An Introduction to the FL Programming Language[1]

John Backus
John H. Williams
Edward L. Wimmers

IBM Almaden Research Center
San Jose, CA 95120

1. Overview

FL is intended to be a programming language in which it is easy to write clear, concise, and efficient programs. It is designed around a set of functional 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 and that may be transformed and optimized according to an underlying algebra of programs.

FL is an interactive language. It incorporates an implicitly controlled history component for dealing with input/output operations and for maintaining a persistent store. This history component provides a mechanism for the localized and disciplined handling of storage and I/O.

FL programs can be higher-order. In addition to the primitive combining forms provided, the language includes powerful mechanisms for defining new higher-order functions. There are primitive operators for currying functions and for lifting functions to functionals; these are useful for writing closed-form, combinatorial definitions of higher-order functions.

FL is a statically scoped language. An FL program consists of an expression together with a list of definitions to be used in evaluating that expression. A definition list entry may itself be a nested definition list that exports only some of its definitions. This provides a convenient hiding mechanism which, together with the facility for programmer defined data types, provides a powerful technique for making encapsulated type definitions. The definition of local function names is further facilitated by a powerful pattern matching mechanism that allows patterns to be constructed out of user defined predicates.

FL has a hierarchical, dynamic type system that divides the value domain into subsets called types. The type system is based on a particular class of predicates that enables many type errors to be discovered at compile time. New data types can be implemented as tagged objects of an underlying representation type. The language is designed so that all such tagged objects are initially in the domain of FL values, thus giving the programmer a convenient, concrete way of thinking about and manipulating data types. Moreover, these tagging oper-

---

1 This chapter is a revised excerpt from a preliminary version of the FL language manual [BWW 86]. The revisions represent changes (mostly deletions) to the language described therein.

ations can be hidden *via* the scoping mechanism, thereby providing the capability for abstract data type definitions.

The meaning of FL programs is specified by an operational semantics that uses rewrite rules *and* by an equivalent denotational semantics that uses an unusually simple domain of values as its basis. The rewrite rule semantics employs a novel scheme for combining higher-order function definitions with call-by-value evaluation order. The denotational semantics is presented in a way that allows the programmer to think about the values denoted by expressions without having constantly to be aware of the usual mathematical complexities of the actual domains. Thus, the denotational semantics is a useful tool for the programmer as well as for the language analyst.

2. Introduction

FL is a function level, functional programming language. The key component of the language is a collection of operations called *combining forms*, operations for building new programs from old programs. These combining forms are carefully chosen to have rich, mutually distributive properties, so that programs written using one combining form as the main connective can often be transformed into an equivalent program whose main connective is some other combining form. These mutually distributive properties form the basis for an *algebra of programs*, a rich collection of identities that hold between different representations of functions in FL. This algebra of programs underlies a program transformation system, which is used both to reason about programs and to provide a basis for an optimizing compiler for FL.

In addition to the combining forms, there are three other fundamental design considerations that distinguish FL from many other programming languages:

1. the emphasis on function level programming
2. a specific history component for handling I/O and memory
3. a denotational semantics designed to be a useful tool for programmers

Function level programming

The language design is based on the fundamental tenet that clarity is achieved when programs are written at the function level, i.e. by putting together existing programs to form new programs, rather than by manipulating objects and then abstracting from those objects to produce programs. This premise is embodied in the name of the language, F(unction) L(evel). For example, to define a program **sumprod** to compute the sum and product of two numbers, one doesn't proceed by beginning with two objects x and y, constructing the object $\langle x+y, x*y \rangle$ (note that the sequence of objects $x_1, \ldots, x_n$ is written $\langle x_1, \ldots, x_n \rangle$), and then abstracting with respect to x and y (e.g. **Def sumprod** $\equiv \lambda x, y.\langle x+y, x*y \rangle$), but rather by beginning with the functions + and * and by applying the combining form *construction* ([ ]), which is defined so that for any functions $f_1, \ldots, f_n$, the function $[f_1, \ldots, f_n]$ applied to x produces

the n element sequence whose ith element is f_i applied to x; i.e., $[f_1,...,f_n]:x = \langle f_1:x,...,f_n:x\rangle$ (note that the application of f to x is written f:x). This results in the function level definition **Def sumprod ≡ [+,*]**.

I/O and persistent memory

In order to accommodate interactive input and output and to facilitate the writing of programs that require persistent storage, all FL programs are required to map pairs into pairs. The first component of the pair can be any FL value, but the second component is required to be a *history*: an FL sequence that contains the current value of the file system and the current status of all input and output devices. Thus, the functionality of any FL program f is $f:\langle val, history\rangle \rightarrow \langle val', history'\rangle$.

Most of the primitive functions are completely independent of the history component and leave it unchanged. For example, **tl:<<1,2,3>,history> = <<2,3>,history>** no matter what the contents of **history** may be. These functions are called typeA.

A few of the primitives depend on the contents of **history** but do not change the file system or the status of any device; e.g. **get** retrieves the most recent file system from the **history**. These primitives are called typeB.

Finally, some primitives both depend on the contents of the history and alter it; e.g., **in** maps **<dev,history>** into **<next_inp,history'>**, where **next_inp** is the next value from device **dev** and the history is changed to reflect the fact that **dev** has been read (the precise details are given in [BWW 86]). These primitives are called typeC.

It is important to note that without the history as an explicit argument, a primitive such as **in** could not be a function, since **in:reader** could return different values depending on when it was called. Including the history as an explicit argument permits a functional treatment of input/output and file system fetches. This implies that *all* programs must be viewed as mapping pairs to pairs, even typeA programs such as **tl**. However, if the history component were to be added explicitly, it would then be possible to treat it like any other object, e.g. to make multiple copies of it and to apply different operations to the various copies. Such manipulations would not be consistent with the intended use of the history component as representing *the* current status of the I/O devices and the file system.

Therefore, some methodology is required whereby exactly one history is in existence at any given time. This is accomplished in FL by making the history component *implicit.* Thus there is no possibility of FL programs treating the history in a way inconsistent with the methodology, since the history component may be accessed or changed only by built in primitives such as **get, put, in,** and **out**, and these primitives are guaranteed to preserve the consistency of the history component.

Whenever possible in this manual the history component will not be mentioned explicitly. For example, in the next section **tl** (a typeA function) is discussed as though it simply mapped

values to values. This suppression of the history component except where necessary for understanding a program is the result of an attempt to reintroduce the notion of an underlying state into an otherwise basically functional language in a disciplined way. Thus, this language can be viewed as an experiment in combining the convenience of languages in which dependence on an underlying state is always implicit (e.g. Pascal, PL/I, etc.) with the mathematical properties of languages in which all functional dependencies must be explicitly written when they are required by the program's specification (e.g. pure functional languages such as FP [Backus 78] and SASL [Turner 82]).

The difficulty with the entirely procedural approach of Pascal et. al. is that the underlying state is pervasive and encourages a style of programming in which essentially all programs depend on and alter the state. The undisciplined use of an underlying state and side effects results in programs that are difficult to understand and reason about. On the other hand, the difficulty with the explicitly functional approach of FP, SASL, et. al. is that the increased complexity of a program's functionality becomes a notational nuisance, which masks the primary purpose of the program. E.g., an interactive SASL program to correct misspelled words in a file using an (updatable) dictionary has the primary purpose of mapping an in_file to a corrected_file but has the proper functionality

<in_file, dictionary, keyboard> → <corrected_file, dictionary′, screen, rest_of_keyboard>

(Note that the program must be given keyboard as input to receive responses to interactive queries it puts on screen, and it must return the "rest" of the keyboard input for use by subsequent programs.)

