SequenceL Language Reference and Problem Solving Guide
# Table of Contents

Table of Contents ........................................................................................................... 2

**SequenceL Language Reference** .................................................................................. 6

**Data Types** .................................................................................................................. 6
  Scalars ............................................................................................................................... 6
  Lists ................................................................................................................................. 6
  Structures ....................................................................................................................... 6
  Declaring structure types ............................................................................................. 7
  Parameterized structure type declarations .................................................................. 7
  Functions ....................................................................................................................... 8

**Arithmetic Operations** ................................................................................................ 8
  Basic Operations ......................................................................................................... 8
  Summations and Products of a List .............................................................................. 9
  Built-in Arithmetic functions ....................................................................................... 9

**Conditional Operations** ............................................................................................. 9
  When Clauses ................................................................................................................ 9
  Comparison Operators ............................................................................................... 10
  Boolean Connectives ................................................................................................. 10
  Boolean Quantifiers ................................................................................................. 11
  Boolean Operations on Lists ...................................................................................... 11

**List Operations** .......................................................................................................... 12
  Concatenation ............................................................................................................ 12
  Size() ......................................................................................................................... 12
  Subscripting ............................................................................................................... 12
  List Functions ........................................................................................................... 13
  Generating a List ...................................................................................................... 14

**Structure Operations** ................................................................................................ 14

**Type Operations** ....................................................................................................... 15

**String and Character Operations** ............................................................................. 15
  Character Functions .................................................................................................... 15
  Functions that do not Normalize-Transpose (NT) ....................................................... 16

**User-Defined Functions** ............................................................................................ 17
  Defining a Function ..................................................................................................... 17
  Specifying a Function Signature ............................................................................... 17
  Using Free Variables to Index Functions ................................................................ 19

**Foreach clauses** .......................................................................................................... 20

**Let Statements** ......................................................................................................... 21

**Overtyped Arguments and the Normalize Transpose operation** .................................. 22

**Recursion** .................................................................................................................. 23

**Commenting Code** .................................................................................................. 24

**Importing SequenceL Libraries** ................................................................................ 24
  Overview ..................................................................................................................... 24
  Syntax ......................................................................................................................... 24
  General ....................................................................................................................... 24
  Details ......................................................................................................................... 25
  The public statement ............................................................................................... 25
  The import statement: ............................................................................................. 25
  Using an imported expression: .............................................................................. 28
  Chaining imports: ................................................................................................. 29
**SequenceL Language Reference and Problem Solving Guide**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>intToFloat(int I)</td>
<td>51</td>
</tr>
<tr>
<td>boolToInt(bool B)</td>
<td>52</td>
</tr>
<tr>
<td>intToBool(int I)</td>
<td>52</td>
</tr>
<tr>
<td>boolToString(bool B)</td>
<td>53</td>
</tr>
</tbody>
</table>

**Math.sl**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>e()</td>
<td>53</td>
</tr>
<tr>
<td>pi()</td>
<td>53</td>
</tr>
<tr>
<td>integerPower(number N, int P)</td>
<td>54</td>
</tr>
<tr>
<td>log(number B, number N)</td>
<td>54</td>
</tr>
<tr>
<td>round(number N)</td>
<td>55</td>
</tr>
<tr>
<td>ceiling(number N)</td>
<td>55</td>
</tr>
<tr>
<td>sign(int I)</td>
<td>56</td>
</tr>
<tr>
<td>abs(number N)</td>
<td>56</td>
</tr>
<tr>
<td>min(number N, number M)</td>
<td>57</td>
</tr>
<tr>
<td>max(number N, number M)</td>
<td>57</td>
</tr>
<tr>
<td>mean(number List N)</td>
<td>58</td>
</tr>
<tr>
<td>median(number List N)</td>
<td>58</td>
</tr>
<tr>
<td>mode(T List L)</td>
<td>58</td>
</tr>
</tbody>
</table>

**Sequence.sl**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>range (int Start, int Stop, int Step)</td>
<td>59</td>
</tr>
<tr>
<td>duplicate(T E, int N)</td>
<td>60</td>
</tr>
<tr>
<td>firstIndexOff(T List L, T E)</td>
<td>60</td>
</tr>
<tr>
<td>lastIndexOff(T List L, T E)</td>
<td>61</td>
</tr>
<tr>
<td>indexesOff(T List L, T E)</td>
<td>61</td>
</tr>
<tr>
<td>split (T List L, T E)</td>
<td>62</td>
</tr>
<tr>
<td>delimit (T List List L, T E)</td>
<td>62</td>
</tr>
<tr>
<td>reverse (T List L)</td>
<td>63</td>
</tr>
<tr>
<td>setElementAt (T List L, int I, T E)</td>
<td>63</td>
</tr>
<tr>
<td>vectorMin (number List L)</td>
<td>63</td>
</tr>
<tr>
<td>vectorMax (number List L)</td>
<td>64</td>
</tr>
<tr>
<td>last (T List L)</td>
<td>64</td>
</tr>
<tr>
<td>allButLast (T List L)</td>
<td>65</td>
</tr>
<tr>
<td>take (T List L, int N)</td>
<td>65</td>
</tr>
<tr>
<td>drop (T List L, int N)</td>
<td>66</td>
</tr>
<tr>
<td>column (T List List L, int I)</td>
<td>66</td>
</tr>
<tr>
<td>sort (number List L)</td>
<td>67</td>
</tr>
<tr>
<td>sortBy (T List L, function F)</td>
<td>67</td>
</tr>
<tr>
<td>fold (T List L, function F)</td>
<td>68</td>
</tr>
<tr>
<td>leftFold (T List L, function F)</td>
<td>69</td>
</tr>
<tr>
<td>rightFold (T List L, function F)</td>
<td>69</td>
</tr>
<tr>
<td>equalListNT (T List L, T List M)</td>
<td>70</td>
</tr>
<tr>
<td>equalBagNT (T List L, T List M)</td>
<td>70</td>
</tr>
<tr>
<td>equalSetNT (T List L, T List M)</td>
<td>71</td>
</tr>
<tr>
<td>appendNT (T List L, T List M)</td>
<td>71</td>
</tr>
</tbody>
</table>

**Set.sl**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>elementOf(T E, T List L)</td>
<td>72</td>
</tr>
<tr>
<td>disjoint(T List L, T List M)</td>
<td>72</td>
</tr>
<tr>
<td>intersection(T List L, T List M)</td>
<td>73</td>
</tr>
<tr>
<td>union(T List L, T List M)</td>
<td>73</td>
</tr>
<tr>
<td>superSet(T List L, T List M)</td>
<td>74</td>
</tr>
</tbody>
</table>

© 2009-2015, Texas Multicore Technologies, Inc.
<table>
<thead>
<tr>
<th>Appendix IV: BLAS and LAPACK Libraries</th>
<th>........................................................................ 81</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAS</td>
<td>Calling a BLAS function from within a SequenceL Program ...................................................................... 81</td>
</tr>
<tr>
<td>.............................................</td>
<td>Linking with a BLAS library .................................................................................................................. 82</td>
</tr>
<tr>
<td>LAPACK</td>
<td>Calling a LAPACK function from within a SequenceL Program ........................................................................ 83</td>
</tr>
<tr>
<td>.............................................</td>
<td>Linking with a LAPACK library ................................................................................................................ 84</td>
</tr>
<tr>
<td>Appendix V: C++ helper functions</td>
<td>........................................................................ 85</td>
</tr>
<tr>
<td>CreateHelp.h</td>
<td>........................................................................ 85</td>
</tr>
<tr>
<td>sli_create(int sd, T min, T max, int x, int n, Sequence&lt;T&gt;&amp; result)................................................. 85</td>
<td></td>
</tr>
<tr>
<td>sli_create(int sd, T min, T max, int x, int y, int n, Sequence&lt;Sequence&lt;T&gt;&gt;&amp; result)...................... 86</td>
<td></td>
</tr>
<tr>
<td>sli_create(int sd, T min, T max, int x, int y, int z, int n, Sequence&lt;Sequence&lt;Sequence&lt;T&gt;&gt;&amp; result) 87</td>
<td></td>
</tr>
<tr>
<td>FileHelp.h</td>
<td>........................................................................ 88</td>
</tr>
<tr>
<td>sli_sequence2file(const Sequence&lt;T&gt;&amp; seq, const char *path) ......................................................... 88</td>
<td></td>
</tr>
<tr>
<td>sli_file2sequence(const char *path, Sequence&lt;T&gt;&amp; result) ................................................................. 88</td>
<td></td>
</tr>
<tr>
<td>sli_sequence2bmp(const Sequence&lt;Sequence&lt;int&gt;&gt;&amp; img, const int bpp, const char *path).................. 89</td>
<td></td>
</tr>
<tr>
<td>sli_bmp2sequence(const char *path, int *bpp, Sequence&lt;Sequence&lt;int&gt;&gt;&amp; result)............................ 90</td>
<td></td>
</tr>
<tr>
<td>sli_sequence2wav(const Sequence&lt;int&gt;&amp; left, const Sequence&lt;int&gt;&amp; right, const int bpcnh, const int rate, const char *path)............ 91</td>
<td></td>
</tr>
<tr>
<td>sli_wav2sequence(const char *path, int *bits_per_chan, int *samples_per_sec, Sequence&lt;int&gt;&amp; left, Sequence&lt;int&gt;&amp; right) .......... 92</td>
<td></td>
</tr>
<tr>
<td>SLTimer.h</td>
<td>........................................................................ 92</td>
</tr>
<tr>
<td>Class SLTimer .................................................................................................................. 93</td>
<td></td>
</tr>
</tbody>
</table>
Data Types

SequenceL has three kinds of data types: scalars, lists and structures.

 Scalars

The scalar types that SequenceL supports are:

- int
- float
- bool  Boolean - either of the two keywords, true or false
- char  Single characters - ‘a’, ‘b’, etc.

Strings are just lists of chars, and can be represented by the shorthand of as characters within quotes. That is, "abc" is the equivalent of ['a', 'b', 'c'], and "a" is the equivalent of ['a'].

 Lists

Lists can contain any number of items that can be scalar values, structures or other lists. However, a list can only contain one data type - in other words, a list is an ordered collection of a single data type. If multiple data types need to be associated, a structure should be used.

Below are some examples of lists.

```
[1, 2, 3]
[[2.0, 4.1, 5.45], [5.6, 2.111, 1.0]]
[]
>[]
[][], [[]]
[[[], [], []]]
[[[[], [], []]]]
[(a:1, b:2), (f:0, c:4)]
[1.2, 1, 4, 5]
```

(a list containing two lists)
(an empty list)
(here [] is an empty list within the list)
(a deeply nested list)
(a list of structures – see below)
(ok to mix int and double)

As noted, a list cannot contain more than one data type, so the following lists would be errors:

```
[1, (a:1, b:3)]  (can’t have an int and a structure)
[1, [1, 2, 3, 4, 5]]  (can’t have an int and a list)
[[1, 2, 3], [[4, 5, 6], [0, 0, 0]]]  (the 2 lists must have the same depth)
```

 Structures

A structure is a group of items where each item is associated with a name. A structure is indicated by a set of parentheses surrounding “name: item” pairs, with each pair separated by a comma.

A name within a structure must begin with a letter (a…z or A…Z). A structure can have any number of items inside of it and these items can be of any type. Below are some examples of structures.
(foo: 1, bar:2)
(a: [2, 3], b: "abc", c: 4)
(x: 2.5, y: 1, z: 9.2)
(book123: (mo:1, yr:1999))

(a structure containing another structure)

Note:

The label names must not be C++ and OpenCL keywords from the following list: INFINITY, Memflags, NaN, __constant, __global, __kernel, __local, __private, __read_only, __read_write, __write_only, abstract, alignas, alignof, array, asm, auto, bool, break, case, catch, char, char16_t, char32_t, class, const, const_cast, constant, constexpr, continue, decltype, default, delegate, delete, deprecated, dllibimport, do, double, dynamic_cast, else, enum, event, explicit, extern, false, finally, float, for, friend, friend_as, gcnew, generic, get_global_id, get_local_id, global, goto, if, initonly, inline, int, interior_ptr, kernel, literal, local, long, mutable, naked, namespace, new, noexcept, noinline, noreturn, noshare, operator, ostream, private, property, protected, public, read_only, read_write, register, reinterpret_cast, return, safecast, sealed, selectany, short, signed, sizeof, static, static_assert, static_cast, string, struct, switch, template, this, thread, thread_local, throw, true, try, typedef, typeid, typename, union, unsigned, using, uuid, virtual, void, volatile, wchar_t, while, write_only.

For SequenceL programs that are compiled by the slc compiler, the label names must also not begin with the following words that are used in the C++ code generated by the slc compiler: SequenceL_, SL_FLOAT, sl_, std, tbb.

Declaring structure types

The user must declare structure types. A declaration will include the name of the type, as well as all of the labels within the structure and their types.

For example, the following declaration creates a structure named Complex that has two members:

    Complex ::= (real: float, imaginary: float);

Structure declarations can be recursive as shown in the following example:

    Tree ::= (value: int, left: Tree, right: Tree);

Parameterized structure type declarations

SequenceL allows for the declaration of parameterized types. The type declarations will include generic parameters, but when an instance of the type is referenced, the types of the parameter must be provided.

Examples:

    Pair<U, V> ::= (first : U, second : V);
NewType1 ::= (a : Pair<int, char(1)>, b : Pair<float(1), int>);

Functions

Functions can be treated as values. The values can be passed to other functions and stored within lists and structures. The type of a function value is stated by putting the function’s signature within parentheses. For more information on function signatures, please see page 17.

Example:

NewType2 ::= (x : (int * float -> char), y : float);

The label x of NewType2 expects a function that takes two values, one of type int and one of type float, and returns a value of type char.

Arithmetic Operations

Basic Operations

The arithmetic operators +, -, *, /, ^, and mod are defined in SequenceL for numbers.

Exponentiation is defined using the ^ operator. The return value is always a float, regardless of the input types.

\[3^2 \Rightarrow 9.0\]
\[3.5^{6.2} \Rightarrow 2361.684476568993\]

Below are some examples and their given result.

\[5 + 4 \Rightarrow 9\]
\[3 - 4 \times 5 \Rightarrow -17\]
\[-3 + 2.5 \Rightarrow -0.5\]
\[4 / 2 \Rightarrow 2\]
\[(2+10^2) \times (3.141592654-7)/2.0 \Rightarrow -196.77877466692055\]

Types of operations are inferred by their arguments. Therefore:

\[1/2 \Rightarrow 0\]
\[1.0/2 \Rightarrow 0.5\]

Mixed mode operations are discouraged because they can cause confusion by giving unexpected results. For example

\[1.2 \times 10^{52} \Rightarrow 0.0\]
\[1.2 \times 10.0^{52} \Rightarrow 1.2e52\]

The keyword mod is used for taking the modulus of two numbers. mod is an infix operator. Below is an example of using mod.

\[5 \mod 2 \Rightarrow 1\]
Summations and Products of a List

sum([2, 4, 3, 12])  ⇒  21
product([2, 4, 3, 12])  ⇒  288

Built-in Arithmetic functions

Below are arithmetic functions that are built into SequenceL. A function is called by entering the name of the function, followed by its arguments in parentheses separated by commas. An example of a function call is \textit{f(arg1, arg2)}.

