

SequenceL Interpreter User's Guide

1. Installing and Running the Interpreter

Currently there are pre-built binaries of the SequenceL interpreter that run on Mac OS X and Windows. However, any system that has a Haskell compiler should be able to build the SequenceL interpreter from source.

1.1 Mac OS X

1. If the GMP framework is not already installed on your computer:
 - a. Download GMP framework from the SequenceL homepage
 - b. Unzip the file GMP.zip
 - c. Drag the GMP framework to /System/Library/Frameworks
2. Download the SequenceL interpreter from the website.
3. Unzip sl_mac.zip.
4. Double-Click on SequenceL in the folder sl_mac.

1.2 Windows

1. Download the Windows version of the SequenceL interpreter from the website.
2. Unzip the file sl_win.zip
3. Double-click on sl.exe in the folder sl_win.

1.3 From Source

The SequenceL interpreter is written in Haskell. To compile it, the Haskell compiler GHC is needed. GHC can be downloaded from <http://www.haskell.org/ghc>.

1. Download and install GHC
2. Build the SequenceL interpreter with the command:

```
ghc -fghc-extensions -make main.hs -o sl
```
3. This will create a binary called sl. To run the interpreter, launch sl.

2. Using the Interpreter

To evaluate an expression in SequenceL, simply type in the expression at the prompt and hit return. This will output the result of the expression followed by another prompt.

To add a function definition to the current run-time environment, the add command is used. Simply type the command:

```
add f(args) := body;
```

After this is done, the function can be used in expressions to be evaluated as well as other user-defined functions. If the function f is already defined during the current run-time environment, the new version will replace the older version.

To load a text file of SequenceL function definitions, the `file` command is used.

```
file filename
```

This will add all of the function definitions in the file to the current run-time environment. Any of these functions can now be used later on.

To quit the SequenceL interpreter, type in the command `quit`.

The SequenceL interpreter can print out a trace of the evaluation of an expression. To do this trace mode must be turned on.

```
trace on
```

Trace mode can also be turned off in a similar manner.

```
trace off
```

3. SequenceL Guide

3.1. Data Types

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

The scalars that it supports are: numbers, strings and boolean values. Strings are represented as characters within quotes. There are two boolean values represented by the keywords `true` and `false`.

Lists can contain any number of items that can be scalar values, structures or other lists. Lists do not have to be a contiguous type. Below are some examples of lists.

```
[1, 2, 3 ]  
["abc", 2, true ]  
[[2, 4, 5], [5, 2, 1]]
```

A record is a group of items where each item is associated with a name. A record 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)
```

3.2 Arithmetic Operations

Basic Operations

The following basic operations are defined in SequenceL and work in the usual manner.

+, -, *, /. Below are some examples and their given result.

```
5 + 4          -> 9
3 - 4 * 5      -> -17
-3 + 2.5       -> -0.5
4 / 2          -> 2
```

Exponentiation is defined using the ^ operator.

```
3^2            -> 9
3.5 ^ 6.2      -> 2361.6838
```

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

```
5 mod 2        -> 1
24 mod 3       -> 0
```

Summations and Products of a List

A summation or product of a list can be taken by putting a + or * symbol, respectively, in front of a list.

```
+ [2, 4, 3, 12] -> 21
* [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 `f(arg1, arg2)`.

The function `floor(x)` returns the greatest integer not exceeding `x`.

```
floor(5.2)     -> 5
floor(5)       -> 5
floor(5.95)    -> 5
```

The function `sqrt(x)` returns the square root of `x`.

```
sqrt(9)        -> 3
sqrt(3)        -> 1.7320508
```

The function `ln(x)` returns the natural logarithm of its argument.

```
ln(1)          -> 0
```

```
ln(34.7) -> 3.5467398
```

The following trigonometric functions are defined in SequenceL: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`.

```
sin(0)      -> 0
cos(34.7)   -> -0.9898667
```

Built-in Constants

Both `pi` and `e` are built-in values in SequenceL.

```
pi  -> 3.1415927
e   -> 2.7182817
```

Out-of-domain Operations

If an arithmetic expression operates on values outside its domain, it will return a value of `nil`. This does not return a runtime error.

```
4 / 0      -> nil
asin(4)    -> nil
nil + 3    -> nil
"abc" + 3  -> nil
```

3.3 Conditional Operations

When Clauses

The `when` command is used to test conditions. The format for the command is: `x when y`. The `y` statement must return a boolean value. When `y` is `true`, `x` is returned, otherwise nothing is returned.

```
5 when true   -> 5
5 when false  -> empty
```

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
```

Comparison Operators

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

= and /= can be performed on any two scalars. The result will be either `true` or `false`.

