produces `true`.)

11.1 Using expanded definitions to indicate the range of a function

Expanded definitions can be used to define a function `f = expr` in which its domain and range are both succintly expressed in a way that guarantees that `f` yields an exception if it is applied to an argument outside of its domain or if its result is outside of the stated range. (For example, one might expect `expr` to receive an integer and produce a char; one might want `f` to fail if it did not.) If $\texttt{p}_{dom}$ and $\texttt{p}_{range}$ are the desired domain and range predicates, the following expanded definition enforces them:

$$\underline{\texttt{exdef}}\ \texttt{f.p}_{range} \leftarrow \texttt{p}_{dom} \equiv \texttt{expr}$$

Since the pattern `f.p` on the left defines only one function, `f`, this expanded definition is equivalent to the single definition

$$\underline{\texttt{def}}\ \texttt{f} \leftarrow \texttt{p}_{dom} \equiv (\texttt{f.p}_{range} \rightarrow \texttt{f};\ \texttt{signal}\circ[\tilde{}\texttt{"f"},\ \tilde{}\texttt{"range"},\ \texttt{id}])\circ\texttt{expr}$$

The `f` after the $\rightarrow$ is defined by the pattern $\texttt{f.p}_{range}$ to be `id`; thus the definition is equivalent to

$$\underline{\texttt{def}}\ \texttt{f} \leftarrow p_{dom} \equiv (p_{range} \rightarrow \texttt{id};\ \texttt{signal}\circ[\tilde{}\texttt{"f"},\ \tilde{}\texttt{"range"},\ \texttt{id}])\circ\texttt{expr}$$

The function f produces signal:<"f", "arg1", x> if its argument x does not satisfy p_{dom} or signal:<"f", "range", expr:x> if expr:x does not satisfy p_{range}.

11.2 Using an expanded definition to make multiple definitions

It is sometimes convenient to condense several definitions into one. Expanded definitions express such multiple definitions. For example,

$$\underline{\texttt{exdef}}\ \texttt{f.}\Rightarrow\texttt{g.} \equiv [\texttt{r}, \texttt{s}, \texttt{t}]$$

is equivalent to the definitions

$$\underline{\texttt{def}}\ \texttt{f} \equiv (\texttt{f.}\Rightarrow\texttt{g.} \rightarrow \texttt{f};\ \texttt{signal}\circ[\tilde{}\texttt{"f"},\ \tilde{}\texttt{"range"},\ \texttt{id}])\circ[\texttt{r}, \texttt{s}, \texttt{t}]$$
$$\underline{\texttt{def}}\ \texttt{g} \equiv (\texttt{f.}\Rightarrow\texttt{g.} \rightarrow \texttt{g};\ \texttt{signal}\circ[\tilde{}\texttt{"g"},\ \tilde{}\texttt{"range"},\ \texttt{id}])\circ[\texttt{r}, \texttt{s}, \texttt{t}]$$

These definitions are equivalent to the following after eliminating patterns and replacing the functions they define by their definitions:

$$\underline{\texttt{def}}\ \texttt{f} \equiv (\texttt{tt}\mapsto\texttt{tt} \rightarrow \texttt{s1};\ \texttt{signal}\circ[\tilde{}\texttt{"f"},\ \tilde{}\texttt{"range"},\ \texttt{id}])\circ[\texttt{r}, \texttt{s}, \texttt{t}]$$
$$\underline{\texttt{def}}\ \texttt{g} \equiv (\texttt{tt}\mapsto\texttt{tt} \rightarrow \texttt{tl};\ \texttt{signal}\circ[\tilde{}\texttt{"g"},\ \tilde{}\texttt{"range"},\ \texttt{id}])\circ[\texttt{r}, \texttt{s}, \texttt{t}]$$

The predicate tt↦tt must be true whenever [r, s, t] is defined (p↦q is true for non-empty sequences $\langle x_1, \ldots, x_n\rangle$ for which p:x_1 and q:$\langle x_2, \ldots, x_n\rangle$ are both true). It is easy to see that f = s1∘[r, s, t] and g = tl∘[r, s, t] and that f = r and g = [s, t] if r, s and t are always defined and have no side effects.

12 Other forms of patterns

There are several kinds of patterns that have not been discussed so far. To describe the predicate and the set of definitions associated with each of them, in the following let PAT be a pattern and let $P_1, \ldots, P_n$ be either patterns or predicates (i.e., expressions) with at least one being a pattern, let Pred(PAT) and Pred(P_i) be the predicates corresponding to PAT and P_i (where Pred(P_i) = P_i if P_i is a predicate), and let Defs(PAT) and Defs(P_i) be the definitions made by PAT and P_i (where Defs(P_i) is empty if P_i is a predicate).

12.1 The general form and meaning of patterns

A pattern combining form F_{pat} is a function that can combine patterns/predicates $P_1, \ldots, P_n$ to form a new pattern PAT that is denoted by the construct

$$\underline{\texttt{pat}}(F_{pat};\ P_1, \ldots, P_n)$$

This new pattern PAT makes definitions that are modifications of the definitions made by $P_1, \ldots, P_n$. Its predicate Pred(PAT) and the modifier functions $\texttt{modifier}_i$ needed to modify the definitions of each P_i are computed by applying the pattern

combining form F_{pat} to the sequence $\langle Pred(P_1), \ldots, Pred(P_n)\rangle$ (F_{pat} cannot be applied to the patterns themselves since patterns are not first class values):

$$F_{pat}{:}\langle Pred(P_1), \ldots, Pred(P_n)\rangle = \langle Pred(PAT), \langle modifier_1, \ldots, modifier_n\rangle\rangle$$

where, if def $f_{ij} \equiv e_{ij}$ is a definition belonging to $Defs(P_i)$, then

$$\text{def } f_{ij} \equiv e_{ij} \circ modifier_i$$

is a definition belonging to Defs(PAT); of course, if P_i is a predicate, then $Defs(P_i)$ is empty and contributes no definitions to Defs(PAT).

12.1.1 Example

Recall that the pattern x.p⇨y.q is true for sequences that satisfy p↦q (non-empty sequences whose first element satisfies p and whose tail satisfies q) and defines x = s1 and y = tl. Therefore the pattern combining form ⇨ can be defined as follows:

$$\text{def } \Rightarrow([\![f.isfunc, g.isfunc]\!]) \equiv \langle f \mapsto g, \langle s1, tl\rangle\rangle$$

Thus ⇨:⟨r, s⟩ = ⟨r↦s, ⟨s1, tl⟩⟩. For example, the pattern f.isnum⇨g. is equivalent, by the pattern infix rule, to the pattern

pat(⇨; f.isnum, g.)

The pattern combining form ⇨ applied to the predicates of the two constituent patterns yields:

⇨:⟨isnum, tt⟩ = ⟨isnum↦tt, ⟨s1, tl⟩⟩

Thus the pattern f.isnum⇨g. has the predicate isnum↦tt, and it makes the definitions

nrdef f ≡ id∘s1
nrdef g ≡ id∘tl

since f.isnum defines f = id and g. defines g = id.

