



# SequenceL 0.3

## Informal Specification

## Denoting code

In this document, SequenceL will appear in typewriter font. When SequenceL code appears in the context of running English text, it will be enclosed in double-angle-brackets, or "chevrons" («»). Code will sometimes be set off by blank lines, and indented or placed in a table. In this case code may appear without chevrons.

## Scalars

Numbers are written in SequenceL in the usual way as decimal numerals. For example, «3», «0.45», and «-6.321» are numbers. A number cannot begin with a decimal point, so, for example, «.45» is not a number.

All numbers in SequenceL are of type «number». Integers are of type «integer» *and also* of type «number». There is no difference, for example, between «5» and «5.0». Both are numbers; both are integers; and they are interchangeable in every way.

An **atom** is a nonempty string of letters, digits, and underscores, containing at least one letter. For example, «foo» and «\_3\_Foo\_Bar\_42» are atoms.

A **scalar** is a number or an atom.

## Arithmetic Operators

The built in **arithmetic operators** are «+», binary and unary «-», «\*», «/» «^», «mod», «floor», «sin», «cos», «tan», «arcsin», «arccos», «arctan», and «ln». «π» and «e» are built in constants.

«+», binary and unary «-», and «\*» behave in the usual ways. For example,

$$3 + 7 = 10 \qquad 4 * -2.5 + 1 = -9$$

«/» is division of real numbers. There is no integer division operation. For example,

$$10/5 = 2 \qquad 5/2 = 2.5$$
$$-1/3 = -0.3333333333$$

«ln» returns the natural logarithm of its argument. For example

$$\ln(e) = 1 \qquad \ln(1) = 0$$
$$\ln(34.7) = 3.54673968695281$$

The trigonometric functions are in radians. For example,

$$\begin{aligned}\sin(0) &= 0 & \sin(\pi) &= 0 \\ \cos(34.7) &= -0.989866769\end{aligned}$$

«^» represents exponentiation. For example,

$$\begin{aligned}5^2 &= 25 & 10^{-2} &= 0.01 \\ 3.5^{6.2} &= 2361.684477\end{aligned}$$

«x^y» may also be written as «x<sup>y</sup>». So, for example,

$$\begin{aligned}5^2 &= 25 & 10^{-2} &= 0.01 \\ 3.5^{6.2} &= 2361.684477\end{aligned}$$

For any integer  $i$  and positive integer  $j$ , « $i \bmod j$ » denotes the remainder when  $i$  is divided by  $j$ . More precisely, it denotes the absolute value of the difference between  $i$  and the largest integer multiple of  $j$  not exceeding  $i$ . For example,

$$\begin{aligned}34 \bmod 10 &= 4. & -4 \bmod 10 &= 6 \\ -1 \bmod 3 &= 2 & 0 \bmod 3 &= 0.\end{aligned}$$

For any number  $x$ , «`floor(x)`» returns the greatest integer not exceeding  $x$ . For example,

$$\begin{aligned}\text{floor}(5.3) &= 5 & \text{floor}(6) &= 6 \\ \text{floor}(-7.1) &= -8\end{aligned}$$

Arithmetic expressions with arguments outside their appropriate domains result in a value of «nil». For example, the terms below all return «nil». Certain computations can be performed on «nil» to yield meaningful results; and a value of nil does *not* generate a runtime error :

$$\begin{aligned}3/0 & & \ln(-3) \\ \arcsin(4) & & \tan(\pi/2) \\ 5 \bmod -3 & & 4.3 \bmod 2 \\ \text{nil} + 3 & & \sin(\text{nil})\end{aligned}$$

## Lists

A *list* can be written as zero or more terms, separated by commas and enclosed in square brackets. The following are lists:

```
[1, 2, 3]          [1]
[]                [1, foo, [2, bar]]
```

The **concatenation operator** `<<+>` is an infix function which operates on lists, and returns the list obtained by appending the second argument to the end of the first. For example,

```
[1, 2, 3] ++ [4, 5] = [1, 2, 3, 4, 5]
[1] + [] ++ [hello, [world]] = [1, hello, [world]]
```

`<<nil>` acts as both a left and right identity under `<<+>`. That is, for any list  $L$ , `<<L ++ nil>` and `<<nil ++ L>` both reduce to  $L$ . Also, `<<[nil]>` reduces to `<<nil>`.

