

Research Report

FL LANGUAGE MANUAL (PRELIMINARY VERSION)

John Backus
John H. Williams
Edward L. Wimmers

IBM Almaden Research Center
San Jose, CA 95120



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

Copies may be requested from:
IBM Thomas J. Watson Research Center
Distribution Services
Post Office Box 218
Yorktown Heights, New York 10598

PREFACE

The programming language described in this manual is part of an experiment in language design. Since, as part of this experiment, we propose to implement the kind of language we describe here and to make it widely available, we are soliciting comments about this preliminary version of the language before we undertake the extensive effort of producing and distributing a usable system.

There are a number of hypotheses/questions that we are trying to explore with this experimental language. Although some of these questions may not be entirely clear without reading some of the FL Language Manual, we want to raise them here so that the interested reader can keep them in mind while reading the manual. The three major questions are:

- Can a "function level" functional language be used to write all kinds of programs, including interactive ones, by a wide variety of users, not just professional programmers?
- Can such a language significantly reduce the time and effort required to think about, write, verify, and maintain programs?
- Can a compiler be constructed for such a language that will produce highly efficient programs?

We believe that these major questions depend on some subsidiary ones:

- Can a program be built by making extensive use of a large collection of general, off-the-shelf programs, rather than by building a program mostly from the language primitives?
- Is the proper functionality for a program one that maps the domain `Objects × States` into itself, where `States` contain file systems and I/O devices? Does this functionality facilitate writing interactive programs while retaining the most important algebraic properties of programs?
- Are the algebraic properties of programs strong enough to support extensive optimization of programs? Can optimization transform source programs operating on regularly-structured data into efficient object programs that use indexing and avoid the use of list structures and extra levels of indirection?
- Can the denotational semantics of programs be made simple enough to be a useful tool in thinking about programs and in proving the required algebraic properties? Does the use of a strict semantics make it easier to understand the meaning of a program? On the other hand, does its use make it more difficult to write programs?
- Can a type system that uses hierarchical types, places no requirements on the user, and does not guarantee complete type checking at compile time still be a useful one that provides powerful facilities for creating and using abstract data types? Can assertions about types in the form of functional equations be effectively used in type checking at compile time?
- The FL language incorporates several syntactic constructs (eg, "patterns", infix notation, precedence rules) to facilitate function level programming using higher order functions. Does this rich syntactic structure contribute significantly to writing programs that are concise, easily readable, and self-documenting?

We welcome any comments or suggestions about the language or about its description in the manual, or any opinions about the above questions.

This manual attempts both to explain the language clearly and to provide a detailed and precise reference manual. Readers should not be dismayed by its length. The introduction (Section 1, 10 pages) gives an overview of the language with a number of examples of programs. By page 20 the reader has all the technical information needed to write simple programs; by page 30, he/she can write complete programs; and by page 39, complete programs with defined data types. The rest of the manual consists of appendices.

FL LANGUAGE MANUAL (PRELIMINARY VERSION)

John Backus
John H. Williams
Edward L. Wimmers

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. It 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 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 is a primitive lifting operator for raising functions to functionals, which is useful for writing closed-form, combinatorial definitions of higher-order functions, and there is a mechanism for abstracting from function level expressions to produce lambda-like definitions of higher-order functions. This latter mechanism is a powerful generalization of the conditional function, which provides the convenience of pattern matching with predicates, a conditional form of abstraction, and a local function definition capability.

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.

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 operations 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 his 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.

Contents

1. Introduction	1
Function level programming	1
I/O and persistent memory	2
The semantics of FL	3
Examples of FL programs	6
 2. Simple Programs	 11
The Syntax of Core Expressions	12
The Semantics of Core Expressions	13
Simple Expressions	16
Simple Programs	19
 3. Additional Language Constructs	 21
Patterns	21
Conditional abstraction	23
Enhanced Definitions	24
Extending Functions	26
Nesting Declaration Lists	27
Expressions	28
 4. Data Types in FL	 31
Introduction	31
Primitive Types	31
Defined Types	32
Examples of Defined Types	36
Abstract Types	38
Primitive Abstract Types	39
 5. Appendix A: The Primitive Functions	 41
Arithmetic Operators	41
Boolean Operators	42
Comparison Operators	43
Combining Forms	44
Predicate Combining Forms	44
Sequence Combining Forms	45
Sequence Functions	46
String Functions	47
Integer Functions	47

Type Functions	47
Store Functions	49
Miscellaneous Functions	49
History Dependent Functions	50
 6. Appendix B: The Rewriting Process	 51
History Independent Functions	51
The History Dependent Functions	57
 7. Appendix C: The Simplification Algorithm	 61
The Function "Core"	61
The Function "SimpA"	62
Declaration List Algorithm	65
Abstraction Algorithm	68
 8. Appendix D: The Primitive Abstract Type store	 71

1. 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, ie 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 $x_1, \dots, x_n$ is written $\langle x_1, \dots, x_n \rangle$), and then abstracting with respect to x and y (eg $\text{Def sumprod} \equiv \lambda x, y. \langle x + y, x * y \rangle$), but rather by beginning with the functions $+$ and $*$ and by applying the combining form *construction* ($[]$), which is defined so that for any functions $f_1, \dots, f_n$, the function $[f_1, \dots, f_n]$ applied to x produces the n element sequence whose i th element is f_i applied to x ; ie, $[f_1, \dots, f_n] : x = \langle f_1 : x, \dots, f_n : x \rangle$ (note that the application of f to x is written $f : x$). This results in the function level definition $\text{Def sumprod} \equiv [+, *]$.

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 $f: \langle \text{val}, \text{history} \rangle \rightarrow \langle \text{val}', \text{history}' \rangle$.

Most of the primitive functions are completely independent of the history component and leave it unchanged. For example, $\text{tl}: \langle \langle 1, 2, 3 \rangle, \text{history} \rangle = \langle \langle 2, 3 \rangle, \text{history} \rangle$ 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; eg `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` $\langle \text{dev}, \text{history} \rangle$ into $\langle \text{next_inp}, \text{history}' \rangle$, 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 appendix A). 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, eg 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 `↑` (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, SASL, etc.).

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. Eg, 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.

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. Eg, 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; eg, the rule $\text{id}:x \rightarrow x$ 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. Eg, the definition $\text{Def } f \equiv E$ results in the rule $f:x \rightarrow E:x$ 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 $\langle \text{id}:2, \text{id}:3 \rangle$ to $\langle 2, 3 \rangle$ using two applications of the rewrite rule $\text{id}:x \rightarrow x$.

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 $\text{out}: \langle \text{'screen'}, 12 \rangle$ rewrites to 12 (the expression component of the result of a successful output operation is just the value printed), and the notation $\langle \text{'outflag'}, \langle \text{'screen'}, 12 \rangle \rangle$ 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 $\text{Def } f \equiv E$ results in the rule $f:x \rightarrow E:x$ being added to the set of rewrite rules. This will treat recursive definitions such as $\text{Def } f \equiv E(f)$ 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, ie 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 $\text{Def } H \equiv E(H)$ were to produce the rewrite rule $H:f \rightarrow E(H):f$, 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 $\text{Def}_2 H \equiv E(H)$. This results in the rewrite rule $H:f:x \rightarrow E(H):f:x$. 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 ($\uparrow l: \langle id, 3 \rangle$ denotes $\langle 3 \rangle$). 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 $\uparrow l$ 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 (eg, $tl: \langle \langle id, 3 \rangle, h \rangle = \langle \langle 3 \rangle, h \rangle$).

$\mathcal{H}$ 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 $\mathcal{F}$ (eg, tl and add) from $\mathcal{D} \times \mathcal{H} \rightarrow \mathcal{D} \times \mathcal{H}$, and is closed under the operations of sequence formation (eg $\langle tl, 3 \rangle$) and tagging with identifiers (eg $\langle 'color', 'red' \rangle$). 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; ie, 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 $\mathcal{D}$. Eg, $\mu(\text{out}:\langle\text{'screen'}, \text{add}:\langle 1, 2 \rangle\rangle, h) = (\langle\text{'screen'}, 3\rangle, h')$ where h' is h with $\langle\text{'outflag'}, \langle\text{'screen'}, 3\rangle\rangle$ 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 $[f_1, \dots, f_n]: x \rightarrow \langle f_1:x, \dots, f_n:x \rangle$.

2. A simple function definition.

- $\text{Def second} \equiv s1 \circ \text{tl}$
- $\text{second}: \langle 2, 3, 4 \rangle \rightarrow (s1 \circ \text{tl}):\langle 2, 3, 4 \rangle \rightarrow (\circ:\langle s1, \text{tl} \rangle):\langle 2, 3, 4 \rangle \rightarrow s1:(\text{tl}:\langle 2, 3, 4 \rangle) \rightarrow s1:\langle 3, 4 \rangle \rightarrow 3$.
- In general s_i is the primitive function that selects the i th 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 `t1`. In general, `f op g` means $op:\langle f, g \rangle$.

3. A recursive factorial program.

- `Def fact  $\equiv$  id =  $\sim 0 \rightarrow \sim 1$ ; id*fact $\circ$ sub1`
- `fact:5  $\rightarrow$  120`
- The `fact` function is built by applying the combining forms *composition* ($\circ$), *constant* ($\sim$) 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 $\sim x:y = x$ (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)  $\rightarrow$  *:<1,2,3,4,5>  $\rightarrow$  120`
- `ints_to` is a primitive function such that `ints_to:n = <1,...,n>` 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.

- `+  $\circ$   $\alpha$ : $\sim 1$`
- `(+  $\circ$   $\alpha$ : $\sim 1$ ) : <a,b,c>  $\rightarrow$  +:( $\alpha$ : $\sim 1$ :<a,b,c>)  $\rightarrow$  +:< $\sim 1$ :a,  $\sim 1$ :b,  $\sim 1$ :c>  $\rightarrow$  +:<1,1,1>  $\rightarrow$  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 $(\alpha:f) : \langle x_1, \dots, x_n \rangle = \langle f:x_1, \dots, f:x_n \rangle$. Note that `:` associates to the left (ie `x:y:z = (x:y):z`) so that $\alpha:\sim 1:a$ means $(\alpha:\sim 1):a$.

6. A program to compute the average of a sequence of numbers.

- `Def ave  $\equiv$  (+  $\div$  length where Def length  $\equiv$  +  $\circ$   $\alpha$ : $\sim 1$  end )`
- `ave : <3,4,8>  $\rightarrow$  (+  $\div$  length) : <3,4,8>  $\rightarrow$  (+:<3,4,8>)  $\div$  (length : <3,4,8>)  $\rightarrow$  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  $\equiv$  +  $\circ$   $\alpha$ : $\sim 1$`).

Such definitions are *local*; ie, their scope of definition is limited to the expression to which they are attached.

7. An order $n \log n$ sorting program.

- $\text{Def sort} \equiv \text{tree}:\langle \text{merge}, \langle \rangle \rangle \circ \alpha:[\text{id}]$
- $\text{sort} : \langle 8, 4, 2, 3 \rangle \rightarrow (\text{tree}:\langle \text{merge}, \langle \rangle \rangle \circ \alpha:[\text{id}]) : \langle 8, 4, 2, 3 \rangle$
 - $\rightarrow \text{tree}:\langle \text{merge}, \langle \rangle \rangle : (\alpha:[\text{id}] : \langle 8, 4, 2, 3 \rangle)$
 - $\rightarrow \text{tree}:\langle \text{merge}, \langle \rangle \rangle : \langle \langle 8 \rangle, \langle 4 \rangle, \langle 2 \rangle, \langle 3 \rangle \rangle$
 - $\rightarrow \text{merge}:\langle \text{merge}:\langle \langle 8 \rangle, \langle 4 \rangle \rangle, \text{merge}:\langle \langle 2 \rangle, \langle 3 \rangle \rangle \rangle$
 - $\rightarrow \text{merge}:\langle \langle 4, 8 \rangle, \langle 2, 3 \rangle \rangle \rightarrow \langle 2, 3, 4, 8 \rangle$
- tree is a primitive combining form that takes a function and an object and produces a function such that $\text{tree}:\langle f, z \rangle : \langle x_1, \dots, x_n \rangle$ rewrites to z if $n=0$, x_1 if $n=1$, and $f:\langle \text{tree}:\langle f, z \rangle : \langle x_1, \dots, x_m \rangle, \text{tree}:\langle f, z \rangle : \langle x_{m+1}, \dots, x_n \rangle \rangle$ if $n > 1$ (where $m = \text{ceil}(n \div 2)$). merge is a primitive function that merges two sorted sequences of numbers into a single sorted sequence; eg, $\text{merge}:\langle \langle 3, 6 \rangle, \langle 1, 3, 5 \rangle \rangle = \langle 1, 3, 3, 5, 6 \rangle$ (Notice that $\alpha:[\text{id}]$ maps an n -element sequence into a sequence of n singleton sequences.)

8. A program for doing a "protected" divide.

- $\text{Def safediv}_1 \equiv \text{ispair} \rightarrow ((\text{isnum} \circ s1 \wedge (\text{isnum} \wedge \text{not} \circ \text{iszero}) \circ s2) \rightarrow s1 \div s2; \sim \text{'diverror});$
 $\sim \text{'diverror}$
- $\text{safediv}_1 : \langle 8, 4 \rangle \rightarrow 2$
- $\text{safediv}_1 : \langle 8, 4, 2 \rangle \rightarrow \text{'diverror}$ (since $\langle 8, 4, 2 \rangle$ is not a pair)
- $\text{safediv}_1 : \langle 8, 0 \rangle \rightarrow \text{'diverror}$ (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.

- $\text{Def safediv}_2 \equiv \llbracket n., m. \rrbracket \rightarrow ((\text{isnum} \circ n \wedge (\text{isnum} \wedge \text{not} \circ \text{iszero}) \circ m) \rightarrow n \div m; \sim \text{'diverror});$
 $\sim \text{'diverror}$
- $\text{safediv}_2 : \langle 8, 4 \rangle \rightarrow 2$
- $\text{safediv}_2 : \langle 8, 4, 2 \rangle \rightarrow \text{'diverror}$ (since $\langle 8, 4, 2 \rangle$ fails to match the pattern $\llbracket n., m. \rrbracket$)
- $\text{safediv}_2 : \langle 8, 0 \rangle \rightarrow \text{'diverror}$ (since $\text{not} \circ \text{iszero}$ is false for m)
- The pattern mechanism is defined so that safediv_2 is just a notational shorthand for safediv_1 . Ie, the pattern $\llbracket n., m. \rrbracket$ 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 safediv3 ≡ [[n.isnum, m.isnum ∧ not ◦ iszero]] → n ÷ m; ~`diverror`
- `safediv3 : <8, 4> → 2`
- `safediv3 : <8, 0> → `diverror` (the pattern fails to match, since `not ◦ iszero` is false for 0)
- Again, the pattern mechanism is defined so that `safediv3` is exactly equivalent to `safediv1`. 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'>

→ "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 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):"enter a"` (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*; ie, when applied to *objects* it performs the catenation function, but when applied to *functions* (eg `++:<f,g>`), it rewrites to `++ ◦ [f,g]`, thus avoiding the need to do explicit lifting (see Section 2). Text enclosed in `{ }` pairs is a comment and is lexically equivalent to a blank space.

12. A higher-order function, the B combinator (Curried composition).

- `Def B:x.:y. ≡ x ◦ y`
- `B:s1:t1:<3,4,5> → (s1 ◦ t1):<3,4,5> → 4`
- `B` is defined to be the higher order function that, when applied to x produces a function that, when applied to y produces the function $x ◦ y$; eg, `B:s1:t1` is `(s1 ◦ t1)`. The x . and y . in the left hand side of this definition are examples of constructs called *templates*, which provide a generalization of λ -abstraction and are useful for defining higher order functions. Templates are fully described in Section 3. This definition results in the new rewrite rule `B:x:y:z → (x ◦ y):z`. Since the (implicit) subscript on `Def` is 1, the extra argument (`<3,4,5>` in this case) is required before `B:s1:t1` can be replaced by `(s1 ◦ t1)`.

2. 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*, ie 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 a *simplification algorithm* that maps 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 3. The rest of this section is organized into the following subsections:

1. The Syntax of Core Expressions

- ## The Syntax of Core Expressions

1. An atom (which may be an identifier, a string, or a number)
2. A primitive function name
3. ? (which is used to indicate an error)
4. A sequence. If $x_1, \dots, x_m$ are core expressions, then the *sequence* $\langle x_1, \dots, x_m \rangle$ is a core expression. The empty sequence $\langle \rangle$ is a core expression.
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 an identifier and x_2 is a core expression, then the *tag pair* $\langle x_1, x_2 \rangle$ is a core expression.

To facilitate the description of identifiers and function names it is useful to have two auxiliary notions: *basic names* and *symbols*.