FL is an attempt to combine the advantages (and avoid the problems) of these two approaches by (semantically) attaching an explicit history component to the functionality of all programs, and (syntactically) suppressing consideration of that component. The success of this approach will be measured by the extent to which programmers can use the convenience of treating the history component implicitly, thus keeping their program notations uncluttered, and still be able to write clear programs. (For a comparison of the underlying algebraic properties of FL and pure FP see [Williams&Wimmers 88].)

The semantics of FL

To understand a program it is sometimes convenient to think about *how* the program computes; at other times it is more convenient to consider *what* it computes. E.g., to understand the temporal dependencies of I/O it may be more illuminating to describe the *process* of computation, whereas the meaning of new higher-order combining forms may be better understood by considering the functions *denoted* by those forms. For this reason, the meanings of FL programs are specified both by an operational semantics based on rewrite rules and by a complementary and precisely equivalent denotational semantics.

The operational semantics. The model of a typical computing machine consists of two parts, a computation engine (usually called a CPU) and a set of input/output and permanent storage

devices (such as keyboards, screens, and disks). Similarly, the model of computation for FL consists of two parts, an expression that is worked on by the CPU and a history component that maintains a chronological record of the activities of the I/O and permanent storage devices. Computation proceeds by transforming the expression by means of rewrite rules. Initially the CPU contains rewrite rules for each of the primitive functions of the language; e.g., the rule id:x → x allows the CPU to simplify expressions containing applications of the identity function id. Other rules may be added to this initial set as the result of programmer defined functions. E.g., the definition Def f ≡ E results in the rule f:x → E:x being added to the set of rewrite rules that can be applied by the CPU.

The computation process (also called the rewriting or reduction process) consists of locating a subexpression that is an instance of the left-hand-side of some rewrite rule and replacing it by the corresponding right-hand-side. For example, the CPU will rewrite the expression <id:2, id:3> to <2, 3> using two applications of the rewrite rule id:x → x.

As part of the computation process, evaluation of the expression will sometimes request information from or send information to an I/O device. When this occurs, an appropriate notation is added to the history stream as part of the rewriting process. For example, the expression out:<'screen', 12> rewrites to <'screen', 12> (the result of a successful output operation is just its argument), and the notation <'outflag', <'screen', 12>> is added to the end of the history stream to indicate that the output 12 has occurred on the device screen. Of course, the history stream is just a formalism used in describing the rewriting process; this output event is actually implemented by printing the number 12 on the screen.

Since I/O can occur as the result of applying a rewrite rule, it is crucial that the order of rewriting be specified precisely. In order to simplify the semantics, FL is designed so that the process that locates the next subexpression to be reduced is always required to select the leftmost innermost such expression. To satisfy this requirement and to allow for the definition of recursive higher-order functions requires a novel treatment of programmer defined functions. As described above, the definition Def f ≡ E results in the rule f:x → E:x being added to the set of rewrite rules. This will treat recursive definitions such as Def f ≡ E(f) correctly, since the CPU will not attempt to rewrite the recursive occurrence of f within E(f) until it actually gets applied to some argument, i.e. until E(f):x gets rewritten to some expression that has f:x as the leftmost innermost subexpression.

However, if the recursive definition of a higher-order function such as Def H ≡ E(H) were to produce the rewrite rule H:f → E(H):f, then in reducing an expression such as (H:f):x, the leftmost innermost reducible expression would always be located in the subexpression (H:f), and the process would diverge, forever developing the function H:f and never getting around to applying it to the argument x. To avoid this problem, definitions can be marked as being second level by writing Def_2 H ≡ E(H). This results in the rewrite rule H:f:x → E(H):f:x. Using this rule, a higher-order function application such as H:f will not be reduced to E(H):f

unless it occurs in a context where the resulting function is being applied to some argument. This device permits the convenient definition of higher-order functions within a strict (call by value) semantics.

The denotational semantics. Every FL expression is considered to denote some semantic object, its *meaning*. This is the case for all syntactic objects, from the simplest such as atoms (the expression 1 has the meaning 1) to the more complex such as applications (tl:<id,3> denotes <3>). Moreover, the meaning of a compound expression is expressed as a function of the meanings of its components; in the above example, the meaning of tl is a function *tl*, which when applied to a nonempty sequence of semantic objects and a history *h* produces the tail of that sequence and the same *h* (e.g., tl:<<*id*,3>,*h*> = <<3>,*h*>).

$\mathcal{H}$ is used to denote the set of all possible histories. The semantic domain $\mathcal{D}$ of FL is the smallest set that contains the element $\perp$ (representing the "undefined" element), the atoms $\mathcal{A}$, and the functions $\mathcal{F}$ (e.g., *tl* and *add*) from $\mathcal{D} \times \mathcal{H} \rightarrow \mathcal{D} \times \mathcal{H}$, and is closed under the operations of sequence formation (e.g. <*tl*,3>) and tagging with strings (e.g. ⟨ 'gram',17 ⟩). Strictness requires that $\perp$ cannot appear in a sequence or a tagged pair.

This domain is sufficiently rich so that the meaning of any FL expression can be given by a *meaning function* μ. Just as the operational semantics deals with two components, an expression and a history, μ maps an expression and a history into a meaning and a history; i.e., the functionality of μ is

$$\mu : \text{expr} \times \mathcal{H} \rightarrow \mathcal{D} \times \mathcal{H}.$$

The history component can be conveniently thought of as a (potentially) infinite sequence of objects from $\mathcal{D}$. E.g., μ(out:<'screen',add:<1,2>>, *h*) = (<'screen',3>,*h'*) where *h'* is *h* with <'outflag',<'screen',3>> appended to the end.

(Note to language specialists: Special care has been taken to design $\mathcal{D}$ so that simple concepts have simple representations. Retracts, isomorphisms, and inverse limits are not required in order to understand the meaning of sequences. The machinery to make the domain mathematically sound is hidden in the representation of functions like *tl*, where the programmer needn't be concerned with them. Concentrating the set theoretic machinations of the domain within the function representations complicates their mathematical structure, but it results in a domain that is a natural, intuitively appealing tool that programmers can use to understand and reason about their programs.)

Because $\mathcal{D}$ is closed under tagging, it contains representations for all the new data objects that might ever be defined in FL programs. This provides programmers with a convenient mental representation for expressions that evaluate to (abstract) data types without requiring them to think of such expressions as denoting function closures or special hiding mechanisms in the semantic domain.

Examples of FL programs

In presenting these examples, language constructs are used without giving complete definitions or descriptions of them in the hope that seeing a few sample programs will serve to give the flavor of the language (and to motivate the following section in which the language is described precisely). Each example consists of a program, an instance of its use, and some brief comments.