If  $L$  is a list and  $n$  is an integer, then `<<Ln>` returns the  $n^{\text{th}}$  member of  $L$ , if there is one. If there is no  $n^{\text{th}}$  member of  $L$ , then `<<Ln>` returns `<<nil>`. Note a list begins with its first member (as opposed to its  $0^{\text{th}}$  member). For example,

```
[10, 20, 30]2 = 20
[1, 2, 3]4 = nil
```

If  $A$  and  $B$  are lists, `<<A \ B>`, read **A takeaway B** returns the list obtained by removing every occurrence of every member of  $B$  from  $A$ . For example,

```
[1, 2, 3, 2] \ [2, 5] = [1, 3]
```

If  $A$  is a list, `<<size(A)>` returns the number of members of  $A$ , counting multiplicity. For example,

```
size([]) = 0           size([a, a, b]) = 3
```

If  $A$  is a list, and  $x$  is a member of  $A$ , the first occurrence of  $x$  in  $A$  is called the **prime occurrence**. `<<remdups(A)>` returns a list obtained from  $A$  by removing all except for the prime occurrences of each of its members. For example,

```
remdups([5, 7, 5, 8, 2, 2]) = [5, 7, 8, 2]
```

## Type declarations

If  $S$  and  $T$  are terms, a statement of the form

$$S :: T$$

is called a **type declaration**. It says that  $S$  is a term of type  $T$ . The left and right hand sides of a type declaration may contain **term variables**, which appear in italics. A type declaration containing term variables is called a **type declaration scheme**, and is considered shorthand for the collection of all type declarations which can be obtained from it by replacing every type variable with a term of appropriate type. For example, the scheme

$$\textit{integer} + \textit{integer} :: \textit{integer}$$

declares that for any two terms  $m$  and  $n$  of type  $\langle\textit{integer}\rangle$ ,  $\langle m+n \rangle$  is a term of type  $\langle\textit{integer}\rangle$ . All of the built in operators of SequenceL have type declarations associated with them, so the programmer would not, for example, have to enter the above declaration.

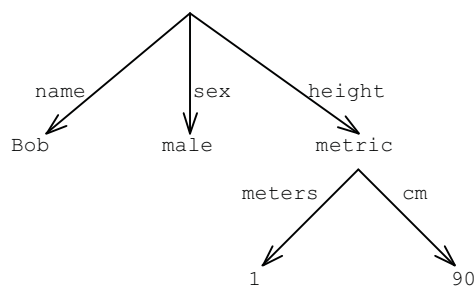
Most user defined SequenceL functions do not need type declarations. Type declarations are needed primarily for **constructors** for user defined types – i.e., terms which are of a given type as part of the definition of that type. This will be illustrated in the following section on structures.

## Structures

SequenceL **structures** are similar to records in Pascal, structs in C, or maps in the C++ standard template library. "Features" and "structures" are defined inductively as follows: A **feature** consists of an atom or integer, followed by a colon, followed by a scalar or structure. A **structure** is one or more features, called the **features of  $s$** , which are separated by commas and enclosed in parentheses if there is more than one. For example, the following are structures:

$(\textit{foo}:1, \textit{bar}:2)$	$\textit{foo}:\textit{bar}$
$(1:4, 2:4)$	$(\textit{bar}:3, 2:\textit{foo}(1:10, 2:20))$

A structure  $s$  can be visualized as a tree. For example, the structure  $\langle(\textit{name}:\textit{Bob}, \textit{sex}:\textit{male}, \textit{height}:\textit{metric}(\textit{meters}:1, \textit{cm}:90))\rangle$  can be visualized as follows:



A structure is determined by its features, paying respect to multiplicity of the features but not their order. That is, a structure may be written with its features in any order without affecting the identity of the structure. For example,  $\langle\langle 1:10, 2:20 \rangle\rangle$  is the same structure as  $\langle\langle 2:20, 1:10 \rangle\rangle$ , but not the same structure as  $\langle\langle 1:10, 1:10, 2:20 \rangle\rangle$ . The equivalence between text representations of structures can be applied recursively, so, for example,  $\langle\langle 21:(foo:1, bar:2) \rangle\rangle$  and  $\langle\langle 21:(bar:2, foo:1) \rangle\rangle$  are the same structure.

Structures may appear as terms in a SequenceL program only if an appropriate type declaration has been given. For example, if a program contains the following type declarations

```
male:: gender
female:: gender

(name:atom, weight: integer, sex: gender):: person
```

then the structure  $\langle\langle name:Bob, sex:male, height: metric(meters:1, cm: 90) \rangle\rangle$  may appear in the program as a term of type  $\langle person \rangle$ .

If  $s$  is a structure and  $\langle x:y \rangle$  is a feature of  $s$ , then  $\langle s:x \rangle$  returns  $y$ . For example,  $\langle\langle 1:10, 2:20 \rangle .2 \rangle$  returns 20.