Since ⇨ is just the function [↦, ~⟨s1, tl⟩], the pattern f.isnum⇨g. could be written as f.isnum [↦, ~⟨s1, tl⟩] g.. The infix rule for patterns treats this as pat([↦, ~⟨s1, tl⟩]; f.isnum, g.), which yields the same pattern as above.

12.1.2 User defined pattern combining forms

Since pattern combining forms are just functions, they may be defined by the user, e.g., for a new data type. Having defined an n-ary pattern combining form F_{pat}, it can be used to build a new pattern with the form $pat(F_{pat}; P_1, \ldots, P_n)$, or if F_{pat} is a binary pattern combining form, it can be used in an infix pattern expression: $P_1\ F_{pat}\ P_2$.

An n-ary pattern combining form F_{pat} must map a sequence of n functions into a pair whose first element is a function (the predicate of the new pattern) and whose second element is a sequence of n functions (the modifier functions).

12.2 Patterns of the form *name.pattern*

A pattern may have the form *name.pattern*; if `f.PAT` is such a pattern, then its predicate is `Pred(PAT)` and its definitions are `def f ≡ id` plus `Defs(PAT)`. Of course, this means that *pattern* must not define *name*; otherwise *name.pattern* would define *name* twice. Patterns of this form allow one to "name" the whole as well as its parts. For example, if `x`, `y` and `z` are defined by the pattern `x.[[y., z.]]` and are applied to a pair `X = <Y, Z>`, `x` selects its argument `X`, `y` selects `Y`, and `z` selects `Z`.

12.3 The pattern combining form pcomp

The primitive pattern combining form `pcomp` may be defined as follows:

```
def pcomp([[f.isfunc, g.isfunc]]) ≡ <f∘g, <g, id>>
```

Since P_1 `pcomp` P_2 = `pat(pcomp;` P_1, P_2`)`, this means that `Pred(`P_1 `pcomp` P_2`)` = `Pred(`P_1`)∘Pred(`P_2`)` and `Defs(`P_1 `pcomp` P_2`)` = definitions of P_1 modified by `Pred(`P_2`)` plus the definitions of P_2. Usually P_2 is just a function `g`, thus `Pred(`P_2`)` = `g` and `Defs(`P_2`)` is the empty set. For example, `[[x., y.]] pcomp s1` is a pattern whose predicate is `[[tt, tt]]∘s1` (true if the first element of its argument is a pair); it defines `x = s1∘s1` and `y = s2∘s1`.

13 Type safety; local and global types

The type analysis system of FL is guaranteed to indicate, either at compile time or run time, that a function is being applied to (or could be applied to) an inappropriate argument. Although there is no guarantee that all potential type errors will be detected at compile time, it is expected that most type errors will be found then.

Two user defined types can have the same type name `t`; however, their type definitions must then be in different scopes, otherwise there would be two clashing definitions of the constructor `mkt`, and the program would be ill-formed. The scope rules prevent any ambiguity about uses of these identically named functions just as they do for all function definitions; however, the scope rules do not apply to the *objects of the new types*. Thus, for example, the destructor function `unt` of one type definition can be applied to an argument constructed by the `mkt` function of another type definition (both defining different types named `t`):

```
((unt where {type t ≡ isval})∘(mkt where {type t ≡ p})):x   (1)
```

For type safety to be preserved, an application of this kind must produce an exception, otherwise functions of one type could access the representation of another.

To enforce the above considerations about type safety, distinct instances of FL type definitions that are not global (see the section on global types below) create distinct functions, and their constructors produce distinct objects; even two identical instances of the type definition `type t ≡ p` define different sets of functions and their `mkt` functions produce different results for the same argument.

Type definitions in a library environment become "global", and when the library is used more than once in a program or in different programs these global definitions produce identical functions. See Section 14.2 on library environments.

If $\texttt{mkt}_1$ and $\texttt{mkt}_2$ are the constructors generated by two identical local type definitions, then, for all x, $\texttt{mkt}_1\texttt{:x}$ and $\texttt{mkt}_2\texttt{:x}$ are both defined or both undefined, and when defined $(\texttt{unt}_1 \circ \texttt{mkt}_1)\texttt{:x} = \texttt{x}$ and $(\texttt{unt}_2 \circ \texttt{mkt}_2)\texttt{:x} = \texttt{x}$. But $(\texttt{unt}_2 \circ \texttt{mkt}_1)\texttt{:x}$ and $(\texttt{unt}_1 \circ \texttt{mkt}_2)\texttt{:x}$ are exceptions and $(\texttt{ist}_2 \circ \texttt{mkt}_1)\texttt{:x} = \underline{\texttt{false}} = (\texttt{ist}_1 \circ \texttt{mkt}_2)\texttt{:x}$. (Subscripts merely distinguish identically-named, but different functions; the actual names are mkt, unt, ist.)

13.1 Type "tags"; storing and transmitting elements of abstract types

The required distinctions between types can be made by associating a unique "tag" with each type definition. If n1 and n2 are distinct tags associated with the two types in (1) above, then the application $(\texttt{unt}_{\texttt{n1}} \circ \texttt{mkt}_{\texttt{n2}})\texttt{:x}$ can be seen to be a type error at compile time. If compile time type analysis fails (and it often does when objects of user defined types are stored in persistent files or transmitted over communication channels), objects are tagged with their type and checking is done at run time.

13.2 Global types

When a type definition is installed in a library environment (this may be done by a system command that is not part of the FL language), it is assigned a unique tag that accompanies each use of the definition. Therefore all instances of a global definition generate identical functions, i.e., constructors generated by each use all produce objects with this tag, and the other functions of the type all test for this tag (unless compile time type analysis eliminates the tagging and testing). Thus global types can be used by different programs to communicate and store objects of a common user defined type.

14 Primitive, library and composite environments

Recall the syntax of environments:

$$
\begin{array}{rcl}
\textit{env} & ::= & \textit{defnlist} \mid \underline{\texttt{export}}(\textit{namelist})\ \textit{env} \mid \underline{\texttt{hide}}(\textit{namelist})\ \textit{env} \mid \\
 & & \textit{composite\_env} \mid \underline{\texttt{lib}}(\textit{string}) \mid \underline{\texttt{PF}} \\
\textit{defnlist} & ::= & \{\ \textit{defn}^{+}_{\textit{space}}\ \} \\
\textit{namelist} & ::= & \textit{name}^{+}_{,}
\end{array}
$$

14.1 The primitive environment

The environment PF is the *primitive environment*; it defines all the primitive function names to be the corresponding primitive FL function and all other function names to

be appropriate exception-producing functions. For example, PF assigns the function *append left* to the function name al and it assigns the function

```
signal∘[~"miranda", ~"No Such Function", id]
```

to the function name miranda[†]. In the environment PF, miranda:3 evaluates to an exception containing the three values: "miranda" (the function name), "No Such Function" (the explanation), and 3 (the argument).

14.2 Library environments; global types

A library environment lib(*string*) is an environment E that has no free function names and whose type definitions have been made "global". Library environments are stored in an FL system with the name given by *string*. The use of lib(*string*) in an expression or composite environment is equivalent to the use of a copy of the environment E to which it refers, except that its type definitions are global. A global type definition differs from an ordinary type definition only in that it has been assigned in advance a unique type tag that accompanies every copy, so that every copy defines exactly the same type. (Recall that this differs from ordinary, local type definitions; identical copies of these receive different tags and hence define different types. See the preceding Section 13.)