1. A program to compute the sum and product of two numbers.
   - $[+,*]$
   - $[+,*]:\langle 2,3\rangle \rightarrow \langle +:\langle 2,3\rangle, *:\langle 2,3\rangle\rangle \rightarrow \langle 5,6\rangle$.
   - This program uses the combining form *construction* ([ ]) which is defined so that in general $[f_1,\ldots,f_n]:x \rightarrow \langle f_1:x,\ldots,f_n:x\rangle$.
2. A simple function definition.
   - Def second $\equiv$ s1 $\circ$ tl
   - second : $\langle 2,3,4\rangle \rightarrow (s1 \circ tl):\langle 2,3,4\rangle \rightarrow (\circ:\langle s1, tl\rangle):\langle 2,3,4\rangle \rightarrow s1:(tl:\langle 2,3,4\rangle) \rightarrow s1:\langle 3,4\rangle \rightarrow 3$.
   - In general si is the primitive function that selects the ith element of a sequence, and tl is the primitive function that returns all but the first element of a sequence. Thus second is defined to be the same as the primitive s2. This is a simple example of the use of infix notation in expressions. The functional $\circ$ (which is defined so that $(\circ:\langle f,g\rangle):x = f:(g:x)$) is applied to the two arguments s1 and tl. In general, f op g means op:$\langle f,g\rangle$.
3. A recursive factorial program.
   - Def fact $\equiv$ id = $\tilde{}0 \rightarrow \tilde{}1$; id*fact $\circ$ sub1
   - fact : 5 $\rightarrow$ 120
   - The fact function is built by applying the combining forms *composition* ($\circ$), *constant* ($\tilde{}$) and *conditional*, to the primitive functions =, id, * and the (recursive) function name fact. As noted above, composition is defined so that $(f \circ g):x = f:(g:x)$. Constant is defined so that $\tilde{}x:y = x$ (provided that y terminates), and conditional is defined so that $(p \rightarrow f; g):x = f:x$ if $p:x$ is true, and $g:x$ otherwise. Thus, fact : 5 rewrites to 5*(fact : 4) (since 5 = 0 is false), etc.
4. Another version of factorial.
   - * $\circ$ ints_to
   - * $\circ$ ints_to : 5 $\rightarrow$ *:(ints_to:5)> $\rightarrow$ *:$\langle 1,2,3,4,5\rangle \rightarrow$ 120
   - ints_to is a primitive function such that ints_to:n = $\langle 1,\ldots,n\rangle$ provided n is a positive integer. Note that * maps a sequence of numbers into the product of those numbers (*:< > = 1).
5. A program to compute the length of a sequence.
   - + $\circ$ $\alpha:\tilde{}1$

- $(+ \circ \alpha{:}\tilde{}1) : \langle a,b,c\rangle \rightarrow +{:}(\alpha{:}\tilde{}1{:}\langle a,b,c\rangle) \rightarrow +{:}\langle \tilde{}1{:}a, \tilde{}1{:}b, \tilde{}1{:}c\rangle$
  $\rightarrow +{:}\langle 1,1,1\rangle \rightarrow 3$
- This program works by simply replacing each element of the sequence by a 1 and then adding them up. The combining form α (for *apply to all*) is defined so that $(\alpha{:}f) : \langle x_1,\ldots,x_n\rangle = \langle f{:}x_1,\ldots,f{:}x_n\rangle$. Note that : associates to the left (i.e. $x{:}y{:}z = (x{:}y){:}z$) so that $\alpha{:}\tilde{}1{:}a$ means $(\alpha{:}\tilde{}1){:}a$.

6. A program to compute the average of a sequence of numbers.
   - Def ave $\equiv$ (+ ÷length where Def length $\equiv$ + $\circ$ $\alpha{:}\tilde{}1$ end)
   - ave : $\langle 3,4,8\rangle \rightarrow$ (+ ÷length) : $\langle 3,4,8\rangle \rightarrow (+{:}\langle 3,4,8\rangle) \div (\text{length} : \langle 3,4,8\rangle) \rightarrow 15 \div 3 \rightarrow 5$
   - Any expression (in this case the right hand side of the definition of ave) can have an associated list of definitions (in this case the simple definition Def length $\equiv$ + $\circ$ $\alpha{:}\tilde{}1$). Such definitions are *local*; i.e., their scope of definition is limited to the expression to which they are attached.
7. An order n log n sorting program.
   - Def sort $\equiv$ tree:$\langle$merge, $\langle\,\rangle\rangle \circ \alpha{:}$[id]
   - sort : $\langle 8,4,2,3\rangle \rightarrow$ (tree:$\langle$merge, $\langle\,\rangle\rangle \circ \alpha{:}$[id]) : $\langle 8,4,2,3\rangle$
     $\rightarrow$ tree:$\langle$merge,$\langle\,\rangle\rangle{:}(\alpha{:}$[id] : $\langle 8,4,2,3\rangle)$
     $\rightarrow$ tree:$\langle$merge,$\langle\,\rangle\rangle{:}\langle\langle 8\rangle,\langle 4\rangle,\langle 2\rangle,\langle 3\rangle\rangle$
     $\rightarrow$ merge:$\langle$merge:$\langle\langle 8\rangle,\langle 4\rangle\rangle$,merge:$\langle\langle 2\rangle,\langle 3\rangle\rangle\rangle$
     $\rightarrow$ merge:$\langle\langle 4,8\rangle,\langle 2,3\rangle\rangle \rightarrow \langle 2,3,4,8\rangle$
   - tree is a primitive combining form that takes a function and an object and produces a function such that tree:$\langle f,z\rangle{:}\langle x_1,\ldots,x_n\rangle$ rewrites to z if $n=0$, x_1 if $n=1$, and $f{:}\langle$tree:$\langle f,z\rangle{:}\langle x_1,\ldots,x_m\rangle$, tree:$\langle f,z\rangle{:}\langle x_{m+1},\ldots,x_n\rangle\rangle$ if $n>1$ (where $m=$ ceil:$(n \div 2)$). merge is a primitive function that merges two sorted sequences of numbers into a single sorted sequence; e.g., merge:$\langle\langle 3,6\rangle,\langle 1,3,5\rangle\rangle = \langle 1,3,3,5,6\rangle$ (Notice that α:[id] maps an *n*-element sequence into a sequence of *n* singleton sequences.)
8. A program for doing a "protected" divide.
   - Def safediv$_1$ $\equiv$ ispair $\rightarrow$ ((isnum $\circ$ s1 $\wedge$ (isnum $\wedge$ not $\circ$ iszero) $\circ$ s2) $\rightarrow$ s1 ÷ s2; $\tilde{}$'diverror');
     $\tilde{}$'diverror'
   - safediv$_1$: $\langle 8,4\rangle \rightarrow 2$
   - safediv$_1$: $\langle 8,4,2\rangle \rightarrow$ 'diverror' (since $\langle 8,4,2\rangle$ is not a pair)
   - safediv$_1$: $\langle 8,0\rangle \rightarrow$ 'diverror' (since s2:$\langle 8,0\rangle$ is 0)
   - safediv first checks that its argument is a pair, then that the first element of the pair is a number and the second element of the pair is a non-zero number; if so, it divides the first by the second, otherwise it returns 'diverror'.
9. An equivalent program that uses a *pattern* to create function names for the arguments.
   - Def safediv$_2$ $\equiv$ $[\![n, m]\!] \rightarrow ((\text{isnum} \circ n \wedge (\text{isnum} \wedge \text{not} \circ \text{iszero}) \circ m) \rightarrow n \div m;\ \tilde{}\text{'diverror'});$
     $\tilde{}$'diverror'

- safediv$_2$: <8, 4> → 2
- safediv$_2$: <8, 4, 2> → 'diverror' (since <8, 4, 2> fails to match the pattern [[n., m.]])
- safediv$_2$: <8, 0> → 'diverror' (since not ∘ iszero is false for m)
- The pattern mechanism is defined so that safediv$_2$ is just a notational shorthand for safediv$_1$. I.e., the pattern [[n., m.]] produces a predicate that tests for the structure of the pattern (ispair in this case) and also defines the function names indicated by "." (n and m in this case) to be selector functions that correspond to their places in the structure (in this case s1 and s2 respectively). These resulting function definitions are local; their scope is limited to the "true" and "false" arms of the conditional.