## Predicates

The atoms  $\langle true \rangle$  and  $\langle false \rangle$  are of type  $\langle Boolean \rangle$ . An operator, such as  $\langle \langle \rangle \rangle$  or  $\langle = \rangle$  which returns a Boolean value is called a **predicate**. A term whose principal operator is a predicate is an **atomic statement**. For example,  $\langle 5 = 0 \rangle$  is an atomic statement. Non-atomic statements may be built from atomic statements using connectives and quantifiers, as discussed later. A statement which returns  $\langle true \rangle$  is called a **true statement**, and one which returns  $\langle false \rangle$  is called a **false statement** (naturally). If  $\langle nil \rangle$  appears as an argument of a predicate, then the resulting statement is always false (even  $\langle nil=nil \rangle$  is false).

For terms  $x$  and  $y$ , the atomic statement  $\langle x = y \rangle$  is true if  $x$  and  $y$  both have non- $\langle nil \rangle$  values and these values are the same. For example,  $\langle [5, [1,2]] = [5, [1,2]] \rangle$  is true, and  $\langle [5, [1,3]] = [21, [1,2]] \rangle$  is false.  $\langle 5 = 1/0 \rangle$  and  $\langle 1/0 = 1/0 \rangle$  are both false.

The **arithmetic comparisons**,  $\langle \langle \rangle \rangle$ ,  $\langle \rangle \rangle$ ,  $\langle \leq \rangle$ , and  $\langle \geq \rangle$ , act on numbers in the usual way. For example,  $\langle 1 \leq 3 \rangle$  is true and  $\langle 1 < 1 \rangle$  is false. Inequalities which "face the same way", i.e.  $\langle \langle \rangle \rangle$  and  $\langle \leq \rangle$  or  $\langle \rangle \rangle$  and  $\langle \geq \rangle$ , can be chained together to form expressions such as  $\langle 1 \leq 3 < 10 < 21 \rangle$ , which is a true statement.

If  $A$  and  $B$  are lists,  $\langle A \subset B \rangle$  is true if every member of  $A$  is a member of  $B$ . For example, the following is true:

$$[1, 2, 3, 2] \subset [5, 4, 3, 2, 1]$$

and the following is false:

$$[1, 6] \subset [5, 4, 3, 2, 1]$$

If  $A$  and  $B$  are lists,  $\langle\langle A =_{\text{set}} B \rangle\rangle$  is true if a  $\langle\langle A \subset B \rangle\rangle$  is true and  $\langle\langle B \subset A \rangle\rangle$  is true. For example,

$$[1, 2, 3, 2] =_{\text{set}} [3, 2, 1]$$

is true. If  $A$  and  $B$  are lists,  $\langle\langle A =_{\text{bag}} B \rangle\rangle$  is true if  $A$  is a permutation of  $B$  – that is, for every  $x$ , the number of occurrences of  $x$  in  $A$  is the number of occurrences of  $x$  in  $B$ . For example, the following is true:

$$[1, 2, 3, 2] =_{\text{bag}} [3, 2, 2, 1]$$

and the following is false:

$$[1, 2, 3, 2] =_{\text{bag}} [3, 2, 1]$$

## Boolean connectives and quantifiers

$\langle\langle \& \rangle\rangle$ ,  $\langle\langle \text{or} \rangle\rangle$ , and  $\langle\langle \text{not} \rangle\rangle$  are **Boolean connectives**.  $\langle\langle \& \rangle\rangle$  and  $\langle\langle \text{or} \rangle\rangle$  are infixes, while  $\langle\langle \text{not} \rangle\rangle$  is a prefix which requires parentheses around its argument. If  $p$  and  $q$  are Boolean terms, then the following hold, where  $s := t$  means that the value of  $t$ , if any, is the value of  $s$ .

$\text{true} \ \& \ p := p$	$p \ \& \ \text{true} := p$
$\text{false} \ \& \ p := \text{false}$	$p \ \& \ \text{false} := \text{false}$
$\text{false} \ \text{or} \ p := p$	$p \ \text{or} \ \text{false} := p$
$p \ \text{or} \ \text{true} := \text{true}$	$\text{true} \ \text{or} \ p := \text{true}$
$\text{not}(\text{true}) := \text{false}$	$\text{not}(\text{false}) := \text{true}$

For terms  $s$  and  $t$ ,  $\langle\langle \text{not}(s = t) \rangle\rangle$  may be written as  $\langle\langle s \neq t \rangle\rangle$ , and  $\langle\langle \text{not}(s \in t) \rangle\rangle$  may be written as  $\langle\langle s \notin t \rangle\rangle$ .

The following are examples of true statements:

$1=1 \ \& \ 2<5$	$\text{not}(5 \notin [3, 4])$
$1+1=2 \ \text{or} \ 1/0=3$	

The **quantifiers**  $\langle\langle \text{some} \rangle\rangle$ ,  $\langle\langle \text{all} \rangle\rangle$ , and  $\langle\langle \text{none} \rangle\rangle$  are unary prefix functions which act on lists of Boolean values. If  $A$  is a set or list of Boolean values,

- $\langle\langle \text{some}(A) \rangle\rangle$  is true if and only if at least one member of  $A$  is true.
- $\langle\langle \text{all}(A) \rangle\rangle$  is true if and only if either  $A$  is empty, or every member of  $A$  is true.
- $\langle\langle \text{none}(A) \rangle\rangle$  is true if and only if no member of  $A$  is true.