- All upper and lower case Roman letters
- All upper and lower case Greek letters
- The special characters ! \$ % & ' () * + , - . : ;

(a blank space)

12

(Note: the symbols & and @ have been reserved for future use in an ASCII based version of the language.)

Atoms. The set of *atoms* is the union of three disjoint sets: the identifiers, the numbers, and the strings.

- An *identifier* is just a ' followed by a basic name. For example, 'abc is an identifier.
- 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 *string* is a juxtaposition of any number of characters or symbols delimited by "s. The symbol ' is used as an escape character so that " may appear in a string. The convention for ' is that the ' itself does not appear as part of the string, but the character after it does even if it is a ' or a ". For example,

"This string has one '" and one '" in it!"

is a string. A string of length one is also called a *char*.

(Primitive) function names. A *function name* is either a basic name (note that uppercase and lowercase letters are distinct in a function name) or one of the symbols

° + * ÷ - = ≤ < ≥ > ≠ / \ ⇨ ⇩ ^ ∨ ++

There are about 70 *primitive functions* in FL. The complete list is given in Appendix A.

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

Comments. A *comment* is any sequence of characters and/or symbols enclosed in braces { }. Comments may be nested; they cannot contain an unmatched brace. A comment may appear anywhere in an expression that a blank space may appear; lexically it is equivalent to a blank space.

The Semantics of Core Expressions

The Denotational Semantics. The *meaning* of a core expression is provided by the denotational semantics of FL. Section 1 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 $\mathcal{D}$ (respectively $\mathcal{H}$, the set of histories) that results from evaluating the expression expr with the history h .

The semantic domain $\mathcal{D}$ for FL is unusual in that it actually *contains* the set $\mathcal{A}$ of atoms, not some isomorphic image of them. (But $\mathcal{A}$ contains only the “canonical forms” of numbers.) Thus $\mathcal{D}$ contains the set of atoms, $\mathcal{A}$, an isomorphic copy of the set $\mathcal{F}$ of “good” functions that map $\mathcal{D} \times \mathcal{H}$ into itself, and $\perp$, and it is closed under sequence formation and tagging with identifiers. $\mathcal{H}$ is the set of all “semantic histories”, ie, those that contain the semantic counterparts of values (eg, the semantic counterpart of the function value id is the mathematical function id such that $\text{id}(d, h) = (d, h)$; 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 $\mathcal{D} \times \mathcal{H}$ into itself. Eg, the meaning of tl is the function tl , where $\text{tl}(\langle x_1, \dots, x_n \rangle, h) = (\langle x_2, \dots, x_n \rangle, h)$

The meaning of $?$ is $\perp$.

The meaning of the sequence $\langle x_1, \dots, x_n \rangle$ is the sequence of the meanings of its components except in the case that one of the components has meaning $\perp$. More formally, $\mu(\langle x_1, \dots, x_n \rangle, h_1) = (\langle y_1, \dots, y_n \rangle, h_{n+1})$ where $\mu(x_i, h_i) = (y_i, h_{i+1})$, provided no y_i is $\perp$. If some $y_i = \perp$, then $\mu(\langle x_1, \dots, x_n \rangle, h_1) = (\perp, h_{k+1})$ 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 $\mathcal{D} \times \mathcal{H}$ 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 tag pair $\langle x_0, x_1 \rangle$ is the meaning of x_1 tagged by x_0 . (Recall that x_0 must be an identifier.) More formally, $\mu(\langle x_0, x_1 \rangle, h_1) = (\langle x_0, y_1 \rangle, h_2)$ where $\mu(x_1, h_1) = (y_1, h_2)$. Tag pairs may be created only by the rewriting system itself and may *not* be written by the programmer (since tag 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 tag pair with a value as its second component is a value. The only other kind of values are irreducible function expressions (ie, expressions to which no rewrite rules apply and whose meanings are functions) such as $\circ: \langle \uparrow, \uparrow \rangle$. 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 (ie, 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 $\text{id}:x \rightarrow x$ indicating that the function id when applied to the value x rewrites to x . Another example of a rewrite rule is $\text{cons}: \langle f_1, \dots, f_n \rangle : x \rightarrow \langle f_1 : x, \dots, f_n : x \rangle$ 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 $\langle x_1, \dots, x_n \rangle$, it is necessary that each x_i be a value for $i = 1, \dots, k - 1$. 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 $\langle \text{out}: \langle \text{'screen'}, \text{'histr'} \rangle, \text{out}: \langle \text{'screen'}, \text{'bye'} \rangle \rangle$ is first rewritten to $\langle \langle \text{'screen'}, \text{'hi'} \rangle, \text{out}: \langle \text{'screen'}, \text{'bye'} \rangle \rangle$ causing "hi" to appear on the screen and then is rewritten to $\langle \langle \text{'screen'}, \text{'hi'} \rangle, \langle \text{'screen'}, \text{'bye'} \rangle \rangle$ causing "bye" to appear on the screen. This final expression is a value.

The complete list of the rewrite rules is given in Appendix B, and an informal description of all the primitive functions is given in Appendix A. 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 $(\text{expr}, \text{hist})$ into $(\text{expr}_1, \text{hist}_1)$, 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 . Eg, $(\langle \text{id}:3, 4 \rangle, \text{hist})$ is transformed by the rewrite rule for id into $(\langle 3, 4 \rangle, \text{hist})$. 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 Appendix A 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 Appendix B.)

Special notation for combining forms.

1. **Prime:** f' abbreviates $\text{lift}:f$ 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:** $[(^{(n)}f_1, \dots, f_m)]$ abbreviates $\text{cons}^{(n)}:\langle f_1, \dots, f_m \rangle$ where the superscript n indicates the number of primes. Thus, $[f_1, \dots, f_m]:x$ rewrites to $\langle f_1:x, \dots, f_m:x \rangle$.
3. **Element-by-element Construction (E-Construction):** $[[^{(n)}f_1, \dots, f_m]]$ abbreviates $\text{econs}^{(n)}:\langle f_1, \dots, f_m \rangle$ where the superscript n indicates the number of primes. E-construction

is a predicate combining form such that $\llbracket f_1, \dots, f_m \rrbracket$ is true just for sequences of length m whose elements satisfy, one-for-one, the predicates f_i . Thus, $\llbracket f_1, \dots, f_m \rrbracket : \langle x_1, \dots, x_k \rangle$ rewrites to $\text{and} : \langle f_1 : x_1, \dots, f_m : x_m \rangle$, provided $m = k$; otherwise if $m \neq k$ it rewrites to 'F. $\text{and} : \langle f_1 : x_1, \dots, f_m : x_m \rangle$ rewrites to 'T provided no $f_i : x$ is 'F; otherwise it rewrites to 'F.

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,

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

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 'F, and "false" only if it is 'F. However, all primitive predicates yield 'T or 'F.

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

$$f : \langle g_1, \dots, g_n \rangle \rightarrow f \circ [g_1, \dots, g_n] \text{ if all the } g_i\text{'s are functions}$$

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 $\text{op} : \langle f, g \rangle$ can be written simply $f \text{ op } g$. 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 $\circ : \langle f_1, \dots, f_n \rangle : x \rightarrow f_1 : (f_2 : \dots (f_n : x) \dots)$; 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 (ie, 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 / e_2 or $e_1 \setminus e_2$

$e_1 \circ e_2$

$e_1 * e_2$ 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 $e_1 \Leftarrow e_2$

$e_1 \wedge e_2$ 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.)

$e_1 | e_2$

e_1 where dl end

let dl in e_1

$fn \equiv e_1$

Some examples of parenthesized expressions:

Before Parenthesizing	After Parenthesizing
$a \rightarrow b; c \rightarrow d; e$	$(a \rightarrow b; (c \rightarrow d; e))$
$a \rightarrow b; c \rightarrow d$	$(a \rightarrow b; (c \rightarrow d))$
$a \rightarrow b \rightarrow c; d$	$(a \rightarrow (b \rightarrow c; d))$
$a \rightarrow b \rightarrow c$	$(a \rightarrow (b \rightarrow c))$
$f + g = \sim h' \circ k$	$((f + g) = ((\sim(h')) \circ k))$
$p \rightarrow f; g \circ h k$	$((p \rightarrow f; (g \circ h)) k)$

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

$\text{Def } g_1 \equiv e_1 \dots \text{Def } g_n \equiv e_n$

where $e_1, \dots, e_n$ are simple expressions and $g_1, \dots, g_n$ are distinct function names. $\text{Def } g \equiv e$ is a *definition* that defines g . (Other kinds of declarations, and additional ways of making definitions are introduced in Sections 3 and 4.) The definitions in the declaration list may be (mutually) recursive. For example, $\text{Def fact} \equiv (\text{id} = \sim 0 \rightarrow \sim 1; \text{id} * \text{fact} \circ \text{sub1})$ recursively defines the factorial function (where sub1 is the function that subtracts one).

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

e

$e \text{ where deflist end}$

$\text{let deflist in } e$

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; ie, the definition $\text{Def } f \equiv e$ in deflist produces the rule $f:x \rightarrow e:x$. The value of $e \text{ where deflist end}$ 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? Eg, if the simple program `out:<'printer, f> where Def f  $\equiv$  tl end` causes "f" to be printed, that might be meaningful to the writer of the program, who might know that f denotes the function tl . 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, ie, that uses no defined function names. Therefore, the precise semantics of any simple program p is given by first mapping p to an equivalent core expression (written $Simp(p)$) and then using the semantics of core expressions. The process for converting p into $Simp(p)$ is called the *simplification algorithm*. It is used to give the precise semantics not only of simple programs, but of the complete class of FL expressions as defined in Section 3. An informal description of it can be found at the end of that section. The complete algorithm is given in Appendix C.

This completes the development of simple programs. In fact, most of the examples in Section 1 were written using simple programs. The rest of the manual provides additional notation and language constructs.

3. 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 given by the *simplification algorithm* that transforms all these features into 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 only appear in the first position of a conditional. Thus conditional is extended to include the form $\text{pattern} \rightarrow e_1; e_2$.

An example of a pattern is $\llbracket x.\text{isident}, \llbracket y.\text{isint}, z.\text{isint} \rrbracket \rrbracket$; it tests its argument to make sure it is a pair whose first element is an identifier 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 makes the definitions $\text{Def } x \equiv s_1$, $\text{Def } y \equiv s_1 \circ s_2$ and $\text{Def } z \equiv s_2 \circ s_2$ and denotes the predicate $\llbracket \text{isident}, \llbracket \text{isint}, \text{isint} \rrbracket \rrbracket$.

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. For example, patterns can be put together with composition and conjunction as well as the "usual" methods of naming positions in a sequence; thus patterns can define functions other than those that simply select elements from a structure.

Every pattern describes a function (called its *function part*) and a list of definitions (called its *definition part*), which define the function names introduced by the pattern (those followed by a ". "). The "function" part of a complete pattern always serves as a predicate; however, in building a pattern $P_1 \circ P_2$ by composition, the right hand pattern P_2 may meaningfully denote an arbitrary function.

The function 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 function 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	Function Part	Definition Part
$fname.$	$\top$	$\{fname \equiv id\}$
$fname.e$	e	$\{fname \equiv id\}$
e	e	$\{\}$
$\llbracket P_1, \dots, P_n \rrbracket$	$\llbracket p_1, \dots, p_n \rrbracket$	$\bigcup_{i=1}^n \{d \equiv D \circ si \mid (d \equiv D) \in Defs(P_i)\}$
$P_1 \mapsto P_2$	$p_1 \mapsto p_2$	$\{d \equiv D \circ s1 \mid (d \equiv D) \in Defs(P_1)\}$ $\cup \{d \equiv D \circ t1 \mid (d \equiv D) \in Defs(P_2)\}$
$P_1 \leftarrow P_2$	$p_1 \leftarrow p_2$	$\{d \equiv D \circ tr \mid (d \equiv D) \in Defs(P_1)\}$ $\cup \{d \equiv D \circ r1 \mid (d \equiv D) \in Defs(P_2)\}$
$P_1 \circ P_2$	$p_1 \circ p_2$	$\{d \equiv D \circ p2 \mid (d \equiv D) \in Defs(P_1)\} \cup Defs(P_2)$
$P_1 \wedge P_2$	$p_1 \wedge p_2$	$Defs(P_1) \cup Defs(P_2)$

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 precise scope rules for these definitions are given by the way in which the simplification algorithm reduces conditionals with patterns to simple conditionals.

Given the conditional $P \rightarrow e_1(d_1, \dots, d_n); e_2(d_1, \dots, d_n)$, where the definition part of the pattern P is $d_1 \equiv D_1 \dots d_n \equiv D_n$, and e_1 and e_2 are expressions involving some or all of the function names defined by P , simplification transforms the pattern P into a simple expression p (corresponding to the function part of P as described in the above table); it also produces the above definition part (as described in the table). Simplification transforms the entire conditional into a core expression equivalent to the following:

$$p \rightarrow E_1(Simp(D_1), \dots, Simp(D_n)); E_2(Simp(D_1), \dots, Simp(D_n))$$

where $E_i(d_1, \dots, d_n)$ is the simple expression $Simp(e_i(d_1, \dots, d_n))$ for $i = 1, 2$.

Notice that the effect of this simplification is that definitions made by a pattern are not (mutually) recursive and that all *uses* of a function name within a pattern refer to its “old” definition, not the definition made by the pattern. The following table gives some examples of expressions written using patterns and their equivalent simple expressions.

Expression using patterns	Equivalent simple expression
$\llbracket x, \text{isnum}, y.\text{isint} \rrbracket \rightarrow f \circ [x, g \circ [\text{id}, y]]; x$	$\llbracket T, \text{isnum}, \text{isint} \rrbracket \rightarrow f \circ [s1, g \circ [\text{id}, s3]]; s1$
$\llbracket T, x.\text{isnum} \rrbracket \circ s1 \rightarrow f \circ x; g \circ x$	$\llbracket T, \text{isnum} \rrbracket \circ s1 \rightarrow f \circ s2 \circ s1; g \circ s2 \circ s1$
$(x.\text{isnum} \mapsto y.) \rightarrow (f \circ x) \text{ al}' (g \circ y); h$	$(\text{isnum} \mapsto T) \rightarrow (f \circ s1) \text{ al}' (g \circ \text{tl}); h$

Observe that, for most patterns, the functions the pattern defines can be applied safely in the “true arm” without causing an error; but these functions may cause an error if not used with care in the “false arm”. For example, if the first expression above is applied to the empty sequence, an error will occur as a result of using x in the false arm of the conditional. For this reason, pattern-defined functions will usually appear only in the first arm; however, there are useful occasions when such functions are known to be defined in the second arm.

This use of patterns must not be mistaken for a kind of λ -abstraction. Notice that $\llbracket x, y. \rrbracket \rightarrow x \circ y; \text{err}$ abbreviates $\llbracket T, T \rrbracket \rightarrow s1 \circ s2; \text{err}$ which, if applied to $\langle f, \langle g, h \rangle \rangle$, rewrites to g . Contrast this with $\lambda x, y. x \circ y$ which, if applied to $\langle f, \langle g, h \rangle \rangle$, would produce $f \circ \langle g, h \rangle$ (which is equivalent to err , since $\langle g, h \rangle$ is not a function).

Conditional abstraction

For those occasions when a form of λ -abstraction is more convenient (ie, when one wants to be able to substitute a value for a variable in an expression) FL provides a generalization of λ -abstraction called *conditional abstraction*. (This is a rather specialized construct, primarily useful for defining higher-order functions, and may safely be skipped at first reading.) Conditional abstraction is written $P \Rightarrow e_1; e_2$ or $P \Rightarrow e_1$, where P is a pattern and e_1, e_2 are expressions.

The result of applying a conditional abstraction to a value x is given by:

$$(P \Rightarrow e_1(d_1, \dots, d_n); e_2(d_1, \dots, d_n)) : x = \begin{cases} E_1(D_1 : x, \dots, D_n : x) & \text{if } p : x \neq 'F, ? \\ E_2(D_1, \dots, D_n) : x & \text{if } p : x = 'F \\ ? & \text{if } p : x = ? \end{cases}$$

where p is the function part of P , and $d_1 \equiv D_1 \dots d_n \equiv D_n$ is the definition part of P , $E_i(d_1, \dots, d_n) = \text{Simp}(e_i(d_1, \dots, d_n))$ for $i = 1, 2$, and $\text{Simp}(e)$ denotes the core expression that

results from simplifying e. (Notice that the “false” arm acts the same in a conditional abstraction as it does in an ordinary conditional). The effect of conditional abstraction is illustrated by:

$$\begin{aligned}
& ([f, g] \Rightarrow [f \circ f, g:f, g:s5]) : \langle t1, \alpha \rangle \\
& = [(s1:\langle t1, \alpha \rangle) \circ (s1:\langle t1, \alpha \rangle), (s2:\langle t1, \alpha \rangle):(s1:\langle t1, \alpha \rangle), (s2:\langle t1, \alpha \rangle):s5] \\
& = [t1 \circ t1, \alpha:t1, \alpha:s5] \\
& \text{since } P \text{ defines } f \equiv s1, g \equiv s2 \text{ and the function part of } P \text{ is true for } \langle t1, \alpha \rangle
\end{aligned}$$

Enhanced Definitions

FL includes some special syntactic notation for enhancing the readability of defined functions. There are three kinds of enhanced definitions. The first kind is useful for defining higher level functions; it causes application of the defined function to be delayed until the appropriate number of arguments are available. The other two kinds provide “arguments” for a defined function on the *left* side of a definition; these symbolic “arguments” limit the kinds and shapes of actual arguments for which the function is defined and name parts of a valid argument, just as patterns do in conditionals.