An environment without free function names can be made a library environment by the use of a special system command (not part of the FL language) that stores it in the system with a specified name; the process of storing it includes the assignment of type tags to its type definitions, thereby making them global.

14.2.1 Example

Let the environment lib("ex") contain the type definition type t ≡ P. Then the function

```
[(mkt where lib("ex")), (mkt where lib("ex"))]
```

will produce a pair <y, y>, where y is of type t, when applied to any argument that satisfies P. But the function

```
[(mkloc where {type loc ≡ P}), (mkloc where {type loc ≡ P})]
```

will produce a pair <z, w>, where z has type $\texttt{loc}_{t1}$ arising from the first occurrence of the definition type loc ≡ P and w has type $\texttt{loc}_{t2}$ from the second definition type loc ≡ P (t1 is the tag assigned to the first definition, t2 is that assigned to the second).

14.3 Composite environments

A composite environment can be built from other environments with one of four operations: uses, where, union and rec(*namelist*), and environments can be grouped

[†]Note: miranda is not the name of a primitive function in FL

with braces:

composite_env ::= *env* **uses** *env* | *env* **where** *env* | *env* **union** *env* |
rec(*namelist*) *env* | { *env* }

As with **hide**(*namelist*) *env* and **export**(*namelist*) *env*, **rec**(*namelist*) *env* is ill-formed if *namelist* is not a subset of the names defined by *env*.

An environment without braces, e.g., $X\ op_1\ Y\ op_2\ Z$, is equivalent to one having left-associated braces, e.g., $\{X\ op_1\ Y\}\ op_2\ Z$.

An environment E defines a set of defined names $D(E)$; it has a set of free names $F(E)$ that occur on the right sides of definitions and have not been defined by some definition within E. To describe a composite environment Z built from environments X and Y, one must describe:

1. the set of names $D(Z)$ that Z defines (and, if both X and Y define a name, which definition to use),
2. the set of free names $F(Z)$ of Z in terms of the defined and free names of X and Y,
3. the set of free names in X or Y that become bound in Z, and how they are bound.

The following section precisely describes each of these three sets for each kind of environment (where X and Y are arbitrary environments).

Briefly, X **uses** Y defines all the names defined in X and Y and if both define a name, its definition in X prevails; free names in X can see their definitions in Y, if they exist. In X **where** Y the situation is the same except it defines only the names defined in X. X **union** Y defines all the names defined by X and Y, but there must be no duplicate definitions; no free name in X can see a definition of that name in Y and *vice versa*. In **rec**(*namelist*) X free names in X that occur in *namelist* can see their definitions in X; every name in *namelist* must be defined by X.

14.4 Defined names, free names and bindings for environments

The primitive environment PF.

$D(\mathbf{PF})$ = the set of all function names

$F(\mathbf{PF})$ = the empty set

PF has no free function names.

A definition list environment. Let L be a definition list, *defnlist*. Then

$D(L)$ = the set of names defined by the definitions in L.

$F(L) = (F_{rec}(L) - D(L)) \cup F_{nonrec}(L)$

where

$F_{rec}(L)$ = free names on the right sides of `def` definitions in L

$F_{nonrec}(L)$ = free names on the right sides of `nrdef` definitions in L

The free names in $F_{rec}(L)$ are bound to their definitions in L, if they exist.

The environment X `uses` Y.

$D(X \texttt{ uses } Y) = D(X) \cup D(Y)$.

If X and Y both define a name, its definition in X prevails.

$F(X \texttt{ uses } Y) = (F(X) - D(Y)) \cup F(Y)$

The free names of X are bound to their definitions in Y, if they exist.

The environment X `where` Y.

$D(X \texttt{ where } Y) = D(X)$

$F(X \texttt{ where } Y) = (F(X) - D(Y)) \cup F(Y)$

The free names of X are bound to their definitions in Y, if they exist.

The environment X `union` Y.

$D(X \texttt{ union } Y) = D(X) \cup D(Y)$

$F(X \texttt{ union } Y) = F(X) \cup F(Y)$

No names are bound by `union`. X `union` Y is ill-formed if $D(X)$ and $D(Y)$ have any names in common.

The environment `rec`(*namelist*) X

$D(\texttt{rec}(namelist)\ X) = D(X)$

$F(\texttt{rec}(namelist)\ X) = F(X) - namelist$

The free names of X are bound to their definitions in X, if their names occur in *namelist* (if a name occurs in *namelist*, it must be defined in X).

The environment `hide`(*namelist*) X

$D(\texttt{hide}(namelist)X) = D(X) - namelist$

$F(\texttt{hide}(namelist)X) = F(X)$

No free names in X are bound in `hide`(*namelist*) X. Every name in *namelist* must be defined by X.

The environment export(*namelist*) X

$D(\texttt{export}(namelist)X) = namelist$

$F(\texttt{export}(namelist)X) = F(X)$

No free names in X are bound in export(*namelist*) X. Every name in *namelist* must be defined by X.

14.5 Examples

Consider the following two environments built from the environments X, Y and U, with the restriction that X and Y do not both define any given name.

$E_1 = X$ where $\{Y$ uses $U\}$
$E_2 = \{X$ union $Y\}$ uses U

The environment E_1 defines the names that X defines; free names in X see their definitions in Y, if they exist there, if not, they see their definitions in U, if they exist there, otherwise the free names of X must be defined in the surrounding context. The free names of Y see their definitions in U if they exist there, otherwise in the surrounding context.

The environment E_2 defines the names that either X or Y defines and that U defines; a free name in X cannot see a definition of that name in Y, but can see its definition in U, if it exists there, otherwise it must be defined in the surrounding context. The situation is symmetrical with respect to X and Y.

There are several identities that govern these operations on environments; here are three that hold for any environments X, Y and Z:

$$\{X \text{ union } Y\} \text{ where } Z = \{X \text{ where } Z\} \text{ union } \{Y \text{ where } Z\}$$
$$X \text{ uses } \{Y \text{ uses } Z\} = \{X \text{ uses } Y\} \text{ uses } Z$$

Both sides of the first identity are ill-formed if X and Y define the same name. The second shows that uses is associative.

15 Precedence of operators in expressions

To avoid the need to write fully parenthesized expressions there are conventions about the precedence of various operations. Expressions can be fully parenthesized by the user to make clear their intended parsing. Alternatively, he may rely on the *precedence rules* of the language to indicate where parentheses should be. For example, the FL expression

`p=q → f + g; h∘α:f`

means the same as the fully parenthesized expression:

`((p=q) → (f+g); (h∘(α:f)))`

since the precedence of the operators in the expression is the following (from the tightest binding to the weakest):

```
:   o   +   =   →
```

Every expression is implicitly fully parenthesized by an algorithm that uses the following precedence table; when in doubt, the programmer is always free to use parentheses to enhance clarity.