For example, the following are true:

```

none([false ])          all([])
some([1=0, 5/2 = 2.5])  none([4 ≤ 3, 5=0])

```

and the following are false:

```

some([false])          all([true, true, false])
none([4 ≤ 3, 5=2+3])  some([1=0, 5/2 = 1/0 ])

```

## Selection and distribution

The **selection operator**,  $\langle | \rangle$ , is an associative, commutative binary infix, which may appear between terms of any type.

Two or more terms separated by  $\langle | \rangle$  constitute an **indefinite term**. The following terms are indefinite:

```

3 | 3 | 4                [1] | {2, 3}
1-4 | 5 | 6 + 7

```

$\langle | \rangle$  "swallows  $\langle \text{nil} \rangle$ ", in the sense that for any  $x$ ,

```
x | nil := x
```

and, in particular,

```
nil | nil := nil
```

When a function is applied to a sequence, the operator is **distributed** across the selections, in much the same way that multiplications are distributed across additions in arithmetic. So, Just as  $10 * (a + b + c) = 10*a + 10*b + 10*c$ , and  $(10 + 20) (a+ b) = 10*a + 10*b + 20*a + 20*b$ , we have

```

1 + (3 | 3 | 4)
⇒ 1+3 | 1+3 | 1+4
⇒ 4 | 4 | 5

```

and

```

(1 | 4) + (5 | 6)
⇒ 1+(5 | 6) | 4 + (6 | 6)
⇒ 1+5 | 1+6 | 4+5 | 4+6
⇒ 6 | 7 | 9 | 10

```

Distributions may be performed in any order. So in the previous example, we could as well have written

```
(1 | 4) + (5 | 6)
```

```

⇒ (1 | 4)+5 | (1 | 4)+6
⇒ 1+5 | 1+6 | 4+5 | 4+6
⇒ 6 | 7 | 9 | 10

```

Distribution occurs for functions of any arity, including unary functions. So, for example,

```

size([1,2] | {3,4,5})
⇒ size([1,3]) | size({3,4,5})
⇒ 2 | 12

```

and if «f», is, for example, a ternary prefix function, then

```

f(a, b | c, d | e)
⇒ f(a, b, d | e) | f(a, c, d | e)
⇒ f(a,b,d) | f(a,b,e) | f(a,c,d) | f(a,d,e)

```

In general, distribution results in a process which is similar to taking a cross product. Prolog programmers will find this process reminiscent of Prolog's backtracking mechanism.

## Indefinite operators and «is\_a»

An operator which may return an indefinite term, even when its arguments are not indefinite, is called an **indefinite operator**. The selection operator « | » is an example of an indefinite operator. Operators which are not definite are called **functions**.

The **member operator** is an indefinite operator which acts on lists:

```

Member([X1, ..., Xm]) := X1 | ... | Xm           Member([]) := nil

```

If C is a term containing no indefinite operators, then

```

v is_a C := v=C

```

Moreover,

```

v is_a C | D := v is_a C or v is_a D

```

So, for example,

```

foo is_a Member({foo, bar})
⇒ foo is_a (foo | bar)
⇒ foo is_a foo or foo is_a bar
⇒ foo = foo or foo = bar
⇒ true or false
⇒ true

```

If a structure  $s$  has features of the form  $\langle\langle x: y_1 \rangle\rangle, \dots, \langle\langle x: y_n \rangle\rangle$ , then the dot operator acts indefinitely on  $s$ , and  $d \langle\langle s.x \rangle\rangle$  returns  $\langle\langle y_1 | \dots | y_n \rangle\rangle$ .

The **membership predicate** is written  $\langle\langle in \rangle\rangle$ . If  $A$  is a list, then

$$\langle\langle E \text{ in } A \rangle\rangle := E \text{ is\_a Member}(A)$$

[NOTE: I have some confusion here. First, sequence is not defined in the document. It is referenced 3 times including the `is_a` above. It seems we have three operators that are similar – the `is_a`, `in`, and `member`. Can we reduce this to a single operator (maybe the `in` operator) that operates on lists in the following way:

$x$  in  $L$  (where  $L$  is a list or a type like integer?) gives true/false when  $x$  is a ground term and does the indefinite thing when  $x$  is not ground.