All three kinds of enhanced definitions correspond in a simple way to an ordinary definition. It is helpful to present this correspondence for all three before explaining the details of the new forms called *templates*, which are used in one of the three. The following table shows how to reduce any enhanced definition to an ordinary one by using the correspondences it gives. To understand the table it is only necessary to know that every template T corresponds to some pattern P_T and that the primitive function *args* has the rewrite rule $\text{args}:n:f:x_1:\dots:x_n \rightarrow f:x_1:\dots:x_n$. In the table, P is a pattern, T is a template (having P_T as the corresponding pattern), F is an “enhanced” left hand side (ie, a function name or, recursively, $F \circ P$ or $F:T$), E is an expression, and i is a non-negative integer.

Enhanced Definition	Corresponding Definition
$\text{Def}_i f \equiv E$	$\text{Def } f \equiv \text{args}:i:E^{\text{abs}}$
$\text{Def}_i F \circ P \equiv E$	$\text{Def}_i F \equiv (P \rightarrow E; \text{err})$
$\text{Def}_i F:T \equiv E$	$\text{Def}_{i+1} F \equiv (P_T \Rightarrow E; \text{err})$

The usual precedence conventions apply to the left-hand-side of an enhanced definition (subject, of course, to the constraint that the resulting parenthesized expression must be a syntactically valid enhanced definition). The first line of the table introduces the new

keywords Def_i , for any non-negative integer, which are then used in the other two lines. In the first line, E^{obs} stands for the simple expression that corresponds to E after all (mutual) recursion has been eliminated. There must be no Def_0 's left after the elimination of enhanced definitions by the use of the last two lines, since this would amount to assigning a function name to an object rather than a function. (The closest one can come to the definition $\text{Def}_0 f \equiv x$ is to write $\text{Def } f \equiv \sim x$; the second definition is the function level analogue of the first.)

The table above gives the complete facts about the first two kinds of enhanced definitions; however an example for each may be helpful. The first form in the above table allows the definition of a curried form of add, ie, $\text{Def}_2 \text{ad} \equiv x.\text{isreal} \Rightarrow \sim x + \text{id}$ such that $\text{ad}:3$ is a value (ie no rewrite rule applies) that denotes the function that adds 3 to its argument. Thus $\text{ad}:3:4$ rewrites as follows: $(x.\text{isreal} \Rightarrow \sim x + \text{id}):3:4 \rightarrow (\sim 3 + \text{id}):4 \rightarrow +: <3, 4> \rightarrow 7$. In general, $\text{Def}_n g \equiv e$ does not allow rewriting of $g:x_1:\dots:x_k$ to $e:x_1:\dots:x_k$ unless $k \geq n$.

The second form in the table allows the definition $\text{Def ave} \circ [[x.\text{isreal}, y.\text{isreal}]] \equiv (x + y) \div \sim 2$ which is, by the table, the same as $\text{Def ave} \equiv [[x.\text{isreal}, y.\text{isreal}]] \rightarrow (x + y) \div \sim 2; \text{err}$; thus *ave* is defined only for pairs of reals and gives their average. This form is the appropriate one for the definition of most object-valued functions.

Templates can be used as symbolic "arguments" on the left hand side of enhanced definitions. For example, $<x, y>$ is a template that represents an argument that is a pair, and it names the two elements x and y ; thus the enhanced definition $\text{Def hd_tl}: <x, y> \equiv \text{al} \circ [x \circ s1, y \circ t1]$ defines hd_tl to be a function that is defined only on pairs of functions and that, when applied to such a pair (for example, $\text{hd_tl}: <f, g>$) produces a function (defined on non-null sequences) that will apply f to the first element of the sequence and g to its tail. The enhanced definition abbreviates the ordinary definition $\text{Def hd_tl} \equiv \llbracket x, y \rrbracket \Rightarrow \text{al} \circ [x \circ s1, y \circ t1]$, as given by the table above and the one below for templates.

For every template, there is a corresponding pattern. All the template forms are given in the following table along with their corresponding patterns. In the table, T_i stands for a template and P_{T_i} stands for the corresponding pattern.

Template	Corresponding Pattern
<code>fname . expr</code>	<code>fname . expr</code>
<code>fname .</code>	<code>fname .</code>
<code>< T₁, ..., T_n ></code>	<code>[[P_{T₁}, ..., P_{T_n}]]</code>
<code>al:<T₁, T₂></code>	<code>P_{T₁} ↦ P_{T₂}</code>
<code>ar:<T₁, T₂></code>	<code>P_{T₁} ⇐ P_{T₂}</code>

Since the simplification algorithm assumes that the expression being simplified has been fully parenthesized, it is necessary to extend the parenthesizing algorithm of the previous section to account for patterns and templates. This is done by treating “.” as an “operator” that binds tighter than any other operator on its left and looser than any operator but $\equiv$ on its right (subject, of course, to syntactic constraints on patterns and templates). Thus, $p \wedge f.a \circ c$ means $p \wedge (f.(a \circ c))$ and $f.[[a., b.]] \circ c$ means $f.([[a., b.]] \circ c)$.

Extending Functions

It is sometimes desirable to be able to extend functions; for example, the programmer might wish to extend the function $+$ to work on sequences of equal-length vectors (where a vector is a sequence of numbers). (This capability is particularly useful for extending functions so that they work on new data types. See Section 4.) An extension definition has the form

$$\text{Ext } f \equiv P_1 \rightarrow e_1; \dots; P_n \rightarrow e_n$$

where $P_1, \dots, P_n$ are patterns and $e_1, \dots, e_n$ are expressions. This extension declaration extends the function f so that the following equation holds: $\text{New} = (P_1 \rightarrow e_1; \dots; P_n \rightarrow e_n; \text{Old})$ where **New** is the new version of f and **Old** is the old version of f . Every explicit mention of the function name f in the declaration list in which the extension takes place refers to the fully extended function (ie, to **New**). The only time the old function definition is used is the case where all the predicates fail and execution “falls through” to **Old** (which is mentioned only implicitly). If there is more than one extension definition for f , the first one listed gets the first opportunity to compute the value, and if all of its predicates fail, execution falls through to the next extension definition listed for f (or to the old definition if there are no more extension definitions). Extension definitions provide the only case in which the order of definitions matters.

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(f1, ..., fn)` (indicating that only the function names $f_1, \dots, f_n$ are visible in the containing declaration list)
3. `hide(f1, ..., fn)` (indicating that all but the function names $f_1, \dots, f_n$ 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.

An extension definition may be exported, but when it is exported the simplification algorithm changes it into an ordinary `Def`; therefore the function name must not clash with any other definitions in the containing declaration list (ie, no declaration list may define the same function name twice).

For example, the following table shows an expression before simplification and an equivalent expression with a simple declaration list. In the following $E(x_1, \dots, x_n)$ represents an expression with the function names $x_1, \dots, x_n$ 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  $\equiv$  err`.

Before	After
<pre> E₀(f, g, h) where export(f, g) Def f ≡ E₁(f, g, h) hide(f) Def f ≡ E₂(f, g, h) Def g ≡ E₃(f, g, h) end Def h ≡ E₄(f, g, h) end end </pre>	<pre> E₀(f₁, g₂, h) where Def f₁ ≡ E₁(f₁, g₂, h₁) Def f₂ ≡ E₂(f₂, g₂, h₁) Def g₂ ≡ E₃(f₂, g₂, h₁) Def h₁ ≡ E₄(f₁, g₂, h₁) end </pre>

It is envisioned that the programming environment (eg, 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 3 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 (which may include representation, assertion, and signature declarations as described in the next section), then the following are the FL *expressions*:

```

se
<e1, ..., en>
e1:e2
< identifier, e >

```

e'
 $[(^{(m)}e_1, \dots, e_n]$
 $\llbracket (^{(m)}e_1, \dots, e_n) \rrbracket$
 $\sim_e$
 $\text{pattern} \rightarrow e_1 ; e_2$
 $\text{pattern} \rightarrow e_1$
 $e_1 \rightarrow (^{(n)}e_1 ; e_2$
 $e_1 \ e_2 \ e_3 \quad (\text{infix notation})$
 $e \text{ where declist end}$
 $\text{let declist in } e$
 $\text{pattern} \Rightarrow e_1 ; e_2$
 $\text{pattern} \Rightarrow e_1$

The semantics of FL expressions. As discussed in Section 2, the semantics of an expression is most conveniently 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 2, 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.

The simplification algorithm. The purpose of simplification is to remove all syntactic abbreviations, resolve all scope issues, eliminate all definitions, and thereby transform any FL program into an exactly equivalent core expression. This simplification is accomplished by working outward from innermost scopes. Thus, when simplifying an expression of the form $e = E \text{ where } \dots \text{Def } f_i \equiv E_i \dots \text{end}$, neither E nor any of the E_i 's contain any definitions (since e is an innermost expression). The following is an informal sketch of the algorithm.

First the algorithm transforms every definition $\text{Def } f_i \equiv E_i$ into $\text{Def } f_i \equiv E_i^{\text{abs}}$ where E_i^{abs} is an expression involving only primitive functions and functions defined *outside* of e (e may be a subexpression of a larger one that contains other definitions). Thus E_i^{abs} denotes the same function that the original definition defines for f_i and it contains no definitions and no locally defined function names. (Note: this is similar to the way that $Y:E$ is used to replace $f = E(f)$ in the λ -calculus.) The algorithm then replaces every occurrence of f_i in E by E_i^{abs} and discards

all the definitions (it can safely do so because it has eliminated all the function names they define). This process is repeated (always selecting an innermost scope) until all definitions have been eliminated. The complete description of the algorithm is given in Appendix C.

4. Data Types in FL

Introduction

The type system of FL has three main purposes: (1) to make it easy for the programmer to write self-documenting programs that indicate the type of arguments a function expects and that contain other, more general assertions about the properties of functions; (2) to allow the programmer to define new types of objects and functions on them; and (3) to make it possible for a compiler to analyze an FL program and produce an efficient object program that wastes little time checking that each function receives an appropriate argument. In order to provide a powerful language with as few restrictions as possible, the FL type system is not designed to guarantee that *all* run-time type checking can be eliminated.

The type system divides the world of values into subsets called *types*. Types are *partially ordered* by set inclusion; thus *integer* is a subtype of *number*, which is a subtype of *atom*. Types are not themselves FL values; instead they are represented by predicates. For example, the predicate *isnum* is true for every number and false for every non-number value. For every primitive (resp., defined) type like *number* there is a corresponding primitive (resp., defined) predicate like *isnum* that is true for every member of the type and false for every non-member; there is also a corresponding *type name*, an identifier like *'num*. A type name has no semantic significance in FL (but it serves as the “stem” of the names of functions on the type and is used to “tag” elements of a defined type in a way that is inaccessible to the programmer).

Primitive Types.

The primitive type predicates used in this chapter are the following (for a complete list, the partial ordering of types, and their closure rules see Appendix A).

Type Name	Predicate	Primitive Type
'val	isval	the set of values
'obj	isobj	the set of objects
'func	isfunc	the set of functions
'atom	isatom	the set of atoms
'ident	isident	the set of identifiers
'num	isnum	the set of numbers
'posreal	isposreal	the set of positive real numbers
'real	isreal	the set of real numbers
'seq	isseq	the set of sequences
(No type names)	islen:n	the set of sequences of length = n
'bool	isbool	the set { 'F, 'T }

There are a number of convenient combining forms for building new predicates from these type predicates or any others. For example, if $p_1, \dots, p_n$ are predicates then the three combining forms $\text{seqof}:p_1$, $\llbracket p_1, \dots, p_n \rrbracket$, and $p_1 \mapsto p_2$ build three new predicates. The first, $\text{seqof}:p_1$, describes the set comprising all sequences whose elements satisfy p_1 ; the second, $\llbracket p_1, \dots, p_n \rrbracket$, all sequences of length n whose i th element satisfies p_i for $i=1, \dots, n$; the third, $p_1 \mapsto p_2$, all sequences whose first element satisfies p_1 and whose tail satisfies p_2 . It is easy to write the predicate for many kinds of sets; for example, for the set *sequences of pairs of numbers* the predicate is $\text{seqof}:\llbracket \text{isnum}, \text{isnum} \rrbracket$; for the set *positive integers* it is $\text{isposreal} \wedge \text{isint}$; for the set *sequences beginning with an identifier followed by numbers* it is $\text{isident} \mapsto \text{seqof}:\text{isnum}$.

Defined Types

The programmer may create a new type by defining a new kind of value and new functions on it. Below is an example of the introduction of a new type, *gram*, along with the functions *isgram*, *mkgram*, *umgram*, (defined implicitly by the first (Rep) declaration) and the function *adgm* (defined explicitly); the following paragraphs explain the new declarations used in this example.

```
Rep 'gram  $\iff$  isreal
Asn umgram  $\circ$  adgm  $\circ$   $\alpha$ :mkgram  $\equiv$  +  $\leftarrow$   $\leftarrow$  seqof:isreal
Sig adgm :: seqof:isgram  $\wedge$  not $\circ$ isnull  $\Rightarrow$  isgram
Def adgm  $\circ$  seqof:isgram  $\wedge$  not:isnull  $\equiv$  mkgram  $\circ$  +  $\circ$   $\alpha$ :umgram
```

Representation declarations. The above representation (Rep) declaration asserts that values belonging to the type named 'gram are to be represented by values that satisfy the predicate *isreal* (ie, by real numbers). Note that reals are not *themselves* grams. Actual grams are *tagged* reals of the form $\langle \text{'gram}, x \rangle$ where x is a real (recall that $\langle t, x \rangle$ denotes the value x tagged

with the identifier t). The declaration defines the predicate `isgram` that is true on grams and false on all non-gram values. It implicitly defines the function `mkgram` that converts a real into a gram, and it implicitly defines the function `umgram` that converts grams back into reals.

The general form of a representation declaration is:

`Rep 'newtype  $\leftrightarrow$  oldpred ,`

where `'newtype` is some identifier and `oldpred` is some predicate expression (that does not depend on or change the history). As a result of this declaration `'newtype` is the name of the new type and `isnewtype` is the name of the implicitly defined total predicate that is true for the values of the new type and is false elsewhere (the predicate name is formed by dropping " ' " from the type name and adding "is"). The scope rules for the definitions made by a representation declaration are the same as for other definitions. Therefore it is an error to make two `Rep` declarations for the same type name in a single scope. `oldpred` is a predicate that is true for the set of values that represent the new type (the *representation set*). The values of the new type are tagged values of the form $\langle 'newtype, y \rangle$ where y satisfies `oldpred`.

`mknewtype` is the name for the implicitly defined function that tags values of the representation set to form values of the new type, and `umnewtype` (for un-make newtype) is the name for the implicitly defined function that untags the new type values to form values of the representation set. The names of the two implicit functions are formed by prefixing the type name (without " ' ") with `mk` and `um`.