15.1 Precedence table

The following list gives expression forms in the order they are parenthesized by precedence. When in doubt, the user should write explicit parentheses. In this list P, P_1, P_2 denote patterns or expressions (as appropriate), e, e_1, e_2, e_3, any smallest (or already parenthesized) expressions, env, any environment, and fn, any function name or left side of a definition. Within a level parentheses associate to the left except for condition, which associates to the right; e.g., f:x:y = (f:x):y but p → f; g → h; r → s = (p → f; (g → h; (r → s)))

Precedence table

e'			
~e			
e_1:e_2			
name.e	or	*name*.P	
$e_1 \circ e_2$			
e_1 * e_2	or	$e_1 \div e_2$	
e_1+e_2	or	e_1-e_2	
e_1=e_2			
$e_1 \mapsto e_2$	or	$e_1 \mapsfrom e_2$	
$P_1 \Rightarrow P_2$	or	$P_1 \Leftarrow P_2$	
$e_1 \wedge e_2$	or	$e_1 \vee e_2$	
e_1 e_2 e_3	(Infix)		
P → e_1; e_2	or	P → e_1	only condition associates to the right
e_1 | e_2			
e where env			
λ(P)e			
fn ≡ e			

16 Comments, assertions and signatures

Comments allow the user to annotate a program with any text at any point. Assertions and signatures serve as more formal comments; they have a formal syntax and allow the user to make precise but possibly undecidable statements about a program. Some FL compilers may issue guidelines indicating classes of assertions and signatures that the compiler expects to be able to check; others may treat them as comments.

16.1 Comments

A *comment* is simply some text enclosed in `/*...*/`. It can be placed in any whitespace in an expression and has no effect on its meaning. Comments can be nested. The character pairs `/*` and `*/` are always used to delimit the beginning and end, respectively, of comments and must occur in nested pairs (except within strings).

Since slash (`/`) is a character that can be used in names, an ambiguity can occur when the *last* character of a name is `/` and that name is *immediately* followed by the self-delimiting operator `*`, or when a name beginning with `/` is immediately preceded by `*`. For example, `x/*y..` might be parsed either as `x` followed by a comment that begins with `y` or as the infix expression `x/ * y` (followed by something), which is equivalent to `*:<x/, y>...` This ambiguity is resolved in favor of comments by the rule:

> Except within a string, the character pair `/*` denotes the beginning of a comment and the character pair `*/` denotes the end of a comment.

16.2 Assertions

An *assertion* is a comment with formal structure; it asserts that two expressions have the same value. An assertion can appear anywhere a definition can; it has no formal effect on the expression containing it, although some compilers may attempt to check its validity.

An assertion has the form:

asn $expr_1 \equiv expr_2$

Its intuitive meaning is that the two expressions denote equal values (for all possible histories), including function values.

16.2.1 Example

The following assertion expresses the law that asserts that for all functions `f`, `tt → f ≡ f`

```
asn  λ(f.isfunc)tt → f   ≡   λ(f.isfunc)f
```

16.3 Signatures and meta-predicates

A *signature* is a comment with formal structure; it asserts that an expression has a property described by a *meta-predicate* (*mpred*). Meta-predicates include ordinary FL predicates but also include some non-computable predicates over functions, and predicates with universally quantified function variables.

A signature has the form:

sig *expr* :: *mpred*

where

mpred ::= *expr* | **meta**(*expr*) | ¬:*mpred* | (*mpred*) |
⟦ *mpred** ⟧ | *mpred* ↤ *mpred* | *mpred* ↦ *mpred* |
seqof:*mpred* | *mpred* ∧ *mpred* | *mpred* ∨ *mpred* |
mpred ⇒ *mpred* | **forall** (*namelist*) *mpred*

If M(x) denotes the truth value of a meta-predicate M at x, then the intuitive meaning of the signature **sig** *expr* :: M is the assertion that M(*expr*) is always true when evaluated starting with any possible history (typically, *expr* is a function name). A signature can appear anywhere a definition can; it has no formal effect on the expression containing it, although some compilers may attempt to check its validity. See Part 3 for the precise semantics of meta-predicates.

16.3.1 Examples

The meta-predicate

⟦isnum, isnum⟧ ⇒ isnum

is true for all functions f such that, whenever the argument is a pair of numbers, the result is a number. Thus + satisfies this meta-predicate, but ÷ does not (since ÷:<1, 0> is an exception, not a number).

The following signatures are valid:

```
sig +    ::  seqof:isint ⇒ isint
sig +    ::  ⟦⟧ ⇒ eqto:0      indicates that +:<> = 0
sig +    ::  seqof:isfunc ⇒ isfunc      since + is a raised function
sig al   ::  ⟦tt, isseq⟧ ⇒ isseq
sig tl   ::  tt↦isseq ⇒ isseq
sig al   ::  forall (p,q)(⟦p, seqof:q⟧ ⇒ p↦seqof:q)
sig tl   ::  forall (p,q)(p↦seqof:q ⇒ seqof:q)
```

17 Exceptions: generation and recovery

17.1 User generated exceptions; the primitive signal

The primitive function **signal** produces an exception that contains its argument:

signal:x = an exception containing x

With **signal** the user can produce exceptions (a) that reproduce system generated exceptions and/or (b) that can be caught with **catch** (see the next section). For example, signal:<"al", "arg1", <3>> is an exception that is the same as the result of al:<3>, since <3> is an inappropriate argument for the primitive al (*append left*) and therefore the exception produced by the system contains the value <"al", "arg1", <3>> of the form <*name_string*, *explanation_string*, *argument*>. All

system generated exceptions contain similar sequences of length 3 whose first two elements are strings. The user may produce other exceptions that could never be system generated. For example, `signal:"fail"` is an exception that contains the string `"fail"`. Any exception (system or user generated) can be caught by `catch` (next section).

17.2 Recovery from exceptions; the primitive `catch`

All FL functions are strict with respect to exceptions; this means that `f:`*exc* = *exc* for any function `f` and any exception *exc*. Nevertheless FL provides a simple, purely functional means for recovering from or "catching" an exception, through the use of the primitive combining form `catch`, where

$$\texttt{catch:<f,g>:x} = \begin{cases} \texttt{x} & \text{if x is an exception} \\ \texttt{f:x} & \text{if f:x is not an exception} \\ \texttt{g:<x,y>} & \text{if f:x = signal:y for some y} \end{cases}$$

Thus, when `f:x` is an exception, `signal:y`, then the result of `catch:<f,g>:x` depends on the function `g` and both `x`, the original argument, and the exception value `y`. If one wishes to catch an exception, the function that produces it must be contained in a `catch` construction; strictness makes an exception produced outside of a `catch` "uncatchable".

Together, `signal` and `catch` can be used to program backtracking and error recovery algorithms.