The function \textit{floor(x)} returns the greatest integer not exceeding \(x\).

\begin{align*}
\text{floor}(5.2) & \Rightarrow 5 \\
\text{floor}(5) & \Rightarrow 5 \\
\text{floor}(-5.2) & \Rightarrow -6
\end{align*}

The function \textit{sqrt(x)} returns the square root of \(x\) as a float.

\begin{align*}
\text{sqrt}(9) & \Rightarrow 3.0 \\
\text{sqrt}(3) & \Rightarrow 1.7320508
\end{align*}

The function \textit{ln(x)} returns the natural logarithm of its argument as a float.

\begin{align*}
\text{ln}(1) & \Rightarrow 0.0 \\
\text{ln}(34.7) & \Rightarrow 3.546739687
\end{align*}

The following trigonometric functions are defined in SequenceL: \textit{sin}, \textit{cos}, \textit{tan}, \textit{asin}, \textit{acos}, \textit{atan}.

\begin{align*}
\text{sin}(0) & \Rightarrow 0.0 \\
\text{cos}(34.7) & \Rightarrow -0.9898667 \\
\text{tan}(1) & \Rightarrow 1.5574077246549023 \\
\text{atan}(0.5) & \Rightarrow 0.4636476090008061 \\
\text{acos}(0.9) & \Rightarrow 0.4510268117962624 \\
\text{asin}(0.7) & \Rightarrow 0.7753974966107531
\end{align*}

Conditional Operations

When Clauses

The \texttt{when} command is used to test conditions. The format for the command is: \texttt{x when y}. The \texttt{y} statement must return a Boolean value. When \texttt{y} is \texttt{true}, \texttt{x} is returned, otherwise nothing is returned.
The `when` command can also be used with an `else` statement. The format for this command is: `x` when `y` else `z`. The `y` statement must return a Boolean value. When `y` is `true`, `x` is returned, otherwise `z` is returned.

5 when true else 4 ⇒ 5
5 when false else 4 ⇒ 4

It should be noted that SequenceL short-circuits `when/else` clauses, but not other logic clauses. SequenceL is designed to maximize parallel performance, so operations that might be short circuited are performed in parallel instead. For instance, given a function that returns true if either one of two conditions are met:

```plaintext
test(a) :=
    cheapOperation(a) or expensiveOperation(a);
```

SequenceL will perform both operations in parallel, so there is no gain to be had by putting the cheap operation first. To achieve that effect, one would write:

```plaintext
test(a) :=
    true when cheapOperation(a) else expensiveOperation(a);
```

In this case, if `cheapOperation()` returns true, `expensiveOperation()` will never be evaluated.

**Comparison Operators**

The following basic comparison operators are defined in SequenceL: `=`, `<`, `<=`, `>`, `>=`, and `/=`.

5 = 5 ⇒ true
5 /= 5 ⇒ false
"abc" = "abc" ⇒ [true, true, true]
"abc" = "deb" ⇒ [false, false, false]
(name: "alex")=(name: "alex") ⇒ true

The operators `<`, `<=`, `>`, and `>=` can only be performed on numbers. They will return either `true` or `false` if used on the correct domain.

5 < 3 ⇒ false
5 >= 3 ⇒ true

**Boolean Connectives**

The following Boolean connective operators are defined in SequenceL: `and`, `or`, `not, not` is a prefix operator that takes one argument while the other two are infix operators. All three operators take Boolean values as their operands and return a Boolean value.
Boolean Quantifiers

The following quantifiers are defined in SequenceL: \texttt{all}, \texttt{some}, \texttt{none}. These are unary prefix operators that take a list of Boolean values as their operand and return a single Boolean value.

\texttt{some(A)} returns \texttt{true} if and only if at least one member of \texttt{A} is \texttt{true}.

\texttt{all(A)} returns \texttt{true} if and only if either \texttt{A} is \texttt{empty} or every member of \texttt{A} is \texttt{true}.

\texttt{none(A)} returns \texttt{true} if and only if no member of \texttt{A} is \texttt{true}.

\begin{verbatim}
  some([ 3=2, 5>4, false]) => true
  all([ 3=2, 5>4, false]) => false
  none([ 3=2, 5>4, false]) => false
  some([]) => true
  all([]) => true
  none([]) => true
\end{verbatim}

Boolean Operations on Lists

The function \texttt{subset(A,B)} returns \texttt{true} if every member of \texttt{A} is also a member of \texttt{B}.

\begin{verbatim}
  subset( [1,2,3,4], [5,4,3,2,1]) => true
  subset( [1,6], [5,4,3,2,1]) => false
  subset("ac","abc") => true
  subset("ad","abc") => false
\end{verbatim}

All operations one can perform on lists also apply to strings, e.g. “this is a string”.

There are three special types of equality tests for lists defined in SequenceL: \texttt{equalList}, \texttt{equalBag}, and \texttt{equalSet}. They each take two lists as their operands and return a Boolean value. In SequenceL, a set is an \textit{unordered} group if items, each of which can occur \textit{exactly} once. A bag is an \textit{unordered} group if items, each of which can occur \textit{more than} once. A list is an \textit{ordered} group if items, each of which can occur \textit{more than} once.

\texttt{equalList(A,B)} returns \texttt{true} if the two lists are identical.

\texttt{equalBag(A,B)} returns \texttt{true} if the number of occurrences of every element \texttt{x} in \texttt{A} is same as the number of occurrences of \texttt{x} in \texttt{B}.

\texttt{equalSet(A,B)} returns \texttt{true} if \texttt{subset(A,B)} is \texttt{true} and \texttt{subset(B,A)} is \texttt{true}.

\begin{verbatim}
  (5 < 3) or (5 > 3) => true
  (5 < 3) or (2 = 3) => false
  not (5 >= 3) => false
  (5 > 3) and (3 = 3) => true
\end{verbatim}
equalList([1, 2, 3, 4], [1, 2, 3, 4]) ⇒ true
equalList([1, 2, 3, 4], [4, 3, 2, 1]) ⇒ false
equalBag([1, 2, 3, 4], [4, 3, 2, 1]) ⇒ true
equalBag([1, 2, 1, 3], [1, 2, 3]) ⇒ false
equalSet([1, 2, 1, 3], [1, 2, 3]) ⇒ true
equalSet([1, 2, 3], [1, 2]) ⇒ false
equalList("abc","abc") ⇒ true
equalList("abc","ac") ⇒ false

Note:

When comparing lists or strings, one should always use equalList(), never “=”.

List Operations

Concatenation

Lists can be concatenated together using the ++ operator. The ++ operator takes two lists and returns the list obtained by appending the second argument to the end of the first argument. The ++ operator is also defined for strings where it takes two strings as operands and returns the string obtained by appending the second argument to the first argument.

Examples:

[1, 2, 3] ++ [4,5] ⇒ [1,2,3,4,5]
[] ++ [1, 2, 3] ⇒ [1,2,3]
"hello" ++ "world" ⇒ "helloworld"

The function join can also be used to concatenate all of the items in a list together. Essentially, it removes the outer-most set of brackets within a list.

join([[1,2,3],[4,5,6],[7,8,9]]) ⇒ [1,2,3,4,5,6,7,8,9]
join([[0,1],[2,3]],[[4,5],[6,7]]) ⇒ [[0,1],[2,3],[4,5],[6,7]]

Size()

The function size(A) returns the number of items in the list A. It is also defined for strings to return the number of characters in a string.

size([4, 5, 6]) ⇒ 3
size([[4, 3], [5, 6]]) ⇒ 2
size([]) ⇒ 0
size("abc") ⇒ 3

Subscripting

If A is a list and b is an integer, then A[b] returns the b\textsuperscript{th} item of A. Lists are always indexed starting at 1, and not at 0. Strings can also be subscripted to return a character within a string.
SequenceL Language Reference and Problem Solving Guide

\(([4, 9, 1] ++ [6, 8])[3]\) \(\Rightarrow 1\)

"abcdef"[3] \(\Rightarrow 'c'\)

**b** can also be a list. When **b** is a list, a list is returned from every index specified in **b**.

\([4, 3, 2, 1, 5, 6, 7, 3][[3, 5, 7]]\) \(\Rightarrow [2, 5, 7]\)

\(A[b_1, ..., b_n]\) indexes into a multi-dimensional array.

\([[[4, 3, 2], [6, 7, 8]]][1, 2]\) \(\Rightarrow 3\)

\([[[4, 3, 2], [6, 7, 8], [12, 13, 15]][[1, 2], [2, 3]]\) \(\Rightarrow [[[3, 2], [7, 8]]\)

### List Functions

The function **head**(*A*) takes a list and returns the first item of that list.

The function **tail**(*A*) takes a list and returns a list that contains every element of *A* except for the first element.

Examples:

\(\text{head([1,2,3,4])} \Rightarrow 1\)
\(\text{head([[1,2],[3,4]])} \Rightarrow [1,2]\)
\(\text{tail([1,2,3,4])} \Rightarrow [2,3,4]\)

The function **transpose**(*A*) takes a list and returns the transpose of that list.

\(\text{transpose([[4,3,2],[6,7,8],[12,13,15]])} \Rightarrow [4,6,12],[3,7,13],[2,8,15]\)

If one passes the list of lists of different sizes, transpose() returns the list without changes:

\(\text{transpose([[1,2,3],[1,4]])} \Rightarrow [[1,2,3],[1,4]]\)

A transpose of a list can also be taken by using the keyword **all** as a subscript. This is useful for grabbing every item in a column.

\([[[4,3,2],[6,7,8],[12,13,15]]\text{all,2}] \Rightarrow [3,7,13]\)
\([[[4,3,2],[6,7,8],[12,13,15]]\text{all} \Rightarrow [[4,3,2],[6,7,8],[12,13,15]]\)

The function **takeAway**(*A, B*) takes two lists as its arguments and returns a list obtained by removing every occurrence of every member of **B** from **A**.

\(\text{takeAway([1,2,3,2],[2,5])} \Rightarrow [1,3]\)

The function **removeDups**(*A*) returns a list which contains only the first occurrence of an element **x** of **A**.
removeDups([5,7,5,8,2,2]) ⇒ [5,7,8,2]

Generating a List

A list of integers can be generated by using the ... operator. This operator creates a list ranging inclusively between its two operands.

2 ... 5 ⇒ [2,3,4,5]
5 ... 2 ⇒ []
(2...3) ... (9...10) ⇒ [[2,3,4,5,6,7,8,9],[3,4,5,6,7,8,9,10]]
((1...3)*0+1) ... 6)*0 ⇒ [0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0]

(No descending lists)

((2...3) ... (9...10)) ⇒ [[2,3,4,5,6,7,8,9],[3,4,5,6,7,8,9,10]]

(generates a 3 by 6 matrix of zeros)

Generated lists can also be used for subscripts:

[[1,2,3,4],[4,5,6,7],[9,8,7,6]][2...3] ⇒ [[4,5,6,7],[9,8,7,6]]
[[1,2,3,4],[4,5,6,7],[9,8,7,6]][2...3,2...3] ⇒ [[5,6],[8,7]]
"abcdefghi"[2...6] ⇒ "bcdef"

Structure Operations

SequenceL structures are used to group values of different types. A structure is a list of labels each associated with a value. These values can be of any type including lists and other structures. Below is an example of a SequenceL structure.

(a: 3, b: "abc", c: [4,3,1], d: (x: 4+3, y: false))

The operator A.B is used to reference values in a structure. A is a structure and B is a label. A.B will return the value associated with B in the structure A.

(a: 3, b: "abc").a ⇒ 3
(a: 3, b: "abc").b ⇒ "abc"
(a: 3, b: (c: 4+5, d:7)).b.c ⇒ 9

When declaring an instance of a structure, the user does not have to fill in values for all of the fields. If a label is a list or a scalar int, float, char or bool it will be initialized with a default value. If a label is a scalar struct or function, there will be no value defined. Below are the default values for each type:

int = 0
float = 0.0
char = ‘\0’
bool = false

Any list = []

Below is a type declaration, followed by an evaluation of an instance of that type.
MyType ::= (myInt : int, myChar : char, myFloat : float, myBool : bool, myList : MyType(1), myOwnType : MyType);

(myInt : 4) ⇒ (myBool:false,myChar:'\0',myFloat:0.0000000000,myInt:4,myList:[])
The function `intToAscii(i)` takes an int and returns a char, corresponding to the character represented by the ASCII value of the integer. This functionality is only defined for numbers from 1-127. Behavior for values out of that range is undefined.

```
intToAscii('77')  ⇒ 'M'
intToAscii([72,101,108,108,111,46])  ⇒ "Hello."
```

Using these built-in functions, it is easy to construct other useful text functions – see Appendix III for examples of string functions.

**Functions that do not Normalize-Transpose (NT)**

Most functions in SequenceL can be Normalize-Transposed so that they can be applied to actual parameters of greater dimension than the dimensions of the formal parameters to the function. For example, the `asciiToInt` function is defined for a character, but it can be applied to a string or a list of strings as shown earlier. However, the following functions that operate on lists cannot be Normalize-Transposed so they only operate on the outermost lists.

```
++ (list append)
join
head
tail
size
takeAway
removeDups
equalBag
equalSet
equallist
subset
transpose
```

Since the builtin functions listed above are not automatically Normalize-Transposed, but occasionally it is necessary to use Normalize-Transposed versions of these functions, the SequenceL library described in Appendix III provides the following Normalize-TTransposed functions that can be used instead of the builtin functions.

```
appendNT
equalBagNT
equalSetNT
equallistNT
subsetNT
```
User-Defined Functions

Defining a Function

A user can define a function in the following manner:

\[ F_n(\text{Arg}_1(\text{Depth}_1), \ldots, \text{Arg}_n(\text{Depth}_n)) := \text{Body}; \]

where \( F_n \) is the name of the function. The name should begin with an alphabet and the rest of the name can contain alphabets, digits, and underscore characters. The name must not contain any other special characters.

\( \text{Arg}_i(\text{Depth}_i) \) defines an argument named \( \text{Arg}_i \) whose depth is \( \text{Depth}_i \). \( \text{Depth}_i \) is defined to be the highest number of nested lists in \( \text{Arg}_i \). For example, the depth of a scalar is 0, the depth of a list full of scalars is 1, and a list of lists of depth 1 has a depth of 2. If no depth is specified, the depth is assumed to be 0 – that is, the argument is assumed to be a scalar.

Examples:

Simple functions

\[ f(n) := n^2 + 1; \]
\[ \text{max}(a,b) := a \text{ when } a>b \text{ else } b; \]
\[ \text{hd}(a(1)) := a[1] \text{ when } \text{size}(a) > 0; \]
\[ \text{tl}(a(1)) := a[2...\text{size}(a)] \text{ when } \text{size}(a) > 1 \text{ else } []; \]

\[ f(5) \Rightarrow 26 \]
\[ f(1...3) \Rightarrow [2,5,10] \text{ (i.e., } [f(1),f(2),f(3)]) \]
\[ \text{max}(5, 4) \Rightarrow 5 \]
\[ \text{hd}([7,3,2,6]) \Rightarrow 7 \]
\[ \text{tl}([7,3,2,6]) \Rightarrow [3,2,6] \]

A more complex function, using the let/in clause

\[ \text{quadratic}(a, b, c) := \]
\[ \text{let discriminant} := b^2-(4.0*a*c); \]
\[ \text{sqrTerm} := \sqrt{\text{discriminant}}; \]
\[ \text{in } [] \text{ when } \text{discriminant} < 0 \]
\[ \text{else } [ -b / (2.0 * a) ] \text{ when } \text{discriminant} = 0 \]
\[ \text{else } [ (-b + \text{sqrTerm})/(2.0*a), (-b - \text{sqrTerm})/(2.0*a) ]; \]

Note: When defining a function inside the interpreter’s console (i.e., not reading from a file), the keyword “:add” must be used, e.g.:

\[ :\text{add } f(a) := a * 3; \]

Specifying a Function Signature

The signature of a function consists of the type of each argument to the function and the return type of the function. The user can optionally specify the signature. If the signature is not specified, it will be inferred. A function signature can be defined in the following manner:
Fn : Type₁(Depth₁) * … * Typeₙ(Depthₙ) -> Typeᵣeturn(Depthᵣeturn);

Where Fn is the name of the function.