NOTE: I tried to eliminate references to sequences, and redefined 'in' as a shorthand for `is_a Member`

## Variables and conditions

A SequenceL **variable declaration** is of the form

$$\text{var } v_1, \dots, v_n :: \textit{type}$$

when  $v_1, \dots, v_n$  are identifiers and *type* is a structure indicating the type of the variables declared. For example, the following are variable declarations:

$$\begin{array}{ll} \text{var } m, i, n :: \textit{integer} & \text{var } x, y :: \textit{number} \\ \\ \text{var } s :: \textit{list} & \end{array}$$

Declared variables are of file scope, and appear in italics in all their occurrences, including the occurrence when they are declared. Also, if  $v$  is a declared variable of a given type, then automatically so is  $v'$ ,  $v''$ ,  $v'''$ , etc. – that is,  $v$  followed by any number of primes.

A **condition** is a statement containing one or more variables. For example, given the above declarations, the following are conditions:

$$\begin{array}{ll} 1 < i < 10 & 3 < x \\ \\ n \text{ is\_a Member}(s) \ \& \ x > 5 & \end{array}$$

If  $\{v_1, \dots, v_n\}$  are the variables occurring in a condition  $C$ , then  $C$  is said to be a **condition on**  $\{v_1, \dots, v_n\}$ . For example,  $\langle 1 \leq m \leq n \rangle$  is a condition on  $\{m, n\}$ . If  $C$  is a condition on  $\{v_1, \dots, v_n\}$ , a **solution of  $C$**  is a set of instantiations  $\{v_1=c_1, \dots, v_n=c_n\}$  which make  $C$  true. For example,  $\langle m=1, n=2 \rangle$  is a solution of  $\langle 1 \leq m \leq n \rangle$ .

«when», «else», and «if»

«when» and «else» are the **branching operators**. If  $E$  and  $F$  are terms, and  $v$  is a value, then

$$\begin{array}{ll} E \text{ when true} := E & E \text{ when false} := \text{nil} \\ v \text{ else } F := v & \text{nil else } F := F \end{array}$$

For example,

$$\begin{array}{l} (5+2 \text{ when } 1 < 5) = 7 \\ (5*2 \text{ when } 1 > 5) = \text{nil} \\ (4 \text{ when } 2=2 \text{ else } 14) = 4 \\ (4 \text{ when } 2=1 \text{ else } 14) = 14 \end{array}$$

If  $T$  is a term, possibly containing one or more of the variables  $v_1, \dots, v_n$ , then  $T \setminus \{v_1=c_1, \dots, v_n=c_n\}$  is the term obtained by replacing every occurrence of  $v_k$  with  $c_k$  for  $1 \leq k \leq n$ .

If  $T$  is term,  $C$  is a condition, and  $S_1, S_2, \dots$  are the solutions of  $C$ , then « $T$  when  $C$ » returns « $T \setminus S_1 \mid T \setminus S_2 \dots$ ». If  $C$  has no solution, then « $T$  when  $C$ » «returns nil». For example,

$$\begin{array}{l} x + 1 \text{ when } x = 2 \Rightarrow 3 \\ x + 5 \text{ when } x \text{ in } [1, 2, 3] \Rightarrow 6 \mid 7 \mid 8 \\ s \text{ when } [8, 9, 10] = s \text{ ++ } [10] \Rightarrow [8, 9] \\ s + [n, 5] \text{ when } [8, 9, 10] = s + [n] + s' \\ \Rightarrow [8, 5] \mid [8, 9, 5] \mid [8, 9, 10, 5] \end{array}$$

« $T$  when  $C$ » and « $T$  when  $C$  else  $B$ » may be written, respectively, as « $T$  if  $C$ » and « $T$  if  $C$  else  $B$ ».

## Ellipses

If  $x$  and  $y$  are terms, an **equalizer from  $x$  to  $y$**  is an ordered pair  $(T_1, T_2)$  of terms such that  $y$  can be obtained from  $x$  by replacing all occurrences of  $T_1$  with  $T_2$ . For example,  $(\langle\langle 5 \rangle\rangle, \langle\langle 7 \rangle\rangle)$  is an equalizer from  $\langle\langle 5+10 \rangle\rangle$  to  $\langle\langle 7+10 \rangle\rangle$ . Note every pair of terms has an equalizer, since  $(x, y)$  is always an equalizer from  $x$  to  $y$ . If there is no proper subterm  $S_1$  of  $T_1$  such that  $(S_1, S_2)$  is an equalizer from  $x$  to  $y$  for some  $S_2$ , then  $(T_1, T_2)$  is said to be the **minimal equalizer from  $x$  to  $y$** . For example,  $(\langle\langle 5 \rangle\rangle, \langle\langle 7 \rangle\rangle)$  is the minimal equalizer from  $\langle\langle 5+10+20 \rangle\rangle$  to  $\langle\langle 7+10+20 \rangle\rangle$ --  $(\langle\langle 5+10 \rangle\rangle, \langle\langle 7+10 \rangle\rangle)$  is another equalizer in this case, but it is not minimal.