The declaration implicitly makes the following definitions:

`Def isnewtype  $\equiv$  istagpair  $\rightarrow \sim 'newtype = tag1 \rightarrow T; F; F$`

`Def mknewtype  $\equiv$  oldpred  $\rightarrow tag: 'newtype; err$`

`Def umnewtype  $\equiv$  istagpair  $\rightarrow \sim 'newtype = tag1 \rightarrow tag2; err; err$`

where the functions `tag`, `tag1`, `tag2`, and `istagpair` are available only to the rewrite system (just as with tag pairs $\langle x, y \rangle$, these functions are not available to the programmer) and have the rewrite rules:

`tag:x:y  $\rightarrow \langle x, y \rangle$`

`tag1:x  $\rightarrow t$` if x has the form $\langle t, y \rangle$

`tag1:x  $\rightarrow ?$` otherwise

`tag2:x  $\rightarrow y$` if x has the form $\langle t, y \rangle$

tag2: $x \rightarrow ?$	otherwise
istagpair: $x \rightarrow 'T$	if x has the form $\langle t, y \rangle$
istagpair: $x \rightarrow 'F$	otherwise

Example of a defined type. This example makes a new type of sequences of numbers (with type name 'nl for "number lists").

```
Rep 'nl  $\leftrightarrow$  seqof:isnum
Def fstnl  $\equiv$  sl  $\circ$  umnl
Def rstnl  $\equiv$  mknl  $\circ$  tl  $\circ$  umnl
```

The representation declaration asserts that values of type 'nl are to be represented by sequences of numbers; it defines mknl, which maps any sequence x of numbers into $\langle 'nl, x \rangle$, and umnl, which untags such a number list to yield the sequence x . The result of the definitions is, eg,

```
fstnl: $\langle 'nl, \langle 4, 5 \rangle \rangle = 4$ 
rstnl: $\langle 'nl, \langle 4, 5 \rangle \rangle = \langle 'nl, \langle 5 \rangle \rangle$ 
```

The system's ability to detect type errors. All that is needed to establish a new type is a Rep declaration together with the definitions of the appropriate functions for manipulating the new type. However, the language provides two other kinds of declarations (assertion and signature, described below) that enable the programmer to provide information that can be used by the type system in doing type checking. This information is in the form of equations; if these equations can be shown to be correct (ie, a consequence of the function definitions), then the programmer's understanding of the meaning of his program is confirmed to some degree; if it is not possible to prove them correct, then the programmer should be so informed. He/she can then examine whether the equations or the definitions are in error or whether the system was simply unable to prove the actual validity of the equations.

The system's ability to do type checking at "compile time" or to confirm the validity of signature or assertion equations is not part of the FL language specification; it is a property of the system implementation. The system will completely type check the argument of every function, either at "compile time" or at "run time".

Assertion declarations. Assertion declarations give certain equations that are supposed to hold for the defined functions. Thus, in the example, the assertion

```
Asn umgram  $\circ$  adgm  $\circ$   $\alpha$ :mkgram  $\equiv$  +  $\leftarrow \leftarrow$  seqof:isreal
```

asserts that the function on the left of " $\equiv$ " is the same as the function $+$ on the domain comprising sequences of reals, since a qualified equation, $E_1 \equiv E_2 \leftarrow\leftarrow P$, asserts that E_1 denotes the same function as E_2 on the domain of values for which P is true. An unqualified equation, $E_1 \equiv E_2$, asserts that E_1 denotes the same function as E_2 .

A *filter* is a function that is the identity on some domain and yields ? everywhere else; filters are useful in writing equations that describe the domains of functions. The combining form ∇ (defined by $\nabla:p \equiv p \rightarrow id; err$) converts any predicate into a filter for the domain on which the predicate is true. Thus $\nabla:isgram:x$ rewrites to x if x is a gram, and to ? otherwise. Filters are often used as if they were the name of the set on which they are defined. An example of a useful domain equation is:

$$f \equiv \nabla:r \circ f \circ \nabla:d$$

This equation expresses the fact that f maps every element x of $\nabla:d$ (ie, $\nabla:d:x$ rewrites to x) into $\nabla:r$ or into ? and that f must not be defined outside of $\nabla:d$. Thus $\nabla:d$ contains the domain of f and $\nabla:r$ contains its range. Note that $id \equiv \nabla:T$ is the filter for the set of all values and $err \equiv \nabla:F$ is the filter for the empty set.

A *qualified equation* $E_1 \equiv E_2 \leftarrow\leftarrow P$ where P is a predicate, is an abbreviation for the equation $E_1 \circ \nabla:P \equiv E_2 \circ \nabla:P$. Thus this qualified equation asserts that E_1 and E_2 denote the same function when restricted to the domain on which P is true.

Assertion declarations may have the forms:

Asn $E_1 \equiv E_2$ (an *unqualified* assertion without variables)

Asn $E_1 \equiv E_2 \leftarrow\leftarrow P$ (a *qualified* assertion without variables)

Asn (L) $E_1 \equiv E_2$ (an *unqualified* assertion with variables)

Asn (L) $E_1 \equiv E_2 \leftarrow\leftarrow P$ (a *qualified* assertion with variables)

where L is a list of *variable entries* separated by commas, and E_1 , E_2 , and P are function expressions using some or all of the variables listed in L (if any). A variable entry has the form *variable* $\in Q$ where *variable* is a new syntactic construct that can appear only in assertions; a variable is either a concatenation of *italic* letters or a function name (if function names are used, it is good practice to use distinctive ones, such as *xyz!*, so that occurrences of a variable in an expression can be easily identified as such). Q is a predicate expression. A variable entry in L indicates that *variable* is a quantified variable in E_1 , E_2 , and P which ranges over the domain given by Q . An assertion with variables asserts that its equation holds for every substitution, for each variable, of a value that satisfies the Q for that variable. For example,

Asn $(x \in \text{isident}) \text{ fetch}:x \circ \text{assign}:<x,1> \equiv \sim 1 \leftarrow \leftarrow \text{isstore}$

asserts that fetching x from the store formed by assigning 1 to x in any store results in 1, provided that x is an identifier. The degree to which assertions will be consistent with the properties of functions in a defined type will depend on the system implementation and its ability to check the consistency of assertions and definitions. (Notice that assertion declarations are not restricted to equations about types; they may involve any functions and may appear in any declaration list.)

Signature declarations. A signature declaration is an abbreviation for a special kind of assertion that asserts that a function maps a given domain into a given range; it has the form:

Sig $f :: D \Rightarrow R$,

where f is the name of the function whose signature is being given; D and R are predicates; D is true on the domain of f and R on its range. Thus the above signature is an abbreviation for the assertion:

Asn $f \equiv \nabla:R \circ f \circ \nabla:D$.

In the example,

Sig $\text{adgm} :: \text{seqof:isgram} \wedge \text{not} \circ \text{isnull} \Rightarrow \text{isgram}$

asserts that adgm maps nonempty sequences of grams into grams; it abbreviates the assertion

Asn $\text{adgm} \equiv \nabla:\text{isgram} \circ \text{adgm} \circ \nabla:(\text{seq:isgram} \wedge \text{not} \circ \text{isnull})$

A given function may appear in any number of signatures and assertions. The following typical Rep declaration always implies the following signatures and assertions for the functions that it defines:

Rep 'newtype $\Leftarrow \Rightarrow$ oldpred

implies:

Sig $\text{isnewtype} :: \text{isval} \Rightarrow \text{isbool}$

Sig $\text{mknewtype} :: \text{oldpred} \Rightarrow \text{isnewtype}$

Sig $\text{umnewtype} :: \text{isnewtype} \Rightarrow \text{oldpred}$

Asn $\text{umnewtype} \circ \text{mknewtype} \equiv \nabla:\text{oldpred}$

Asn $\text{mknewtype} \circ \text{umnewtype} \equiv \nabla:\text{isnewtype}$

Examples of Defined Types

The following partial examples illustrate various ways in which new types can be defined.

The enumerated type color. This example defines a type consisting of three colors.

Rep 'color $\iff$ memseq $\circ$ [id, ~<'red, 'green, 'blue>]

Here memseq is true only if its first argument is an element of its second argument; thus there are only three members of the type 'color. mkcolor is defined only for the three identifiers listed; it converts 'red into the color <'color, 'red> .

The unit type gram. In this treatment of grams the primitive + is extended to add non-empty sequences of grams (ie, <> is treated as the empty sequence of numbers, not grams).

Rep 'gram $\iff$ isreal

Ext + $\equiv$ seqof:isgram $\wedge$ not $\circ$ isnull $\rightarrow$ mkgram $\circ$ + $\circ$ α :umgram

The *extension definition* Ext extends the definition of + when its argument is a nonempty sequence of grams (one should use +/0 or +/mkgram:0 instead of + if <> can be an argument, to indicate whether numbers or grams are being added).

Recall that an extension definition $\text{Ext } f \equiv p \rightarrow E$ extends the definition of f to be E in the domain for which p is true, and f retains its former definition when p is false. Thus, in this case, when + is applied to a non-empty sequence of grams, it is taken to be $\text{mkgram} \circ + \circ \alpha:\text{umgram}$, which will add a sequence of grams as follows: the argument of + in this latter expression will be a sequence of numbers (ie, not grams), the extension definition will be used again for this occurrence of +, but the predicate will be false and hence the primitive + will be used to add the sequence of numbers provided by $\alpha:\text{umgram}$. Finally, the result will be converted to grams by mkgram.

The result of these declarations is that values of type 'gram can be formed by applying mkgram to a real, eg, $\text{mkgram}:1.3 = \langle \text{'gram}, 1.3 \rangle$; + will add gram quantities and produce a gram quantity (as well as adding numbers), eg, $+: \langle \langle \text{'gram}, 2.1 \rangle, \langle \text{'gram}, 3 \rangle \rangle = \langle \text{'gram}, 5.1 \rangle$ and $+: \langle 2.1, 3 \rangle = 5.1$, whereas $+: \langle \langle \text{'gram}, 2.1 \rangle, 3 \rangle = ?$. Grams can be converted to pure reals by applying umgram.

The recursive type tree. The declaration

Rep 'tree $\iff$ isatom $\vee$ [[istree, istree]]

asserts that a tree is to be represented either by an atom or (recursively) by a pair of trees. mktree will convert either of these into a tree and umtree will convert any tree into an atom or a pair of trees. For example, $\text{mktree}:1 = \langle \text{'tree}, 1 \rangle$; $\text{mktree}: \langle \langle \text{'tree}, 1 \rangle, \langle \text{'tree}, 2 \rangle \rangle = \langle \text{'tree}, \langle \langle \text{'tree}, 1 \rangle, \langle \text{'tree}, 2 \rangle \rangle \rangle$; and $\text{mktree}: \langle 1, 2 \rangle = ?$.

Def isatmtree $\equiv$ isatom $\circ$ umtree $\rightarrow$ T; F

Def ltree $\equiv$ s1 $\circ$ umtree

Def rtree $\equiv$ s2 $\circ$ umtree

The result of these definitions is that *isatmtree* is defined only on trees and is true for atomic trees and false for other trees. *ltree* is defined for non-atomic trees and yields the left subtree of a such a tree.

Def flattree $\circ$ istree $\equiv$ isatmtree $\rightarrow$ [umtree]; cat $\circ$ [flattree $\circ$ ltree, flattree $\circ$ rtree]

This definition of *flattree* makes it defined only on trees; it maps every tree into a sequence that contains all its leaves. *cat* is the catenate function that produces a sequence from a sequence of sequences. The definition of *flattree* implies the following signature:
Sig flattree :: istree $\Rightarrow$ seqof:isatom.

Abstract Types

If the above definitions in the tree example were included in a declaration list, such as:

begin

begin

Rep 'tree $\Leftarrow$ isatom $\vee$ [[istree, istree]]

Def isatmtree $\equiv$ isatom $\circ$ umtree $\rightarrow$ T; F

Def ltree $\equiv$ s1 $\circ$ umtree

Def rtree $\equiv$ s2 $\circ$ umtree

Def flattree $\circ$ istree $\equiv$ isatmtree $\rightarrow$ [umtree]; cat $\circ$ [flattree $\circ$ ltree, flattree $\circ$ rtree]

end

Def f₁ $\equiv$ e₁

.....

Def f_n $\equiv$ e_n

end

then the expressions e_i could use any of the functions *istree*, *mktree*, *umtree*, *isatmtree*, *ltree*, *rtree*, *flattree*. Thus tree would be a new ("concrete") type, whose representation is accessible to the "users" of the type (that is, the expressions e_i). Simply by replacing the second, nested *begin* by *hide(mktree, umtree)*, this becomes an example of an *abstract* (or "encapsulated") type definition, ie one in which the underlying representation is invisible

outside of the nested scope in which the type is defined. In this case the expressions e_i can manipulate trees using only the exported functions (*istree*, *isatomtree*, *ltree*, *rtree*, *flattree*).

Primitive Abstract Types

Primitive abstract types follow a desirable principal for any abstract type: the assertions and signatures constitute an exact specification of the operations. This is not necessarily the case with defined types in which the definitions of the operations determine their actual properties, not the assertions.

Appendix D defines the primitive abstract type *store* that provides the following functions that are “visible” as primitive functions: *isstore*, *mkempty_st*, *isempty_st*, *find*, *fetch*, *save*, *assign*.

5. Appendix A: The Primitive Functions

This appendix gives an informal description of the primitive functions. See Appendix B for a list of all the rewrite rules.

Arithmetic Operators

Sample Applications	Comments
$+:<x_1, \dots, x_n> = y$ $+:<2, 3, 4, 6> = 15$ $+:<f_1, \dots, f_n> = + \circ [f_1, \dots, f_n]$ $+:<s_1, s_4, s_5>:<1, 2, 3, 4, 5> = 10$	y is the sum of the numbers $x_1, \dots, x_n$ if $f_1, \dots, f_n$ are functions by previous rule
$* \text{ multiplies its arguments}$	similar to $+$
$-:<x_1, x_2> = y$ $-:<3, 2> = 1$ $-:<f_1, f_2> = - \circ [f_1, f_2]$ $-:<\sim 3, \sim 2>:x = 1$	y is the difference of the numbers x_1 and x_2 if f_1, f_2 are functions by previous rule
$\div \text{ divides its arguments}$	similar to $-$
$\text{neg}: -2 = 2$	neg subtracts its argument from 0

Since they are associative functions, $+$ and $*$ apply to sequences of arbitrary length. However, $-$ and $\div$ are binary functions. In order to facilitate infix notation, all the arithmetic functions have been extended so that they apply to sequences of functions as well as sequences of numbers.

Boolean Operators

Rewrite Rules	Comments
$\text{and}: \langle x_1, \dots, x_n \rangle = \text{'T'}$ $\text{and}: \langle x_1, \dots, x_n \rangle = \text{'F'}$	if none of the x_i is 'F' if at least one of the x_i is 'F'
$\text{or}: \langle x_1, \dots, x_n \rangle = \text{'T'}$ $\text{or}: \langle x_1, \dots, x_n \rangle = \text{'F'}$	if not all of the x_i are 'F' if all of the x_i are 'F'
$\text{T}: x = \text{'T'}$	
$\text{F}: x = \text{'F'}$	
$\wedge: \langle f_1, \dots, f_n \rangle = \text{and} \circ [f_1, \dots, f_n]$	
$\vee: \langle f_1, \dots, f_n \rangle = \text{or} \circ [f_1, \dots, f_n]$	
$\text{not}: \text{'F'} = \text{'T'}$ $\text{not}: x = \text{'F'}$	if x is not 'F'

Notice that **and**, **or**, and **not** treat all values other than 'F' as if they were 'T'.

Comparison Operators

Operators	Comments
$\text{eq}:\langle x_1, x_2 \rangle = \text{'T}$ $\text{eq}:\langle x_1, x_2 \rangle = \text{'F}$ $\text{eq}:\langle x_1, x_2 \rangle = \text{'F}$ $\text{eq}:\langle x_1, x_2 \rangle = ?$	if x_1 and x_2 are the same object if x_1 and x_2 are not the same object if x_1 is an object and x_2 is not if x_1 and x_2 are functions
$=:\langle f_1, f_2 \rangle \text{ is eq}^\circ[f_1, f_2]$	
$\neq$ is $\text{not}^\circ =$	
$\text{isnull}:\langle \rangle = \text{'T}$ $\text{isnull}:x = \text{'F}$	otherwise
$<:\langle x_1, x_2 \rangle = \text{'T}$ $<:\langle x_1, x_2 \rangle = \text{'F}$ $<:\langle f_1, f_2 \rangle = <^\circ[f_1, f_2]$	if x_1 is less than x_2 if x_1 is not less than x_2 if f_1 and f_2 are functions
$\leq > \geq$ are similar to $<$	

Combining Forms

Operators	Infix notation
$\circ: \langle f_1, \dots, f_n \rangle : x = f_1 : (f_2 : (f_3 : (\dots (f_n : x) \dots)))$	$(f \circ g) : x = f : (g : x)$
$: \langle f_1, \dots, f_n \rangle : x = f_n : (f_{n-1} : (f_{n-2} : (\dots (f_1 : x) \dots)))$	$(f g) : x = g : (f : x)$
$\text{cons}: \langle f_1, \dots, f_n \rangle : x = \langle f_1 : x, \dots, f_n : x \rangle$	$[f_1, \dots, f_n] = \text{cons}: \langle f_1, \dots, f_n \rangle$
$\text{apply}: \langle f, x \rangle = f : x$	$f \text{ apply } x = f : x$
$K : x : y = x$	
$f' = \text{lift}: f$ $\text{lift}: f : \langle x_1, \dots, x_n \rangle = f \circ [x_1, \dots, x_n]$	$g \ f' \ h = f \circ [g, h]$ $(f \ \text{lift}: g \ h) : x = (f : x) \ g \ (h : x)$
$\text{condapp}: \langle 'F, x_2, x_3 \rangle = x_3$ $\text{condapp}: \langle x_1, x_2, x_3 \rangle = x_2 \text{ if } x_1 \text{ is not } 'F$	
$\text{cond}: \langle f_1, f_2, f_3 \rangle : x = \text{condapp}: \langle f_1 : x, f_2, f_3 \rangle : x$	$(f_1 \rightarrow f_2; f_3) = \text{cond}: \langle f_1, f_2, f_3 \rangle$

Predicate Combining Forms

Predicate Combining Forms
$\mapsto: \langle f, g \rangle = (\text{is_nonnull_sequence} \rightarrow (f \circ s1) \wedge (g \circ t1); F)$
$\leftarrow: \langle f, g \rangle = (\text{is_nonnull_sequence} \rightarrow (f \circ t1r) \wedge (g \circ r1); F)$
$\llbracket f_1, \dots, f_n \rrbracket = \text{econs}: \langle f_1, \dots, f_n \rangle$ $\text{econs}: \langle f_1, \dots, f_n \rangle = (\text{is_sequence_of_length_n} \rightarrow \text{and} \circ [f_1 \circ s1, \dots, f_n \circ sn]; F)$
$\text{seqof}: f = (\text{isseq} \rightarrow \text{and} \circ \alpha : f; F)$

All of the above predicate combining forms produce total predicates when given total predicates as their arguments.