Typeᵢ(Depthᵢ) defines an argument or return type, where Typeᵢ is the name of the type and Depthᵢ is the depth of the type. If no depth is specified, the depth is assumed to be 0.

If the function does not have any arguments, the signature has the following format:

Fn : Typeᵣeturn(Depthᵣeturn);

Examples:

Simple functions

v : float;
v := 1.0;

f : float -> float;
f(n) := n^2 + v;

max : int * int -> int
max(a,b) := a when a>b else b;

f(5) => 26.0
f(1...3) => [2.0,5.0,10.0]
max(5, 4) => 5

Parameterized Function Signatures

A function signature can take a list of parameters in the same manner as a structure definition. This allows for generic function signatures. A parameterized function signature can be declared in the following manner:

Fn< T₁, …, Tₙ > : Type₁(Depth₁) * … * Typeₙ(Depthₙ) -> Typeᵣeturn(Depthᵣeturn);

Where Tᵢ can be used as any of the type names after the colon.

Parameterized examples:

    hd<U> : U(1) -> U;
    hd(a(1)) := a[1] when size(a) > 0;

    tl<U> : U(1) -> U(1);
    tl(a(1)) := a[ 2...size(a)] when size(a) > 1 else [];

    hd([7,3,2,6]) => 7
    tl([7,3,2,6]) => [3,2,6]
Using Free Variables to Index Functions

Functions that return lists can also be defined in terms of what a value should be at a certain index, using a “free variable.” This is useful for functions that need to use index values in calculations. These function definitions are the same as normal function definitions except that the index list is written after the argument list:

\[ \text{Fn}(\text{Arg}_1(\text{Depth}_1), \ldots, \text{Arg}_n(\text{Depth}_n))[\text{Index}_1, \ldots, \text{Index}_m] := \text{Body}; \]

\text{Index}_i\ is a symbol used to reference the index value. The range of the index value is not specified explicitly – they are free to range over the entire set of possible values (hence the name). That is, the range is from 1, to the size of whatever array is referenced using the free variable (index) inside the body of the function. Below are some examples.

\textbf{Matrix Multiply:}

\[ \text{mm}(A(2),B(2))[i,j] := \text{sum}(A[i,\text{all}] \times B[\text{all},j]); \]

In this example, \(i\) ranges from 1 through the number of rows in matrix \(A\), while \(j\) ranges from 1 through the number of columns in matrix \(B\). This function is stating that the value of the \(i\)th row and \(j\)th column of the matrix returned by the function \(\text{mm}()\) is equal to the value described in the body of the function. Note that this function also takes advantage of over-typed arguments, since \(A[i,\text{all}]\) and \(B[\text{all},j]\) both return lists.

Note that free variables must appear in the body of the function as the index of an array or list, so that the range can be determined. Though they may appear outside of an array or list subscript as general variables, they cannot appear only as variables.

For instance,

\[ \text{aa}(A(2),B(2))[i,j] := i + j; \]

is an error, because no range can be defined from the context. However, this is correct:

\[ \text{bb}(A(2),B(2))[i,j] := A[i] + B[j]\ \text{when}\ i < 2; \]

as is even this:
\[
cc(A(2),B(2))[i,j] := A[i]*i + B[j]*j \text{ when } i < 2;
\]

**Jacobi:**

\[
\text{jacobi}(A(2), \text{delta})[i,j] :=
\begin{cases}
A[i,j] \text{ when (i=1 or size(A)=i or j=1 or size(A)=j)} \\
\end{cases}
\]

In this example, \(i\) ranges from 1 through the number of rows in \(A\), and \(j\) ranges through the number of columns in \(A\). Note that a **when** clause is used for the border cases which would otherwise be nil. The result of this function will be a matrix.

### Foreach clauses

Normally, SequenceL can infer the range of the free variable(s) from its usage in the code. However, in some cases, SequenceL lacks context, and so must be given a hint. For example, consider a function which takes an integer \(n\), and returns a 2-dimensional square matrix of size \(n\), with all elements set to zero, except for the diagonal, which is set to 1.

\[
\text{identityMatrix}(n)[i,j] := \begin{cases} 
1 \text{ when } i=j \text{ else } 0
\end{cases}
\]

In this case, there is no context for \(i\) and \(j\) to determine their ranges, so the programmer must give that information explicitly:

\[
\text{identityMatrix}(n)[i,j] :=
\begin{cases}
1 \text{ when } i=j \text{ else } 0 \\
\text{foreach } i \text{ within } 1...n, \\
\text{ } j \text{ within } 1...n;
\end{cases}
\]

\[
\text{identityMatrix}(3) \Rightarrow \begin{bmatrix} 1,0,0 \end{bmatrix}, \begin{bmatrix} 0,1,0 \end{bmatrix}, \begin{bmatrix} 0,0,1 \end{bmatrix}
\]

The **form of the foreach clause** is **foreach** \(\text{var1} \text{ within } \text{list1}, \text{var2} \text{ within } \text{list2}, \ldots\)**, where \(\text{var1}\) names a free variable, \(\text{list1}\) contains the values that \(\text{var1}\) can take, \(\text{var2}\) names a second free variable, \(\text{list2}\) contains the values that \(\text{var2}\) can take, and so on. If there are multiple free variables, there can be multiple within clauses with at most one clause per variable. Further, the expressions for later clauses can depend on the free variables in earlier clauses as shown in the following example that creates a triangular array.

\[
f2(a(1),b(1))[i, j] := a[i] + b[j]
\text{foreach } i \text{ within } 1 \ldots \text{size}(a),
\jtext{within } 1 \ldots i;
\]

\[
f2([10,20,30],[1,2,3,4]) \Rightarrow \begin{bmatrix} 11,21,22 \end{bmatrix}, \begin{bmatrix} 31,32,33 \end{bmatrix}
\]

The **list** component of the foreach clause can contain lists of different types as shown in the following example that returns a struct containing an integer, a character, and a boolean.

\[
f3(n)[x, y, z] := (\text{param1} : x, \text{param2} : y, \text{param3} : z)
\]
foreach x within 1 .. n,  
y within ['a', 'b'],  
z within [true, false];

\[
f_3(2) \Rightarrow [[[\text{param1:1,param2:'a',param3:true}),(\text{param1:1,param2:'a',param3:false})],  
\quad [[[\text{param1:1,param2:'b',param3:true}),(\text{param1:1,param2:'b',param3:false})]],  
\quad [[[\text{param1:2,param2:'a',param3:true}),(\text{param1:2,param2:'a',param3:false})],  
\quad [[[\text{param1:2,param2:'b',param3:true}),(\text{param1:2,param2:'b',param3:false})]]
\]

**Let Statements**

Let statements are used to define values to be used only within a function. This is useful when the result of an evaluation is to be returned several times within a function. By using let statements they only have to be written once. The syntax for let statements in SequenceL is as follows:

\[
f(\text{args}) := \text{let v}_1:=\text{expression}_1; \ldots \text{v}_n:=\text{expression}_n; \text{ in } \text{Body};
\]

The name \(v_i\) can be used in Body to represent \(\text{expression}_i\). Below is an example using let statements.

\[
f(a,b) :=
\quad \text{let } x := a+5;
\quad \quad y := x*b;
\quad \text{in } x + y * a;
\]

\[
f(3,5) \Rightarrow 128
\]
\[
x \Rightarrow 8
\]
\[
y \Rightarrow 40
\]

Notice that let variables can be referenced in subsequent, or previous, variable definitions – as in this implementation of the power method for eigen values:

\[
eigen(\text{m}(2),\text{v}(1)) :=
\quad \text{let } s := \text{size}(\text{v}); \ e := \text{sum}(\text{m} * \text{v}); \ \text{v1} := \text{e}/\text{e}[s];
\quad \quad \text{in } \text{eigen(m,v1)}
\quad \quad \text{when all( math.abs(v[1...s-1] - v1[1...s-1] ) > 0.0099999) else v1;}
\]

Functions may also return structures:

\[
f(\text{v}(1),\text{m}(2)) := (x:v, y:m);
\]

and values referenced in other functions:

\[
g(\text{n}(0)) := f([1,2,3],["abc","def"]).x+n;
\]
\[
g(3) \Rightarrow [4,5,6]
\]

Structures can also contain functions:
\[ f(n(0)) := (\text{fact}:f(n-1).\text{fact}*n \text{ when } n > 1 \text{ else } n); \]
\[ f(3).\text{fact} \Rightarrow 6 \]

**Note:**
A key feature of the \texttt{let} statement is that the expressions within it are guaranteed to be evaluated exactly once, or never. In the quadratic function from above:

\[
\text{quadratic}(a, b, c) :=
\begin{align*}
\text{let } \text{discriminant} & := b^2-(4.0*a*c); \\
\text{sqrtTerm} & := \sqrt{\text{discriminant}}; \\
\text{in } & \begin{cases} 
[ ] & \text{when } \text{discriminant} < 0 \\
[-b / (2.0 \ast a)] & \text{when } \text{discriminant} = 0 \\
[-b + \sqrt{\text{Term}}] / (2.0 \ast a), (-b - \sqrt{\text{Term}}) / (2.0 \ast a) & \text{when } \text{discriminant} > 0
\end{cases}
\end{align*}
\]

the \texttt{sqrtTerm} term will not be evaluated if \texttt{discriminant} is less than zero. As mentioned before, \texttt{when/else} clauses short-circuit, so if \texttt{discriminant} is zero, the function will return immediately after the \texttt{in}. If, on the other hand, \texttt{discriminant} is positive, the \texttt{sqrtTerm} term will only be evaluated once, though it is used twice. Likewise, \texttt{discriminant} will only be evaluated once, even though it is used in different lines within the \texttt{in} clause.

As a programming practice, this makes it easy to write readable code by keeping the \texttt{in} clause clean by using variables for readability, without worrying about code efficiency:

\[
\text{quadratic}(a, b, c) :=
\begin{align*}
\text{let } & \begin{align*}
\text{discriminant} & := b^2-(4.0*a*c); \\
\text{sqrtTerm} & := \sqrt{\text{discriminant}}; \\
\text{root0} & := -b / (2.0 \ast a); \\
\text{root1} & := (-b + \sqrt{\text{Term}}) / (2.0 \ast a); \\
\text{root2} & := (-b - \sqrt{\text{Term}}) / (2.0 \ast a)
\end{align*}; \\
\text{in } & \begin{cases} 
[ ] & \text{when } \text{discriminant} < 0 \\
[\text{root0}] & \text{when } \text{discriminant} = 0 \\
[\text{root0, root2}] & \text{when } \text{discriminant} > 0
\end{cases}
\end{align*}
\]

The clean structure of the \texttt{in} clause makes it obvious what the program does, while the calculation details can be found in the \texttt{let} clause. But when compiled and run, the code will execute with perfect efficiency (and in parallel).

**Overtyped Arguments and the Normalize Transpose operation**

SequenceL can handle functions being passed arguments that are of a larger depth than specified in the function definition. This is called passing an overtyped argument. When a function call has overtyped arguments an operation called Normalize-Transpose is performed.

The Normalize stage has two steps.

1) An extra level of depth is added to all non-overtyped arguments by converting them to lists.
2) Every argument appends duplicates of its contents to itself until its length is equal to the argument with the largest length.

Example:

\[ 3 + [4, 5, 6] \]

The second argument to the plus operation is overtyped. First, an extra level of depth is added to the non-overtyped argument, as well as the operand.

\[ [3] [+][4, 5, 6] \]

Next, the first argument duplicates members of its list until its length is equal to the second argument’s length.

\[ [3, 3, 3] [+][+, +][4, 5, 6] \]

Finally, the Transpose stage takes the transpose of all of the arguments and performs the original function on each row of the transpose.

\[ [3 + 4, 3 + 5, 3 + 6] \]

This will return a list whose length is the length of the arguments.

So altogether:

\[ 3 + [4, 5, 6] \]
\[ [3, 3, 3], [+], [4, 5, 6] \]
\[ [3 + 4, 3 + 5, 3 + 6] \]
\[ 7, 8, 9 \]

Normalize the 3 and the ‘+’

Transpose

Evaluate

Overtyped arguments are a good tool for performing the same operation on different data without having to explicitly create any looping structures.

Arguments to a value being called as a function may not be overtyped. The depths of the arguments must match the signature of the value exactly.

**Recursion**

SequenceL supports recursion. For example:

\[ fact(n(0)) := fact(n-1) * n \text{ when } n>0 \text{ else } 1; \]

Here the quicksort algorithm is implemented to sort an array using recursion.
filtermin(p,s) := s when (s < p);
filtermax(p,s) := s when (s >= p);
qsort(x(l)):=
    qsort(filtermin(head(x),tail(x)))
  ++[head(x)]
  ++qsort(filtermax(head(x),tail(x)))
when size(x)>1 else x;

Always use when/else clauses instead of just when to avoid unexpected results when the when
conditions aren’t met. Using a when without an else should only be done when an empty result is
explicitly desired in the case of a when failure.

Commenting Code

SequenceL uses the same commenting system as C. To add a comment that lasts until the next new line,
use // . To create a comment block, enclose it in /* and */.

Importing SequenceL Libraries

Overview

SequenceL allows you to write an “expression” (which is a function or a constant) once, in one file, and
use it elsewhere in another file. To reuse a function or constant it must be declared “public” within the
file which defines it, and it must be imported by the file that uses it. Finally, when used, it must be
referred to appropriately. The mechanics for doing these things are laid out below.

Syntax

public <expression>[, <expression2>, …, <expressionN>];

import <expression>"*" from <file> [as [<library>.]<newExpressionName>]
[<library>::]<expression>()

The syntax falls into 3 categories, corresponding to the numbering above:
1) Declaring that an expression contained within file A is importable by other files
2) Importing an expression from file A into another file B
3) Using the imported expression within file B

General

“Import” always either imports
1) all public expressions in a file, or
2) just one of them.

In case #1 above, you may rename the entire library at the time of import, for ease of typing, or
disambiguation of similarly named libraries, by using the “as” clause.
In case #2 above, you may rename the expression, and/or the library containing it, at the time of import, for similar reasons, in the same manner.

**Details**

**The public statement**

```plaintext
public <expression>[, <expression2>, ..., <expressionN>];
```

Expressions must be “public” to be imported. Expressions are declared as public by inserting one or more “public” lines from the syntax above in the file in which they are defined, e.g.:

```plaintext
public foo, bar;
```

That is, you can declare either one or several expressions public at a time, and you can have multiple “public” lines in a file. You cannot declare an expression public “inline” – i.e.

```plaintext
public foo(a) := a * 3;
```

That is, the public declaration must appear as a separate statement.

**Style note:**

Although you could put the public statement prior to each and every expression you declare public (almost like using an inline “public” keyword”), the intent is for developers to declare their public expressions all at once, at the top of the file (either all on 1 line, or expression-by-expression) so that others can easily see what functionality the file provides, rather than having to wade through the code.

**The import statement:**

```plaintext
import <expression>|”*” from <file> [as [<library>].<newExpressionName>]
```

Here are the parts of the import statement broken down.

**<expression>**

Is either an expression’s (i.e., a function’s or a constant’s) name, or a “*”, meaning “all public expressions.” If an expression name is given, it must have been declared public, or the line will generate a compile-time error (or read-time error in the interpreter). Comma-delimited expression names are not supported – you either import one function at a time, or all of them at once.

**<file>**

Is the name of a file, and may take one of two forms:

```plaintext
“[path/]filename”
```

or
Note that in both cases, “filename” includes the extension, e.g., “math.sl”

The double-quotes indicate that the path/filename indicated is rooted in the same directory as the importing file. The angle-brackets indicate that the path/filename indicated is rooted in the SequenceL library directory.