10. Another equivalent version of safediv that uses embedded predicates within the pattern.
    - Def safediv$_3$ ≡ [[n.isnum, m.isnum∧not ∘ iszero]] → n ÷ m; ~'diverror'
    - safediv$_3$: <8, 4> → 2
    - safediv$_3$: <8, 0> → 'diverror' (the pattern fails to match, since not ∘ iszero is false for 0)
    - Again, the pattern mechanism is defined so that safediv$_3$ is exactly equivalent to safediv$_1$. To match a pattern with embedded predicates, not only must the argument have the proper structure, but its (sub)components must match the corresponding predicates as well. This ability to embed predicates in patterns greatly increases the usefulness of the pattern matching mechanism for writing terse, easy to read function definitions. Note that the precedence of the infix operators ., ∧, and ∘ is such that m.isnum∧not ∘ iszero means m.(isnum ∧ (not ∘ iszero))
11. An interactive program to prompt for two numbers and print their sum.
    - Def adder ≡ out_screen ∘ (~'sum = ' ++ + ∘ getnums)

      where

      Def out_screen ≡ out ∘ [~'screen', id]

      Def getnums ≡ [prompt ∘ ~'enter a', prompt ∘ ~'enter b']

      Def prompt ≡ in ∘ ~'keyboard' ∘ out ∘ [~'screen', id]

      end

      { (This is a comment) ++ catenates strings }
    - adder : 'x'

      → (out_screen ∘ (~'sum = ' ++ + ∘ getnums)) : 'x'

      → out_screen:(++ :<'sum = ', +:(getnums:'x')>)

      → out_screen:(++ :<'sum = ', +:<prompt:'enter a', prompt:'enter b'>>)

      → out_screen:(++ :<'sum = ', +:<1, prompt:'enter b'>>)

      (assuming that the user types 1 in response to the prompt message enter a)

      → out_screen:(++ :<'sum = ', +:<1,2>>)

      (assuming that the user types 2 in response to the prompt message enter b)

      → out_screen:(++ :<'sum = ', 3>)

      → out_screen:'sum = 3'

→ (out∘[~'screen', id]):'sum = 3'

→ out:<'screen','sum = 3'>

→ <'screen','sum = 3'> (and sum = 3 appears on the screen)

- This program is independent of the value to which it is applied ('x' in this case). Its real purpose is to prompt for two numbers and output their sum. out is a primitive function that, when applied to a pair <devicename, x>, evaluates to <devicename, x> and causes x to be written on the output device named devicename. in is a primitive function that, when applied to an argument devicename, reads the next input token from the device named devicename and evaluates to that input token. Thus, prompt:'enter a' rewrites to (in∘~'keyboard'):(out:<'screen', 'enter a'>), which rewrites to (in∘~'keyboard'):<'screen', 'enter a'>, which (and writes enter a on the device named 'screen'). This latter expression then rewrites to in:'keyboard', which gets the next input from the device named 'keyboard' and rewrites to that input value. Note that ++ , the primitive for catenating strings, will coerce non-string arguments (in this case the integer 3) into strings automatically. Like the primitives + and *, ++ is *multilevel*; i.e., when applied to *objects* it performs the catenation function, but when applied to *functions* (e.g. ++ :<f,g>), it rewrites to ++ ∘[f,g], thus avoiding the need to do explicit lifting (see Section 3). Text enclosed in { } pairs is a comment and is lexically equivalent to a blank space.

3. Simple Programs

To describe the syntax and semantics of FL expressions, it is convenient to proceed in stages, beginning with a primitive kernel language called *core expressions*, whose syntax and semantics are easily described. Although it is universal (in the sense that any computable function is expressible in it), this kernel is extremely Spartan and has no concessions to ease of programming or readability. In fact, it doesn't even contain the notion of auxiliary program definitions; all core expressions must be written solely in terms of primitive functions.

The first step in enriching core expressions is to include some convenient syntactic abbreviations for some of the commonly occurring forms and to allow expressions to be built up from programmer defined functions as well as primitive functions. Such enriched expressions are called *simple expressions*, which when combined with a simple list of declarations for programmer defined functions (known as a *simple declaration list*), becomes a *simple program*, i.e. an expression of the form simple_expr where simple_decl_list end.

Simple programs form a conceptually appealing half-way point in the development of complete FL expressions. They provide the programmer with a sufficiently rich syntax so that most small to medium sized programs can be expressed clearly and conveniently as simple programs, and, at the same time, allow the continued use of the relatively straightforward semantic techniques for core expressions in reasoning about programs.

Finally, simple programs are extended to allow declarations to be nested within expressions and other declaration lists and to allow function definitions to create local "names" for their arguments. These extensions result in the collection of full FL *expressions*; an FL program is such an expression. Rather than enhance the semantic mechanisms to handle such issues as definition hiding and nested scopes, the semantics of full FL is given by first describing how to map all FL expressions to their core expression equivalents and then applying the easy to understand semantics of core expressions.

The description of this development is divided into two sections. The development up through simple programs is given in this section, and the remainder is given in Section 4. The rest of this section is organized into the following subsections:

1. The Syntax of Core Expressions
2. The Semantics of Core Expressions
3. Simple Expressions
4. Simple Programs

The Syntax of Core Expressions

A *core expression* is (recursively) one of the following:

1. An atom (which may be a char, a number, or a truth value)
2. A primitive function name
3. ? (which is used to indicate an error)
4. A sequence. If $x_1, ..., x_m$ are core expressions, then the *sequence* $\langle x_1, ..., x_m \rangle$ is a core expression. The empty sequence $\langle\ \rangle$ is a core expression. A sequence of chars is called a *string*, which may be abbreviated as described below.
5. An application. If x_1, x_2 are core expressions, then the *application* $x_1 : x_2$ is a core expression.
6. A tagged pair. If x_1 is a string and x_2 is a core expression, then the *tagged pair* $\prec x_1, x_2 \succ$ is a core expression.

Expressions may contain comments (described below). The following describes the syntax of atoms, function names, abbreviations for strings, and comments.

Atoms. The set of *atoms* is the union of three disjoint sets: the chars, the numbers, and the truth values.

- A *char* is any single printable letter, symbol, or punctuation mark preceded by the double quote symbol ("). For example, "a and "" are chars.
- A *number* has the form sIFE where s represents a sign and is either +, −, or empty, I represents an integer and is a string of (zero or more) digits (0 through 9), F represents a fractional part and is either empty or a period (.) followed by a string of (zero or more) digits, and E represents an exponent and is either empty or an (upper or lower case) e followed by something of the form sI. At least one of the strings I or F in sIFE must

have at least one digit. For example, −40500e−2 is the integer -405, 3. is the integer 3, and −.2056E1 is the real number -2.056.

- A *truth value* is one of the symbols true or false.
- A sequence that is a *string* may be abbreviated by juxtaposing the chars in the sequence and removing the double quote marks. For example, <"a,"b> is abbreviated 'ab'. The symbol " is used as an escape character so that ' or " may appear in a string. For example, <"a,"',"","b> is abbreviated 'a"'""b'. Note that <> and '' both denote the empty string (i.e. the empty sequence).