Sequence Combining Forms

Example Uses	Comments
$\backslash : \langle f, a \rangle : \langle \rangle = a$ $\backslash : \langle f, a \rangle : \langle x \rangle = x$ $\backslash : \langle f, a \rangle : \langle x_1, x_2, x_3, x_4 \rangle = f : \langle f : \langle f : \langle x_1, x_2 \rangle, x_3 \rangle, x_4 \rangle$	left linear insert
$/ : \langle f, a \rangle : \langle \rangle = a$ $/ : \langle f, a \rangle : \langle x \rangle = x$ $/ : \langle f, a \rangle : \langle x_1, x_2, x_3, x_4 \rangle = f : \langle x_1, f : \langle x_2, f : \langle x_3, x_4 \rangle \rangle \rangle$	right linear insert
$\text{tree} : \langle f, a \rangle : \langle \rangle = a$ $\text{tree} : \langle f, a \rangle : \langle x \rangle = x$ $\text{tree} : \langle f, a \rangle : \langle x_1, x_2, x_3, x_4 \rangle = f : \langle f : \langle x_1, x_2 \rangle, f : \langle x_3, x_4 \rangle \rangle$	binary tree insert
$\alpha : f : \langle x_1, \dots, x_n \rangle = \langle f : x_1, \dots, f : x_n \rangle$	
$\text{compress} : \text{iseven} : \langle 1, 2, 3, 4 \rangle = \langle 2, 4 \rangle$	compress: f yields subsequence of values that pass f

Each of the above five sequence combining forms when applied to its argument yields a function that works on sequences of arbitrary length.

Sequence Functions

Sequence Functions	Explanation of name
$al: \langle x_0, \langle x_1, \dots, x_n \rangle \rangle = \langle x_0, x_1, \dots, x_n \rangle$	Append to the Left of a sequence
$ar: \langle \langle x_1, \dots, x_{n-1} \rangle, x_n \rangle = \langle x_1, \dots, x_{n-1}, x_n \rangle$	Append to the Right of a sequence
$cat: \langle \langle x_1^1, \dots, x_{n_1}^1 \rangle, \dots, \langle x_1^m, \dots, x_{n_m}^m \rangle \rangle = \langle x_1^1, \dots, x_{n_1}^1, \dots, x_1^m, \dots, x_{n_m}^m \rangle$	CATenate sequences
$distl: \langle x, \langle y_1, \dots, y_n \rangle \rangle = \langle \langle x, y_1 \rangle, \dots, \langle x, y_n \rangle \rangle$	DISTribute from Left
$distr: \langle \langle y_1, \dots, y_n \rangle, z \rangle = \langle \langle y_1, z \rangle, \dots, \langle y_n, z \rangle \rangle$	DISTribute from Right
$ints_to: n = \langle 1, 2, 3, \dots, n \rangle$	INTEgerS TO
$trans: \langle \langle x_1^1, \dots, x_n^1 \rangle, \dots, \langle x_1^m, \dots, x_n^m \rangle \rangle = \langle \langle x_1^1, \dots, x_1^m \rangle, \dots, \langle x_n^1, \dots, x_n^m \rangle \rangle$	TRANSPose a matrix
$merge: \langle \langle 2, 9, 4 \rangle, \langle 1, 5, 6 \rangle \rangle = \langle 1, 2, 5, 6, 9, 4 \rangle$	MERGEs sequences
$sel: \langle i, \langle x_1, \dots, x_n \rangle \rangle = x_i \quad \text{if } 1 \leq i \leq n$ $sel: \langle i, \langle x_1, \dots, x_n \rangle \rangle = x_{n+i+1} \quad \text{if } -n \leq i \leq -1$	SELEct
$si: x = sel: \langle i, x \rangle$ for positive integer i	Select Ith element
$ri: x = sel: \langle neg: i, x \rangle$ for positive integer i	select from Right the Ith element
$tl: \langle x_1, \dots, x_n \rangle = \langle x_1, \dots, x_{n-1} \rangle$ if $1 \leq n$	TaiL of a sequence
$tlr: \langle x_1, \dots, x_n \rangle = \langle x_2, \dots, x_n \rangle$ if $1 \leq n$	TaiL Right of a sequence
$makeseq: x \rightarrow \langle x \rangle$ if x is not a sequence $makeseq: \langle x_1, \dots, x_n \rangle \rightarrow \langle x_1, \dots, x_n \rangle$	MAKE into a SEQUENCE
$length: \langle x_1, \dots, x_n \rangle = n$	

String Functions

Functions	Comments
<code>str_to_seq: "c₁...c_n" = <"c₁", ..., "c_n"></code>	Converts strings to sequences of chars
<code>seq_to_str: <"c₁", ..., "c_n"> = "c₁...c_n"</code>	Converts sequences of chars to strings
<code>scat: <"c₁"...c_{n₁}", ..., "c₁^m"...c_{n_m}^m"> = "c₁"...c_{n₁}"...c₁^m"...c_{n_m}^m"</code>	String catenation
<code>makestring: "c₁...c_n" = "c₁...c_n"</code> <code>makestring: i = "i" if i is a number</code>	Makes everything into a string by enclosing with " "
<code>++ : x = raise: (scat ∘ α: makestring): x</code>	
<code>locate: <x, y> = n</code>	if substring x starts at position n
<code>extract: <m, n, x> = substring of y</code>	starting at position m of length n

Integer Functions

Examples	Examples	Examples
<code>ceil: 4.5 = 5</code>	<code>ceil: -2.5 = -2</code>	<code>ceil: 6 = 6</code>
<code>floor: 4.5 = 4</code>	<code>floor: -2.5 = -3</code>	<code>floor: 6 = 6</code>

Type Functions

The following primitive predicates are true on the corresponding primitive types of FL and false elsewhere. All primitive predicates are *total* functions, ie, defined on all arguments.

Predicate	Primitive Type
isval	the set of values
isobj	the set of objects
isfunc	the set of functions
isatom	the set of atoms
isbool	the set {'T', 'F'}
isident	the set of identifiers
isnum	the set of numbers
islongnum	the set of extended precision numbers
isreal	the set of real numbers
isint	the set of integers
isposreal	the set of real numbers > 0
isnegreal	the set of real numbers < 0
isnonnegreal	the set of real numbers ≥ 0
isnonposreal	the set of real numbers ≤ 0
isstore	the set of stores
isstring	the set of strings
ischar	the set of characters
isseq	the set of sequences
islen:n	the set of sequences of length n ($n \geq 0$)

At present, the set of numbers and the set of real numbers are the same. In the future, the set of numbers may include complex numbers. These primitive types are partially ordered by set inclusion as follows (using a predicate to stand for the type it denotes):

$\text{isint} \subset \text{isreal} \subset \text{isnum} \subset \text{isatom} \subset \text{isobj} \subset \text{isval}$
 $\text{islongnum} \subset \text{isnum}$
 $\text{isnonnegreal} \subset \text{isposreal} \subset \text{isnum}$
 $\text{isnonposreal} \subset \text{isnegreal} \subset \text{isnum}$
 $\text{isbool} \subset \text{isident} \subset \text{isatom}$
 $\text{ischar} \subset \text{isstring} \subset \text{isatom}$
 $\text{islen:n} \subset \text{isseq} \subset \text{isval}$ (for all non-negative integers)
 $\text{isfunc} \subset \text{isval}$

The rewrite rules for the above primitive type predicates all have the form:

$\text{istype}_t: x \rightarrow 'T$ if x belongs to $'\text{type}_t$
 $\text{istype}_t: x \rightarrow 'F$ otherwise

The following types are described in Section 2: values, functions, atoms, identifiers, objects, numbers, strings, chars, sequences, sequences of length n . It is hoped that for the other types

mentioned the names are self explanatory. All tag pairs belong to only one primitive type, the set of values.

Store Functions

Functions	Comments
$\text{isstore}:x = \text{'T}$ $\text{isstore}:x = \text{'F}$	if x is a store if x is not a store
$\text{mkempty_st}:x$ produces an empty store	
$\text{isempty_st}:x = \text{'T}$ $\text{isempty_st}:x = \text{'F}$	if x is an empty store otherwise
$\text{find}:<x,s> = y$	if the pair $<x,y>$ is in the store s
$\text{fetch}:x:y = \text{find}:<x,y>$	fetch is a Curried version of find
$\text{save}:<<x,y>,s> = t$	t is the store s with the pair $<x,y>$ added
$\text{assign}:<x,y>:s = \text{save}:<<x,y>,z>$	assign is a Curried version of save

Miscellaneous Functions

No language would be complete without a few miscellaneous functions.

Functions	Comments
$\text{raise}:g:<f_1,\dots,f_n> = g \circ [f_1,\dots,f_n]$	if $f_1,\dots,f_n$ are all functions
$\text{fix}:f:x = f:(\text{fix}:f):x$	the fixpoint operator
$\text{mrfix}:s:i:x = (s_i:s):<\text{mrfix}:s:1,\dots,\text{mrfix}:s:n>:x$	the mutually recursive fixpoint operator
$\text{args}:i:f:x_1:\dots:x_i = f:x_1:\dots:x_i$	$\text{args}:i:f:x_1:\dots:x_{i-1}$ is a value. Delays application.
$\Delta:f = (f \rightarrow \text{id}; \text{err})$	
$\nabla:f = (f \rightarrow (\text{id} \rightarrow \text{id}; T); F)$	
$\text{doio}:f = (f \rightarrow \text{id}; \text{id})$	preserves argument but changes history
$\text{err}:x = ?$	
$\text{id}:x = x$	the identity function

History Dependent Functions

The history consists of a stream h_s of the entire record of the activity of the I/O devices and file system and flags used to control access to h_s . For example, $h_s = (<\text{'outflag'}, <\text{'screen'}, \text{"Give me a number"}>>, <\text{'inflag'}, \text{'keyboard'}>)$ is a history stream of length 2 indicating that first Give me a number was printed on the device named 'screen' and then some input was read from the device named 'keyboard'.

The function `get` ignores its argument and returns the file system. The function `put` leaves its argument unchanged but installs that argument as the new value of the file system (which typically resides on a disk or some other permanent storage device).

The function `in` takes a device name as its argument and gets the next input from the device with that name. The function `InReady` takes a device name as its argument and checks to see if that input device is ready to produce an argument.

The function `out` takes a pair consisting of a device name and an arbitrary value and outputs that value on the device. If the output operation was successful, then `out` returns the value that was output; otherwise, `out` returns `?`. The function `OutReady` tests to see if the output will be successful.

When applied to a function, β produces a new function that acts the same as the argument to β except that I/O operations and changes to the file system are prohibited (ie, they cause an error). The function γ is similar to β but is even more restrictive in that γ prohibits accesses to the file system (as well as prohibiting I/O operations and changes to the file system). These functions are quite useful in expressing algebraic laws. For example, $[f_1, f_2] \circ g$ is *not* necessarily the same function as $[f_1 \circ g, f_2 \circ g]$, because g does its I/O twice in the second function but only once in the first function. However, the equations $[\beta:f_1, \beta:f_2] \circ \beta:g = [\beta:f_1 \circ \beta:g, \beta:f_2 \circ \beta:g]$ and $[f_1, f_2] \circ \gamma:g = [f_1 \circ \gamma:g, f_2 \circ \gamma:g]$ *do* hold.

The function `pmark` leaves its argument and the file system unchanged, except it makes the same mark in the history stream that `put` makes. The function `pmark` is useful primarily in expressing algebraic laws. For example, $\text{put} \circ \text{put} = \text{pmark} \circ \text{put}$, but the second function is much faster since the first function writes the same file system out to disk twice while the second function writes it out only once.

6. Appendix B: The Rewriting Process

The rewrite rules are divided into two sets: those that are independent of the history stream and those that depend on and possibly change the history stream.

History Independent Functions

The history independent rewrite rules are listed below with a , b , f , g , i , s , x , and y representing arbitrary (fully parenthesized) values.

$? \rightarrow ?$

$a:x \rightarrow ?$ for any atom a

$\langle x_1, \dots, x_n \rangle : y \rightarrow ?$

$\langle x_1, x_2 \rangle : y \rightarrow ?$

$+:x \rightarrow \text{raise:add}:x$

$-:x \rightarrow \text{raise:minus}:x$

$*:x \rightarrow \text{raise:multiply}:x$

$\div:x \rightarrow \text{raise:divide}:x$

$\wedge:x \rightarrow \text{lift:and}:x$

$\vee:x \rightarrow \text{lift:or}:x$

$=:x \rightarrow \text{lift:eq}:x$

$\neq:x \rightarrow \text{not}(=:x)$

$\geq:x \rightarrow \text{raise:greaterequal}:x$

$>:x \rightarrow \text{raise:greaterthan}:x$

$\leq:x \rightarrow \text{raise:lessequal}:x$

$<:x \rightarrow \text{raise:lessthan}:x$

$\circ:\langle f_1, \dots, f_n \rangle : y \rightarrow f_1:(f_2:\dots(f_{n-1}:(f_n:y))\dots)$ for $n \geq 1$

$\circ:\langle \rangle \rightarrow \text{id}$

$\circ:x \rightarrow ?$ if x is not a sequence

$|\langle f_1, \dots, f_n \rangle : y \rightarrow f_n:(f_{n-1}:\dots(f_2:(f_1:y))\dots)$ for $n \geq 1$

$|\langle \rangle \rightarrow \text{id}$

$|:x \rightarrow ?$ if x is not a sequence

$\vdash: \langle f_1, f_2 \rangle : x \rightarrow \text{'F}$ if x is not a sequence of length at least one
 $\vdash: \langle f_1, f_2 \rangle : \langle x_1, \dots, x_n \rangle \rightarrow \text{condapp:} \langle \text{and:} \langle f_1 : x_1, f_2 : \langle x_2, \dots, x_n \rangle \rangle, T, F \rangle : \langle x_1, \dots, x_n \rangle$
for $n \geq 1$
 $\vdash: x \rightarrow ?$ if x does not have the form $\langle x_1, x_2 \rangle$
 $\Leftarrow: \langle f_1, f_2 \rangle : x \rightarrow \text{'F}$ if x is not a sequence of length at least one
 $\Leftarrow: \langle f_1, f_2 \rangle : \langle x_1, \dots, x_n \rangle \rightarrow \text{condapp:} \langle \text{and:} \langle f_1 : \langle x_1, \dots, x_{n-1} \rangle, f_2 : x_n \rangle \rangle, T, F \rangle : \langle x_1, \dots, x_n \rangle$
for $n \geq 1$
 $\Leftarrow: x \rightarrow ?$ if x does not have the form $\langle x_1, x_2 \rangle$
 $\backslash: \langle f, a \rangle : \langle \rangle \rightarrow a$
 $\backslash: \langle f, a \rangle : \langle x \rangle \rightarrow x$
 $\backslash: \langle f, a \rangle : \langle x_1, \dots, x_n \rangle \rightarrow f: \langle \backslash: \langle f, a \rangle : \langle x_1, \dots, x_{n-1} \rangle, x_n \rangle$ for $n \geq 2$
 $\backslash: \langle f, a \rangle : x \rightarrow ?$ if x is not a sequence
 $\backslash: x \rightarrow ?$ is x is not of the form $\langle x_1, x_2 \rangle$
 $/: \langle f, a \rangle : \langle \rangle \rightarrow a$
 $/: \langle f, a \rangle : \langle x \rangle \rightarrow x$
 $/: \langle f, a \rangle : \langle x_1, \dots, x_n \rangle \rightarrow f: \langle x_1, /: \langle f, a \rangle : \langle x_2, \dots, x_n \rangle \rangle$ for $n \geq 2$
 $/: \langle f, a \rangle : x \rightarrow ?$ if x is not a sequence
 $/: x \rightarrow ?$ is x is not of the form $\langle x_1, x_2 \rangle$
 $++ : x \rightarrow \text{raise:} (\circ: \langle \text{scat}, \alpha: \text{makestring} \rangle): x$
 $\Leftarrow: x \rightarrow \text{assign:} x$
 $\nabla: f \rightarrow \text{cond:} \langle f, \text{cond:} \langle \text{id}, \text{id}, T \rangle, F \rangle$
 $\alpha: f: \langle x_1, \dots, x_n \rangle \rightarrow \langle f: x_1, \dots, f: x_n \rangle$
 $\alpha: f: x \rightarrow ?$ if x is not a sequence
 $\Delta: f \rightarrow \text{cond:} \langle f, \text{id}, \text{err} \rangle$
 $\text{add:} \langle x_1, \dots, x_n \rangle \rightarrow y$ where y is the sum of the numbers $x_1, \dots, x_n$
 $\text{add:} x \rightarrow ?$ if x is not a sequence of numbers
For example, $+: \langle 2.5, 3.5, 4 \rangle \rightarrow 10$
 $\text{al:} \langle x_0, \langle x_1, \dots, x_n \rangle \rangle \rightarrow \langle x_0, \dots, x_n \rangle$ for $n \geq 0$
 $\text{al:} x \rightarrow ?$ if x does not have the form $\langle x_0, \langle x_1, \dots, x_n \rangle \rangle$
 $\text{and:} \langle x_1, \dots, x_n \rangle \rightarrow \text{'T}$ if no x_i is 'F.
 $\text{and:} x \rightarrow \text{'F}$ otherwise.
 $\text{apply:} \langle f, x \rangle \rightarrow f: x$
 $\text{apply:} x \rightarrow ?$ if x does not have the form $\langle x_1, x_2 \rangle$