More specifically, in the first case, the double-quotes indicate that the directory of the importing file should be the root of the path. So, for example, if the file foo.sl included the line

```sequence
import "bar.sl";
```

then bar.sl would need to be in the same directory as foo.sl. Similarly,

```sequence
import "../bar.sl";
import "biff/bar.sl";
```

will look in <path to foo.sl>/../ and <path to foo.sl>/biff (respectively) for bar.sl

In the second case, angle-brackets are used to indicate that the search for the file should be rooted in the SequenceL library directory, rather than the importing file’s directory. The SequenceL library directory is defined as $SL_HOME/Libraries. So for example, if the file foo.sl included the line

```sequence
import <bar.sl>;
```

the interpreter/compiler would just look in the $SL_HOME/Libraries directory for bar.sl. Likewise for

```sequence
import <../bar.sl>;
import <biff/bar.sl>
```

will look for the file $SL_HOME/Libraries/../bar.sl, or $SL_HOME/Libraries/biff/bar slu, respectively.

Note that in all cases, forward-slash (“/”) is used as a path-delimiter - the Windows-style backslash (“\”) is not used. This is to maximize cross-platform compatibility of code. For example, if you want to indicate a sub-directory of $SL_HOME/Libraries, you can say <foo/bar/math.sl> without worrying what will happen if you compile on a Windows machine.

<library> and <newExpressionName>

The default behavior is to refer to imported expressions using their library name as a prefix, with a double-colon (“::”) delimiting the library and the expression. E.g.

```sequence
import factorial from math.sl;  \xrightarrow{} x := math::factorial(y);
import * from math.sl;          \xrightarrow{} x := math::factorial(y);
                                  \xrightarrow{} a := math::cubeRoot(b);
```
Note that while the full file names (including extensions) must be given in the <file> section, only the base name (i.e. the path-less file name, minus the extension) is used when referring to it. This implies that you cannot generically import two files with the same basename:

```
import <bar.sl>
import <bar.xyz>
```

Since both would map to the “bar” library name. In that case, you’d have to use the “as <name>” clause (below) to rename one or both libraries.

The “as <name>” clause is used to rename either the library, or the specific expression to something else. As before, remember “Import” always either imports

1) all public expressions in a file (i.e. the whole library), or
2) just one of them.

Using the “as “ clause either renames the entire library (in case 1), or just that expression (and optionally the library as well) in case 2.

That means, for case #1 (renaming the entire library), you can refer to the expressions by a new library name, or by no library name at all (i.e., make them all “local” expressions), by using the “as” clause, as shown in the following examples:

**Examples**

1) **Rename the library for all its expressions**

Use the “*” wildcard to show that you want to import all expressions, but give the library a new name after the “as” clause.

```
import * from math.sl as m::*
  \( \rightarrow x := m::factorial(y); \)
  \( \rightarrow x := math::factorial(y); // this won’t work, need to use m::* \)
```

2) **Rename all expressions as local**

When importing all expressions via “*”, you can make them all local, just by not specifying a library name. Use with care: you may not know what all of the public expressions are, and if any conflict with any other expressions you’ve defined or imported, the whole thing will fail.

```
import * from math.sl as *
  \( \rightarrow x := factorial(y); \)
  \( \rightarrow x := math::factorial(y); // this won’t work, they’re all local \)
```

3) **Rename the library (and optionally, the one expression)**

You can use this technique to require the use of a different library name, for ease of access while still ensuring clarity. You may or may not rename the actual expression.

```
import factorial from myGradStudentsMathLib.sl as math::factorial;
  \( \rightarrow x := math::factorial(y); \)
```
import factorial from myGradStudentsMathLib.sl as math::fact;
  \rightarrow x := math::fact(y);
  \rightarrow x := myGradStudentsMathLib::factorial(y); // wouldn't work, because
  // the library isn’t called
  // “myGradStudentsMathLib” anymore, it’s just “math”

4) Rename the one expression as a local expression

Doing this implies that you don’t use the library name.

import factorial from math.sl as fact;
  \rightarrow x := fact(y);
  \rightarrow x := math::factorial(y); // this wouldn’t work, because it was made
  // a local on import

Renaming an expression, after the import

Using the “as” clause allows you to rename an expression at import time, which may be necessary to avoid conflicts. However, even without using the “as” clause, you can always rename an expression after the fact, as follows:

import * from myGradStudentsMathLib.sl as m::*;

foo := m::foo;

This renames the entire library, and THEN renames the function foo() to be local. This could also be accomplished by doing:

import * from myGradStudentsMathLib.sl as m::*;
import foo from myGradStudentsMathLib as foo;

In both cases, the following two commands would then be equivalent:

x := m::foo(y);
  x := foo(y);

Using an imported expression:

[<library>::]<expression>()

This is discussed in detail in the <library> and <newExpressionName> section above, but so that it’s clearly called out here, an imported expression is (usually) accessed by referring to its library name, and the expression name, separated by a “::” symbol. For example:

x := math::factorial(y);

Typically, the library name is the imported file’s base name (i.e. minus any path information and extension). However, as described in the <library> and <newExpressionName> section above, the “as” clause may be used to rename both the library and the expression itself, so that the same expression might be accessed in various ways, depending on how it was imported:
import factorial from math.sl \Rightarrow x := math::factorial(y);
import * from math.sl \Rightarrow x := math::factorial(y);
import * from math.sl as m::* \Rightarrow x := m::factorial(y);
import * from math.sl as * \Rightarrow x := factorial(y);
import factorial from math.sl as m::fact \Rightarrow x := m::fact(y);
import factorial from math.sl as fact \Rightarrow x := math::fact(y);

**Chaining imports:**

If your public code relies on code you import, e.g.:

`mymath.sl:`

```plaintext
import sqrt from math.sl;
public quadratic;
quadratic(a, b, c) :=
let discriminant := math::sqrt((b*b)-(4.0*a*c));
in [ (-b + discriminant)/(2.0*a), (-b - discriminant)/(2.0*a) ];
```

then SequenceL will follow that dependency chain and import the appropriate code from math.sl into the file that imports mymath.sl. It will do this efficiently, and not import the entire math.sl library when only math::sqrt() is used. But the end user will not have to import math.sl themselves – though obviously, math.sl must be available for their use.

**Re-exporting:**

You may not chain imports. That is, you cannot merely re-export an expression, e.g., in the file “foo.sl”, say:

```plaintext
import fact from math.sl;
public math::fact;
```

In that case, the user would have to use

```plaintext
import math::fact from foo.sl;
x := foo::math::fact;
```

to get to it.

However, you can make an imported expression “your own,” by:

```plaintext
import fact from math.sl;
public fact;
fact := math::fact;
```

or just

```plaintext
import fact from math.sl as fact;
public fact;
```

It might seem that this is in bad form – essentially passing off a library expression as yours – but there are scenarios where it might be useful: for instance, grouping a set of small libraries together into one
larger, more complete library, so that the user can access all of their expressions in one place, rather than remembering which library different things belong to.

**Foreign Functions**

In addition to importing SequenceL libraries, SequenceL supports the ability to call out to a non-SequenceL function from within compiled code.

**Warning:** Foreign functions can be extremely dangerous within SequenceL code, because they break the functional paradigm of SequenceL. That is, unlike SequenceL functions, foreign functions may have side effects, which means that SequenceL code that calls them will no longer be guaranteed to be race-condition-free, nor even necessarily parallelize correctly in general.

It is much preferable to write whatever foreign function is contemplated in SequenceL, to maintain the integrity of the code, and to allow the compiler to correctly exploit parallelisms.

However, it is recognized that there may be cases foreign functions are desirable, or even necessary – for instance, when

- complex code has already been written to perform a specific function, and regeneration in SequenceL is not practical, or
- code has been hand-written/tuned that exploits parallelisms or other efficiencies to a greater degree than the automatically generated C++ code produced by the SequenceL compiler would

In such cases, it is important to remember that if they must be used, foreign functions should:

- never introduce side effects of any kind,
- not communicate with other instances of themselves (pass tokens, etc.),
- use only local variables or constants,
- never perform any I/O, and
- be used only as a last resort

With that warning in mind, using a foreign function is straightforward. The function must be declared, and – if it is to be debugged in the interpreter – an equivalent or dummy or stand-in SequenceL function should be written.

To declare a foreign function call, use the following syntax:

```plaintext
f(<type>{<depth>}, ... ) -> type;
```

Example:

```plaintext
Ceiling(float) -> int;
```

Legal types are any scalar type, namely:

- int
- float
Foreign functions must have at least one argument. As always, “(<depth>)”, if omitted, defaults to 0. Otherwise, you may indicate the depth of an argument the same way as you would in the “intended use” argument (see Section 3.3, “Compiling a SequenceL file,” later in this guide):

- `int(1)` → Sequence of integers
- `int(0)` → Scalar integer
- `int` → Scalar integer

When the foreign function declaration is added to your code, the compiler will generate a function declaration in the header file. For example, the following declaration in SequenceL:

```sequence
fun1(int, char(1)) -> int;
```

would generate the following C++ code in the header file:

```cpp
int _foreign_SequenceL_fun1(int, const Sequence< char >&);
```

Foreign Functions are only implemented in the compiler – the interpreter will ignore the declaration. This means that in order to test code in the interpreter that relies on foreign functions, a SequenceL function must stand in for the foreign function.

For instance, this example will parse in the interpreter, and `foo()` will execute properly.

```sequence
1 fact(int) -> int;                  // FF declaration
2
3 fact(x) := x when x <= 2 else x * fact(x-1); // stand in function
4
5 foo(a) :=
6    let factorial := fact(a);
7    in factorial;
```

To compile it, you would just comment out line 3 (the `fact(x)` definition, which acts as a stand-in for the compiled function in the interpreter), compile it with the SequenceL compiler, and link the resulting C++ code with the code or library that implements the `fact(int)` declaration in line 1.
Appendix I: Problem Solving in SequenceL

SequenceL is a different way to solve problems. Consequently one approaches problems differently in SequenceL than the approaches used in other languages. It shares many of the same qualities of a pure functional language, but is based on very different computational laws. This difference results in a language that is very small, very transparent, and surprisingly simple solutions to even complex problems. For more information on transparency the reader is referred to Computer Magazine, September 2009.¹ For more information on the computational laws the reader is referred to ACM Transactions on Programming Languages and Systems 30.2 (March 2008).²

The SequenceL compiler takes as input a SequenceL solution (all solutions in this document included) and produces high performance multicore code. For more information on performance, see the proceedings of the Declarative Aspects of Multi-core Programming (DAMP) conference of 2010.³ The DAMP paper was written in 2009. Since that time performance has improved significantly.

Abstraction.

Work on SequenceL began in 1990. The goal was to see whether one could describe a problem solution strictly in terms of data products, without having to provide much in the way of algorithmic detail - i.e., most languages follow Wirth’s abstraction:

Data Structures + Algorithms = Programs

The abstraction level chosen was set builder notation. The mathematical structure chosen was the sequence, which can be viewed as a list. The desire was to see how far this abstraction could be pushed; to see if data products could be described in terms of form and content with little procedural description.

The best way to view SequenceL programming is to think in terms of operators that guide the interaction of data structures. Stripped away is the need to break data structures apart or reassemble them (which is required of other languages). There are four ways to approach a problem in SequenceL:

1. If there is a formal, precise definition of a problem, transcribe the solution in SequenceL.
2. If the sequences involved do not require knowledge of subscripts simply define the “base case” of the solution.
3. If indices of Sequences need to be referenced provide an indexed function.


³ Nemanich, Brad, Dan Cooke, and Nelson Rushton. Proceedings of DAMP (Declarative Aspects of Multi-core Programming) 2010, Madrid, Spain
4. When all else fails, provide a recursive definition to the problem solution, which may require the sequences to be broken apart and/or reassembled.

Examples:

1. Set membership

Definition: \( x \in S \) is true if some element of \( S = x \) otherwise false

SequenceL: \( \text{member}(X(0),S(1)) := \text{some}(X = S) \);

2. Subset

Definition: \( S_1 \subseteq S_2 \) is true if all elements of \( S_1 \) are also elements of \( S_2 \) otherwise false

SequenceL: \( \text{subset}(S_1(1),S_2(1)) := \text{all}(\text{member}(S_1,S_2)) \);

3. Even Numbers

Definition: \( \{ \ x \mid x \in I \ & \ x \mod 2 = 0 \} \)

SequenceL: \( \text{even}(X(0)) := X \) when \( X \mod 2 = 0 \);

Note: This is a good example of providing the base case - or fundamental definition.

4. Matrix Multiplication

Definition: \( (AB)_{ij} = \sum_{r=1}^{n} A_{ir} B_{rj} \)

SequenceL: \( \text{matmul}(A(2),B(2))[i,j] := \text{sum}(A[i,all]*B[all,j]) \);

Note: In this case subscripts must be provided to direct the interaction between the two data structures.

5. String Functions

Using just the two character built-in functions \( \text{asciiToInt()} \) and \( \text{intToAscii()} \), it is easy to construct an array of handy string functions:

\[
\text{isLowerCase(c)} := \\
\text{let i := asciiToInt(c);} \\
\text{in i >= asciiToInt('a') and i <= asciiToInt('z');}
\]
6. Game of Life

Definition:

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbors, which are the cells that are directly horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbors dies, as if caused by under-population.
2. Any live cell with more than three live neighbors dies, as if by overcrowding.
3. Any live cell with two or three live neighbors lives on to the next generation.
4. Any dead cell with exactly three live neighbors becomes a live cell.

The initial pattern constitutes the seed of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed—births and deaths happen simultaneously, and the discrete moment at which this happens is sometimes called a tick (in other words, each generation is a pure function of the one before). The rules continue to be applied repeatedly to create further generations. Notice Borders elements do not have 8 neighbors, so they remain zero and are only part of computations involving non-border elements.

SequenceL:

```sequence

isLowerCase('a') ⇒ true
isLowerCase('B') ⇒ false

allLowerCase(s(1)) :=
    all( isLowerCase(s) );

allLowerCase("Hello world") ⇒ false
allLowerCase("HELLO") ⇒ false
allLowerCase("hi there") ⇒ false
allLowerCase("hello") ⇒ true

(There's a space)

toUpper(c) :=
    intToAscii(asciiToInt(c) - 32) when isLowerCase(c)
    else c;

toUpper("Hello world") ⇒ "HELLO WORLD"
toUpper("hi there") ⇒ "HI THERE"

titleCase(s(1))[i] :=
    let beginningOfWord := asciiToInt(s[i-1]) = 32 when i > 1
    else i = 1;
    in toUpper(s[i]) when beginningOfWord else s[i];

titleCase("a tale of two cities") ⇒ "A Tale Of Two Cities"
```
Cells[I, J-1] + Cells[I, J+1];
in (0 when borders < 2
   else 0 when borders > 3
   else 1 when all( [Cells[I, J] = 1, some( borders = [2, 3] )] )
   else 1 when all( [Cells[I, J] = 0, borders = 3] )
   else Cells[I, J])
when all( [I /= 1, J /= 1, I /= size(Cells), J /= size(Cells[I]) ]
   else Cells[I, J];

Note that borders defined in the let are borders of an interior cell. The outer when condition shown in boldface type (whens are nested in this example) rules out computations involving the first and last rows and columns.