If the minimal equalizer from  $x$  to  $y$  is  $(\langle\langle 1 \rangle\rangle, n)$  for some nonnegative integer  $n$ , then  $\langle\langle [x, \dots, y] \rangle\rangle$  denotes the set whose  $k^{\text{th}}$  member, for all  $1 \leq k \leq n$ , is obtained from  $x$  by replacing all occurrences of  $\langle\langle 1 \rangle\rangle$  with  $k$ . From this definition, it follows that if  $n > 1$  the  $n^{\text{th}}$  member of  $[x, \dots, y]$  will be equal to  $y$ . If  $n$  is negative then  $\langle\langle [x, \dots, y] \rangle\rangle$  returns  $\langle\langle [] \rangle\rangle$ . For example,

$$\begin{aligned} [1, \dots, 4] &= [1, 2, 3, 4] & [2^1, \dots, 2^4] &= [2, 4, 8, 16] \\ [1, \dots, 2+2] &= [1, 2, 3, 4] & [1, \dots, -5] &= [] \end{aligned}$$

Where the first three examples have minimal equalizers (after reduction) of  $(1, 4)$  and the fourth has equalizer  $(1, -5)$ . If the minimal equalizer from  $x$  to  $y$  is  $(m, n_1)$  for integers  $m$  and  $n_1$ , the minimal equalizer from  $x$  to  $z$  is  $(m, n_2)$  for some integer  $n_2$ , then  $\langle\langle [x, y, \dots, z] \rangle\rangle$  denotes the list whose  $k^{\text{th}}$  member is obtained from  $x$  by replacing all occurrences of  $m$  with  $\langle\langle \{m+(n_1-m) \times (k-1)\} \rangle\rangle$

for  $k = 1 \dots \frac{n_2 - m}{n_1 - m} + 1$ . For example, if  $\langle\langle f \rangle\rangle$  is a unary prefix function on integers,

$$[2, 4, \dots, 10] = [2, 4, 6, 8, 10]$$

where  $x=2, y=4, z=10, m=2, n_1=4, n_2=10$

$$[f(1), f(3), \dots, f(9)] = [f(1), f(3), f(5), f(7), f(9)]$$

where  $x=f(1), y=f(3), z=f(9), m=1, n_1=3, n_2=9$

$$[f(3), f(1), \dots, f(-5)] = [f(3), f(1), f(-1), f(-3), f(-5)]$$

where  $x=f(3), y=f(1), z=f(-5), m=3, n_1=1, n_2=-5$

Ellipses may be used similarly with any infix function or predicate. So for example,

$$1 + \dots + 5 = 15 \qquad 2 < 4 < \dots < 100 = \text{true}$$

$$1 * 3 * \dots * 9 = 1 * 3 * 5 * 7 * 9$$

If  $x$  contains  $\langle\langle 1 \rangle\rangle$  as a subterm, then define  $x^n$  to be the term obtained by replacing every occurrence of  $\langle\langle 1 \rangle\rangle$  in  $x$  with  $\langle\langle n \rangle\rangle$ . If  $x^n$  returns nil for all  $n > K$ , then  $\langle\langle [x, \dots] \rangle\rangle$  returns the list whose  $n^{\text{th}}$  member is  $x^n$  for all  $n \leq K$ . For example,

$$[[2, 4, 6]_1 + [10, 20, 30]_1, \dots] = [12, 24, 36]$$

« $[x, y, \dots]$ » is defined in terms of a minimal equalizer, similarly to above. So, for example,

$$\begin{aligned} & [[10, 20, 30, 40, 50, 60]_2, [[10, 20, 30, 40, 50, 60]_4, \dots] \\ &= [20, 40, 60] \end{aligned}$$

## Promotion and extension

**Promotion** occurs when an operation is performed on respective members of one or more lists, to obtain a list of the respective results. In Haskell, this would be done with 'map' for unary operators, and 'zipwith' for binary operators. In SequenceL, an operator of any arity may be automatically **promoted** to operate on lists, simply by applying it to a list – we are planning an IDE that will display promoted operators in boldface. For example,

$$-[1, 2, 3] = [-1, -2, -3]$$

$$[5, 5, 5] \mathbf{+} [10, 20, 30] = [15, 25, 35]$$

$$[1, 2] \mathbf{+} [10, 20] \mathbf{+} [100, 200] = [111, 222]$$

The semantics of the promotion functional are given by a schema of recursive definitions:

$$\begin{aligned} \mathbf{F}(s) &:= \\ & [] \text{ when } s=[] \text{ else} \\ & [\mathbf{F}(s_1)] \mathbf{++} \mathbf{F}([s_2, s_3, \dots]) \end{aligned}$$

$$\begin{aligned} \mathbf{F}(s, s') &:= \\ & [] \text{ when } s = s'=[] \text{ else} \\ & [\mathbf{F}(s_1, s'_1)] \mathbf{++} \mathbf{F}([s_2, s_3, \dots], [s'_2, s'_3, \dots]) \end{aligned}$$

etc.