$\text{ar}: \langle \langle x_1, \dots, x_{n-1} \rangle, x_n \rangle \rightarrow \langle x_1, \dots, x_n \rangle$ for $n \geq 0$
 $\text{ar}: x \rightarrow ?$ if x does not have the form $\langle \langle x_1, \dots, x_{n-1} \rangle, x_n \rangle$
 $\text{args}: n: f: x_1: \dots: x_n \rightarrow f: x_1: \dots: x_n$ for $n \geq 0$
 $\text{assign}: x$ see Appendix D
 $\text{cat}: x \rightarrow \text{tree}: \langle \text{cat2}, \langle \rangle \rangle: x$
 $\text{cat2}: \langle \langle x_1, \dots, x_m \rangle, \langle y_1, \dots, y_n \rangle \rangle \rightarrow \langle x_1, \dots, x_m, y_1, \dots, y_n \rangle$
 $\text{cat2}: x \rightarrow ?$ if x is not a pair of sequences
 $\text{ceil}: x \rightarrow y$ where y is the smallest integer $\geq x$ if x is a number
 $\text{ceil}: x \rightarrow ?$ if x is not a number
 $\text{compress}: f: \langle \rangle \rightarrow \langle \rangle$
 $\text{compress}: f: \langle x_1, \dots, x_n \rangle \rightarrow$
 $\text{condapp}: \langle f: x_1, \dots, f: x_n \rangle, \text{compress}: f: \langle x_2, \dots, x_n \rangle$ for $n \geq 1$
 $\text{compress}: f: x \rightarrow ?$ if x is not a sequence
 $\text{cond}: \langle f_0, f_1, f_2 \rangle: x \rightarrow \text{condapp}: \langle f_0: x, f_1, f_2 \rangle: x$
 $\text{cond}: x \rightarrow ?$ if x is not of the form $\langle x_1, x_2, x_3 \rangle$
 $\text{condapp}: \langle 'F, f_1, f_2 \rangle \rightarrow f_2$
 $\text{condapp}: \langle y, f_1, f_2 \rangle \rightarrow f_1$ if y is not $'F$
 $\text{condapp}: x \rightarrow ?$ if x does not have the form $\langle x_1, x_2, x_3 \rangle$
 $\text{cons}: \langle f_1, \dots, f_n \rangle: x \rightarrow \langle f_1: x, \dots, f_n: x \rangle$
 $\text{cons}: x \rightarrow ?$ if x is not a sequence
 $\text{distl}: \langle y, \langle x_1, \dots, x_n \rangle \rangle \rightarrow \langle \langle y, x_1 \rangle, \dots, \langle y, x_n \rangle \rangle$
 $\text{distl}: \langle x, y \rangle \rightarrow ?$ if y is not a sequence
 $\text{distl}: x \rightarrow ?$ if x is not a pair
 $\text{distr}: \langle \langle x_1, \dots, x_n \rangle, y \rangle \rightarrow \langle \langle x_1, y \rangle, \dots, \langle x_n, y \rangle \rangle$
 $\text{distr}: \langle x, y \rangle \rightarrow ?$ if x is not a sequence
 $\text{distr}: x \rightarrow ?$ if x is not a pair
 $\text{divide}: \langle x_1, x_2 \rangle \rightarrow y$ where y is the quotient of the numbers x_1, x_2
 $\text{divide}: \langle x, 0 \rangle \rightarrow ?$
 $\text{divide}: x \rightarrow ?$ if x does not have the form $\langle x_1, x_2 \rangle$ with x_1, x_2 both numbers
For example, $\text{divide}: \langle 7, 2 \rangle \rightarrow 3.5$
 $\text{doio}: f \rightarrow \text{cond}: \langle f, \text{id}, \text{id} \rangle$
 $\text{econs}: \langle f_1, \dots, f_n \rangle: \langle x_1, \dots, x_n \rangle \rightarrow \text{and}: \langle f_1: x_1, \dots, f_n: x_n \rangle$

$\text{econs}: \langle f_1, \dots, f_n \rangle : x \rightarrow \text{'F}$ if x is not a sequence of length n
 $\text{econs}: y \rightarrow ?$ if y is not a sequence
 $\text{eq}: x \rightarrow ?$ if x does not have the form $\langle x_1, x_2 \rangle$
 $\text{eq}: \langle x_1, x_2 \rangle \rightarrow \text{'F}$ if
 $\quad x_1$ is a function and x_2 is not
 $\quad x_1$ is a tagged value and x_2 is not
 $\quad x_1$ is a sequence and x_2 is not
 $\quad x_1$ is an atom and x_2 is not the same atom
 $\text{eq}: \langle f_1, f_2 \rangle \rightarrow ?$ if both f_1 and f_2 are functions
 $\text{eq}: \langle \langle a, x_1 \rangle, \langle b, x_2 \rangle \rangle \rightarrow \text{'F}$ if a and b are different identifiers
 $\text{eq}: \langle \langle a, x_1 \rangle, \langle a, x_2 \rangle \rangle \rightarrow ?$
 $\text{eq}: \langle a, a \rangle \rightarrow \text{'T}$ if a is an atom
 $\text{eq}: \langle \langle x_1, \dots, x_m \rangle, \langle y_1, \dots, y_n \rangle \rangle \rightarrow \text{'F}$ if $m \neq n$
 $\text{eq}: \langle \langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_n \rangle \rangle \rightarrow \text{and}: \langle \text{EqBool}: \langle x_1, y_1 \rangle, \dots, \text{EqBool}: \langle x_n, y_n \rangle \rangle$
 $\text{err}: x \rightarrow ?$
 $\text{F}: x \rightarrow \text{'F}$
 $\text{fetch}: x$ see Appendix D
 $\text{find}: x$ see Appendix D
 $\text{fix}: f: x \rightarrow f: (\text{fix}: f): x$
 $\text{floor}: x \rightarrow y$ where y is the largest integer $\leq x$ if x is a number
 $\text{floor}: x \rightarrow ?$ if x is not a number
 $\text{greaterequal}: \langle x, y \rangle \rightarrow \text{'T}$ if x, y are real numbers and x is greater than or equal to y
 $\text{greaterequal}: \langle x, y \rangle \rightarrow \text{'F}$ if x, y are real numbers and x is less than y
 $\text{greaterequal}: x \rightarrow \text{'F}$ if x is not a pair of numbers
 $\text{greaterthan}: \langle x, y \rangle \rightarrow \text{'T}$ if x, y are real numbers and x is greater than y
 $\text{greaterthan}: \langle x, y \rangle \rightarrow \text{'F}$ if x, y are real numbers and x is less than or equal to y
 $\text{greaterthan}: x \rightarrow \text{'F}$ if x is not a pair of numbers
 $\text{id}: x \rightarrow x$
 $\text{ints_to}: i \rightarrow \langle 1, \dots, i \rangle$ for positive integers i
 $\text{ints_to}: 0 \rightarrow \langle \rangle$
 $\text{ints_to}: x \rightarrow ?$ if x is not a non-negative integer
For example, $\text{ints_to}: 4 \rightarrow \langle 1, 2, 3, 4 \rangle$
 $\text{isempty_st}: x$ see Appendix D

isnull:<> → 'T

isnull:x → 'F otherwise

isempty_st:x see Appendix D

isXXXX:y → 'T if y is in the type XXXX

isXXXX:y → 'F otherwise.

These primitive functions, as described in Appendix A, are:

isatom, isbool, ischar, isfunc, isident,

isimag, isimagint, islen:n, islongnum, isnegreal,

isnonnegreal, isnonposreal, isnum, isobj, isposreal, isreal,

isseq, isstring, isval

For example, isint:x → 'T if x is an integer

For example, islen:n:<x₁, ..., x_n> → 'T

For example, islen:n:x → 'F otherwise

K:x:y → x

length:<x₁, ..., x_n> → n

length:x → ? if x is not a sequence

lessequal:<x, y> → <x, y> if x, y are real numbers and x is less than or equal to y

lessequal:<x, y> → 'F if x, y are real numbers and x is greater than y

lessequal:x → 'F if x does not have the form <x₁, x₂> with x₁, x₂ both numbers

lessthan:<x, y> → <x, y> if x, y are real numbers and x is less than y

lessthan:<x, y> → 'F if x, y are real numbers and x is greater than or equal to y

lessthan:x → 'F if x does not have the form <x₁, x₂> with x₁, x₂ both numbers

lift:f:x → ◦:<f, cons:x>

locate:<x, y> → n if x is a substring that starts at position n (and not earlier).

x may contain the char ? which matches any single char

and □ which matches any string.

locate:x → ? if x is not a pair of strings

makeseq:<x₁, ..., x_n> → <x₁, ..., x_n>

makeseq:x → <x> if x is not a sequence

makestring:x → x if x is a string

makestring:i → "i" for the number i (in canonical form)

makestring:'aaa → "aaa" for any identifier 'aaa

makestring:f → "Function Expression" for the function f

makestring: $x \rightarrow$ "Tag Pair" for the tag pair x
 makestring: $\langle x_1, \dots, x_n \rangle \rightarrow$ scat: $\langle " < ", \text{makestring}:x_1, " ", \dots, " ", \text{makestring}:x_n " > " \rangle$
 merge: $x \rightarrow \backslash : \langle \text{merge2}, \langle \rangle \rangle : x$
 merge2: $\langle \langle \rangle, \langle y_1, \dots, y_n \rangle \rangle \rightarrow \langle y_1, \dots, y_n \rangle$
 merge2: $\langle \langle x_1, \dots, x_m, \langle \rangle \rangle \rangle \rightarrow \langle x_1, \dots, x_m \rangle$
 merge2: $\langle \langle x_1, \dots, x_m \rangle, \langle y_1, \dots, y_n \rangle \rangle \rightarrow$
 condapp: $\leq : \langle x_1, x_2 \rangle, K : (\text{al} : \langle x_1, \text{merge2} : \langle \langle x_2, \dots, x_m \rangle, \langle y_1, \dots, y_n \rangle \rangle \rangle),$
 $K : (\text{al} : \langle y_1, \text{merge2} : \langle \langle x_1, \dots, x_m \rangle, \langle y_2, \dots, y_n \rangle \rangle \rangle) : 'T$ if $m, n \geq 1$
 merge2: $x \rightarrow ?$ if x is not a pair of sequences
 mrfix: $s : i : x \rightarrow \text{sel} : \langle i, s \rangle : (\alpha : (\text{mrfix} : s) : (\text{ints_to} : (\text{length} : s))) : x$
 multiply: $\langle x_1, \dots, x_n \rangle \rightarrow y$ where y is the product of the numbers $x_1, \dots, x_n$
 $* : x \rightarrow ?$ if x is not a sequence of numbers
 For example, multiply: $\langle 2, 3, 4 \rangle \rightarrow 24$
 neg: $x \rightarrow - : \langle 0, x \rangle$
 not: $x \rightarrow 'F$ if x is not $'F$
 not: $'F \rightarrow 'T$
 or: $\langle x_1, \dots, x_n \rangle \rightarrow 'T$ if at least one of the x_i is not $'F$.
 or: $x \rightarrow 'F$ otherwise.
 raise: $f : x \rightarrow \text{cond} : \langle \wedge : \langle \text{seqof} : \text{isfunc}, \circ : \langle \text{not}, \text{isnull} \rangle \rangle, L : (\text{raise} : f), f \rangle : x$
 ri: $x \rightarrow \text{sel} : \langle \text{neg} : i, x \rangle$ for each positive integer i
 si: $x \rightarrow \text{sel} : \langle i, x \rangle$ for each positive integer i
 save: x see Appendix D
 scat: $x \rightarrow \text{tree} : \langle \text{scat2}, \langle \rangle \rangle : x$
 scat2: $\langle "c_1 \dots c_m", "d_1 \dots d_n" \rangle \rightarrow "c_1 \dots c_m d_1 \dots d_n"$ for chars $"c_1", \dots, "c_m", "d_1", \dots, "d_n"$
 scat2: $x \rightarrow ?$ if x is not a pair of strings
 sel: $\langle i, \langle x_1, \dots, x_n \rangle \rangle \rightarrow x_i$ if i is an integer and $1 \leq i \leq n$
 sel: $\langle i, \langle x_1, \dots, x_n \rangle \rangle \rightarrow x_{n+i+1}$ if i is an integer and $-n \leq i \leq -1$
 sel: $x \rightarrow ?$ otherwise
 For example, sel: $\langle -2, \langle x_1, x_2, x_3, x_4 \rangle \rangle \rightarrow x_3$
 seqof: $f : \langle x_1, \dots, x_n \rangle \rightarrow \text{cond} : \langle \circ : \langle \text{and}, \alpha : f \rangle, \text{id}, K : 'F \rangle : \langle x_1, \dots, x_n \rangle$ for $n \geq 0$
 seqof: $f : x \rightarrow 'F$ if x is not a sequence
 seq_to_str: $\langle "c_1", \dots, "c_n" \rangle \rightarrow "c_1 \dots c_n"$ for $"c_1", \dots, "c_n"$ chars
 seq_to_str: $x \rightarrow x$ if x is not a sequence of chars

$\text{slength}:x \rightarrow \text{length}:(\text{str_to_seq}:x)$
 $\text{str_to_seq}:"c_1...c_n" \rightarrow \langle "c_1", \dots, "c_n" \rangle$ where $c_1, \dots, c_n$ are the characters in the string " $c_1...c_n$ "
 $\text{str_to_seq}:x \rightarrow ?$ if x is not a string
 $\text{subtract}:\langle x_1, x_2 \rangle \rightarrow y$ where y is the difference of the numbers x_1, x_2
 $\text{subtract}:x \rightarrow ?$ if x is not a pair of numbers
 For example, $\text{subtract}:\langle 2, 5 \rangle \rightarrow -3$
 $T:x \rightarrow 'T$
 $\text{tl}:\langle x_1, \dots, x_n \rangle \rightarrow \langle x_2, \dots, x_n \rangle$ for $n \geq 1$.
 $\text{tl}:x \rightarrow ?$ if x is not a sequence with length ≥ 1
 $\text{tlr}:\langle x_1, \dots, x_n \rangle \rightarrow \langle x_1, \dots, x_{n-1} \rangle$ for $n \geq 1$.
 $\text{tlr}:x \rightarrow ?$ if x is not a sequence with length ≥ 1
 $\text{trans}:\langle \langle x_1^1, \dots, x_n^1 \rangle, \dots, \langle x_1^m, \dots, x_n^m \rangle \rangle \rightarrow \langle \langle x_1^1, \dots, x_1^m \rangle, \dots, \langle x_n^1, \dots, x_n^m \rangle \rangle$ for $m \geq 1$
 $\text{trans}:\langle \rangle \rightarrow \langle \rangle$
 $\text{trans}:x \rightarrow ?$ otherwise
 $\text{tree}:\langle f, a \rangle:\langle \rangle \rightarrow a$
 $\text{tree}:\langle f, a \rangle:\langle x \rangle \rightarrow x$
 $\text{tree}:\langle f, a \rangle:\langle x_1, \dots, x_{2n+1} \rangle \rightarrow$
 $f : \langle \text{tree}:\langle f, a \rangle:\langle x_1, \dots, x_{n+1} \rangle, \text{tree}:\langle f, a \rangle:\langle x_{n+2}, \dots, x_{2n+1} \rangle \rangle$ for $n \geq 1$
 $\text{tree}:\langle f, a \rangle:\langle x_1, \dots, x_{2n} \rangle \rightarrow f : \langle \text{tree}:\langle f, a \rangle:\langle x_1, \dots, x_n \rangle, \text{tree}:\langle f, a \rangle:\langle x_{n+1}, \dots, x_{2n} \rangle \rangle$ for $n \geq 1$

The History Dependent Functions

The rewrite rules for the primitive functions that depend on (and possibly change) the history stream are defined using the auxiliary functions G , I , and O . The history consists of three parts: a stream h_s of the entire record of the activity of the I/O devices and file system, a flag h_a to inhibit accesses or changes to the history stream, and a flag h_c to inhibit changes to the history stream. (These flags have the value either *stop* or *go*).