17.3 Example

Suppose one has a program `P` in which there are many unchecked divisions ($\div$) and, if any divide-by-zero exception occurs in evaluating `P:X`, one then wants to evaluate `P1:X` instead of `P:X`, where `P1` prevents division by zero. The following program accomplishes this:

```
catch:<P,G> where
   {def G ← [[x., [[eqto:"÷", eqto:"arg1", [[isnum, iszero]]]]] ≡ P1∘x}
```

thus

$$\texttt{catch:<P, G>:X} = \begin{cases} \texttt{P:X} & \text{if P:X is not an exception} \\ \texttt{P1:X} & \text{if P:X = signal:Y and Y =} \\ & \texttt{<"÷", "arg1", <}\textit{number}\texttt{, 0>>} \\ \texttt{signal:Z} & \text{if P:X = signal:Y and} \\ & \text{Y is not of the above form} \\ & \text{where Z = <"G", "arg1", <X, Y>>} \end{cases}$$

If `P:X` is a divide by zero exception — for example, it might arise from an attempt to divide 3 by 0: thus `P:X = signal:Y` and `Y = <"÷", "arg1", <3, 0>>` — then in `G:<X, Y>`, `G`'s domain predicate is true and the result is `(P1∘x):<X, Y>` =

(P1∘s1):<X, Y> = P1:X. If Y does not come from a divide by zero exception, then G:<X, Y> fails because G's predicate is false; the resulting exception indicates that G failed on <X, Y>.

18 List of FL primitive functions

FL is intended to have an extensive set of primitive functions that will simplify programming in many areas. The following list of primitive functions gives an informal indication of what each function does; however, indications are seldom given of the exceptions produced when a function is applied to an inappropriate argument. The functions are grouped into the following categories: predicates, boolean, comparison, arithmetic, combining forms, predicate combining forms, pattern combining forms, sequence combining forms, sequence functions, input/output/file, miscellaneous.

All primitive function names consist of all lower case letters except the names of three combining forms — K (*constant*), C (*Curry*) and Δ (*make filter*). Names are case sensitive. The only one-character primitive function names are the self-delimiting names (∧ , ∨ , ↦, ↤, +, -, *, ÷, ∘, =, |, ⇨, ⇦) plus the one-character identifiers K, C, Δ and α.

18.1 Predicates

All primitive predicates yield either **true** or **false** for all normal arguments.

Predicates for numbers

Function/examples	Function true only for; Comment
isint:5 = true, isint:3.2 = false	integers; integers are not reals
isreal:3.2 = true, isreal:5 = false	reals;
isnum:3.2 = true, isnum:<5> = false	numbers = integers or reals;
ispos	numbers greater than 0;
isneg	numbers less than 0;
iszero	0 or 0.0;

Predicates for other atoms, functions and user types

Function/examples	Function true only for; Comment
isatom	numbers, truthvalues and chars;
isbool	true or false;
ischar:`a = true ischar:"a" = false	chars; "a" = <`a> is a string
isutype	elements of any user defined type;
isfunc	functions;
isobj	objects; = closure of atoms under sequence-formation

Predicates for sequences

Function/examples	Function true only for; Comment
`isnull:<>` = `isnull:""` = `true`	the empty sequence or string;
`ispair:<2, 3>` = `ispair:"ab"` = `true`	sequences of length 2;
`isseq:<3>` = `true`, `isseq:"a"` = `true`	sequences; strings are sequences
`isstring:<'a, 'b>` = `true` `isstring:"abcd"` = `true`	strings; `isstring` = `isseq` ↠ `and` ∘ α:`ischar`; `ff`

Identically true and false predicates

Function/examples	Function true only for; Comment
`ff`	never true; identically false
`isval`	all normal values including functions and user types;
`tt`	all normal values; `tt` = `isval`

18.2 Boolean and comparison functions

The Boolean primitives treat any normal value other than `false` as true. All the Boolean and comparison functions except `not` take sequences of any length as arguments. Users will often prefer to use the lifted or higher order versions of the Boolean functions (i.e., ∧ for `and`, ∨ for `or`, ¬ for `not`) and the raised versions of the comparison functions (e.g., = for `eq`, `lt` for `less`, `le` for `lesseq`, etc.).

All the FL atoms are completely ordered — in ascending order: `false`, `true`, numbers in their usual order, chars in the order given by the ISO standard DP 10646 for characters. Any atom is less than any sequence, and sequences are ordered lexicographically; thus for any two *objects* `x` and `y`, (i.e., values that belong to the closure of the atoms under sequence-formation) either `x<y` or `y<x` or `x = y`. Functions and user defined types are not ordered. This is the ordering that is recognized by primitives such as `less` (*less than*). For example, `'a less <1>`, `<1> less <1, 2, 3>`, `"abc" less "ef"` and `<1, 2, 3> less <2>` are all true.

Any primitive comparison function yields an exception when comparing functions or elements of a user defined type; however these primitives can be extended by the user to compare elements of a user defined type. Although direct comparison of functions and/or user defined types gives an exception, comparison of sequences containing such elements may succeed if the result does not depend on comparing two of these incomparable elements. For example, even if `x` and `y` are elements of a user defined type, `<1, x, 3>` is less than `<2, y>` since 1 is less than 2, and in the lexicographical ordering of sequences it is therefore not necessary to compare `x` and `y`. Similarly, if `f`, `g` and `h` are functions, `<f> eq <g, h>` is false; since sequences of different lengths cannot be equal, it is not necessary to compare their elements. But `<f> less <g, h>` is an exception because ordering these sequences requires comparing two functions, `f` and `g`.

Boolean functions

Function/examples	Function true only for; Comment
and:<true, true, true> = true and:<1, 2, 3, 4> = true and:<1, 2, false> = false and:<> = true	any sequence not containing false;
not:false = true	the value false;
or:<false, false, 1> = true or:<> = false	any sequence with an element that is not false;

See also the combining form (i.e., lifted) versions of and, or and not under Predicate Combining Forms; these are $\wedge$, $\vee$ and $\neg$.

Comparison functions (see also raised comparison functions)

Function/examples	Function true only for; Comment
eq:<1, 1, 1> = true	sequences of identical objects;
eq:<1, 1, 2> = false	
eq:<> = true	
eq:<id, id> = exception	no comparison of functions or user types (unless eq is extended for a user type)
eq:<id> = exception	id is not an object
eq:<id, 'a> = false	a function is not equal to an atom or a sequence
eq:<<id>, <id, id>> = false	sequences of unequal lengths are not equal
neq:<1, 1, 2> = true	sequences for which eq is false;
less:<1, 2, 'a, <>> = true	sequences of strictly ascending order;
less:<1, 1> = false	
less:<> = true	
less:<3> = true	
less:<id, id> = exception	incomparable
less:<id> = exception	
less:<<>, <id>> = true	<> is less than any other sequence
greater:<3, 2, 1> = true	sequences of strictly descending order;
lesseq:<1, 1, 2> = true	sequences of non-decreasing order;
greatereq:<2, 1, 1> = true	sequences of non-increasing order;

Raised comparison functions (compare objects or combine functions to form a comparison function)

Function/examples	Comment
=:<1, 1, 1> = true =:<f, g, h> = eq∘[f, g, h]	= = raise:eq (f=g):x = f:x eq g:x
lt:<1, 2, <>> = true lt:<f, g, h> = less∘[f, g, h]	lt = raise:less; *less than* (f lt g):x = f:x less g:x
gt:<f, g, h> = greater∘[f, g, h]	gt = raise:greater
le:<f, g, h> = lesseq∘[f, g, h]	le = raise:lesseq
ge:<f, g, h> = greatereq∘[f, g, h]	ge = raise:greatereq

18.3 Arithmetic functions

This group of functions includes the basic arithmetic operations (e.g., `add`); most users will use their raised versions (e.g., `+`) instead of these. All arithmetic functions are defined for both integers and reals; if the argument contains only integers the result is an integer (except for `div`, whose result is always a real); otherwise the result is a real (except for `floor` and `ceiling`, whose results are always integers). Some (`add`, `mul`) are defined on sequences of numbers of any length; some (`sub`, `div`) are defined only on pairs of numbers; others (`neg`, `floor`, `ceiling`, `abs`) are defined on single numbers.