« $\underline{x}$ » is a potentially infinite list of  $x$ 's, and is called the **extension** of  $x$ . Extension can be used with promotion as in the following examples:

$$[1, 2, 3] \mathbf{+} \underline{10} = [11, 12, 13]$$

$$\underline{10}^{\mathbf{+}} [1, 2, 3] = [10, 100, 1000]$$

Promotion and extension scales to nested operations, in the sense that an operator can be automatically promoted to operate on lists, lists of lists, etc. Extension scales similarly. For example:

$$\begin{aligned}
 & [[1, 2, 3], [4, 5]] + \underline{10} = \\
 & [[1, 2, 3] + \underline{10}, [4, 5] + \underline{10}] = \\
 & [[1 + \underline{10}, 2 + \underline{10}, 3 + \underline{10}], [4 + \underline{10}, 5 + \underline{10}]] = \\
 & [[11, 12, 13], [14, 15]]
 \end{aligned}$$

These also scale to function bodies and functions. NOTE: I'm not sure what this means

The following examples illustrate the interplay between promotion and `«when»`. This achieves the effects of a 'filter', due to the semantics of promotion, `«when»`, and the fact that concatenation of `«nil»` is the identity on lists:

$$\begin{aligned}
 & [1, \dots, 100] \text{ when } [1, \dots, 100] \bmod \underline{2} = 0 \\
 & = [2, 4, \dots, 100] \\
 \\
 & [\text{foo}, \text{bar}, \text{hello}] \text{ when } [3, 1, 7] > \underline{2} \\
 & = [\text{foo}, \text{hello}]
 \end{aligned}$$

## User defined functions

A **function definition** has the form `«T := B»`, when *T* and *B* are terms, respectively called the **head** and **body** of the definition. If *T* contains variables, the definition is considered shorthand for the collection of all definitions which can be obtained from it by consistently replacing every variable of type *S* in *T* with a term of type *S*.

If a program contains the definition `«T := B»` then the value of *T* is the value of *B*. For example, if a program contains

```

var n  :: int

f(n)  := n2 + 1

g(n)  := 2*n

positive_filter(n) := n when n > 0

```

Then the value of `«f(5)»` in that program is 26, the value of `«g(5)»` is 10, and the value of `«f(g((3)))»` is 37. `«positive_filter(5)»` returns 5, and `«positive_filter(-4)»` returns `«nil»`. Notice that variable *n* is declared only once.

A term is called **ground** if it contains no variables. If *T* is a term and *G* is a ground term which can be obtained from *T* by consistently replacing variables with ground terms of appropriate type,

then we say that  $G$  **matches**  $T$ . For example,  $\ll f(5) \gg$  matches  $\ll f(n) \gg$ , but  $\ll f(5) \gg$  does not match  $\ll f(6) \gg$ , or  $\ll g(n) \gg$ . A function may have more than one definition in a program, as long as no ground term matches the head of more than one definition. This can be checked at compile time. For example, the following is valid:

```
foo(4) := 8
foo(5) := 15
```

While the following is not:

```
foo(4) := 8
foo(n) := 15
```

## Functionals

A **functional** appears as a superscript following a function symbol. In effect, a functional acts on a function to return another function. A familiar example is the derivative functional from calculus, which acts on one function, say  $\lambda x. x^2$ , to return another function, say  $\lambda x. 2x$ . SequenceL contains two built-in functionals, described as follows.

$\ll \cdot \gg$  is the **ancestral extension** functional. If  $F$  is a unary function and  $x$  a term, then  $\ll F^\cdot(x) \gg$  returns the sequence  $\ll x \mid F(x) \mid F(F(x)) \dots \gg$ . More formally, we may define  $\ll \cdot \gg$  by the following recursive definition:

$$F^\cdot(x) := x \mid F^\cdot(F(x))$$

For example,

```
Member^\cdot([5, [6, 7], [[8]]])
⇒ [5, [6, 7], [[8]]] | 5 | [6, 7] | 6 | 7 | [[8]] | [8] | 8
```

$\ll \cdot \gg$  may be used to define ancestral relationships. For example,

```
Ancestor(p) := Parent(Parent^\cdot(p))
```

$\ll \omega \gg$  is the **fixpoint** functional, and acts on any unary function  $F$  as follows:

$$F^\omega(x) := \begin{cases} x & \text{when } F(x) = x \\ F^\omega(F(x)) & \text{else} \end{cases}$$

The effect of this is basically to *apply F until it has no effect*.  $\omega$  can be used like a while-loop in C or Fortran. For example, if a program contains

```
f(x) := 2*x when x < 100 else x
```

Then in that program,  $\ll f^{\omega}(3) \gg$  returns 192, which is obtained by starting with 3, and doubling until the result is no longer less than 100.