The auxiliary function G returns the latest value of the file system that was stored in the history stream. The function G , when applied to the history h , finds the last occurrence of an entry of the form $\langle \text{FileFlag}, x \rangle$ in h_s and returns x . If there is no such occurrence in h_s , $G(h) = \#$.

Inputs to the rewriting process typically come from a user sitting at a keyboard or from another computer. As such, it is not possible to specify directly what the inputs to the rewriting process are. The purpose of the function I is to provide these inputs (and, in

general, to model the behavior of input devices and people using them). The value of the expression $\text{in}:\alpha$ is obtained by applying I to the pair (α, h) , where α is the name (which must be an identifier) of the device that is to provide the input and h is the history stream. Of course, the details of exactly how this input is produced (eg, the device might wait for input if none is available) and what form it has are device dependent and are not described here. If there is no device named α , then I returns $?$.

Similarly, the function O is the output-handler. When applied to (x, h) , O returns $s2:x$ if the output was successful and $?$ otherwise. In order for the function out to be successful, its argument must be a pair $\langle \alpha, \gamma \rangle$ with α the name of the output device and γ the value to be output. Of course, in order for the output to be successful, it is necessary that there actually be a device named α , which in turn implies that α is an identifier.

In order to implement the functions β and γ , it is necessary to inhibit changes to the history stream while evaluating $\beta:f:x$ and to further inhibit accesses while evaluating $\gamma:f:x$. Once these expressions have been fully evaluated, accesses and changes to the history stream are again permitted. The new expressions $\text{GoA}(e)$ and $\text{GoC}(e)$ (where e is an arbitrary expression) permit such accesses. These expressions set the h_a and h_c flag (respectively) to *go* (after e is reduced to a value). These expressions may *not* be written by the programmer since they are intended solely for the purpose of again permitting access to the history stream.

In these rewrite rules, x , γ , and α represent arbitrary values and h represents the history. The new history is represented by h' , and only the changes to h needed to produce h' are listed in the "New History" column. Consequently, a blank entry in the "New History" column indicates that $h' = h$. Adding a new element x to the history stream h_s is denoted by $h_s \text{ AR } x$. The rules are the following:

Rewrite Rule	New History	Comments
$\beta:x \rightarrow x$ $\beta:x:y \rightarrow \text{GoC}(x:y)$ $\beta:x:y \rightarrow x:y$	$h'_c = \text{stop}$	if x is not a function if $h_c = \text{go}$ if $h_c = \text{stop}$
$\gamma:x \rightarrow x$ $\gamma:x:y \rightarrow \text{GoA}(x:y)$ $\gamma:x:y \rightarrow x:y$	$h'_a = \text{stop}$	if x is not a function if $h_a = \text{go}$ if $h_a = \text{stop}$
$\text{get}:x \rightarrow G(h)$ $\text{get}:x \rightarrow ?$		if $h_a = \text{go}$ if $h_a = \text{stop}$
$\text{GoA}(x) \rightarrow x$	$h'_a = \text{go}$	
$\text{GoC}(x) \rightarrow x$	$h'_c = \text{go}$	
$\text{in}:a \rightarrow I(a, h)$ $\text{in}:a \rightarrow ?$	$h'_s = h_s \text{ AR } <\text{'InFlag, } a>$	if $h_a = h_c = \text{go}$ if $h_a = \text{stop}$ or $h_c = \text{stop}$
$\text{InReady}:a \rightarrow \text{'T}$ $\text{InReady}:a \rightarrow \text{'F}$ $\text{InReady}:a \rightarrow ?$	$h'_s = h_s \text{ AR } <\text{'InTest, } a>$ $h'_s = h_s \text{ AR } <\text{'InTest, } a>$	if $h_a = h_c = \text{go}$ and $I(a, h') \neq ?$ if $h_a = h_c = \text{go}$ and $I(a, h') = ?$ if $h_a = \text{stop}$ or $h_c = \text{stop}$
$\text{out}:x \rightarrow O(x, h)$ $\text{out}:x \rightarrow ?$	$h'_s = h_s \text{ AR } <\text{'OutFlag, } x>$	if $h_a = h_c = \text{go}$ if $h_a = \text{stop}$ or $h_c = \text{stop}$
$\text{OutReady}:x \rightarrow \text{'T}$ $\text{OutReady}:x \rightarrow \text{'F}$ $\text{OutReady}:x \rightarrow ?$	$h'_s = h_s \text{ AR } <\text{'OutTest, } x>$ $h'_s = h_s \text{ AR } <\text{'OutTest, } x>$	if $h_a = h_c = \text{go}$ and $O(x, h') \neq ?$ if $h_a = h_c = \text{go}$ and $O(x, h') = ?$ if $h_a = \text{stop}$ or $h_c = \text{stop}$
$\text{pmark}:x \rightarrow x$ $\text{pmark}:x \rightarrow ?$	$h'_s = h_s \text{ AR } <\text{'FileFlag, } G(h)>$	if $h_a = h_c = \text{go}$ if $h_a = \text{stop}$ or $h_c = \text{stop}$
$\text{put}:x \rightarrow x$ $\text{put}:x \rightarrow ?$	$h'_s = h_s \text{ AR } <\text{'FileFlag, } x>$	if $h_a = h_c = \text{go}$ if $h_a = \text{stop}$ or $h_c = \text{stop}$

7. Appendix C: The Simplification Algorithm

This appendix describes the simplification algorithm by giving a function *Simp* that maps arbitrary expressions to an equivalent core expression. The function *Simp* is described by giving two auxiliary functions *SimpA* which maps arbitrary expressions to simple expressions and *Core* which maps simple expressions to core expressions. The definition of *Simp* is $Simp(e) = Core(SimpA(e))$.

The Function "Core"

The function *Core* maps the set of simple expressions into the set of core expressions. The function *Core* is defined as follows:

$Core(a) = a$ if a is an atom

$Core(f) = f$ if f is a primitive function name

$Core(f_{pf}) = f$ if f_{pf} is a labelled primitive function name.

SimpA sometimes needs to label a primitive function name

so that it may not be redefined. This is the only reason for the label pf .

$Core(f) = \text{err}$ if f is a non-primitive function name

$Core(\langle e_1, \dots, e_n \rangle) = \langle Core(e_1), \dots, Core(e_n) \rangle$

$Core(x:y) = Core(x):Core(y)$

$Core(f') = \text{lift}:Core(f)$

$Core([^{(n)}e_1, \dots, e_n]) = (\text{lift}:\dots(\text{lift}:(\text{lift}:\text{cons}))\dots):\langle Core(e_1), \dots, Core(e_n) \rangle$

where the superscript (n) in the argument indicates the number of primes and

there are n occurrences of the function *lift* in the result

$Core([^{(n)}e_1, \dots, e_n]) = (\text{lift}:\dots(\text{lift}:(\text{lift}:\text{econs}))\dots):\langle Core(e_1), \dots, Core(e_n) \rangle$

where the superscript (n) in the argument indicates the number of primes and

there are n occurrences of the function *lift* in the result

$Core(\sim e) = K:Core(e)$

$Core(e_1 \rightarrow ^{(n)}e_2; e_3) = (\text{lift}:\dots(\text{lift}:(\text{lift}:\text{cond}))\dots):\langle Core(e_1), Core(e_2), Core(e_3) \rangle$

where the superscript (n) in the argument indicates the number of primes and

there are n occurrences of the function lift in the result

$$\text{Core}(e_1 \rightarrow e_2) = \text{Core}(e_1 \rightarrow e_2; \text{err})$$

The Function “SimpA”

The function *SimpA* maps the set of patterns (recall that every expression is, by definition, also a pattern) into the set of simple expressions. In order to define *SimpA*, it is necessary to define the auxiliary function *SDefs*, which maps the set of patterns into the set of definition lists (which the patterns generate) of the form $d_1 \equiv D_1 \dots d_n \equiv D_n$ where $D_1, \dots, D_n$ are all simple expressions. The function *SDefs* was informally described as *Defs* in the “Patterns” subsection of Section 2. Of course, if e is merely an expression and not a pattern, then $\text{SDefs}(e) = \{ \}$.

SimpA is defined recursively by cases (on the structure of its argument) in the list below. The list (which is numbered for readability but whose order is irrelevant) is organized so that step a in each case is the definition of *SimpA*, step b is the definition of *SDefs*, step c contains some conditions, and step d contains some comments. In order to save space, if some constructs behave the same way, they are grouped together in the list. For example, case 1 contains all the “easy” cases in the definition of *SimpA*.

In order to streamline the definition of *SimpA*, it is only necessary to deal with the cases that programmers can write (eg, *SimpA* need not handle tag pairs). Furthermore, *SimpA* assumes that all expressions have been fully parenthesized and that all comments have been removed. For technical reasons, it is necessary to assign a unique positive integer to each *Rep* declaration, and *SimpA* assumes that this has been done.

1. a. $SimpA(f) = f$ if f function name
 $SimpA(a) =$ the canonical form of a if a is an atom
 $SimpA(\langle x_1, \dots, x_n \rangle) = \langle SimpA(x_1), \dots, SimpA(x_n) \rangle$
 $SimpA(x:y) = SimpA(x):SimpA(y)$
 $SimpA(f') = (SimpA(f))'$
 $SimpA([^{(n)}f_1, \dots, f_m]) = [^{(n)}SimpA(f_1), \dots, SimpA(f_m)]$ for $n \geq 0$
 $SimpA(\llbracket ^{(n)}f_1, \dots, f_m \rrbracket) = \llbracket ^{(n)}SimpA(f_1), \dots, SimpA(f_m) \rrbracket$ for $n \geq 1$
 $SimpA(\sim x) = \sim SimpA(x)$
 $SimpA(e_1 \rightarrow ^{(n)}e_2; e_3) = (SimpA(e_1) \rightarrow ^{(n)}SimpA(e_2); SimpA(e_3))$ for $n \geq 1$
 - b. $SDefs(e) = \{ \}$ for any of the above expressions
 - c. Where: the superscript (n) indicates the number of primes
 - d. Comments:
 - i. Note that none of the above combining forms is a valid pattern combining form.
 - ii. The only atoms that are not necessarily in canonical form are numbers. Each number has a canonical form (which is left unspecified in this manual).
2. a. $SimpA(P_1 \rightarrow e_2; e_3) = (SimpA(P_1) \rightarrow E_2(D_1, \dots, D_n); E_3(D_1, \dots, D_n))$
b. $SDefs(P_1 \rightarrow e_2; e_3) = \{ \}$
c. Where:
 - i. $SDefs(P_1) = \{d_1 \equiv D_1, \dots, d_n \equiv D_n\}$, and
 - ii. for $i = 2, 3$, $E_i(D_1, \dots, D_n)$ is $SimpA(e_i)$ with the simple expression D_k substituted for all occurrences of the function name d_k for $k = 1, \dots, n$.
d. Example:
$$SimpA(\llbracket f.isint, g.isnum \rrbracket \rightarrow (f + g)*g; [F, [f, g]]) =$$

$$(\llbracket isint, isnum \rrbracket \rightarrow (s1 + s2)*s2; [F, [s1, s2]])$$
3. a. $SimpA(P \rightarrow e) = SimpA(P \rightarrow e; err)$
b. $SDefs(P \rightarrow e) = SDefs(P \rightarrow e; err)$

4. a. $SimpA(P_1 \Rightarrow e_2; e_3) = SimpA(P_1 \rightarrow Abs(SimpA(e_2), l); e_3)$
b. $SDefs(P_1 \Rightarrow e_2; e_3) = \{ \}$
c. Where:
 $SDefs(P_1) = \{d_1 \equiv D_1, \dots, d_n \equiv D_n\}$,
 l is the list $((d_1, d_1), \dots, (d_n, d_n))$, and
the function Abs is described in the "Abstraction Algorithm" subsection
d. Comments:
i. Note that the "false" arm acts the same way in conditional abstraction that it does in regular conditional.
e. Example:

$$SimpA(\llbracket f.isfunc, g.isfunc \rrbracket \Rightarrow s1 \circ f \circ g; f \circ g) =$$

$$SimpA(\llbracket f.isfunc, g.isfunc \rrbracket \rightarrow \sim s1 \circ' f \circ' g; f \circ g) =$$

$$(\llbracket isfunc, isfunc \rrbracket \rightarrow \sim s1 \circ' s2 \circ' s1; s1 \circ s2)$$
Thus, $(\llbracket f.isfunc, g.isfunc \rrbracket \Rightarrow s1 \circ f \circ g; f \circ g):<tl, tlr> = (\sim s1 \circ' s1 \circ' s2):<tl, tlr> = s1 \circ tl \circ tlr$
5. a. $SimpA(P_1 \Rightarrow e_2) = SimpA(P_1 \Rightarrow e_2; err)$
b. $SDefs(P_1 \Rightarrow e_2) = SDefs(P_1 \Rightarrow e_2; err)$
6. a. $SimpA(e \text{ where } dl \text{ end}) = E(E_1, \dots, E_n)$
b. $SDefs(P_1 \rightarrow e_2; e_3) = \{ \}$
c. Where:
i. $\{(fn_1, E_1), \dots, (fn_n, E_n)\}$ is the set of substitution pairs derived from the declaration list dl as described in the "Declaration List Algorithm" subsection of this Appendix, and
ii. $E(E_1, \dots, E_n)$ is $SimpA(e)$ with the simple expression E_k substituted for all occurrences of the function name fn_k for $k = 1, \dots, n$.
d. Comments:
i. the declaration list algorithm guarantees that the expressions $E_1, \dots, E_n$ are all values.
e. Example:

$$SimpA([f, g] \text{ where } Def f \equiv s4 \text{ Def } g \equiv tlr \text{ end})$$
is equivalent to $[s4, tlr]$ and is equal to

$$[mrfix_{pf}:<args_{pf}:2:(\sim s4), args_{pf}:2:(\sim tlr)>:1, \\ mrfix_{pf}:<args_{pf}:2:(\sim s4), args_{pf}:2:(\sim tlr)>:2]$$

7. a. $SimpA(\text{let } dl \text{ in } e) = SimpA(e \text{ where } dl \text{ end})$
 b. $SDefs(\text{let } dl \text{ in } e) = SDefs(e \text{ where } dl \text{ end})$
8. a. $SimpA(fn.) = T$
 $SimpA(fn.e) = SimpA(e)$
 b. $SDefs(fn.) = \{fn \equiv id\}$
 $SDefs(fn.e) = \{fn \equiv id\}$
9. a. $SimpA(\llbracket P_1, \dots, P_n \rrbracket) = \llbracket SimpA(P_1), \dots, SimpA(P_n) \rrbracket$
 b. $SDefs(\llbracket P_1, \dots, P_n \rrbracket) = \bigcup_{i=1}^n \{d \equiv D \circ si \mid d \equiv D \in SDefs(P_i)\}$
 c. Example:
 $SimpA(\llbracket f.isint, g.isnum \rrbracket) = \llbracket isint, isnum \rrbracket$
 $SDefs(\llbracket f.isint, g.isnum \rrbracket) = \{f \equiv s1, g \equiv s2\}.$
10. a. $SimpA(x \ y \ z) = SimpA(x) \ SimpA(y) \ SimpA(z)$ for infix notation
 b. $SDefs(x \mapsto z) = \{d \equiv D \circ s1 \mid (d \equiv D) \in SDefs(x)\} \cup \{d \equiv D \circ t1 \mid (d \equiv D) \in SDefs(z)\}$
 $SDefs(x \leftarrow z) = \{d \equiv D \circ tlr \mid (d \equiv D) \in SDefs(x)\} \cup \{d \equiv D \circ r1 \mid (d \equiv D) \in SDefs(z)\}$
 $SDefs(x \circ z) = \{d \equiv D \circ SimpA(z) \mid (d \equiv D) \in SDefs(x)\} \cup SDefs(z)$
 $SDefs(x \wedge z) = SDefs(x) \cup SDefs(z)$
 $SDefs(x \ y \ z) = \{ \}$ for any other infix notation
 c. Example:
 $SimpA((f.isoolarge) \circ (g.factorial)) = isoolarge \circ factorial$
 $SDefs((f.isoolarge) \circ (g.factorial)) = \{ f \equiv id \circ factorial, g \equiv id \}$

Declaration List Algorithm

The declaration list algorithm is a subroutine for the function *SimpA*. The purpose of the declaration list algorithm is to transform declaration lists by resolving all scope issues, removing Ext and Rep declarations, and removing all (mutual) recursion. Formally, it transforms an arbitrary declaration list into a list of “substitution pairs”, each of which has the form (fn, e) for some function name fn and some simple expression e (which is always a value). The expression e that corresponds to the function name fn is suitable for substituting into simple expressions. For example, the expression $E(fn_1, fn_2) \text{ where } dl \text{ end}$ is equivalent to the expression $E(e_1, e_2)$ provided E is a simple expression and $((fn_1, e_1), (fn_2, e_2))$ is the list

produced by the declaration list algorithm.