Arithmetic functions

Function/examples	Domain; Comment
`add:<1, 2, 3, 4>` = 10 `add:<1.3, 4, -5>` = 0.3 `add:<>` = 0	sequences of numbers; all integers to integer result, otherwise: real result
`sub:<4, 5>` = -1 `sub:<4, 3.0e2>` = -296.0 `sub:<7, 2, 1>` = `exception`	pairs of numbers; integers to integer result otherwise: real result
`mul:<1, 2, 3, 4>` = 24 `mul:<1.2, 5>` = 6.0 `mul:<>` = 1	sequences of numbers;
`div:<4, 2>` = 2.0 `div:<5, 2>` = 2.5 `div:<2.7, -3>` = -0.9	pairs of numbers; result always real
`neg:3` = -3 `neg:5.3e-3` = -.0053	numbers;
`floor:3` = 3 `floor:3.7` = 3 `floor:-3.7` = -4	numbers; result = largest integer ≤ argument
`ceiling:3` = 3 `ceiling:3.7` = 4 `ceiling:-3.7` = -3	numbers; result = smallest integer ≥ argument
`abs:3` = 3 `abs:` − 3 = 3 `abs;` − 3.7 = 3.7	numbers; result = absolute value of argument

Raised arithmetic functions (do arithmetic on numbers or combine functions to form an arithmetic function)

Function/examples	Comment
+:<1, 2, 3> = 6 +:<f, g, h> = add∘[f, g, h]	+ = raise:add (f+g):x = f:x + g:x
-:<f, g> = sub∘[f, g]	- = raise:sub, (f-g):x = f:x - g:x
* :<f, g, h> = mul∘[f, g, h]	* = raise:mul
÷:<f, g> = div∘[f, g]	÷ = raise:div

18.4 Combining forms

Each combining form is presented as a higher order function. The first line for each combining form gives the equivalent special syntax for the form, if it has one. The second line (or the first, if there is no special form) shows the application of a combining form `cf` as follows: `(cf:arg1):arg2` where `arg1` is an argument of the combining form function (which must be in the stated domain) and `arg2` is an argument of the function resulting from `cf:arg1`.

Combining forms

Function/examples	Domain; Comment
∘:<f, g, h>:x = f:(g:(h:x)) ∘:<> = id	sequences of functions; infix: f∘g = ∘:<f, g>
|:<f, g, h>:x = h:(g:(f:x))	sequences of functions; | = ∘∘reverse
cons:<f, g, h> = [f, g, h] [f, g, h]:x = <f:x, g:x, h:x> []:x = cons:<>:x = <>	sequences of functions;
cond:<p, f, g> = p → f; g (p → f; g):x = f:x (p → f; g):x = g:x (p → f; g):x = exception (p → f; g):x = ⊥	triples of functions; if p:x is true if p:x is false if p:x is exception if p:x is non-terminating
apply:<f, x> = f:x	pairs satisfying [[isfunc, isval]];
K:x = ˜x ˜x:y = x	values; K = λ(x.)λ(y.)x except ˜x:exception = exception
lift:f = f′ f′:<g, h> = f∘[g, h]	functions; lift = K∘″˜cons = λ(f.isfunc)λ(x.)(f ∘ cons:x)
C:f:x:y = f:<x, y>	functions; C is *curry* C = λ(f.isfunc)λ(x.)λ(y.)f:<x, y>
raise:add:<f, g>:3 = add:<f:3, g:3> raise:add:<3, 4> = 7	functions defined on sequences of objects; raise = λ(f.isfunc)(seqof:isfunc ∧ ¬:isnull → lift:f; f)
catch:<signal∘˜"err", id>:<3, 4> = <<3, 4>, "err"> catch:<add, id>:<3, 4> = 7	pairs of functions; catch:<f, g>:x is f:x if not an exception, o.w. it is g:<x, v> where v is the value in the exception f:x
delay:f:x:y = f:x:y	functions; delay:f and delay:f:x are normal forms; delay = λ(f.isfunc)λ(x.)λ(y.)f:x:y

18.5 Predicate combining forms

Predicate combining forms

Function/examples	Domain; Comment
pcons:<$f_1,\ldots,f_n$> = [$f_1,\ldots,f_n$] pcons:<$f_1,\ldots,f_n$>:x = true [isint, tt, tt]:<3, "a", 0> = true []:<> = pcons:<>:<> = true	sequences of functions; iff x is a sequence <$x_1,\ldots,x_n$> and f_i:x_i is true for all i = 1,..,n true only for <>
=:<f, g>:x f=g = =:<f, g> (len=~1):<5> = true	pairs of functions; true iff f:x = g:x infix
seqof:p:x seqof:isint:<3, 4> = true seqof:ff:<> = true	p a function; true iff x = <$x_1,\ldots,x_n$> and p:x_i is true for all i = 1,..,n
eqto:x:y eqto:<3>:<3> = true	x an object; *equal to* true iff x = y a curried version of eq
lenis:x:y lenis:0:<> = true	x an integer; *length is* true iff y a sequence of length x
∧:<$f_1,\ldots,f_n$>:x p∧q = ∧:<p, q> ∧:<>:3 = true	f_i functions; true iff f_i:x is true for all i, i = 1,..,n infix
∨:<$f_1,\ldots,f_n$>:x p∨q = ∨:<p, q> ∨:<>:x = false	f_i functions; true iff f_i:x is true for some i, i = 1,..,n infix
¬:p:x ¬:isint:3 = false	p a function; true iff p:x = false ¬:p = not∘p
↦:<p, q>:x p↦q = ↦:<p, q> (isnum↦tt):<3, <>, 1> = true	domain = pairs of functions; true iff x = <$x_1,\ldots,x_n$>, n≥1, p:x_1 and q:<$x_2,\ldots,x_n$> are true infix
↤:<p, q>:x p↤q = ↤:<p, q> (isseq↤(eqto:~x)):"bcx" = true	domain = pairs of functions; true iff x = <$x_1,\ldots,x_n$>, n≥1, p:<$x_1,\ldots,x_{n-1}$> and q:x_n are true infix

18.6 Pattern combining forms

A pattern combining form F maps n predicates (of the n patterns being combined) into a pair <p, <m_1, .., m_n>>, where p is the predicate of the new pattern and m_i is the function that modifies the definitions of the ith pattern being combined to form the definitions of the new pattern. In the following table $P_1, .., P_n$ denote patterns or predicates, with at least one being a pattern, and $p_1, .., p_n$ denote the predicates of $P_1, .., P_n$ (where $p_i = P_i$ if P_i is a predicate). See Section 12.1 for a description of the general form pat(*expr*; $P_1, .., P_n$) for patterns.