Function names. A *function name* is either one of the symbols

∘ + * ÷ − = ≤ < ≥ > ≠ / \ ↦ ↤ ∧ ∨ ++

or the juxtaposition of one or more of the following

- All upper or lower case Roman letters
- All upper or lower case Greek letters
- The digits 0, 1, ... , 9
- The special characters !, $, ¢, %, #, ¬, ∇, □, _

provided that the initial character is not a digit. There are about 70 primitive functions in FL. The complete list is given in [BWW 86].

Comments. A *comment* is any sequence of characters and/or symbols enclosed in braces { }. Comments may be nested. A comment may appear anywhere in an expression that a blank space may appear; lexically it is equivalent to a blank space.

Objects. It is useful to identify a special subset of core expressions. An *object* is an atom or (recursively) a sequence of objects.

The Semantics of Core Expressions

The Denotational Semantics. The *meaning* of a core expression is provided by the denotational semantics of FL. Section 2 has already explained how the meaning function μ maps a pair (expr,h) into (d_1, h_1), where h is the initial history that provides the I/O and file system environment in which the expression expr is to be evaluated, and d_1 (respectively h_1) is the element of the semantic domain $\mathscr{D}$ (respectively $\mathscr{H}$, the set of histories) that results from evaluating the expression expr with the history h.

The semantic domain $\mathscr{D}$ for FL is unusual in that it actually *contains* the set $\mathscr{A}$ of atoms, not some isomorphic image of them. (But $\mathscr{A}$ contains only the "canonical forms" of numbers.) Thus $\mathscr{D}$ contains the set of atoms, $\mathscr{A}$, an isomorphic copy of the set $\mathscr{F}$ of "good" functions that map $\mathscr{D} \times \mathscr{H}$ into itself, and $\perp$, and it is closed under sequence formation and tagging with strings. $\mathscr{H}$ is the set of all "semantic histories", i.e., those that contain the semantic counterparts of values (e.g., the semantic counterpart of the function value id is the mathematical function id such that $id(d,h) = (d,h)$; the counterpart of the string value 'abc' is

just that string itself). The following gives, informally, the denotational semantics of core expressions.

The meaning of an atom is the canonical form of that atom. (Every non-number atom is in canonical form; every number has a canonical form, which is a number.)

The meaning of a function name corresponds to a function from $\mathscr{D} \times \mathscr{H}$ into itself. E.g., the meaning of tl is the function tl, where $tl(\langle x_1, ..., x_n \rangle, h) = (\langle x_2, ..., x_n \rangle, h)$

The meaning of ? is $\perp$.

The meaning of the sequence $\langle x_1, ..., x_n \rangle$ is the sequence of the meanings of its components except in the case that one of the components has meaning $\perp$. More formally, $\mu(\langle x_1, ..., x_n \rangle, h_1) = (\langle y_1, ..., y_n \rangle, h_{n+1})$ where $\mu(x_i, h_i) = (y_i, h_{i+1})$, provided no y_i is $\perp$. If some $y_i = \perp$, then $\mu(\langle x_1, ..., x_n \rangle, h_1) = (\perp, h_{k+1})$ provided $y_k = \perp$ and $y_j \neq \perp$ for $j < k$.

The meaning of the application $x_1 : x_2$ is the result of applying the function that is the meaning of x_1 to the meaning of x_2. More formally, $\mu(x_1 : x_2, h_1) = y_1(y_2, h_3)$ provided y_1 is a function (mapping $\mathscr{D} \times \mathscr{H}$ to itself) where $\mu(x_i, h_i) = (y_i, h_{i+1})$. If y_1 is not a function and not $\perp$, then $\mu(x_1 : x_2, h_1) = (\perp, h_3)$. If $y_1 = \perp$, then $\mu(x_1 : x_2, h_1) = (\perp, h_2)$.

The meaning of the tagged pair $\prec x_0, x_1 \succ$ is the meaning of x_1 tagged by x_0. (Recall that x_0 must be a string.) More formally, $\mu(\prec x_0, x_1 \succ, h_1) = (\prec x_0, y_1 \succ, h_2)$ where $\mu(x_1, h_1) = (y_1, h_2)$. Tagged pairs may be created only by the rewriting system itself and may *not* be written by the programmer (since tagged pairs are designed for use in the implementation of data types).

The Operational Semantics of Core Expressions. The operational semantics of FL is based on a rewriting process. In such a process, an expression is rewritten to another according to a set of rewrite rules. This process continues until it is not possible to rewrite the expression any further. An expression that cannot be rewritten is called a *value* and represents the (fully evaluated) "answer" for the original expression. As a consequence of the rewrite rules for FL, every atom is a value, any sequence all of whose components are values is a value, and any tagged pair with a value as its second component is a value. The only other kind of values are irreducible function expressions (i.e., expressions to which no rewrite rules apply and whose meanings are functions) such as $\circ : \langle tl, tl \rangle$. An expression x is said to have the value v iff v is a value and x rewrites to v. Not all expressions have values since it is possible for the rewriting process to continue forever; in this case, the expression has meaning $\perp$.

The rewriting process is *sound*: it preserves the (denotational) meaning of expressions. If e_1 can be rewritten to e_2, then the meaning of e_2 is the same as the meaning of e_1; moreover, if e_2 is an object, then it *is* the meaning of e_1 (provided numbers are in canonical form). Furthermore, the rewriting process is *complete*; this implies that if the meaning of e_1 is the object e_2, then the rewriting process will rewrite e_1 to e_2.

The rewriting process consists of locating a redex (i.e., a subexpression matching the left hand side of some rewrite rule), replacing it by the corresponding right hand side, and repeating until there are no more redexes in the final expression. For example, the rewrite rule for the identity function id is $\mathsf{id}{:}x \rightarrow x$ indicating that the function id when applied to the value x rewrites to x. Another example of a rewrite rule is $\mathsf{cons}{:}\langle f_1, \ldots, f_n\rangle{:}x \rightarrow \langle f_1{:}x, \ldots, f_n{:}x\rangle$ indicating that cons when applied to a sequence of n functions yields a function that always produces a sequence of length n as its answer.

Since the evaluation of an expression can require I/O operations, it is important to specify the order in which the rewrite rules are applied. The redex chosen is always the leftmost innermost redex. This requires that all proper subexpressions of the left hand side of a rewrite rule be values and that rewriting be done left to right. So in order to rewrite x_k in the expression $\langle x_1, \ldots, x_n\rangle$, it is necessary that each x_i be a value for $i = 1, \ldots, k-1$. Similarly, in order to rewrite x_2 in the expression $x_1{:}x_2$, it is necessary that x_1 be a value. So, for example, the expression $\langle \mathsf{out}{:}\langle \mathsf{'screen'}, \mathsf{'hi'}\rangle, \mathsf{out}{:}\langle \mathsf{'screen'}, \mathsf{'bye'}\rangle\rangle$ is first rewritten to $\langle\langle \mathsf{'screen'}, \mathsf{'hi'}\rangle, \mathsf{out}{:}\langle \mathsf{'screen'}, \mathsf{'bye'}\rangle\rangle$ causing "hi" to appear on the screen and then is rewritten to $\langle\langle \mathsf{'screen'}, \mathsf{'hi'}\rangle, \langle \mathsf{'screen'}, \mathsf{'bye'}\rangle\rangle$ causing "bye" to appear on the screen. This final expression is a value.

The complete list of the rewrite rules and an informal description of all the primitive functions are given in [BWW 86]. In general, these functions act by rewriting the expression and adding some notations (if appropriate) to the history stream to indicate their actions.