In order to have multiple generations, recursion is unavoidable:

\[
gol(Cells(2),N(0)) :=
\]
\[
\text{let } R := \text{gol2}(Cells);\]
\[
in \ \text{gol}(R, N-1) \text{ when } N > 0 \text{ else } R;\]

7. A more complex example

In all cases above we have seen that functions provide operators that direct the interaction of data structures with data structures. In this more complex example the interaction requires more thought and demonstrates the approach one takes to solve problems in SequenceL - that of guiding data structures interacting with data structures. We will write a simple tokenizer in SequenceL where the only delimiters are spaces. To show the interaction, consider the original string:

" the fox jumped the stream "

The first step is to identify the locations of all the delimiters:

["1the5fox9jumped16the20stream27"] to obtain \([1,5,9,16,20,27]\)

This can be accomplished by the SequenceL function:

\[
\text{spaces(string(1))}[i] := i \text{ when } \text{string}[i] = \text{'}\text{'};\]

Next we need to filter out the individual words of the original string using the indices of the spaces:

" the fox jumped the stream " × \([1,5,9,16,20,27]\)

if \(s = \text{" the fox jumped the stream "},\) then

\[
s[1+1...5-1] \Rightarrow \text{"the"} \]
\[
s[5+1...9-1] \Rightarrow \text{"fox"} \]
\[
s[9+1...16-1] \Rightarrow \text{"jumped"} \]
\[
s[16+1...20-1] \Rightarrow \text{"the"} \]
\[
s[20+1...27-1] \Rightarrow \text{"stream"} \]

to obtain

["the","fox","jumped","the","stream"]
This is accomplished using the function:

\[
\text{tokens}(\text{string}(l), \text{space}(l))[i] := \text{string}[\text{space}[i]+1...\text{space}[i+1]-1] \text{ when } i < \text{size(space)};
\]

A final function orchestrates the interaction:

\[
\text{tokenizer}(s(l)) := \text{tokens}(s, \text{spaces}(s));
\]

So altogether:

```
add s := " the fox jumped the stream ";
add spaces(string(l))[i] := i when string[i] = ' ';
add tokens(string(l),space(l))[i] := string[space[i]+1...space[i+1]-1] when i < size(space);
add tokenizer(s(l)) := tokens(s,spaces(s));
tokenizer(s) ["the","fox","jumped","the","stream"]
```

Appendix II: Regular Expression Library

The Regular Expression library offers commonly used functions for SequenceL programmers. This library can be imported using a SequenceL statement such as import <RegEx/RegEx.sl>;

Character Sets

There are a number of sets of characters described in this section.

Ordinary Characters

"ABCDEFGHIJKLMNOPQRSTUVWXYZ"  
"abcdefghijklmnopqrstuvwxyz"  
"0123456789"  
";<=@\_`{}~"  
"!#$&',-/:"  
- Space Character: \"\"  
- Tab Character: \"\t\"  
- New-Line Character: \"\n\"  
- Double Quote Character: \"\"  

Special Characters

- Backslash: '\\'  
- Period: '.'  
- Square Brackets: '[ ' '] '  
- Caret: '^'  
- Dollar Sign: '\$'  
- Asterisk: '* '  
- Plus Sign: '+'  
- Question Mark: '?'  
- Parentheses: ' ( ' ' ) '  
- Pipe: '|'  

Bracketed Character Sets

Positive Character Set

- A list of characters enclosed within Square Brackets is referred to as a Positive Character Set as long as the first character in that list is not a Caret.
- A Positive Character Set contains all characters in the list of characters.
- Example:
  "[abc]"
  Defines the Bracketed Character Set containing the characters ‘a’, ‘b’, and ‘c’.

**Negative Character Set**
- A list of characters enclosed within Square Brackets is referred to as a Negative Character Set as long as the first character in that list is a Caret.
- A Negative Character Set contains all characters in the Alphabet which are NOT in the list of characters.
- The first Caret is not part of the list of characters. It is used only to specify a Negative Character Set.
- Example:
  "[^abc]"
  Defines the Bracketed Character Set containing every character EXCEPT ‘a’, ‘b’, and ‘c’.

**Built-In Character Sets**

<table>
<thead>
<tr>
<th>RegEx String</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d</td>
<td>Contains all Numeric Characters.</td>
</tr>
<tr>
<td></td>
<td>[0123456789]</td>
</tr>
<tr>
<td>\D</td>
<td>The negation of \d. Contains all Non-Numeric Characters.</td>
</tr>
<tr>
<td>\a</td>
<td>Contains all Upper and Lower Case Letter Characters.</td>
</tr>
<tr>
<td></td>
<td>[abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]</td>
</tr>
<tr>
<td>\A</td>
<td>The negation of \a. Contains all Non-Letter Characters.</td>
</tr>
<tr>
<td>\w</td>
<td>Contains all “Word” Characters.</td>
</tr>
<tr>
<td></td>
<td>[0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]</td>
</tr>
<tr>
<td>\W</td>
<td>The negation of \w. Contains all Non-“Word” Characters.</td>
</tr>
<tr>
<td>\s</td>
<td>Contains all Whitespace Characters.</td>
</tr>
<tr>
<td></td>
<td>[Space, Tab, New-Line]</td>
</tr>
<tr>
<td>\S</td>
<td>The negation of \a. Contains all Non-Whitespace Characters.</td>
</tr>
</tbody>
</table>
Matching

A Regular Expression is a String of Ordinary and/or Special Characters which can Match another String based on the following rules.

- The Regular Expression containing only the Ordinary Character $C$ matches the string of length one containing only the character $C$.
- The Regular Expression containing only the Special Character $C$ preceded by a Backslash, matches the string of length one containing only the character $C$.
- The Regular Expression containing only the Bracketed Character Set $B$ matches the string of length one containing only the character $C$, where $C$ is a member of $B$.
- The Regular Expression containing only the Period, or Wildcard Character, matches any string of length one.
- The Regular Expression “$(X)$”, where $X$ is a Regular Expression, matches all strings matched by $X$.
- The Regular Expression “$X|Y$”, where $X$ and $Y$ are Regular Expressions, matches the string $S$, if either $X$ matches $S$ or $Y$ matches $S$.
- The Regular Expression “$XY$”, where $X$ and $Y$ are Regular Expressions, matches the string $S$, of length $N$, if there exists some integer $I$ such that $X$ matches $S[1 \ldots I]$ and $Y$ matches $S[I+1 \ldots N]$.
- The Regular Expression “$X*$”, where $X$ is a Regular Expression, matches the string $S$ if $S$ is composed only of one or more concatenated strings that are matched by $X$, or $S$ is the empty string.
- The Regular Expression “$X?$”, where $X$ is a Regular Expression, matches the string $S$ if $X$ matches $S$ or $S$ is the empty string.
- The Regular Expression “$X+$”, where $X$ is a Regular Expression, matches the string $S$ if $S$ is composed only of one or more concatenated strings that are matched by $X$.
- The Regular Expression “$^X$”, where $X$ is a Regular Expression, matches the substring $T$ of string $S$ if $T$ occurs at the beginning of $S$.
- The Regular Expression “$X\$”, where $X$ is a Regular Expression, matches the substring $T$ of string $S$ if $T$ occurs at the end of $S$.

Operation Precedence

When a regular expression contains multiple operations, they are interpreted in the following order of precedence from highest precedence to lowest.

1. Character Escaping ‘\’
2. Bracketed Character Classes ‘[ ]’
3. Parenthetical Sub-Expressions ‘( )’
4. Repetition Operators ‘*’ ‘+’ ‘?’
5. Concatenation
6. Union ‘|’
7. Beginning and Ending Anchoring ‘^’ ‘$’

Struct and Functions

This section provides a quick reference to the public struct and functions provided by the RegEx library.

Matched Struct

<table>
<thead>
<tr>
<th>Member Name</th>
<th>Member Type</th>
<th>Member Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag</td>
<td>Boolean</td>
<td>True when a valid match is found. False otherwise.</td>
</tr>
<tr>
<td>Begin</td>
<td>Integer</td>
<td>The starting index of a match if Flag is true. -1 otherwise.</td>
</tr>
<tr>
<td>End</td>
<td>Integer</td>
<td>The ending index of a match if Flag is true. -1 otherwise.</td>
</tr>
</tbody>
</table>

matches(string S, string E)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>string S</td>
<td>A String to match against the Regular Expression</td>
</tr>
<tr>
<td>Regular Expression string E</td>
<td>The Regular Expression to match against.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>Boolean</td>
<td>True if the Regular Expression E accepts the String S. False otherwise.</td>
</tr>
</tbody>
</table>

For example matches("ca", "(b|c) a?")

Returns true
firstMatch(string S, string E)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>string S</td>
<td>A String to search for a substring that matches the Regular Expression pattern.</td>
</tr>
<tr>
<td>Regular Expression string E</td>
<td>The Regular Expression to match against.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>Matched</td>
<td>A Struct ( M, (Flag, Begin, End) ), representing the results of the search. If ( M.Flag ) is True, then ( M.Begin ) and ( M.End ) are the indices in ( S ) of the first and last characters of the first substring of ( S ) string that matches ( E ). If ( M.Flag ) is False there are no substrings of ( S ) Matched by ( E ).</td>
</tr>
</tbody>
</table>

For example \( \text{firstMatch("cabcrab", "ab")} \)

Returns \( (\text{Begin:2,End:3,Flag: true}) \)

replaceFirst(string \( S_1 \), string E, string \( S_2 \))

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>string ( S_1 )</td>
<td>A String that will have the first occurrence of ( E ) replaced by ( S_2 ).</td>
</tr>
<tr>
<td>Regular Expression string E</td>
<td>The Regular Expression to match against.</td>
</tr>
<tr>
<td>string ( S_2 )</td>
<td>The new String used to replace.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>The string obtained from ( S_1 ) by replacing the first occurring substring that is accepted by ( E ) with ( S_2 ). If ( \text{FirstMatch}(S_1, E).Flag ) is true, this returns the String obtained from ( S_1 ) by replacing the characters between positions ( \text{FirstMatch}(S_1, E).Begin ) and ( \text{FirstMatch}(S_1, E).End ) with the String ( S_2 ). Otherwise it returns ( S_1 ).</td>
</tr>
</tbody>
</table>

For example \( \text{replaceFirst("cabcrab", "ab", "xyz")} \)
Returns "cxyzcrab"

**replaceAll(string S₁, string E, string S₂)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>string S₁</td>
<td>A String that will have all distinct occurrences of E replaced by S₂.</td>
</tr>
<tr>
<td>Regular Expression string E</td>
<td>The Regular Expression to match against.</td>
</tr>
<tr>
<td>string S₂</td>
<td>The new String used to replace.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>The string obtained from S₁ by replacing all distinct substrings that are</td>
</tr>
<tr>
<td></td>
<td>accepted by E with S₂.</td>
</tr>
</tbody>
</table>

For example `replaceAll("cabcrab", "ab", "xyz")`

Returns "cxyzcrxyz"

**findFirst(string S, string E)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>string S</td>
<td>A String that will be searched.</td>
</tr>
<tr>
<td>Regular Expression string E</td>
<td>The Regular Expression to match against.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>The first substring of S that is matched by E.</td>
</tr>
</tbody>
</table>

For example `findFirst("cabbcrabbb", "ab+")`

Returns "abb"
### findAll(string S, string E)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>string S</td>
<td>A String that will be searched.</td>
</tr>
<tr>
<td>Regular Expression string E</td>
<td>The Regular Expression to match against.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>List string</td>
<td>The list containing all distinct substrings of S that are matched by E.</td>
</tr>
</tbody>
</table>

For example `findAll("cabbcrabb", "ab+")`

Returns `"abb", "abbb"`
Appendix III: Utilities Library

The Utilities libraries offer several commonly used functions for SequenceL programmers. These libraries can be imported using a SequenceL statement such as `import <Utilities/Math.sl>;

Complex.sl

The Complex library provides a Complex structure type and functions that operate on complex numbers.

Complex Struct

<table>
<thead>
<tr>
<th>Member Name</th>
<th>Member Type</th>
<th>Member Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real</td>
<td>float</td>
<td>Real component of complex number.</td>
</tr>
<tr>
<td>Imaginary</td>
<td>float</td>
<td>Imaginary component of complex number.</td>
</tr>
</tbody>
</table>

makeComplex(float R, float I)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>float R</td>
<td>The real component of the complex number.</td>
</tr>
<tr>
<td>float I</td>
<td>The imaginary component of the complex number.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>Complex</td>
<td>The complex number with components R and I.</td>
</tr>
</tbody>
</table>

Example `makeComplex(1.2, 3.4)`

Returns `(Imaginary:3.4000000000, Real:1.2000000000)`

complexMagnitude(Complex C)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>Complex C</td>
<td>The complex number input.</td>
</tr>
</tbody>
</table>
Output

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>The magnitude of input $C$ is $</td>
</tr>
</tbody>
</table>

Example:

```plaintext
c := makeComplex(3.0, 4.0);
complexMagnitude(c)
```

Returns 5.0000000000

**complexConjugate(Complex C)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>Complex $C$</td>
<td>The complex number input.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>The conjugate of input $C$ is $C^\ast$.</td>
</tr>
</tbody>
</table>

Example:

```plaintext
c := makeComplex(3.0, 4.0);
complexConjugate(c)
```

Returns (Imaginary:-4.0000000000, Real:3.0000000000)

**complexExp(Complex C)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>Complex $C$</td>
<td>The complex number input.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>The exponential of input $C$ is $e^C$.</td>
</tr>
</tbody>
</table>

Example:

```plaintext
c := makeComplex(0.0, pi);
complexExp(c)
```
Returns \((\text{Imaginary:0.0000000000, Real:-1.0000000000})\)

### complexScale(Complex C, number F)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>Complex C</td>
<td>The complex multiplicand.</td>
</tr>
<tr>
<td>number F</td>
<td>The scaling factor (of type float or integer).</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>Complex</td>
<td>The product (F \times C).</td>
</tr>
</tbody>
</table>

**Example**

c := makeComplex(1.2, 3.4);

f := 10;

complexScale(c, f)

Returns \((\text{Imaginary:34.0000000000, Real:12.0000000000})\)

### complexAdd(Complex C, Complex D)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>Complex C</td>
<td>The augend is the first operand for addition.</td>
</tr>
<tr>
<td>Complex D</td>
<td>The addend is the second operand for addition.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>Complex</td>
<td>The sum (C + D).</td>
</tr>
</tbody>
</table>

**Example**

c := makeComplex(1.2, 3.4);

d := makeComplex(5.6, 7.8);

complexAdd(c, d)
Returns \((\text{Imaginary:}11.2000000000, \text{Real:}6.8000000000)\)

**complexSubtract(Complex C, Complex D)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>Complex C</td>
<td>The minuend is the first operand for subtraction.</td>
</tr>
<tr>
<td>Complex D</td>
<td>The subtrahend is the second operand for subtraction.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>Complex</td>
<td>The difference (C - D).</td>
</tr>
</tbody>
</table>

Example

\[
c := \text{makeComplex}(1.2, 3.4);
d := \text{makeComplex}(5.6, 7.8);
\text{complexSubtract}(c, d)
\]

Returns \((\text{Imaginary:}-4.4000000000, \text{Real:}-4.4000000000)\)

**complexMultiply(Complex C, Complex D)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>Complex C</td>
<td>The multiplicand is the first operand for multiplication.</td>
</tr>
<tr>
<td>Complex D</td>
<td>The multiplier is the second operand for multiplication.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>Complex</td>
<td>The product (C \times D).</td>
</tr>
</tbody>
</table>

Example

\[
c := \text{makeComplex}(1.0, 2.0);
d := \text{makeComplex}(3.0, 4.0);
\text{complexMultiply}(c, d)
\]
Returns (Imaginary:10.0000000000,Real:-5.0000000000)

**complexDivide(Complex C, Complex D)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>Complex C</td>
<td>The dividend is the first operand for division.</td>
</tr>
<tr>
<td>Complex D</td>
<td>The divisor is the second operand for division.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>Complex</td>
<td>The quotient $C / D$.</td>
</tr>
</tbody>
</table>

Example \(c := \text{makeComplex}(1.0, 2.0);\)

\(d := \text{makeComplex}(0.03, 0.04);\)

\(\text{complexDivide}(c, d)\)

Returns (Imaginary:8.0000000000,Real:44.0000000000)

**complexSum(Complex List C)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>Complex List C</td>
<td>The complex numbers to be added.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
</tbody>
</table>

Example \(c := \text{makeComplex}([1.0, 3.0, 5.0], [2.0, 4.0, 6.0]);\)

\(/* c is [(Imaginary:2.0000000000,Real:1.0000000000),\)  
\((\text{Imaginary:4.0000000000,Real:3.0000000000)}\)  
\((\text{Imaginary:6.0000000000,Real:5.0000000000})\) */\)

\(\text{complexSum}(c)\)
Returns (Imaginary:12.0000000000, Real: 9.0000000000)

**complexProduct(Complex List C)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>Complex List C</td>
<td>The complex numbers to be multiplied.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
</tbody>
</table>

Example

```c
complexProduct(makeComplex([1.0, 3.0, 5.0], [2.0, 4.0, 6.0]));
```

/* c is [(Imaginary:2.0000000000, Real:1.0000000000),
   (Imaginary:4.0000000000, Real:3.0000000000)
   (Imaginary:6.0000000000, Real:5.0000000000)] */

Returns (Imaginary:20.0000000000, Real: -85.0000000000)

### Conversion.sl

The Conversion library provides utility functions to convert values of one type to another type.