Pattern combining forms

Pattern Function/examples	Equivalent pat form Domain; Comment
$[\![P_1, .., P_n]\!]$ patcons:<$p_1, .., p_n$> = <pcons:<$p_1, .., p_n$>, <s1, .., sn>>	pat(patcons; $P_1, .., P_n$) nonempty sequences of functions; defs of P_i modified by si
$P_1 \Rightarrow P_2$ $\Rightarrow$:<p_1, p_2> = <$p_1 \mapsto p_2$, <s1, t1>>	pat($\Rightarrow$; P_1, P_2) pairs of functions; defs of P_1 modified by s1, of P_2 by t1
$P_1 \Leftarrow P_2$ $\Leftarrow$:<p_1, p_2> = <$p_1 \mapsto p_2$, <t1r, r1>>	pat($\Leftarrow$; P_1, P_2) pairs of functions; defs of P_1 modified by t1r, of P_2 by r1
P_1 pand P_2 pand:<p_1, p_2> = <$p_1 \wedge p_2$, <id, id>>	pat(pand; P_1, P_2) pairs of functions; defs of P_1 and P_2 not modified
P_1 por P_2 por:<p_1, p_2> = <$p_1 \vee p_2$, <id, id>>	pat(por; P_1, P_2) pairs of functions; defs not modified; caution: defined functions may fail on objects that satisfy pattern
P_1 pcomp P_2 pcomp:<p_1, p_2> = <$p_1 \circ p_2$, <p_2, id>>	pat(pcomp; P_1, P_2) pairs of functions; defs of P_1 modified by p_2

18.7 Sequence combining forms

Sequence combining forms

Function/examples	Domain; Comment
/1:f:<x_1, .., x_n> = f:</1:f:<x_1, .., x_{n-1}>, x_n> /1:f:<x> = x /1:id:<1, 2, 3> = <<1, 2>, 3>	functions f defined on pairs; /1:f not defined for <> /1 = sl insert left
/r:f:<x_1, .., x_n> = f:<x_1, /r:f:<x_2, .., x_n>> /r:f:<x> = x /r:sub:<1, 2, 3, 4> = −2	functions f defined on pairs; /r:f not defined for <> /r = sl insert right
tree:f:<x_1, .., x_n> = f:<tree:f:<x_1, .., x_k>, tree:f:<x_{k+1}, .., x_n>> tree:f:<x_1> = x_1 tree:id:<1, 2, 3, 4> = <<1, 2>, <3, 4>>	functions f on pairs; for n>1 where k = ceiling:(n÷2) for n=1 tree:f not defined for <>
α:f:<x_1, . . ., x_n> = <f:x_1, . . ., f:x_n> α:isint:<1, 3.2> = <true, false>	functions f; α = *apply to all*
merge:f:<x, y> = x or y = al:<x_1, merge:f:<tl:x, y> = al:<y_1, merge:f:<x, tl:y> merge:lt:<<1, 7>, <2, 4, 9>> = <1, 2, 4, 7, 9>	functions f; x, y sequences if y or x is <> if f:<x_1, y_1> is true if f:<x_1, y_1> is false

18.8 Sequence functions

Sequence functions

Function/examples	Domain predicate; Comment
al:<x, <y_1, .., y_n>> = <x, y_1, .., y_n> al:<1, <2>> = <1, 2>	[[tt, isseq]]; *append left*
ar:<<x_1, .., x_n>, y> = <x_1, .., x_n, y>	[[isseq, tt]]; *append right*
cat:<x_1, .., x_n> = y cat:<<1>, <2>> = <1, 2>	seqof:isseq; y is the *concatenation* of the x_is
distl:<x, <y_1, .., y_n>> = <<x, y_1>, .., <x, y_n>> distl:<3, <4, 5>> = <<3, 4>, <3, 5>>	[[tt, isseq]]; *distribute left*
distr:<<x_1, .., x_n>, y> = <<x_1, y>, .., <x_n, y>> intsto:n = <1, . . ., n>	[[isseq, tt]]; *distribute right* isint ∧ ¬:isneg; *integers to*
len:<x_1, .., x_n> = n	isseq; *length*
reverse:<x_1, .., x_n> = <x_n, .., x_1>	isseq;
sel:<i, <x_1, .., x_n>> = x_i sel:<3, <1, 2>> = exception sel:< − 2, <3, 4, 5, 6>> = 5	[[isint, isseq∧(s1 le lenos2)]]; i negative: count from right; o.w., from left. sel = *select*

(Continued, next page)

Sequence functions, continued

Function/examples	Domain predicate; Comment
trans:<x_1, .., x_n> = <y_1, .., y_m>	seqof:isseq ∧ eq∘α:len; *transpose* y_j is sequence of jth elements of x_is
Selector functions si, for i=1,2,.. s3:<x_1, x_2, x_3, x_4> = x_3	isseq ∧ (len le ~i); *select left, ith* si:x = sel:<i, x> for i=1,2,..
Selector functions ri, for i=1,2,.. r1:<x_1, x_2, x_3, x_4> = x_4	isseq ∧ (len le ~i); *select right, ith* ri:x = sel:<neg:i, x> for i=1,2,..
tl:<$x_1, x_2, .., x_n$> = <$x_2, .., x_n$>	tt↦tt; *tail*
tlr:<$x_1, .., x_{n-1}, x_n$> = <$x_1, .., x_{n-1}$>	tt↤tt; *tail right*

18.9 Input, output and file functions

Input, output and file functions

Function/examples	Domain predicate; Comment
in:devicename = input	isstring; input = next input from device devicename
in:"kbd" = c	c = next character typed at keyboard
out:<devnm, x> = <devnm, x>	[[isstring, tt]]; x to device devnm
out:<"scr", "hello"> = <"scr", "hello">	"hello" appears on screen
get:filename = file	isstring; file = value stored at filename (= a string)
put:<filename, x> = <filename, x>	[[isstring, tt]]; x is stored as the file named filename

18.10 Miscellaneous functions: id, Δ, signal

Miscellaneous functions

Function/examples	Domain predicate; Comment
id:x = x	isval; the identity function
Δ:p:x = x if p:x true Δ:p:x = exception otherwise	isfunc; Δ:p = p → id; signal∘[~"Delta", ~"arg2", id]
signal:x = y = exception	isval; exception y contains x
f:(signal:x) = signal:x	see combining form catch for all functions f and values x

19 The Syntax of FL

19.1 Syntax conventions

Symbol	Interpretation
italics	non-terminals
|	"or"
{..}	grouping, as distinguished from {..} (FL characters)
x^*_s	zero or more x's separated by the separation mark s e.g., 1, 2, 3 and abc are values for $expr^*_,$ and $character^*$
x^+_s	one or more x's separated by the separation mark s
x^p	x followed by zero or more primes $'$ where x is one of: [[[→ : ~ E.g., $\rightarrow''$ is an instance of $\rightarrow^p$
construct *small italics*	*small italics* identifies "construct" when the latter has not been named by a non-terminal. E.g., $expr{:}^p\,expr$ indicates that $expr{:}^p\,expr$ is an application $_{application}$

For example,

$$seqs ::= \langle \{atom \mid name \mid \text{"}\underset{string}{character^*}\text{"} \mid seqs\}^*_, \rangle$$

describes sequences of constants (one of whose elements may be a *string*, a row of characters enclosed in quotes); these include the following expressions:

```
<>    <1,2>    <square,5,6,"1+ab">    <f,<1,"abc">,"",beta>
```

19.2 Syntax of FL expressions

See the next section, Lexical Structure, for the definitions of *character*, *identifier*, *name* and *number*.