The rewriting process has been presented as one of rewriting expressions. However, the purpose of this process is to transform a pair (expr, hist) into $(\mathsf{expr}_1, \mathsf{hist}_1)$, since the rewriting of expr may involve a history change. The pair transformation associated with any rewrite rule is just that of applying the rule to rewrite expr to obtain expr_1 and to make any additions to hist that the rule requires to obtain hist_1. E.g., ($\langle \mathsf{id}{:}3, 4\rangle$, hist) is transformed by the rewrite rule for id into ($\langle 3, 4\rangle$, hist). Observe that, if e is an expression that does I/O but rewrites forever, then (e, h) will have the meaning $(\perp, h_1)$ and the (potentially infinite) history h_1 is an accurate record of the I/O activity of e.

As a result of the rewrite rule $? \rightarrow ?$, once the rewriting process causes ? to appear in an expression, the rewriting process will forever rewrite ? to itself. Thus, ? has meaning $\perp$ and is used to indicate that some sort of error has occurred. For example, the whole sequence $\langle a, ?\rangle$ has meaning $\perp$ (and rewrites to itself forever) since its second component (?) has meaning $\perp$ (and rewrites to itself forever). Error handling is an extra-language notion, and the exact manner in which errors are handled is implementation-dependent. It is envisioned that implementations will handle errors by informing the user about the nature of each error.

Simple Expressions

Although the primitive functions provided in [BWW 86] are sufficient to express any computable function, it is important to have a more convenient notation for expressing functions. Therefore, the language provides special notations for some of the commonly used combining forms, and it provides a convention for infix notation, designed to improve the readability of function expressions. In addition, to reduce the need for parenthetical groupings, the language also employs precedence conventions for these special forms. (All the rewrite rules for these combining forms can be found in [BWW 86].)

Special notation for combining forms.

1. **Prime:** f' abbreviates lift:f and is useful in infix notation (see below). For example, $f':\langle x,y\rangle$ has the same meaning as both $x\ f'\ y$ and $f\circ[x,y]$.
2. **Construction:** $[^{(n)}f_1,\ldots,f_m]$ abbreviates $\text{cons}^{(n)}:\langle f_1,\ldots,f_m\rangle$ where the superscript n indicates the number of primes. Thus, $[f_1,\ldots,f_m]:x$ rewrites to $\langle f_1:x,\ldots,f_m:x\rangle$.
3. **Predicate-Construction :** $[\![^{(n)}f_1,\ldots,f_m]\!]$ abbreviates $\text{pcons}^{(n)}:\langle f_1,\ldots,f_m\rangle$ where the superscript n indicates the number of primes. Predicate-construction is a predicate combining form such that $[\![f_1,\ldots,f_m]\!]$ is true just for sequences of length m whose elements satisfy, one-for-one, the predicates f_i. Thus, $[\![f_1,\ldots,f_m]\!]:\langle x_1,\ldots,x_k\rangle$ rewrites to $\text{and}:\langle f_1:x_1,\ \ldots\ ,f_m:x_m\rangle$, provided $m=k$; otherwise if $m\neq k$ it rewrites to <u>false</u>. $\text{and}:\langle f_1:x_1,\ \ldots\ ,f_m:x_m\rangle$ rewrites to <u>true</u> provided no $f_i:x$ is <u>false</u>; otherwise it rewrites to <u>false</u>.
4. **Constant:** $\sim x$ abbreviates $K:x$. Thus, ~ 2 abbreviates $K:2$, and $\sim$'abc':y rewrites to 'abc' (provided y does not have meaning $\perp$).
5. **Conditional:** $(p\rightarrow^{(n)}f;\ g)$ abbreviates $\text{cond}^{(n)}:\langle p,f,g\rangle$ where the superscript n indicates the number of primes. Informally,

$$(p\rightarrow f;\ g):x=\begin{cases} f:x & \text{if } p:x\neq \underline{\text{false}} \text{ or } ?\\ g:x & \text{if } p:x=\underline{\text{false}}\\ ? & \text{otherwise}\end{cases}$$

Because it occurs so frequently, FL also allows $p\rightarrow f$ as an abbreviation for $(p\rightarrow f;\ \text{err})$. (This produces a "dangling else" phenomenon, which is resolved below in the discussion of precedence conventions.) Note that conditional allows predicates p such that p:x is true if it is any value except <u>false</u>, and false only if it is <u>false</u>. However, all primitive predicates yield <u>true</u> or <u>false</u>.

Multilevel functions. For a few commonly used functions f whose arguments are always sequences of *objects* (e.g., +), it is useful to extend their domain to include sequences of functions by adding the rewrite rule

$f:\langle g_1,\ldots,g_n\rangle \rightarrow f\circ[g_1,\ldots,g_n]$ if all the g_i's are *functions*

With this extension f is called a *multilevel* function; for example, + is multilevel, so that, if g and h are functions then $+:\langle g,h\rangle:x = +\circ[g,h]:x = +:\langle g:x,\ h:x\rangle$. Multilevel functions are

particularly convenient in connection with infix notation as described below. The multilevel functions are: $+ \; - \; * \; \div \; < \; \leq \; > \; \geq \; ++$.

Infix notation. In addition to the extensions described so far, the language includes a convention for infix notation in which $op:\langle f, g\rangle$ can be written simply $f \; op \; g$. Thus, $\circ:\langle \circ:\langle f, g\rangle, h\rangle$ can be expressed in the more legible form, $f \circ g \circ h$. (The primitive $\circ$, composition, has the rewrite rule $\circ:\langle f_1, \ldots, f_n\rangle:x \rightarrow f_1:(f_2:\ldots(f_n:x)\ldots)$; rules for the precedence and associativity of infix functions are described below.) Since + and other common functions are multilevel functions, the following properties of such infix operations apply (using + as an example):

$1 + 2 = 3$ since $1 + 2 = +:\langle 1, 2\rangle$

$(f + g):x = f:x + g:x$ if f and g are functions

Precedence. All of the abbreviations just introduced have a precedence. The precedence rules are realized by fully parenthesizing an arbitrary expression in accordance with the following precedence list. The parenthesizing algorithm is: find the smallest, leftmost, non-parenthesized subexpression that has the form of the first entry in the precedence list (i.e., e') and put parentheses around it; repeat until all subexpressions with the given form have been parenthesized. The algorithm then repeats this process for each entry in the precedence list in order, except that for the conditional entry, the search is for the smallest *rightmost* non-parenthesized conditional. (Thus all the forms are left-associative except conditional which is right-associative.)

e'

$e_1:e_2$

$\sim e$

$e_1 \circ e_2$

$e_1 * e_2$ 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 $e_1 \dashv e_2$ (See Patterns in the next section for the definitions of $\mapsto$ and $\dashv$.)

$e_1 \wedge e_2$ or $e_1 \vee e_2$

$e_1 \; e_2 \; e_3$ provided this form is not covered by any other case

$p \rightarrow f; g$ or $p \rightarrow f$ (This is the only right-associative case.)

e_1 **where dl end**

let dl in e_1

$fn \equiv e_1$

Some examples of parenthesized expressions:

Before Parenthesizing	After Parenthesizing
a → b; c → d; e	(a → b; (c → d; e))
a → b; c → d	(a → b; (c → d))
a → b → c; d	(a → (b → c; d))
a → b → c	(a → (b → c))
f + g = ~h′ ∘ k	((f + g) = ((~(h′)) ∘ k))

Simple expressions. The *simple expressions* are just the core expressions with two differences:

1. All function names are allowed in simple expressions rather than just the primitive function names.
2. All the special notation described in this subsection is allowed in simple expressions.