**stringToInt(String S)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>string S</td>
<td>The string to be converted.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>integer</td>
<td>The integer representing the string.</td>
</tr>
</tbody>
</table>

Example

```c
stringToInt(" -123.45 meters")
```

Returns -123
Example `stringToInt("hello")`

Returns 0

**intToString(int I)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>int I</td>
<td>The integer to be converted.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>The string representing the integer.</td>
</tr>
</tbody>
</table>

Example `intToString(-4611686018427387904) // -2^62`

Returns "-4611686018427387904"

**stringToFloat(String S)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>string S</td>
<td>The string to be converted.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>The float representing the string.</td>
</tr>
</tbody>
</table>

Example `stringToFloat(" -123.45 meters")`

Returns -123.4500000000

Example `stringToFloat("hello")`

Returns 0.0000000000
### floatToString(float F, int precision)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>float F</td>
<td>The float to be converted.</td>
</tr>
<tr>
<td>int precision</td>
<td>The number of decimal places in the float.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>The string representing the float rounded to the precision.</td>
</tr>
</tbody>
</table>

**Example** `floatToString(-1234.56, 1)`

Returns "-1234.6"

### floatToInt(float F)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>float F</td>
<td>The float to be converted.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>The integer part of the float.</td>
</tr>
</tbody>
</table>

**Example** `floatToInt(-1234.56)`

Returns -1234

### intToFloat(int I)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>int I</td>
<td>The int to be converted.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
</tbody>
</table>
float  The float representing the int.

Example  intToFloat(-4611686018427387904)  //  -2^62
Returns  -4611686018427388000.0000000000

boolToInt(bool B)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>bool B</td>
<td>The bool to be converted.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>The integer is 1 if B else 0.</td>
</tr>
</tbody>
</table>

Example  boolToInt(true)
Returns 1

intToBool(int I)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>int I</td>
<td>The int to be converted.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>The bool is true when I is non-zero.</td>
</tr>
</tbody>
</table>

Example  intToBool(-4611686018427387904)  //  -2^62
Returns  true
boolToString(bool B)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>bool B</td>
<td>The bool to be converted.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>The string representation of B.</td>
</tr>
</tbody>
</table>

Example `boolToString(true)`

Returns "true"

**Math.sl**

The Math library provides commonly used constants and utility functions for mathematical operations.

**e()**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>Euler's number.</td>
</tr>
</tbody>
</table>

Example `e()`

Returns 2.7182818285

Example `e`

Returns 2.7182818285
float Ratio of a circle’s circumference to its diameter.

Example \( \pi \)

Returns 3.1415926536

**integerPower(number N, int P)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>number ( N )</td>
<td>The float or integer base.</td>
</tr>
<tr>
<td>int ( P )</td>
<td>The non-negative integer exponent.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>number</td>
<td>The float or number exponentiation ( N^p ).</td>
</tr>
</tbody>
</table>

Example `integerPower(12, 2.99)`

Returns 144

Example `integerPower(\pi, 2)`

Returns 9.8696044011

**log(number B, number N)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>number ( B )</td>
<td>The float or integer base.</td>
</tr>
<tr>
<td>number ( N )</td>
<td>The float or integer input.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>The logarithm ( \log_B N ).</td>
</tr>
</tbody>
</table>

Example `log(10, 100)`
Returns 2.0000000000

Example $\log_\text{e}(10.0)$

Returns 2.3025850930

**round(number N)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>number N</td>
</tr>
<tr>
<td>Output</td>
<td>int</td>
</tr>
</tbody>
</table>

Example `round(10.5)`

Returns 11

Example `round(-10.5)`

Returns -11

**ceiling(number N)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>number N</td>
</tr>
<tr>
<td>Output</td>
<td>int</td>
</tr>
</tbody>
</table>

Example `ceiling(10.1)`

Returns 11

Example `ceiling(-10.1)`
Returns -10

**sign(int I)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>int I</td>
<td>The integer whose sign is computed.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>-1 if I is negative, 0 if I is 0, 1 if I is positive.</td>
</tr>
</tbody>
</table>

Example `sign(10)`

Returns 1

Example `sign(-10)`

Returns -1

**abs(number N)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>number N</td>
<td>The float or integer whose absolute value is computed.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>number</td>
<td>The float or integer absolute value</td>
</tr>
</tbody>
</table>

Example `abs(10)`

Returns 10

Example `abs(-10.1)`

Returns 10.1000000000
**min(number N, number M)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>number N</td>
<td>The first float or integer input.</td>
</tr>
<tr>
<td>number M</td>
<td>The second float or integer input.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>number</td>
<td>The float or integer minimum value of N and M.</td>
</tr>
</tbody>
</table>

Example: \( \text{min}(10, e) \)

Returns: 2.7182818285

Example: \( \text{min}(10, 10.1) \)

Returns: 10

**max(number N, number M)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>number N</td>
<td>The first float or integer input.</td>
</tr>
<tr>
<td>number M</td>
<td>The second float or integer input.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>number</td>
<td>The float or integer maximum value of N and M.</td>
</tr>
</tbody>
</table>

Example: \( \text{max}(10, e) \)

Returns: 10

Example: \( \text{max}(10, 10.1) \)

Returns: 10.1000000000
### mean(number List N)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>number List N</td>
<td>The float or integer input list containing at least one element.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>The average of values in list $N$.</td>
</tr>
</tbody>
</table>

**Example** `mean([])`

**Returns** Error: Divide by zero

**Example** `mean([3,1,4,1])`

**Returns** 2.2500000000

### median(number List N)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>number List N</td>
<td>The float or integer input list containing at least one element.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>The median of values in list $N$.</td>
</tr>
</tbody>
</table>

**Example** `median([])`

**Returns** Error: Subscripting out of bounds [][0]

**Example** `median([3,1,4,1])`

**Returns** 2.0000000000

### mode(T List L)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>

© 2009-2015, Texas Multicore Technologies, Inc.
**SequenceL Language Reference and Problem Solving Guide**

<table>
<thead>
<tr>
<th>Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List $L$</td>
<td>The input list containing at least one element of type $T$.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>The most common element of values in list $L$.</td>
</tr>
</tbody>
</table>

**Example** `mode([])`

**Returns** Error: Subscripting out of bounds $[][1]$

**Example** `mode([3,1,4,1])`

**Returns** 1

**Example** `mode("hello")`

**Returns** 'l'

**Sequence.sl**

The Sequence library adds commonly used utility functions on Sequences to the builtin functions provided by SequenceL.

**range (int Start, int Stop, int Step)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>int $Start$</td>
<td>The integer that starts the sequence.</td>
</tr>
<tr>
<td>int $Stop$</td>
<td>The integer that ends the sequence.</td>
</tr>
<tr>
<td>int $Step$</td>
<td>The non-zero integer increment between elements of the sequence.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Output</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List int</td>
<td>List containing a sequence of integers from $Start$ to $Stop$ (inclusive). This function generalizes the builtin ... function to use $Step$ other than 1.</td>
</tr>
</tbody>
</table>

**Example** `range(-10, 10, 3)`
Returns [-10,-7,-4,-1,2,5,8]

Example range(10,-11,-3)
Returns [10,7,4,1,-2,-5,-8,-11]

duplicate(T E, int N)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>T E</td>
<td>Element E of type T.</td>
</tr>
<tr>
<td>int N</td>
<td>The number of times to duplicate element E.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>List T</td>
<td>List in which the element E is duplicated N times.</td>
</tr>
</tbody>
</table>

Example duplicate('h', 3)
Returns "hhh"

firstIndexOf(T List L, T E)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>List T L</td>
<td>The input list containing elements of type T.</td>
</tr>
<tr>
<td>T E</td>
<td>Element E of type T.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>0 if the element E is not found in list L, otherwise the least index value at which E is found in L.</td>
</tr>
</tbody>
</table>

Example firstIndexOf([3,1,4,1], 5)
Returns 0
Example `firstIndexOf([3,1,4,1], 1)`

Returns 2

**lastIndexOf(T List L, T E)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>List T L</td>
<td>The input list containing elements of type T.</td>
</tr>
<tr>
<td>T E</td>
<td>Element E of type T.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>0 if the element E is not found in list L, otherwise the greatest index value at which E is found in L.</td>
</tr>
</tbody>
</table>

Example `lastIndexOf([3,1,4,1], 5)`

Returns 0

Example `lastIndexOf([3,1,4,1], 1)`

Returns 4

**indexesOf(T List L, T E)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>List T L</td>
<td>The input list containing elements of type T.</td>
</tr>
<tr>
<td>T E</td>
<td>Element E of type T.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>List int</td>
<td>List of all the index values at which E is found in L.</td>
</tr>
</tbody>
</table>

Example `indexesOf([3,1,4,1], 5)`

Returns []
Example `indexesOf([3,1,4,1], 1)`
Returns `[2,4]`

**split (T List L, T E)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>List T L</td>
<td>The input list containing elements of type T.</td>
</tr>
<tr>
<td>T E</td>
<td>Element E of type T.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>List List T</td>
<td>List of contiguous subsets of L that are separated by element E.</td>
</tr>
</tbody>
</table>

Example `split("hello there how are you", ' ')`  
Returns `"hello","","","there","how","are","you"]`

**delimit (T List List L, T E)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>List List T L</td>
<td>The input list containing lists of elements of type T.</td>
</tr>
<tr>
<td>T E</td>
<td>Element E of type T.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>List T</td>
<td>List of elements of L that are delimited by element E.</td>
</tr>
</tbody>
</table>

Example `delimit(["hello","","","there","how","are","you"], ' ')`  
Returns `"hello there how are you"`
reverse (T List L)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>List T L</td>
<td>The input list containing elements of type T.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>List T</td>
<td>List of elements of L in reverse order.</td>
</tr>
</tbody>
</table>

Example `reverse(1..10)`

Returns `[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]`

setElementAt (T List L, int I, T E)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>List T L</td>
<td>The input list containing elements of type T.</td>
</tr>
<tr>
<td>int I</td>
<td>The index of the element to be replaced.</td>
</tr>
<tr>
<td>T E</td>
<td>The new value at index I.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>List T</td>
<td>List of elements of L with element E at index I.</td>
</tr>
</tbody>
</table>

Example `setElementAt(1..10, 5, 100)`

Returns `[1, 2, 3, 4, 100, 6, 7, 8, 9, 10]`

Example `setElementAt(1..10, 11, 100)`

Returns `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`

vectorMin (number List L)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
SequenceL Language Reference and Problem Solving Guide

<table>
<thead>
<tr>
<th><strong>Inputs</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List number ( L )</td>
<td>The input list containing at least one float or integer element.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Output</strong></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>The minimum float or integer element of list ( L ).</td>
</tr>
</tbody>
</table>

Example \( \text{vectorMin}([]) \)

Returns Error: Subscripting out of bounds \([][1]\)

Example \( \text{vectorMin}([-1, 2, \pi, e]) \)

Returns \(-1.0000000000\)

\( \text{vectorMax} \ (\text{number List L}) \)

<table>
<thead>
<tr>
<th><strong>Type</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>List number ( L )</td>
<td>The input list containing at least one float or integer element.</td>
</tr>
</tbody>
</table>

Example \( \text{vectorMax}([]) \)

Returns Error: Subscripting out of bounds \([][1]\)

Example \( \text{vectorMax}([-1, 2, \pi, e]) \)

Returns \(3.1415926536\)

\( \text{last} \ (\text{T List L}) \)

<table>
<thead>
<tr>
<th><strong>Type</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>List ( T ) ( L )</td>
<td>The input list containing at least one element of type ( T ).</td>
</tr>
</tbody>
</table>

© 2009-2015, Texas Multicore Technologies, Inc.
**Output**

| T       | The last element of list L. |

**Example** `last([])`

**Returns** Error: Subscripting out of bounds `[][0]`

**Example** `last([-1, 2, pi, e])`

**Returns** `2.7182818285`

---

### allButLast (T List L)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>List T L</td>
<td>The input list containing elements of type T.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>List T</td>
<td>List containing all elements of list L except the last element.</td>
</tr>
</tbody>
</table>

**Example** `allButLast([])`

**Returns** `[ ]`

**Example** `allButLast([-1, 2, pi, e])`

**Returns** `[-1.0000000000,2.0000000000,3.1415926536]`

---

### take (T List L, int N)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>List T L</td>
<td>The input list containing elements of type T.</td>
</tr>
<tr>
<td>int N</td>
<td>Non-negative number of elements of list L to retain.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
</tbody>
</table>
List T  List containing the first N elements of list L.

Example take(1...10, 0)

Returns []

Example take([-1, 2, pi, e], 3)

Returns [-1.0000000000, 2.0000000000, 3.1415926536]

drop (T List L, int N)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>List T</td>
<td>L  The input list containing elements of type T.</td>
</tr>
<tr>
<td>int N</td>
<td>Non-negative number of elements of list L to drop.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>List T</td>
<td>List containing all but the first N elements of list L.</td>
</tr>
</tbody>
</table>

Example drop(1...10, 0)

Returns [1,2,3,4,5,6,7,8,9,10]

Example drop([-1, 2, pi, e], 3)

Returns [2.7182818285]

column (T List List L, int I)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>List List T</td>
<td>L  The input list containing lists of size N elements of type T.</td>
</tr>
<tr>
<td>int I</td>
<td>Index I between 1 and N of the column of L to extract.</td>
</tr>
</tbody>
</table>
### Output

| T | List of elements at index \( i \) within each list of list \( L \). |

Example `column([[1,2,3],[4,5,6]], 4)`

Returns Error: Subscripting out of bounds \([1,2,3][4]\)

Example `column([[1,2,3],[4,5,6]], 2)`

Returns \([2,5]\)

### `sort(number List L)`

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>List number ( L )</td>
<td>The list of float or integer elements to be sorted.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>List number</td>
<td>The list of float or integers elements in ( L ) in ascending order.</td>
</tr>
</tbody>
</table>

Example `sort([2,-1,pi,e])`

Returns \([-1.0000000000,2.0000000000,2.7182818285,3.1415926536]\)

Example `sort([2,-1,3,2])`

Returns \([-1,2,2,3]\)

### `sortBy(T List L, function F)`

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>List T ( L )</td>
<td>The list of elements of type T to be sorted.</td>
</tr>
<tr>
<td>Function</td>
<td>A function that takes a first and second element and returns -1, 0, or 1 depending on whether the first element is less than, equal to, or greater</td>
</tr>
</tbody>
</table>

© 2009-2015, Texas Multicore Technologies, Inc.
than the second element.