```
5 = 5           -> true
5 /= 5          -> true
"abc" = "abc"   -> true
"abc" = "deb"   -> false
"abc" = 3       -> false
```

<, <=, >, and >= can only be performed on numbers. They will return either `true` or `false` if used on the correct domain. If a non-number is used in a comparison, `nil` will be returned.

```
5 < 3          -> false
5 >= 3         -> true
"abc" < 3      -> nil
```

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.

```
(5 < 3) or (5 > 3) -> true
(5 < 3) or (2 = 3) -> false
not (5 >= 3)       -> false
(5 > 3) and (3 = 3) -> true
```

Boolean Quantifiers

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

`some(A)` returns true if and only if at least one member of A is true.

`all(A)` returns true if and only if either A is empty or every member of A is true.

`none(A)` returns true if and only if no member of A is true.

```
some([ 3=2, 5>4, false]) -> true
all([ 3=2, 5>4, false])  -> false
none([ 3=2, 5>4, false]) -> false
some([])                  -> false
all([])                   -> true
none([])                  -> true
```

Boolean Operations on Lists

The function `subset(A,B)` returns true if every member of A is also a member of B.

```
subset( [1,2,3,4], [5,4,3,2,1])    -> true
subset( [1,6], [5,4,3,2,1])       -> false
```

There are three special types of equality tests for lists defined in SequenceL: `eq_list`, `eq_bag`, `eq_set`. They each take two lists as their operands and return a boolean value.

`eq_list(A,B)` returns true if the two lists are identical.

`eq_bag(A,B)` returns true if the number of occurrences of x in A is the number of occurrences of x in B.

`eq_set(A,B)` returns true if `subset(A,B)` is true and `subset(B,A)` is true.

```
eq_list([ 1, 2, 3, 4], [1, 2, 3, 4])    -> true
eq_list([ 1, 2, 3, 4], [4, 3, 2, 1])    -> false
eq_bag([ 1, 2, 3, 4], [4, 3, 2, 1])     -> true
eq_bag([ 1, 2, 1, 3], [1, 2, 3])        -> false
eq_set([ 1, 2, 1, 3], [1, 2, 3])        -> true
eq_set([ 1, 2, 3], [1, 2])              -> false
```

3.4 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.

```
[1, 2, 3] ++ [4,5]                    -> [1,2,3,4,5]
[1] ++ [] ++ ["hello", ["world"]]     -> [1, "hello", ["world"]]
"hello" ++ "world"                    -> "helloworld"
```

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, [5, 6]])                      -> 2
size([])                                -> 0
size("abc")                             -> 3
```

Subscripting

If A is a list and B is an integer, then A[B] returns the Bth item of A. Lists are always indexed starting at 1 and not at 0. If B is less than 1 or larger than size(A), then nil is returned.

```
( [ 4, 9, 1] ++ [ 6, 8] ) [ 3]      -> 1
( [ 4, 9, 1] ++ [ 6, 8] ) [10]     -> nil
"abcdef" [ 3]                       -> "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] ] -> [ 2, 5, 7]
```

A[B₁, ..., B_n] indexes into a multi-dimensional array. If n is larger than the depth of A, nil is returned.

```
[ [ 4, 3, 2], [ 6, 7, 8] ] [ 1, 2]      -> 3
[ 4, 3, 2] [ 1, 2]                      -> nil
[ [ 4, 3, 2], [ 6, 7, 8], [ 12, 13, 15] ] [ [ 1, 2], [ 2, 3] ] -> [[ 3, 2], [ 7, 8]]
```

List Functions

The function transpose(A) takes a list and returns the transpose of that list. When A is a one-dimensional list, A is returned. If A is a scalar, nil is returned.

```
transpose([ 5, 3, 6, 2])                -> [ 5, 3, 6, 2]
transpose([ [ 4, 3, 2], [ 6, 7, 8], [ 12, 13, 15] ]) -> [[ 4, 6, 12], [ 3, 7, 13], [ 2, 8, 15]]
```

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] ] [ all]      -> [[ 4, 6, 12], [ 3, 7, 13], [ 2, 8, 15]]
[ [ 4, 3, 2], [ 6, 7, 8], [ 12, 13, 15] ] [ all, 2]   -> [ 3, 7, 13]
```

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.

```
takeaway([ 1, 2, 3, 2], [ 2, 5])         -> [ 1, 3]
```

The function remdups(A) returns a list which contains only the first occurrence of an element x of A.