Thus, the simple expressions are a superset of the core expressions.

Simple Programs

Simple declaration lists. A simple declaration list has the form

Def g_1 ≡ e_1 ... Def g_n ≡ e_n

where $e_1, ..., e_n$ are simple expressions and $g_1, ..., g_n$ are distinct function names. Def g ≡ e is a *definition* that defines g. (Other kinds of declarations, and additional ways of making definitions are introduced in Section 4.) The definitions in the declaration list may be (mutually) recursive. For example, Def fact ≡ id = ~0 → ~1; id * fact ∘ sub1 recursively defines the factorial function (where sub1 is the function that subtracts one).

Simple Programs. A *simple program* has one of three forms:

e

e where deflist end

let deflist in e

where e is a simple expression and deflist is a simple declaration list. The last two are equivalent.

Since simple programs are a superset of core expressions, it is natural to extend the semantics of core expressions to handle simple programs. This can be done by adding new rewrite rules for each of the function names defined in the simple declaration list; i.e., the definition Def f ≡ e in deflist produces the rule f:x → e:x. The value of e where deflist end can then be obtained by rewriting e with this augmented set of rewrite rules. Such an extension provides a conceptually simple treatment of the semantics of simple programs and is correct provided that the program evaluates to an object and that only objects are read from or written into the history. However, simple programs can contain operations that do not satisfy this proviso.

The representation of functions. If a program outputs a function, exactly how will it be represented? If a function is printed, how will it appear? E.g., if the simple program `out:<'printer', f> where Def f ≡ fl end` causes "f" to be printed, that might be meaningful to the writer of the program, who might know that f denotes the function fl. However, if the representation "f" were used to store that function in the file system and later that representation were to be recalled and applied to form f:x in another program with a different definition list, then the function applied to x might not be the same as the one originally stored.

Such considerations make it necessary to represent a function as an expression that is not context dependent, i.e., that has no free function names. Therefore, in the above example, instead of the expression f, the expression `f where Def f ≡ fl end` would be printed.

This completes the development of simple programs. In fact, most of the examples in Section 2 were written using simple programs.

4. Additional Language Constructs

This section contains features useful for higher-order programming as well as useful ways for defining functions. As with simple programs, rather than extend the operational and denotational semantics to cover these new features, their meaning is in terms of their core expression equivalents.

Patterns

It often happens that one wants to write a conditional that tests whether an argument has a particular structure, and if so, to apply a function to the various components of that structure. This can be accomplished using the primitive selector functions, but it is convenient to have a mechanism that at once tests for structure *and* permits the local definition of functions for accessing those components. For this reason the language provides an additional syntactic construct, *patterns*, which can appear only in the first position of a conditional. Thus conditional is extended to include the form $\text{pattern} \rightarrow e_1; e_2$.

A pattern is a structured expression that (a) denotes a predicate that is true for arguments whose structure is similar to that of the pattern, and (b) defines function names to select argument elements whose positions in the argument are those of the defined function names in the pattern. For example, the pattern `[[x.isstring, [[y.isint, z.isint]]]]` tests its argument to make sure it is a pair whose first element is a string and whose second is a pair of integers. The functions x, y, and z are defined by the pattern (indicated by ".") to be functions that select the parts of their argument that occupy the same positions that they themselves occupy in the pattern. Thus the pattern denotes the predicate `[[isident, [[isint, isint]]]]` and makes the definitions `Def x ≡ s1`, `Def y ≡ s1 ∘s2` and `Def z ≡ s2 ∘s2`. The predicate denoted by a pattern is called its *predicate part*, and the set of definitions it makes is called its *definition part.*

The most elementary patterns define a function name; thus the pattern x. denotes the identically true predicate T and defines x to be the identity function id. The pattern x.isint denotes the predicate isint and defines x to be id. Compound patterns are built from elementary ones using *predicate combining forms.* A pattern can also be built by composing a pattern with a function. (See the following table that gives the predicate and definition parts for patterns.)

The predicate combining forms are: $[\![\ldots]\!]$ (predicate-construction), $\mapsto$ (predicate append left), $\dashv$ (predicate append right), and $\wedge$ (Boolean and). $[\![p_1, \ldots, p_n]\!]$ is true only for a sequence of length n for which each p_i is true of the ith element; $p \mapsto q$ is true for a non-empty sequence for which p is true for its first element and q is true for its tail; $p \dashv q$ is true for a non-empty sequence for which q is true for its last element and p is true for its tail-right.

When patterns are combined with predicate combining forms, the resulting pattern denotes a predicate that is the combination of the predicates denoted by its constituents; the functions it defines are those its constituent patterns would have defined individually, but modified to reflect their positions in the new pattern. There must be only one instance of each defining function name in a pattern. Thus in $[\![$x.isstring, $[\![$y.isint, z.isint$]\!]]\!]$, while the elementary pattern x.isstring alone defines x to be id, this compound pattern defines x to be the function that selects the first element of a sequence (i.e., id$\circ$s1) and it defines y to select the first element of the second element of a sequence (i.e., id$\circ$s1$\circ$s2).

The fundamental tenet of FL, that programs be written at the function level, means that patterns can take advantage of the power of FL without having to depend on a subsidiary unification algorithm. Thus, as in the example, in addition to just giving a pattern to denote structures of the form <x,<y,z>>, qualifying predicates for x, y, and z can also be included, as well as other predicates for the whole argument or for its parts.

The predicate and definition parts of an arbitrary pattern are given in the following table. Since patterns are defined recursively, the following notational conventions are used: p_i is the predicate part of the sub-pattern P_i; *Defs*(P) is the definition part of the pattern P (in which a typical definition is d $\equiv$ D); fname denotes an arbitrary function name and e an arbitrary expression (which usually denotes a predicate).

Pattern	Predicate Part	Definition Part
fname.	T	{fname ≡ id}
fname.P	p	{fname ≡ id} U *Defs*(P)
e	e	{ }
[[$P_1, \ldots, P_n$]]	[[$p_1, \ldots, p_n$]]	$\bigcup_{i=1}^{n} \{d \equiv D \circ si \mid (d \equiv D) \in Defs(P_i)\}$
$P_1 \rightarrowtail P_2$	$p_1 \rightarrowtail p_2$	$\{d \equiv D \circ s1 \mid (d \equiv D) \in Defs(P_1)\}$ U $\{d \equiv D \circ t1 \mid (d \equiv D) \in Defs(P_2)\}$
$P_1 \rightarrowtail P_2$	$p_1 \rightarrowtail p_2$	$\{d \equiv D \circ t1r \mid (d \equiv D) \in Defs(P_1)\}$ U $\{d \equiv D \circ r1 \mid (d \equiv D) \in Defs(P_2)\}$
P ∘ e	p ∘ e	$\{d \equiv D \circ e \mid (d \equiv D) \in Defs(P_1)\}$
$P_1 \wedge P_2$	$p_1 \wedge p_2$	$Defs(P_1)$ U $Defs(P_2)$

Patterns may appear only in the predicate part of conditionals, and the definitions they make may be seen only in the two arms of the conditional. The meaning of a conditional (P → e_1; e_2) with a pattern P in the predicate position is exactly the same as the conditional p → e_1 where *Defs*(P) end ; e_2 where *Defs*(P) end (assuming that the names in the right-hand-sides of the definitions of *Defs*(P) never refer to the functions defined in *Defs*(P)). Note that the scope of the functions defined by the pattern includes the false arm; it is sometimes useful to use the definitions even though the argument does not match the pattern.