**Output**

<table>
<thead>
<tr>
<th>List number</th>
<th>The list of elements in ( L ) in ascending order.</th>
</tr>
</thead>
</table>

Example $dec(f, s) := \text{sign}(s - f)$;

```
sortBy([2,-1,pi,e], dec)
```

**Returns** $[3.1415926536, 2.7182818285, 2.0000000000, -1.0000000000]$

Example $inc(f, s) := \text{sign}$(asciiToInt$(f) - \text{asciiToInt}(s))$;

```
sortBy("hello there", dec)
```

**Returns** "$\text{eeehhllort}$"

**fold(T List L, function F)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>List T L</td>
<td>The list of at least one element of type T.</td>
</tr>
<tr>
<td>Function</td>
<td>An <strong>associative</strong> function that takes a first and second operand of type T and returns a result of type T.</td>
</tr>
</tbody>
</table>

**Output**

| T | The result of applying the function \( F \) to consecutive elements of list \( L \) in an unspecified order. The execution time of fold is usually less than the execution times of functions leftFold and rightFold, but fold requires an associative function. |

Example $mul(f, s) := f * s$;

```
fold(1...4, mul)
```

**Returns** $24$

Example $fold([2,-1,pi,e], \text{max})$

**Returns** $3.1415926536$
### leftFold (T List L, function F)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>List T L</td>
<td>The list of at least one element of type T.</td>
</tr>
<tr>
<td>Function</td>
<td>A function that takes a first and second operand of type T and returns a result of type T.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>The result of applying the function $F$ to consecutive elements of list $L$ from left to right.</td>
</tr>
</tbody>
</table>

**Example**

```plaintext
muladd(f, s) := 10 * f + s;

leftFold(1...3, muladd)
```

**Returns** 123

### rightFold (T List L, function F)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>List T L</td>
<td>The list of at least one element of type T.</td>
</tr>
<tr>
<td>Function</td>
<td>A function that takes a first and second operand of type T and returns a result of type T.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>The result of applying the function $F$ to consecutive elements of list $L$ from right to left.</td>
</tr>
</tbody>
</table>

**Example**

```plaintext
muladd(f, s) := 10 * f + s;

rightFold(1...3, muladd)
```

**Returns** 33
# equalListNT (T List L, T List M)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>List T L</td>
<td>The first list containing elements of type T.</td>
</tr>
<tr>
<td>List T M</td>
<td>The second list containing elements of type T.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>List bool</td>
<td>The result of applying the NT of the function <code>equalList</code> to elements of list L and list M.</td>
</tr>
</tbody>
</table>

**Example** `equalListNT([[2,1],[1,2,1],[1,1,2]],[1,1,2])`

**Returns** `[false,false,true]`

**Example** `equalListNT([1,1,2],[[2,1],[1,2,1],[1,1,2]])`

**Returns** `[false,false,true]`

## equalBagNT (T List L, T List M)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>List T L</td>
<td>The first list containing elements of type T.</td>
</tr>
<tr>
<td>List T M</td>
<td>The second list containing elements of type T.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>List bool</td>
<td>The result of applying the NT of the function <code>equalBag</code> to elements of list L and list M.</td>
</tr>
</tbody>
</table>

**Example** `equalBagNT([[2,1],[1,2,1],[1,2,2]],[1,1,2])`

**Returns** `[false,true,false]`

**Example** `equalBagNT([1,1,2],[[2,1],[1,2,1],[1,2,2]])`

**Returns** `[false,true,false]`
### equalSetNT (T List L, T List M)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>List T L</td>
<td>The first list containing elements of type T.</td>
</tr>
<tr>
<td>List T M</td>
<td>The second list containing elements of type T.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>List bool</td>
<td>The result of applying the NT of the function <code>equalSet</code> to elements of list L and list M.</td>
</tr>
</tbody>
</table>

**Example** `equalSetNT([[2,1],[1,2,1],[1,2,2]], [1,1,2])`

**Returns** `[true,true,true]`

**Example** `equalSetNT([1,1,2], [[2,1],[1,2,1],[1,2,2]])`

**Returns** `[true,true,true]`

### appendNT (T List L, T List M)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>List T L</td>
<td>The first list containing elements of type T.</td>
</tr>
<tr>
<td>List T M</td>
<td>The second list containing elements of type T.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>List T</td>
<td>The result of applying the NT of the function <code>append</code> to elements of list L and list M.</td>
</tr>
</tbody>
</table>

**Example** `appendNT([[2,1],[1,2,1],[1,2,2]], 3...5)`

**Returns** `[[2,1,3,4,5],[1,2,1,3,4,5],[1,2,2,3,4,5]]`
Set.sl

The Set library adds commonly used utility functions to the builtin operations for sets that are provided by SequenceL.

**elementOf(T E, T List L)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>T E</td>
<td>The element of type T.</td>
</tr>
<tr>
<td>List T L</td>
<td>The list containing elements of type T.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>True if E is an element of list L.</td>
</tr>
</tbody>
</table>

**Example** `elementOf(1, [])`

Returns **false**

**Example** `elementOf(1, [3,1,4,1])`

Returns **true**

**disjoint(T List L, T List M)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>List T L</td>
<td>The first list containing elements of type T.</td>
</tr>
<tr>
<td>List T M</td>
<td>The second list containing elements of type T.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>True if list L and list M have no element in common.</td>
</tr>
</tbody>
</table>

**Example** `disjoint([1], [])`

Returns **true**
Example `disjoint([1], [3,1,4,1])`

Returns `false`

**intersection(T List L, T List M)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>List T L</td>
<td>The first list containing elements of type T.</td>
</tr>
<tr>
<td>List T M</td>
<td>The second list containing elements of type T.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>List T</td>
<td>The list containing all elements common to list L and list M.</td>
</tr>
</tbody>
</table>

Example `intersection([3,1,4,1], [])`

Returns `[]`

Example `intersection([3,1,4,1], [3, 1, 4, 1, 5])`

Returns `[3,1,4]`

**union(T List L, T List M)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>List T L</td>
<td>The first list containing elements of type T.</td>
</tr>
<tr>
<td>List T M</td>
<td>The second list containing elements of type T.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>List T</td>
<td>The list containing all elements in list L and list M.</td>
</tr>
</tbody>
</table>

Example `union([3,1,4,1], [])`

Returns `[3,1,4]`
Example union([3,1,4,1], [3, 1, 4, 1, 5])

Returns [3,1,4,5]

**superSet(T List L, T List M)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>List T L</td>
<td>The first list containing elements of type T.</td>
</tr>
<tr>
<td>List T M</td>
<td>The second list containing elements of type T.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>True if list L contains all elements of list M.</td>
</tr>
</tbody>
</table>

Example superSet([], [])

Returns true

Example superSet([3,1,4], [3,1,4,1])

Returns true

**subsetNT(T List L, T List M)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>List T L</td>
<td>The first list containing elements of type T.</td>
</tr>
<tr>
<td>List T M</td>
<td>The second list containing elements of type T.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>List bool</td>
<td>List of bool where each element is true if the corresponding element of list L is a subset of the corresponding element of list M. This function applies an NT to the builtin function subset.</td>
</tr>
</tbody>
</table>

Example subsetNT([3,1,4,1],[3,1,4,5])
Returns true

Example subsetNT([[3,1,4,1],[3,1,4]], [[1,4],[3,1,4,1,5,9]])
Returns [false, true]

**powerSet(T List L)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>List T L The list containing elements of type T.</td>
</tr>
<tr>
<td>Output</td>
<td>List List T List containing all subsets of elements of list L.</td>
</tr>
</tbody>
</table>

Example powerSet([1,2,3,1,2,1,1])
Returns [[], [3], [2], [2, 3], [1], [1, 3], [1, 2], [1, 2, 3]]

**powerSetOfStrings(List string L)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>List string L The list containing strings.</td>
</tr>
<tr>
<td>Output</td>
<td>List List string List containing all subsets of strings of list L.</td>
</tr>
</tbody>
</table>

Example powerSetOfStrings(["hello", "there", "how"])
Returns [[], ["how"], ["there"], ["there", "how"], ["hello"], ["hello", "how"], ["hello", "there"], ["hello", "there", "how"]]
### cartesianProduct(T List L, T List M)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>List T L</td>
<td>The first list containing elements of type T.</td>
</tr>
<tr>
<td>List T M</td>
<td>The second list containing elements of type T.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>List T</td>
<td>A list containing the concatenation of each element from the first list L</td>
</tr>
<tr>
<td></td>
<td>with each element from the second list M.</td>
</tr>
</tbody>
</table>

**Example** `cartesianProduct([1,2],[3,4,5])`

**Returns** `[[1,4],[1,5],[2,3],[2,4],[2,5],[1,3]]`

**Example** `cartesianProduct("hi","bye")`

**Returns** `"hy","he","ib","iy","ie","hb"`

### naryCartesianProduct(T List List L)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>List List T L</td>
<td>The list containing lists of elements of type T.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>List List T</td>
<td>List containing the concatenation of each element from each of the lists in</td>
</tr>
<tr>
<td></td>
<td>L.</td>
</tr>
</tbody>
</table>

**Example** `naryCartesianProduct(["pt", "aio", "ng"])`

**Returns** `"pag","pin","pig","pon","pog","tan","tag","tin","tig","ton","tog","pan"`
dominates(T List List L, T List List M)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>List List T L</td>
<td>The first list of lists containing elements of type T.</td>
</tr>
<tr>
<td>List List T M</td>
<td>The second list of lists containing elements of type T.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>True if every list in M is a subset of at least one list in L.</td>
</tr>
</tbody>
</table>

Example dominates(["helotrw"], ["hello", "there", "how"])
Returns true

Example dominates(["helo", "trw"],["hello", "there", "how"])
Returns false

Example dominates(["helo", "thrower"],["hello", "there", "how"])
Returns true

String.sl

The String library provides utility functions for characters and strings.

isDigit(char C)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>char C</td>
<td>The character input.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>True if C is a digit between 0 and 9.</td>
</tr>
</tbody>
</table>

Example isDigit('A')
Returns false
Example `isDigit("1\t2\n")`  // Use Normalize Transposed version

Returns [true,false,true,false]

**isWhiteSpace(char C)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>The character input.</td>
</tr>
<tr>
<td>char C</td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>True if C is a space, tab, newline, or carriage return.</td>
</tr>
<tr>
<td>bool</td>
<td></td>
</tr>
</tbody>
</table>

Example `isWhiteSpace(' ')`

Returns `true`

Example `isWhiteSpace("1\t2\n")`  // Use Normalize Transposed version

Returns `[false,true,false,true]`

**trimStart(string S)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>The string to be trimmed.</td>
</tr>
<tr>
<td>string S</td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>The string S with leading whitespace removed.</td>
</tr>
<tr>
<td>string</td>
<td></td>
</tr>
</tbody>
</table>

Example `trimStart(" -123.45 meters ")`

Returns "-123.45 meters 

Example `trimStart(" \t \r \n \t ")`

Returns " "

© 2009-2015, Texas Multicore Technologies, Inc.
charCompare(char C, char D)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>char C</td>
<td>The first char input.</td>
</tr>
<tr>
<td>char D</td>
<td>The second char input.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>-1, 0, or 1 depending on whether the ASCII code of C is less than, equal to, or greater than the ASCII code of character D.</td>
</tr>
</tbody>
</table>

Example: charCompare('a', 'b')

Returns: -1

stringCompare(string S, string T)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>string S</td>
<td>The first string input.</td>
</tr>
<tr>
<td>string T</td>
<td>The second string input.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>-1, 0, or 1 depending on whether the string S is less than, equal to, or greater than the string T.</td>
</tr>
</tbody>
</table>

Example: stringCompare("a", "aa")

Returns: -1

Example: stringCompare("a", "")

Returns: 1
sortStrings(string List L)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>List string <em>L</em></td>
<td>The list of strings to be sorted.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>List string</td>
<td>The list of strings <em>L</em> in sorted order.</td>
</tr>
</tbody>
</table>

Example: `sortStrings("
["asdfg", "aa", ",", "a", ","]
Returns [",", ",", "a", "aa", "asdfg"]`
Appendix IV: BLAS and LAPACK Libraries

Wrappers to the standard linear algebra libraries BLAS and LAPACK are provided. These libraries can only be called from SequenceL programs that are compiled; they cannot be called from within the SequenceL Interpreter or the SequenceL Debugger.

BLAS

The BLAS (Basic Linear Algebra Subprograms) are routines that provide standard building blocks for performing basic vector and matrix operations. For more information on the BLAS library see Netlib (www.netlib.org/blas) or documentation for the Intel MKL (https://software.intel.com/en-us/mkl_11.2_ref)

Calling a BLAS function from within a SequenceL Program

The BLAS library can be used by importing the library <Numerical/BLAS.sl>. The function signatures should match the CBLAS (C interface to the Basic Linear Algebra Subprograms) interface. For information on the CBLAS interface, please see the Intel MKL BLAS documentation. For each signature in the CBLAS interface, there is a corresponding SequenceL function with the following rules:

- Same name as the CBLAS function
- Every scalar argument should have the same type in SequenceL as it does in the CBLAS interface
- Every argument to the function that is only an output should have a type of int(1) and contain a single integer with the expected size of the output
- Every argument to the function that contains an input and is a pointer to a floating point number should have a type of float(1)
- Every argument to the function that contains an input and is a pointer to an integer should have a type of int(1)
- Every argument to the function that contains an input and is a pointer to a character should have a type of char(1)
- Every argument to the function that contains an input and is a pointer to a complex number (represented as void* in C) should have a type of Complex(1) (The complex type can be used by importing <Utilities/Complex.sl>)
- The function returns a struct with a type name of: NAME_return_type
- The return type contains a label for each output called argN, where N is the order that the arguments appear in. For example, if the second argument to a function is an output, the return type will have a label called arg2.

Example:

The function signature for cblas_zaxpy is:
void cblas_zaxpy (const int N, const void *alpha, const void *X, const int incX, void *Y, const int incY);

Y is the only output of cblas_zaxpy.

Below is an example of calling cblas_zaxpy from a SequenceL program and returning its result:

```sequence
import <Numerical/BLAS.sl>;
import <Utilities/Complex.sl>;

makeVectorComplex(n)[i] := (Real : i + 0.5, Imaginary : i - 0.5) foreach i within 1 ... n;

test(n) :=
  let
    a := makeVectorComplex(n);
    b := makeVectorComplex(n);
    res := cblas_caxpy(n, a, 1, b, 1);
  in
    res.arg5;
```

**Linking with a BLAS library**

A program must be linked with the slnumerical library to utilize functions from the BLAS library. It must also be linked with an implementation of the CBLAS library. The CBLAS library provided by the Intel MKL is included in the SequenceL installation; however, other implementations may also be used.