```
remdups([ 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. The list can be in ascending or descending order.

```
2 ... 5          -> [2,3,4,5]
5 ... 2          -> [5,4,3,2]
```

3.5 Indefinite Operations

SequenceL has support for indefinite terms. Indefinite terms can have more than one possible value. These values are connected using the | operator. For example, the following is an indefinite term that can be the values of 3, 5 or 8: 3 | 5 | 8. The order that the possible values are listed in does not matter. When an indefinite term is used as an argument to a function, another indefinite term is returned that holds all of the possible values that the function can return.

```
(3 | 5 | 8) + 4          -> 7|9|12
(3 | 5 | 8) + (4 | 13)   -> 7|16|9|18|12|21
size([2,5,6] | [1,2,3,4,5,6,7]) -> 3|7
```

The function member(A) takes a list as an argument and returns an indefinite term whose possible values are every element of A.

```
member([4, 6, 2, 8])    -> 4|6|2|8
member([])              -> nil
```

The function is_a(A,B) returns true if A is a possible value of B. B can be an indefinite term.

```
is_a(2,2)              -> true
is_a(2,4)              -> false
is_a(2, member(1...5)) -> true
is_a(8, member(1...5)) -> false
```

3.6 Structure Operations

SequenceL structures are similar to map data structures. 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: "abc").c      -> nil
(a: 3, b: (c: 4+5, d:7)).b.c -> 9
```

Structures can actually have multiple occurrences of the same label. When $A.B$ is called and there are multiple occurrences of B an indefinite term is returned. This term includes all possible values in A associated with B .

```
(a: 3, b: 6, a: 5).a      -> 3|5
```

3.7 User-Defined Functions

Defining a Function

A user can define a function in the following manner:

```
F(Arg1(Depth1), ... , Argn(Depthn)) := Body;
```

F is the name of the function.

$Arg_i(Depth_i)$ is defining an argument named Arg_i whose depth is $Depth_i$. Depth is defined to be the highest number of nested lists in 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 argument is assumed to be a scalar. If the depth of the argument can be anything, then the symbol ? is used instead of a number.

```
f(n) := n^2 + 1;
max(a,b) := a when a>b else b;
head(a(1)) := a[1] when size(a) > 0;
tail(a(1)) := a[ 2...size(a)] when size(a) > 1 else [];
```

```
f(5)          -> 26
max(5, 4)     -> 5
head([7,3,2,6]) -> 7
tail([7,3,2,6]) -> [3,2,6]
```

Defining Indexed Functions

Functions that return lists can also be defined in terms of what a value should be at a certain index. 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:

```
F(Arg1(Depth1), ..., Argn(Depthn))[Index1, ..., Indexm] := Body;
```

Index_i is a symbol used to reference the index value. The range of the index value is not specified explicitly. Instead, the range is from 1 to the size of whatever array is referenced using the index inside the body of the function. Below are some examples.

Matrix Multiply:

```
mm(a(2), b(2)) [i,j] := +(a[i,all] * b[all,j]);
```

In this example *i* ranges from 1 through the number of rows in *a* and *j* ranges from 1 through the number of columns in *b*. This function is stating that at the *i*th row and *j*th column of the matrix returned by the function *mm* the value is the body of the function. Note that this function also takes advantage of over-typed arguments since *a*[*i*, *all*] and *b*[*all*, *j*] both return lists.

Jacobi:

```
jacobi(a(2), delta)[i,j] :=
(a[i,j] when (i=1 or size(a)=i or j=1 or size(a)=j) else
(a[i+1,j]+a[i-1,j]+a[i,j+1]+a[i,j-1])/4)-
(a[i,j]*delta^2)/4;
```

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.

Let Statements

Let statements are used to define values to be used only within a function. This is useful for when the same evaluation is done 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(args) := let v1:=expression1; ... vn:=expressionn; in Body;
```

The name *v_i* can be used in *Body* to represent *expression_i*. Below is an example using let statements.

```
f(a,b) := let  x:= a+5;
              y:= x*b;
            in x+y*a;
```

```
f(3,5)      -> 128
  x         ->  8
  y         -> 40
```

3.8 Overtyped Arguments

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 putting a pair of square brackets around each one.
- 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.

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

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

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

The Transpose stage then takes the transpose of all of the arguments and performs the original function on each row of the transpose. This will return a list whose length is the length of the arguments.

Example:

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

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

$[7, 8, 9]$

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

3.9 Functions as values

SequenceL supports using functions as values. This can be used to implement more complicated functions whose arguments can be functions.

Example:

```
foldr(f,s,lst(1)) := s when size(lst)=0
                  else foldr(f, f(s,head(lst)), tail(lst));
```

In this example `foldr` takes a function that takes two arguments as its first argument. This function can be used to process a list with various functions. Below is an example to find the maximum value in a list.

```
foldr(max, 0, [7, 3, 9, 12, 4])    -> 12
```

A function can also return another function as a value. This can be useful when combined with when clauses to choose different functions to use. It is also useful when combined with structures to store different functions within a structure.

```
a := (arg1:4, fun: max);  
b := (arg1:5, fun: minimum);  
f(x,n) := x.fun(x.arg1, n);
```

```
f(a,3)    -> 4
```

```
f(b,3)    -> 3
```

3.10 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 */.