Functionals may also be defined by a program. For example,

```
var x, y : number  
  
plusy(x) := x + y
```

In a program containing this definition,  $\ll \text{plus}^{10}(5) \gg$  would return 15. It is doubtful a programmer would need to redefine addition in this way -- the example simply illustrates the use of functionals.

If  $L$  is a (possibly empty) list of unary functions, then  $\ll L(x) \gg$  returns  $x$  if  $L$  is empty, and otherwise reduces to  $\ll [L_2, L_3, \dots] (L_1(x)) \gg$ . Basically, the functions in the list  $L$  are applied to  $x$  sequentially, in the order in which they appear. So, for example,

```
[sin, cos, tan](5) = tan(cos(sin(5)))
```

The ability to apply lists of functions in this way can be combined with user-defined functionals to achieve the effects of a for-loop. For example, if a program contains

```
var x, y : number  
  
timesy(x) := x * y
```

then we may write, using ellipses,

```
factorial(n) := [times1, ..., timesn](1)
```

or, using promotion and extension

```
factorial(n) := times^[1...n](1)
```

A more straightforward way to write factorial might be

```
factorial(n) := 1 * ... * n
```

but the first two methods illustrates how functionals can be used with lists of functions.

## Future Work

It is possible to write a **defeasible function definition** by replacing  $\langle\langle := \rangle\rangle$  with  $\langle\langle \approx \rangle\rangle$ . Any number of defeasible definitions of a function are allowed to match the same ground term. However, if two defeasible function definitions have heads that can match the same ground term, then the head of one must be more specific than the head of the other. If  $s_1$  and  $s_2$  are schemas,  $s_1$  is **more specific** than  $s_2$  if every ground term which matches  $s_1$  also matches  $s_2$ . If a term matches the head of one or more defeasible function definitions, and matches the head of no strict definition, then the defeasible definition with the most specific head is used.

## Appendix I : Complete table of operators

Appearance	Name	Example
+ - * - < ≤ > ≥ sin cos tan = asin acos atan ln π, e	as usual	3+3=6 (4 < 8) = true (1 < 2 < 3 < 40) = true
/	real division	8/4 = 2, 3/6 = 0.5
mod	remainder	34 mod 10 = 4
floor	greatest integer	floor(5.7)=5
[ , ]	list formation	[1,2,3]
++	append	[1,2]+[3,4] = [1,2,3,4]
x <sub>n</sub>	subscript	[10,20,30] <sub>2</sub> =20
\	takeaway	[1,2,2,3]\[2,4] = [1,3]
size	cardinality	size([10,20,30]) = 3
remdups	remove duplicates	remdups([1,2,2,3]) = [1,2,3]
:	structure formation	(foo:1, bar:2)
.	structure indexing	(foo:1, bar:2).foo = 1
in	membership predicate	(3 in [1,2,3]) = true
= <sub>bag</sub>	equality as bags	[1,2] = <sub>bag</sub> [2,1]
= <sub>set</sub>	equality as sets	[1,2,2] = <sub>set</sub> [2,1,1]
⊂	subset	([1,3] ⊂ [3,2,1]) = true
&, or, not	Boolean connectives	true & true = true, not(true)=false, true or false = true, etc.
some, all, none	quantifiers	some([false, true, false]) = true all([false, true, true]) = false
	selection operator	foo   bar
Member	member operator	Member([a,b,c]) <b>reduces to</b> (a   b   c)
is_a	indefinite comparison	(b is_a Member([a,b,c])) = true
when	conditional	x when 1=1 = x, x when 1=0 <i>evaluates to</i> nil
else	contingency	nil else x = x, y else x = y -- <i>whenever y is not nil</i>
if then	conditional	if x then y <i>is the same as</i> y when x
...	ellipses	[1,...,4] = [1,2,3,4] 1+...+4 = 1 + 2 + 3 + 4
<b>boldface</b>	promotion	x <b>+</b> y = [x <sub>1</sub> +y <sub>1</sub> , ...]
overline	extension	x <b>+</b> <u>3</u> = [x <sub>1</sub> +3, ...]
ω	fixpoint	<i>repeat a function until a fixpoint is reached</i>
..	ancestral closure	Ancestor(p) := Parent(Parent(p))
[f, g, h](x)	sequential application	[f, g, h](x) = h(g(f(x)))

## Appendix II : precedence of infixes

Operators appearing higher in the table have higher precedence.

All associate left to right.

^
.
* / mod
+ - +
< > = in = <sub>set</sub> = <sub>bag</sub> is_a C ≤ ≥
&
...
or
when
else

.