The following table gives some examples of expressions written using patterns and their equivalent simple expressions.

Expression using patterns	Equivalent simple expression
[[x., isnum, y.isint]] → f ∘ [x, g ∘ [id, y]]; x	[[T, isnum, isint]] → f ∘ [s1, g ∘ [id, s3]]; s1
[[T, x.isnum]] ∘ s1 → f ∘ x; g ∘ x	[[T, isnum]] ∘ s1 → f ∘ s2 ∘ s1; g ∘ s2 ∘ s1
(x.isnum ↦ y.) → (f ∘ x) al′ (g ∘ y); h	(isnum ↦ T) → (f ∘ s1) al′ (g ∘ t1); h

This use of patterns must not be mistaken for a kind of λ-abstraction. Notice that [[x., y.]] → x ∘ y; err abbreviates [[T, T]] → s1 ∘ s2; err which, if applied to <f, <g, h>>, rewrites to g. Contrast this with λx, y . x ∘ y which, if applied to <f, <g, h>>, would produce f ∘ <g, h> (which is equivalent to err, since <g, h> is not a function).

Enhanced Definitions

FL includes some special syntactic notation for enhancing the readability of defined functions. There are two kinds of enhanced definitions. The first is useful for defining higher-order functions; it causes application of the defined function to be delayed until the

appropriate number of arguments is available. The second allows patterns to be written on the *left* side of a function definition to limit the kinds and shapes of actual arguments for which the function is defined and to name selectors for parts of a valid argument.

Both kinds of enhanced definitions correspond in a simple way to an ordinary definition. The following table shows how to reduce any enhanced definition to an ordinary one by using the correspondences it gives. The primitive function **args** has the rewrite rule **args**:n:f:x_1:....:x_n → f:x_1:....:x_n. In the table, P is a pattern, F is an "enhanced" left hand side (i.e., a function name or, recursively, F ← P), E is an expression, and i is a non-negative integer.

Enhanced Definition	Corresponding Definition
$\mathbf{Def}_i$ f ≡ E	Def f ≡ **args**:i:E
$\mathbf{Def}_i$ F ← P ≡ E	$\mathbf{Def}_i$ F ≡ (P → E; err)

For example, using the rules in the table, the enhanced definition $\mathbf{Def}_2$ ReverseComp ← [[f., g.]] ≡ g ∘ ′f reduces to $\mathbf{Def}_2$ ReverseComp ≡ ([[f., g.]] → g ∘ ′f; err) by using the second rule, which in turn reduces to the simple definition Def ReverseComp ≡ **args**:2:([[f., g.]] → g ∘ ′f; err) by using the first rule.
(Further elimination of the pattern and the ′ symbol gives the function Def ReverseComp ≡ **args**:2:([[T, T]] → ∘ ∘ [s2, s1]; err); the presence of **args**:2 prevents ReverseComp from rewriting except in the context ReverseComp:x_1:x_2.)

Nesting Declaration Lists

Since it is frequently useful to make auxiliary definitions that are visible only to a few select functions, declaration lists can be nested to allow this capability.

Instead of enclosing a nested declaration list with the keywords **where** and **end**, such a nested declaration list can begin with one of the following key phrases (and is always ended with the keyword **end**):

1. **begin** (indicating that all the functions defined in the declaration list are visible in the containing declaration list)
2. **export**(f_1, ..., f_n) (indicating that only the function names f_1, ..., f_n are visible in the containing declaration list)
3. **hide**(f_1, ..., f_n) (indicating that all but the function names f_1, ..., f_n are visible in the containing declaration list)

To aid the programmer in catching misspellings, the functions named in the **export** or **hide** clause must be distinct and must be a subset of the function names that are given definitions in the nested declaration list.

For example, the following table shows an expression before simplification and an equivalent expression with a simple declaration list. In the following E(x_1, ..., x_n) represents an expression

with the function names $x_1, ..., x_n$ occurring in it. Notice that, in the "After" column, the h in the main expression refers to the primitive definition supplied by the system if h is the name of a primitive function; and if h is not the name of a primitive function, then h is given the implicit definition **Def h ≡ err.**

Expression with Enhanced Definitions	Equivalent Simple Expression

$E_0(f, g, h)$ **where**
  export(f, g)
  Def $f \equiv E_1(f, g, h)$
  hide(f)
  Def $f \equiv E_2(f, g, h)$
  Def $g \equiv E_3(f, g, h)$
  end
  Def $h \equiv E_4(f, g, h)$
  end
end
	$E_0(f_1, g_2, h)$ **where**
	  Def $f_1 \equiv E_1(f_1, g_2, h_1)$
	  Def $f_2 \equiv E_2(f_2, g_2, h_1)$
	  Def $g_2 \equiv E_3(f_2, g_2, h_1)$
	  Def $h_1 \equiv E_4(f_1, g_2, h_1)$
	end

It is envisioned that the programming environment (e.g., the operating system, an editor, a language preprocessor, etc) will provide the capability of including one file in another by means of an "include" or "imbed" command. This command would probably insert the contents of one file into the file where the include command was located; a file representing an expression could then "include" another containing a declaration list. Since this is a feature of the programming environment, it is not described in this manual as part of the language itself.

Expressions

Section 4 has introduced several additional constructs to the expressions of the complete FL language. It is helpful to review the structure of expressions that incorporate these new constructs.

If **se** is a simple expression, $e, e_1, ..., e_n$ are expressions, and **declist** is a declaration list, then the following are FL *expressions*:

se

$\langle e_1, ..., e_n \rangle$

$e_1 : e_2$

$\prec$ string, e $\succ$

e'

$[^{(m)} e_1, ..., e_n]$

$[\![^{(m)}e_1, \ldots, e_n]\!]$

$\sim_e$

pattern $\rightarrow e_1 ; e_2$

pattern $\rightarrow e_1$

$e_1 \rightarrow^{(n)} e_1 ; e_2$

$e_1\ e_2\ e_3$ (infix notation)

e where declist end

let declist in e

The semantics of FL expressions. As discussed in Section 3, the semantics of an expression can be given indirectly, by rewriting the expression to its core language equivalent. The value/meaning of this core expression is then, by definition, the value/meaning of the original program, and the history it produces is, by definition, the one that the original program produces. As described in Section 3, the value (and resulting history) of a core expression is obtained by the rewrite rules of the operational semantics, and its meaning (and history) by the denotational semantics. An informal description of this rewriting process was given for each additional construct as it was introduced. A more formal description is given in [BWW 86].

5. Bibliography

[Backus 78] J. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *CACM* **21:8** (August, 1978), pp. 613-641.

[BWW 86] J. Backus, J.H. Williams, and E.L. Wimmers, FL Language Manual (Preliminary version), RJ 5339, IBM Almaden Research Center (November, 1986).

[Turner 82] D.A. Turner, Recursive equations as a programming language, *Functional Programming and its Applications*, Darlington *et al.* (editors), Cambridge University Press (1982).

[Williams&Wimmers 88] J.H. Williams, E.L. Wimmers, Sacrificing simplicity for convenience: Where do you draw the line? *Proceedings of the 15th ACM Conference on Principles of Programming Languages* (1988).