To use the included MKL version of CBLAS, link the following libraries with your program:

- **Linux**: slnumerical, mkl_intel_lp64, mkl_gnu_thread, mkl_core, iomp5
- **Mac**: slnumerical, mkl_core, mkl_avx, mkl_intel_lp64, mkl_intel_thread, mkl_mc
- **Windows**: slnumerical.lib, mkl_intel_thread_dll.lib, mkl_intel_lp64_dll.lib, mkl_core_dll.lib, mkl_rt.lib libiomp5md.lib

**LAPACK**

LAPACK (Linear Algebra Package) provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. For more information on the BLAS library see Netlib (www.netlib.org/lapack) or documentation for the Intel MKL (https://software.intel.com/en-us/mkl_11.2_ref)
Calling a LAPACK function from within a SequenceL Program

The LAPACK library can be used by importing the library <Numerical/LAPACK.sl>. The function signatures should match the LAPACK C interface. For information on the LAPACK C interface, please see the Intel MKL LAPACK documentation. For each signature in the LAPACK C interface, there is a corresponding SequenceL function with the following rules:

- Same name as the LAPACK C function
- Every scalar argument should have the same type in SequenceL as it does in the LAPACK C interface
- Every argument to the function that is only an output should have a type of int(1) and contain a single integer with the expected size of the output
- Every argument to the function that contains an input and is a pointer to a floating point number should have a type of float(1)
- Every argument to the function that contains an input and is a pointer to an integer should have a type of int(1)
- Every argument to the function that contains an input and is a pointer to a character should have a type of char(1)
- Every argument to the function that contains an input and is a pointer to a complex number should have a type of Complex(1) (The complex type can be used by importing <Utilities/Complex.sl>)
- The function returns a struct with a type name of: NAME_return_type
- The return type contains a label for each output called argN, where N is the order that the arguments appear in. For example, if the second argument to a function is an output, the return type will have a label called arg2.
- The return type also contains a label called returned. This contains the value of the actual result returned by the LAPACK C function.

Example:

The function signature for LAPACKE_dgetrf is:

```plaintext
int LAPACKE_dgetrf(int matrix_layout, int m, int n,
                       double* a, int lda, int* ipiv);
```

The input arguments are matrix_layout, m, n, a and lda.

The output arguments are a and ipiv.

The function signature for LAPACKE_dgetri is:

```plaintext
int LAPACKE_dgetri(int matrix_layout, int n,
                       double* a, int lda, const int* ipiv);
```

The input arguments are matrix_layout, n, a, lda and ipiv.
The output argument is a.

Below is an example of calling LAPACKE_dgetrf and LAPACKE_dgetri from a SequenceL program and returning its result:

```sequence
import <Numerical/LAPACK.sl>;

test :=
  let
    mat := [4.0, 3.0, 6.0, 3.0];
    size_n := 2;
    factorization := LAPACKE_dgetrf(LAPACK_ROW_MAJOR, size_n, size_n, mat, size_n, [size_n]);
    inverse := LAPACKE_dgetri(LAPACK_ROW_MAJOR, size_n, factorization.arg4, size_n, factorization.arg6);
  in
    inverse.arg3 when inverse.returned = 0 else [];
```

**Linking with a LAPACK library**

A program must be linked with the slnumerical library to utilize functions from the LAPACK library. It must also be linked with an implementation of the LAPACK C library. The LAPACK C library provided by the Intel MKL is included in the SequenceL installation; however, other implementations may also be used.

To use the included MKL version of LAPACK C, link the following libraries with your program:

- **Linux**: slnumerical, mkl_intel_lp64, mkl_gnu_thread, mkl_core, iomp5
- **Mac**: slnumerical, mkl_core, mkl_avx, mkl_intel_lp64, mkl_intel_thread, mkl_mc
- **Windows**: slnumerical.lib, mkl_intel_thread_dll.lib, mkl_intel_lp64_dll.lib, mkl_core_dll.lib, mkl_rt.lib libiomp5md.lib
Appendix V: C++ helper functions

The SequenceL runtime offers helper functions that programmers can use in their C++ programs. The prototypes for these functions are obtained by using `#include <SequenceL/SequenceL.h>`

CreateHelp.h

The CreateHelp helper functions provide methods for creating 1 dimensional, 2-dimensional, and 3-dimensional Sequences containing random values between a specified minimum and maximum value. Functions are provided for Sequences of char, int, float, and double.

sl_create(int sd, T min, T max, int x, int n, Sequence<T>& result)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>int sd</td>
<td>An integer seed that is greater than 0.</td>
</tr>
<tr>
<td>T min</td>
<td>A minimum value of type T, which can be char, int, float, or double.</td>
</tr>
<tr>
<td>T max</td>
<td>A maximum value of type T, which can be char, int, float, or double.</td>
</tr>
<tr>
<td>int x</td>
<td>The number of elements required in the output Sequence.</td>
</tr>
<tr>
<td>int n</td>
<td>The number of processors to use while creating the Sequence.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>Sequence&lt;T&gt;</td>
<td>A one-dimensional sequence containing x elements where each element is</td>
</tr>
<tr>
<td></td>
<td>between min and max (inclusive).</td>
</tr>
</tbody>
</table>

Example program

```cpp
#include <iostream>
#include <SequenceL/SequenceL.h>
main(){
  int seed = 123;
  int min = 0;
  int max = 9;
  int dim1 = 3;
  int cores = 2;
  Sequence< int > result;
  sl_create(sead, min, max, dim1, cores, result);
  std::cout << result << std::endl;
}
```

Output

```
[4,4,0]
```
**sl_create(int sd, T min, T max, int x, int y, int n, Sequence<Sequence<T>>& result)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
<td></td>
</tr>
<tr>
<td>int sd</td>
<td>An integer seed that is greater than 0.</td>
</tr>
<tr>
<td>T min</td>
<td>A minimum value of type T, which can be char, int, float, or double.</td>
</tr>
<tr>
<td>T max</td>
<td>A maximum value of type T, which can be char, int, float, or double.</td>
</tr>
<tr>
<td>int x</td>
<td>The number of Sequences required in the output Sequence.</td>
</tr>
<tr>
<td>int y</td>
<td>The number of elements per Sequence required in the output Sequence.</td>
</tr>
<tr>
<td>int n</td>
<td>The number of processors to use while creating the Sequence.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Output</strong></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence&lt;Sequence&lt;T&gt;&gt;</td>
<td>A two-dimensional sequence containing x Sequences of y elements where each element is between min and max (inclusive).</td>
</tr>
</tbody>
</table>

**Example program**
```cpp
#include <iostream>
#include <SequenceL/SequenceL.h>
main()
    {
        int seed = 123;
        float min = 0.0;
        float max = 1.0;
        int dim1 = 2;
        int dim2 = 3;
        int cores = 2;
        Sequence<Sequence<float>> result;
        sl_create(seed, min, max, dim1, dim2, cores, result);
        std::cout << result << std::endl;
    }
```

**Output**
```
[[0.413908, 0.44019, 0.071586], [0.457517, 0.481425, 0.439719]]
```
**sl_create**(
	int sd, T min, T max, int x, int y, int z, int n,
	Sequence<Sequence<Sequence<T>>>& result)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>int sd</td>
<td>An integer seed that is greater than 0.</td>
</tr>
<tr>
<td>T min</td>
<td>A minimum value of type T, which can be char, int, float, or double.</td>
</tr>
<tr>
<td>T max</td>
<td>A maximum value of type T, which can be char, int, float, or double.</td>
</tr>
<tr>
<td>int x</td>
<td>The number of 2 dimensional Sequences required in the output Sequence.</td>
</tr>
<tr>
<td>int y</td>
<td>The number of Sequences required in the output Sequence.</td>
</tr>
<tr>
<td>int z</td>
<td>The number of elements per Sequence required in the output Sequence.</td>
</tr>
<tr>
<td>int n</td>
<td>The number of processors to use while creating the Sequence.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>Sequence&lt;Sequence&lt;Sequence&lt;T&gt;&gt; &amp; result</td>
<td>A three-dimensional sequence containing x Sequences of y Sequences of z elements where each element is between min and max (inclusive).</td>
</tr>
</tbody>
</table>

**Example program**
```
#include <iostream>
#include <SequenceL/SequenceL.h>
main()
{
    int seed = 123;
    char min = 'a';
    char max = 'f';
    int dim1 = 2;
    int dim2 = 3;
    int dim3 = 4;
    int cores = 2;
    Sequence<Sequence<Sequence< char >> result;
    sl_create(seed, min, max, dim1, dim2, dim3, cores, result);
    std::cout << result << std::endl;
}
```

**Output**
[["ccac","ccca","bcaf"],["afcc","afbd","bdcc"]]

FileHelp.h

The FileHelp helper functions provide methods for reading and writing Sequences from files. Functions are provided for Sequences of char, int, float, and double as well for audio files in WAV format and image files in BMP format.

\textbf{sl\_sequence2file(const Sequence\textless T\textgreater \& seq, const char \*path)}

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>const Sequence\textless T\textgreater  seq</td>
<td>A one-dimensional sequence containing elements of type T to write. Type T can be char, int, float, or double.</td>
</tr>
<tr>
<td>const char *path</td>
<td>The name of the file to write to.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>The file path is created and the sequence seq is written to the file.</td>
<td></td>
</tr>
</tbody>
</table>

Example program

```c++
#include <iostream>
#include <SequenceL/SequenceL.h>
main()
{
    Sequence< char > seq;
    seq.setSize(3);
    seq[1] = 'S';
    seq[2] = 'L';
    seq[3] = '.';
    \textcolor{red}{sl\_sequence2file(seq, "out.txt");}
}
```

Output file “out.txt” of length 3 contains SL.

\textbf{sl\_file2sequence(const char \*path, Sequence\textless T\textgreater \& result)}

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>const char *path</td>
<td>The name of the file to read.</td>
</tr>
</tbody>
</table>
Output
Sequence<T> A one-dimensional sequence containing elements of type T read from the file path. Type T can be char, int, float, or double.

Example program
```cpp
#include <iostream>
#include <SequenceL/SequenceL.h>
main(){
  Sequence< char > seq;
  sl_file2sequence("out.txt", seq);
  std::cout << "length " << seq.size() << std::endl;
}
Output
length 3
SL.
```

**sl_sequence2bmp(const Sequence<Sequence<int>>& img, const int bpp, const char *path)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>const Sequence&lt;Sequence&lt;int&gt;&gt;&amp; img</td>
<td>A two-dimensional sequence containing integers.</td>
</tr>
<tr>
<td>const int bpp</td>
<td>Bits per pixel. Expected values are 8, 16, 24, and 32.</td>
</tr>
<tr>
<td>const char *path</td>
<td>The name of the file to write to.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The file path is created and the Sequence img is written to the file as a Windows bitmap.</td>
</tr>
</tbody>
</table>

Example program
```cpp
#include <iostream>
#include <SequenceL/SequenceL.h>
main(){
  Sequence< Sequence<int> > img;
  img.setSize(2);
  img[1].setSize(3);
  img[2].setSize(3);
```
img[1][1] = 1;
img[1][2] = 2;
img[1][3] = 3;
img[2][1] = 4;
img[2][2] = 5;
img[2][3] = 6;
    sl_sequence2bmp(img, 8, "out.bmp");
}

Output file “out.bmp” of length 62 shows file type
PC bitmap, Windows 3.x format, 3 x 2 x 8

\textbf{slbmp2sequence(const char *path, int *bpp, Sequence<Sequence<int>> & result)}

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>const char *path</td>
<td>The name of the file to read. The file should be a Windows bitmap file.</td>
</tr>
<tr>
<td></td>
<td>Not all Windows bitmap files are read by this function because it works</td>
</tr>
<tr>
<td></td>
<td>only for a small subset of bitmap files.</td>
</tr>
<tr>
<td>int *bpp</td>
<td>Bits per pixel.</td>
</tr>
<tr>
<td>Sequence&lt;Sequence&lt;int&gt;&gt;</td>
<td>A two-dimensional sequence containing integers read from the file \textit{path}.</td>
</tr>
</tbody>
</table>

\textbf{Example program}

\begin{verbatim}
#include <iostream>
#include <SequenceL/SequenceL.h>
main()
{
    Sequence< Sequence <int> > img;
    int bpp;
    slbmp2sequence("out.bmp", &bpp, img);
    std::cout << "length " << img.size() << std::endl;
    std::cout << "bits per pixel " << bpp << std::endl;
    std::cout << img[1][1] << img[1][2] << img[1][3] << std::endl;
}
\end{verbatim}

\textbf{Output}

length 2
bits per pixel 8
123
456
sl_sequence2wav(const Sequence<int>& left, const Sequence<int>& right, const int bpchn, const int rate, const char *path)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td></td>
</tr>
<tr>
<td>const &lt;Sequence &lt;int&gt; &amp; left</td>
<td>A one-dimensional sequence containing integers for the left channel.</td>
</tr>
<tr>
<td>const &lt;Sequence &lt;int&gt; &amp; right</td>
<td>A one-dimensional sequence containing integers for the right channel.</td>
</tr>
<tr>
<td>const int bpchn</td>
<td>Bits per channel. Expected values are 8, 16, 24, and 32.</td>
</tr>
<tr>
<td>const int rate</td>
<td>Number of samples per second.</td>
</tr>
<tr>
<td>const char *path</td>
<td>The name of the file to write to.</td>
</tr>
<tr>
<td>Output</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The file path is created and the Sequences left and right are written to the file as a Windows WAV file.</td>
</tr>
</tbody>
</table>

Example program

```cpp
#include <iostream>
#include <SequenceL/SequenceL.h>
main(){
    Sequence< int > left;
    Sequence< int > right;
    left.setSize(2);
    left[1] = 1;
    left[2] = 2;
    right.setSize(2);
    right[1] = 3;
    right[2] = 4;
    sl_sequence2wav(left, right, 16, 8000, "out.wav");
}
```

Output file “out.wav” of length 52 shows file type
RIFF (little-endian) data, WAVE audio, Microsoft PCM, 16 bit, stereo 8000 Hz
sl_wav2sequence(const char *path, int *bits_per_chan, int *samples_per_sec, Sequence<int>& left, Sequence<int>& right)

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Inputs</strong></td>
<td></td>
</tr>
<tr>
<td>const char *path</td>
<td>The name of the file to read. The only WAV formats that are correctly read by this function are Microsoft PCM files.</td>
</tr>
<tr>
<td>int *bits_per_chan</td>
<td>Bits per channel. Expected values are 8, 16, 24, and 32.</td>
</tr>
<tr>
<td>int *samples_per_sec</td>
<td>Number of samples per second.</td>
</tr>
<tr>
<td>Sequence&lt;int&gt;&amp; left</td>
<td>A one-dimensional sequence containing integers for the left channel read from the file path.</td>
</tr>
<tr>
<td>Sequence&lt;int&gt;&amp; right</td>
<td>A one-dimensional sequence containing integers for the right channel read from the file path.</td>
</tr>
</tbody>
</table>

Example program

```c++
#include <iostream>
#include <SequenceL/SequenceL.h>
main(){
    Sequence< int > left;
    Sequence< int > right;
    int bits, samples, bits_sample;
    sl_wav2sequence("out.wav", &bits, &samples, left, right);
    std::cout << "length " << left.size() << std::endl;
    std::cout << "bits per channel " << bits << std::endl;
    std::cout << "samples per second " << samples << std::endl;
    std::cout << left[1] << left[2] << std::endl;
}
```

Output

```
length 2
bits per channel 16
samples per second 8000
12
34
```

**SLTimer.h**

The SLTimer class provides methods for measuring the execution time of C++ functions.
Class SLTimer

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void start()</td>
<td>Start measuring time.</td>
</tr>
<tr>
<td>void stop()</td>
<td>Stop measuring time.</td>
</tr>
<tr>
<td>double getTime()</td>
<td>Return measured time in seconds between start() and stop().</td>
</tr>
<tr>
<td>double timeSinceStart()</td>
<td>Return elapsed time in seconds since most recent call to start().</td>
</tr>
</tbody>
</table>

Example program

```cpp
#include <iostream>
#include <SequenceL/SequenceL.h>
main(){
    SLTimer timer;
    timer.start();
    usleep(500000);  // 0.5 seconds delay
    timer.stop();
    usleep(1000000); // 1 second delay
    std::cout << "measured " << timer.getTime() << std::endl;
    std::cout << "elapsed " << timer.timeSinceStart() << std::endl;
}
```

Output

measured 0.500771  
elapsed  1.50198