1. Discard all **Asn** and **Sig** declarations from **dl**.
2. Remove extension definitions from **dl** by replacing
 $\text{Ext } f \equiv p_1 \rightarrow e_1; \dots; p_{i_1} \rightarrow e_{i_1} \dots \text{Ext } f \equiv p_1^n \rightarrow e_1^n; \dots; p_{i_1}^n \rightarrow e_{i_n}^n$ by
 $\text{Def } f \equiv p_1 \rightarrow e_1; \dots; p_{i_1} \rightarrow e_{i_1}; \dots p_1^n \rightarrow e_1^n; \dots; p_{i_1}^n \rightarrow e_{i_n}^n; f_{\text{old}}$ where f_{old} is the function name
“labelled” with the word “old” to indicate that this refers to the former definition. For
the rest of this algorithm, f_{old} is considered a different function name than f .
3. Replace all **Rep** declarations of the form **Rep** $t \leftrightarrow e$ by the three definitions:

Def $\text{ist} \equiv (\text{istagpair}_{pf} \rightarrow (\text{tag1}_{pf} = \sim t_n \rightarrow T_{pf}; F_{pf}); F_{pf})$

Def $\text{maket} \equiv (\gamma: E \rightarrow \text{tag}_{pf}: t_n; \text{err}_{pf})$

Def $\text{umt} \equiv (\text{istagpair}_{pf} \rightarrow (\text{tag1}_{pf} = \sim t_n \rightarrow \text{tag2}_{pf}; \text{err}_{pf}); \text{err}_{pf})$

where n is the unique positive integer assigned to this **Rep** declaration and E is the simple expression formed from $\text{SimpA}(e)$ by replacing each subexpression (of $\text{SimpA}(e)$) of the form $x:y$ by $\gamma:x.y$.

4. Move all patterns and templates from the left-hand-side of enhanced definitions to the right-hand-side as described in the “Enhanced Definitions” subsection of Section 3.
5. Replace the keyword **Def** by **Def_i**. Notice that all definitions in **dl** now start with the keyword **Def_i** for some positive integer i .
6. Simplify the right-hand-sides of definitions by applying the function SimpA . Now all definitions in **dl** have the form **Def_i** $f \equiv e$ for some positive integer i and some simple expression e .
7. If there are n definitions in **dl**, assign a unique integer between 1 and n to each function name that is given a definition in **dl**. (Note that each function name is given at most one definition or else the declaration list is illegal.) The order in which the integers is assigned is irrelevant. (Ie, the denotational meaning of the final result is independent of

which integer is assigned to which function name). The function names will henceforth be referred to as $fn_1, \dots, fn_n$.

8. a. For $i = 1, \dots, n$, let $Def_{a_i} fn_i \equiv e_i$ be the definition of fn_i . (Recall that e_i is a simple expression by step 6.) For $i = 1, \dots, n$, replace the definition $Def_{a_i} fn_i \equiv e_i$ by the equivalent (see part b) definition $Def fn_i \equiv mrfix_{pf}:<rhs_1, \dots, rhs_n>:i$ where $rhs_k = args_{pf}: (a_k + 1) : Abs(e_k, ((fn_1, sl_{pf}), \dots, (fn_n, sn_{pf})))$ for $k = 1, \dots, n$. Notice that the new right hand sides are all values that contain no mention of $fn_1, \dots, fn_n$.
- b. Informal justification of this step: Let E_i be an expression that has the property that $E_i:<fn_1, \dots, fn_n> = e_i$. It is possible to obtain such an expression by using the function Abs (defined in the "Abstraction Algorithm" subsection of this appendix). $E_i = Abs(e_i, ((fn_1, sl_{pf}), \dots, (fn_n, sn_{pf})))$ has the desired property. This means that $Def_{a_i} fn_i \equiv E_i:<fn_1, \dots, fn_n>$ is equivalent to the original definition of fn_i . This has not yet eliminated the function names from the right-hand-sides, but it has isolated them. The next order of business is to take care of the Def_{a_i} keywords. Let $rhs_i = args_{pf}: (a_i + 1) : E_i$. Thus, $def_{a_i} fn_i \equiv e_i$ is equivalent to $Def fn_i \equiv rhs_i:<fn_1, \dots, fn_n>$. Note that it is necessary to add 1 to a_i because of the extra argument $<fn_1, \dots, fn_n>$. Since the goal is to obtain an expression F_i that is equivalent to the function named fn_i (and makes no mention of $fn_1, \dots, fn_n$), just set $fn_i = F_i$ and derive some necessary conditions on F_i . If $fn_i = F_i$, then $F_i = fn_i = rhs_i:<fn_1, \dots, fn_n> = rhs_i:<F_1, \dots, F_n>$. Thus, if the expression F_i is to equal the function named by fn_i , it is necessary that $F_i = rhs_i:<F_1, \dots, F_n>$. (This is not a sufficient condition, but the purpose is to justify this step, not prove it.) This necessary condition is satisfied by setting $F_i = mrfix_{pf}:<rhs_1, \dots, rhs_n>:i$ since the rewriting proceeds as follows (ignoring the subscripts pf):

```

F_i:x
→ mrfix:<rhs_1,...,rhs_n>:i:x
→ sel:<i,<rhs_1,...,rhs_n>>:(α:(mrfix:<rhs_1,...,rhs_n>):<1,...,n>):x
→ rhs_i:(α:(mrfix:<rhs_1,...,rhs_n>):<1,...,n>):x
→ rhs_i:<F_1,...,F_n>:x

```

9. If dl is initiated by either of the keywords `export` or `hide`, replace the initial keyword by `begin` after taking the following steps for all hidden function names fn_i (where a function

name is hidden if it is named in the **hide** clause or not named in the **export** clause).

- a. Unlabel all occurrences (in dl) of fn_i that were labelled with the word "old".
- b. Discard the definition of the fn_i .

10. If dl is contained in another declaration list, merge dl into the containing declaration list by discarding the **begin...end** pair of keywords from dl and then go back to step 1 and start the process over with any innermost declaration list. If dl is the outermost declaration list (thus, the only declaration list that remains), terminate the algorithm and return the list $((f_1, E_1), \dots, (f_m, E_m))$ where dl has the form $\text{Def } f_1 \equiv e_1 \dots \text{Def } f_m \equiv e_m$ where E_i is formed from e_i by unlabeled all function names in e_i that were labelled with the subscript "old".

For example, the declaration list algorithm when applied to the declaration list

export(FromCr, ToScreen)

{ the function FromCr never terminates
and transfers all input from the card reader ('CR)
to the screen ('screen) }

Def FromCr $\equiv$ ToScreen $\circ$ in $\circ$ $\sim$ 'CR

Def ToScreen $\equiv$ FromCr $\circ$ OutScreen

Def OutScreen $\equiv$ out $\circ$ [$\sim$ 'screen, id]

end

produces the substitution list:

$((\text{FromCr}, \text{mrfix}_{pf}:L:1), (\text{ToScreen}, \text{mrfix}_{pf}:L:2))$

where L stands for the sequence (assuming that $\circ$ refers to the original primitive)

$\langle \text{args}_{pf}:2:((s2_{pf} \circ \sim \text{in}) \circ \sim \sim \text{'CR}), \text{args}_{pf}:2:(s1_{pf} \circ \sim s3_{pf}), \text{args}_{pf}:2:\sim(\text{out} \circ [\sim \text{'screen}, \text{id}]) \rangle$

Abstraction Algorithm

The abstraction algorithm is given by defining a function *Abs* that maps a simple expression and a list l of pairs to a simple expression. The list l must have the form $((fn_1, e_1), \dots, (fn_n, e_n))$ where $fn_1, \dots, fn_n$ are function names and $e_1, \dots, e_n$ are expressions. Informally, the function *Abs* acts by abstracting the function names in the list out of the original simple expression. More formally, the function *Abs* has the following property for all values $e_1, \dots, e_n, x$

$$Abs(E(fn_1, \dots, fn_n), ((fn_1, e_1), \dots, (fn_n, e_n))) : x = E(e_1 : x, \dots, e_n : x)$$

Some optional steps (resulting in a smaller simple expression) are given provided the appropriate part of the expression is *l*-safe. An *l*-safe expression is one that has no occurrence of any function names in *l* and is a value regardless of any possible definitions or redefinitions of any function name, including primitive function names (that have not been labelled with the subscript pf). For example, an expression without : and without any function names from *l* is *l*-safe.

1. (Optional) If *x* is *l*-safe, $Abs(x, l) = K_{pf} : x$.
2. $Abs(fn, l) = e$ if $(fn, e) \in l$.
3. $Abs(fn, l) = K_{pf} : fn$ if (fn, e) does not occur in *l*.
4. $Abs(a, l) = K_{pf} : a$ if *a* is an atom.
5. $Abs(?, l) = err_{pf}$.
6. (Optional) If *x* is *l*-safe, $Abs(x : y, l) = x \circ_{pf} Abs(y, l)$.
7. $Abs(x : y, l) = Abs(x, l) \text{ apply}_{pf}' Abs(y, l)$.
8. $Abs(\langle x_1, \dots, x_m \rangle, l) = [Abs(x_1, l), \dots, Abs(x_m, l)]$.
9. (Optional) If *y* is *l*-safe, $Abs(x y^{(n)} z, l) = Abs(x, l) y^{(n+1)} Abs(z, l)$ for infix notation *y*.
10. $Abs(x y z, l) = Abs(y, l) \text{ apply}_{pf}' [Abs(x, l), Abs(z, l)]$ for any infix notation *y*.
11. $Abs([^{(n)}x_1, \dots, x_m], l) = [^{(n+1)}Abs(x_1, l), \dots, Abs(x_m, l)]$.
12. $Abs(\llbracket ^{(n)}x_1, \dots, x_m \rrbracket, l) = \llbracket ^{(n+1)}Abs(x_1, l), \dots, Abs(x_m, l) \rrbracket$.
13. $Abs(x_1 \rightarrow^{(n)} x_2; x_3, l) = (Abs(x_1, l) \rightarrow^{(n+1)} Abs(x_2, l); Abs(x_3, l))$.
14. $Abs(x_1 \rightarrow x_2, l) = (Abs(x_1, l) \rightarrow' Abs(x_2, l); \sim err_{pf})$.
15. $Abs(x', l) = lift_{pf} \circ_{pf} Abs(x, l)$.
16. $Abs(\sim x, l) = K_{pf} \circ_{pf} Abs(x, l)$.

8. Appendix D: The Primitive Abstract Type store

Store is a primitive abstract type, which provides the following primitive functions for manipulating values of type *store*: *isstore*, *mkempty_st*, *isempty_st*, *find*, *fetch*, *save*, *assign*. It is a *primitive* type, because its operations are built into the language. It is an *abstract* type, because the details of the implementation are not visible to the programmer; all that is known about these primitives is that they satisfy a particular set of signature and assertion declarations (some of which are given below).

However, to suggest a possible implementation for *store* and to provide a larger example of an abstract type definition, a representation declaration and the associated function definitions are also given, as if it were a programmer defined type.

Export (*isstore*, *mkempty_st*, *isempty_st*, *find*, *fetch*, *save*, *assign*)

Rep 'store $\Leftarrow$ seqof: [[*isobj*, *isval*]]

{ stores are tagged sequences of pairs <name, content> }

{ Signatures of the exported functions: }

Sig *isstore* :: *isval* $\Rightarrow$ *isbool* { implicit Sig from Rep }

Sig *mkempty_st* :: *isval* $\Rightarrow$ *isstore*

Sig *isempty_st* :: *isstore* $\Rightarrow$ *isbool*

Sig *find* :: [[*isobj*, *isstore*]] $\Rightarrow$ *isval*

Sig *fetch* :: *isobj* $\Rightarrow$ *isfunc*

Sig *save* :: [[[[*isobj*, *isval*]], *isstore*]] $\Rightarrow$ *isstore*

Sig *assign* :: [[*isobj*, *isval*]] $\Rightarrow$ *isfunc*

{ Sample assertions about the exported functions: }

Asn (*name* $\in$ *isobj*, *val* $\in$ *isval*, *store* $\in$ *isstore*)

find:<*name*, save:<<*name*, *val*>, *store*>> $\equiv$ *val*

Asn (*name* $\in$ *isobj*, *val* $\in$ *isval*)

find:<*name*, mkempty_st:*val*> $\equiv$ '#

Asn (*name* $\in$ *isobj*, *store* $\in$ *isstore*)

save:<<*name*, find:<*name*, *store*>>, *store*> $\equiv$ *store*

Asn *isstore* $\circ$ *mkempty_st* $\equiv$ T

```

Asn (name ∈ isobj, val ∈ isval, store ∈ isstore)
  isstore ◦ save ◦ [[~name, ~val], ~store] ≡ T
Asn isempty_st ◦ mkempty_st ≡ T
Asn (name ∈ isobj, val ∈ isval, store ∈ isstore)
  isempty_st ◦ save ◦ [[~name, ~val], ~store] ≡ F
Asn (name ∈ isobj, store ∈ isstore)
  fetch:name:store ≡ find:<name,store>
Asn (name ∈ isobj, val ∈ isval, store ∈ isstore)
  assign:<name, val>:store ≡ save:<<name, val>, store>
{ Signatures of local functions: }
  Sig mkstore :: seqof:[isobj, isval] ⇒ isstore { implicit Sig from Rep }
  Sig umstore :: isstore ⇒ seqof:[isobj, isval] { implicit Sig from Rep }
  Sig s1st :: isstore ∧ not◦isempty_st ⇒ [isobj, isval] { analog of s1 for stores }
  Sig t1st :: isstore ∧ not◦isempty_st ⇒ isstore { analog of t1 for stores }
  Sig alst :: [[isobj, isval], isstore] ⇒ isstore { analog of al for stores }
  Sig purge :: [isobj, isstore] ⇒ isstore { deletes pairs with name = obj from store }
{ Possible definitions for local functions: }
  Def s1st ◦ isstore ≡ s1 ◦ umstore
  Def t1st ◦ isstore ≡ mkstore ◦ t1 ◦ umstore
  Def alst ◦ [[c.[isobj, isval], s.isstore]] ≡
    mkstore ◦ al ◦ [c, umstore ◦ s]
  Def purge ◦ [nm.isobj, st.isstore] ≡
    isempty_st ◦ st → st;
    ([name., content.] ⊢ (rst.) ◦ mkstore) ◦ umstore ◦ st →
      nm = name → purge ◦ [nm, rst];
      alst ◦ [[name, content], purge ◦ [nm, rst]]
{ Possible definitions for exported functions: }
  { isstore is implicitly defined by the Rep declaration }
  Def mkempty_st ◦ isval ≡ mkstore ◦ []
  Def isempty_st ◦ isstore ≡ isnull ◦ umstore → T; F
  Def find ◦ [nm.isobj, st.isstore] ≡ isempty_st ◦ st → ~\#;
    ([name., content.] ⊢ (rst.) ◦ mkstore) ◦ umstore ◦ st → nm = name → content;
    find ◦ [nm, rst]

```

```

Def fetch: nm.isobj  $\equiv$  isstore  $\rightarrow$  find  $\circ$  [ $\sim$ nm, id]
Def save  $\circ$  [[[nm.isobj, vl.isval], st.isstore]]  $\equiv$ 
  alst  $\circ$  [[nm, vl], purge  $\circ$  [nm, st]]
Def assign: <nm.isobj, vl.isval>  $\equiv$  isstore  $\rightarrow$  save  $\circ$  [[ $\sim$ nm,  $\sim$ vl], id]
End

```