Syntax of FL expressions

expr ::= *atom* | *name* | *seq* | *expr* $:^{p}$ *expr* (*application*) | (*expr*) |
expr **where** *env* | *expr*′ (*primed expr*) | $\neg^{p}$*expr* (*constant*) | $[^{p}$ *expr*$^{*}_{,}$] (*construction*) |
$[\![^{p}$ *expr*$^{*}_{,}$ $]\!]$ (*predicate constr*) | *cond* | *expr expr expr* (*infix expr*) | λ(*pat*) *expr* (*lambda expr*)

atom ::= `*character* (*a "char"*) | *number* | **true** | **false** (*truth values*)

seq ::= < *expr*$^{*}_{,}$ > | *string*

string ::= "*character**"

cond ::= *expr* $\rightarrow^{p}$ *expr* ; *expr* | *expr* $\rightarrow^{p}$ *expr* |
pat $\rightarrow$ *expr* ; *expr* | *pat* $\rightarrow$ *expr*

pat ::= *name.* | *name.expr* (*elementary patterns*) | $[\![$ *patlist* $]\!]$ (*pat construction*) | *pat_expr expr pat* (*infix pat*) |
pat expr pat_expr (*infix pat*) | *name.pat* | **pat**(*expr* ; *patlist*) (*general pattern*)

patlist ::= {*pat_expr*, }* *pat*{, *pat_expr*}*

pat_expr ::= *pat* | *expr*

env ::= {*defn*$^{+}_{blank}$} (*defn list*) | **export**(*name*$^{+}_{,}$) *env* (= **export**(*namelist*) *env*) | **hide**(*name*$^{+}_{,}$) *env* (= **hide**(*namelist*) *env*) |
lib(*string*) | **PF** | *env* **uses** *env* | *env* **where** *env* |
env **union** *env* | **rec**(*name*$^{+}$) *env* (= **rec**(*namelist*) *env*) | { *env* }

defn ::= **def** *name argexp* ≡ *expr* | **nrdef** *name argexp* ≡ *expr* |
exdef *pat argexp* ≡ *expr* | **type** *identifier* ≡ *pat_expr* |
asn *expr* ≡ *expr* (*assertion*) | **sig** *expr* :: *mpred* (*signature*)

argexp ::= *empty* | (*pat*)$^{+}$ | (*pat*)* ← *pat_expr*

mpred ::= *expr* | **meta**(*expr*) | *mpred* ⇒ *mpred* | $[\![$*mpred*$^{*}_{,}$$]\!]$ |
mpred ↤ *mpred* | *mpred* ↦ *mpred* | **seqof**:*mpred* |
mpred ∧ *mpred* | *mpred* ∨ *mpred* | ¬:*mpred* | (*mpred*)
forall(*name*$^{+}_{,}$) *mpred* (= **forall**(*namelist*) *mpred*)

19.3 Lexical structure of *name, identifier, number, character, comment*

In this section (x) denotes an optional x.

Lexical structure

name	::=	*identifier* | ∧ | ∨ | ↤ | ↦ | + | - | * | ÷ | ∘ | = | ! | ⇨ | ⇦
identifier	::=	*ident_char* {*ident_char* | *digit*}*
ident_char	::=	*letter* | / | $ | % | # | _ | ? | ↑ | ↓ | ^ | ¬
number	::=	(+ | -)$_{sign}$ *digit*$^+$ (. *digit**)(*exponent*) | . *digit*$^+$ (*exponent*)
exponent	::=	{e | E} (+ | -) *digit*$^+$
character	::=	*letter* | *digit* | *special_char*
letter	::=	{ *see note below* }
digit	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
comment	::=	/ * {*text*} * /
		within *text* / * and * / must be nested

The set of letters and special characters is not fixed but must contain the first 128 in the ASCII set. A full implementation should include the upper and lower case Roman and Greek letters plus the following special characters:

Special characters

special_char	::=	*visible_char* | *code_char*
visible_char	::=	{*characters listed without* " | "}:
		‾ ` ! @ # $ % ^ & * ( )
		- _ = + | { } [ ] : ; '
		' < , > . ? / ← → ↤ ↦ ⊥
		⟦ ⟧ ≠ ∧ ∨ ↑ ↓ ¬ ≡ ÷ ⊂ ⊃
		∩ ∪ ∘ ≺ ≻ ≤ ≥ ⇒ ⇨ ⇦
code_char	::=	\ *code* \ | \\ | \"\
code	::=	{*letter* | *digit* | -}$^+$

The set of meaningful code characters (*code_chars*) is not fixed but includes all the ASCII codes such as \CR\ (carriage return or newline). The code character \\ denotes \ and \"\ denotes ". The code characters built with integers, \n\ , up to some limit, are codes for the characters in some set of characters. Other codes may be assigned meanings at a later time.

Comments may appear any place in an expression that whitespace can appear. They have no effect on the meaning of the expression.

19.4 ASCII representation of expressions

Many FL function names and symbols are not in the ASCII set of symbols. The following table presents a transliteration of full FL into ASCII. Some operators, e.g., → , are translated into compound ASCII symbols, ->, thus some self delimiting operators must be delimited by spaces to avoid ambiguity, e.g. in ASCII - and > must have a space between them, else -> is a right arrow.

Symbol	ASCII	Description (FL symbol :: ASCII character(s))
( )	( )	parentheses
~	~	tilde
< >	< >	sequence brackets
"	"	double quote
`	`	back quote
,	,	comma
;	;	semicolon
:	:	application sign :: colon
.	.	period
[ ]	[ ]	construction brackets :: square brackets
=	=	equal
+	+	add :: plus sign
-	-	subtract :: minus sign
*	*	multiply :: asterisk
FL symbols with different ASCII representations		
'	'	prime :: right single quote
÷	/	divide :: slash
∘	@	composition :: "at" sign
|	!	reverse composition :: exclamation point
λ	`lambda`	lambda :: reserved word "`lambda`"
⟦ ⟧	[| |]	predicate construction :: square brackets with |'s
≡	==	definition symbol :: pair of ='s
→	->	right arrow :: - with >
←	<-	left arrow :: < with -
↦	|->	predicate append left :: | and - and >
↤	<-|	predicate append right :: < and - and |
∧	/\	and :: slash and backslash
∨	\/	or :: backslash and slash
¬	`Not`	not :: function name "`Not`"
⇨	|=>	pattern append left :: | and = and >
⇦	<=|	pattern append right :: < and = and |
⇒	=f=>	function meta-predicate :: = and `f` and = and >

19.4.1 FL keywords and their reserved words in ASCII

The FL keywords have the following corresponding reserved words in ASCII, which are used without underlines in an ASCII version of FL; one ASCII reserved word corresponds to the FL symbol `lambda`. The reserved words are listed alphabetically.

```
asn   def   exdef   export   false   hide   lambda   lib   nrdef
pat   PF    rec     sig      true    type   union    uses  where
```