



TEXAS MULTICORE
TECHNOLOGIES, INC.

SequenceL Language Guide
Problem Solving in SequenceL
SequenceL Interpreter Reference
SequenceL Compiler Reference



Table of Contents

SequenceL Language Guide	3
1.1 Data Types	3
1.2 Arithmetic Operations	3
Basic Operations	3
Summations and Products of a List	4
Built-in Arithmetic functions.....	4
Built-in Constants	5
1.3 Conditional Operations	5
When Clauses.....	5
Comparison Operators	5
Boolean Connectives	6
Boolean Quantifiers	6
Boolean Operations on Lists	6
1.4 List Operations.....	7
Concatenation.....	7
Size().....	7
Subscripting	7
List Functions	8
Generating a List	8
1.5 Structure Operations.....	9
1.6 User-Defined Functions	9
Defining a Function.....	9
Defining Indexed Functions	10
Let Statements	11
1.7 Overtyped Arguments.....	11
1.8 Recursion	12
1.9 Commenting Code	13
Appendix I: Problem Solving in SequenceL	14
Appendix II: SequenceL Interpreter Reference	19
2 The SequenceL Interpreter	19
2.1 Using the Interpreter	19
2.2 Command Line Arguments.....	20
Appendix III: SequenceL Compiler Reference	21
3.1 Compiling a SequenceL file	21
Command Line Arguments	21
3.2 SequenceL Data Types in C++	22
SLString.....	22
Sequence.....	22
3.3 Including a SequenceL call in a C++ program.....	23
3.4 Building a C++ Program with SequenceL functions.....	24



SequenceL Language Guide

1.1 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 are represented by the keywords **true** and **false**.

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

1.2 Arithmetic Operations

Basic Operations

The operations **+**, **-**, *****, and **/** 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
```

Types of operations are inferred by their arguments. Therefore,:

```
1/2             ⇒ 0
1.0/2          ⇒ 0.5
```



Exponentiation is defined using the ^ operator.

```
3^2           ⇒ 9
3.5 ^ 6.2     ⇒ 2361.684476568993
```

Mixed mode operations are discouraged. For example

```
1.2 * 10^52   ⇒ 0.0
1.2*10.0^52   ⇒ 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       ⇒ 1
24 mod 3      ⇒ 0
```

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 **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)       ⇒ 3.0
sqrt(3)       ⇒ 1.7320508
```

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

```
ln(1)         ⇒ 0.0
ln(34.7)      ⇒ 3.546739687
```



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

```
sin(0)           ⇒ 0.0
cos(34.7)        ⇒ -0.9898667
tan(1)           ⇒ 1.5574077246549023
atan(0.5)        ⇒ 0.4636476090008061
acos(0.9)        ⇒ 0.4510268117962624
asin(0.7)        ⇒ 0.7753974966107531
```

Built-in Constants

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

```
pi              ⇒ 3.1415927
e               ⇒ 2.7182818
```

1.3 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 **/=**.

```
5 = 5           ⇒ true
5 /= 5          ⇒ false
"abc" = "abc"   ⇒ [true, true, true]
"abc" = "deb"   ⇒ [false, false, false]
```

<, **<=**, **>**, 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.

```
(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
subset("ac", "abc")           ⇒ true
subset("ad", "abc")           ⇒ false
```

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: **eq_list**, **eq_bag**, and **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
eq_list("abc", "abc")                   => true
eq_list("abc", "ac")                    => false
```

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

1.4 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, 2, 3]                           => [1,2,3]
"hello" ++ "world"                        => "helloworld"
```

The function **appends** can also be used to concatenate all of the items in a list together.

```
appends([[1,2,3],[4,5,6],[7,8,9]])       => [1,2,3,4,5,6,7,8,9]
```

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 **Bth** 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.

```
([4, 9, 1] ++ [6, 8])[3]      ⇒ 1
"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.

```
[[4, 3, 2], [6, 7, 8]][1, 2]      ⇒ 3
[[4, 3, 2], [6, 7, 8], [12, 13, 15]][[1, 2], [2, 3]] ⇒ [[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.

```
head([1, 2, 3, 4])      ⇒ 1
tail([1, 2, 3, 4])     ⇒ [2, 3, 4]
```

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

```
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, 2]      ⇒ [3, 7, 13]
[[[4, 3, 2], [6, 7, 8], [12, 13, 15]][all]]       ⇒ [[4, 6, 12], [3, 7, 13], [2, 8, 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**.

```
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.

```

2 ... 5           ⇒ [2,3,4,5]
5 ... 2           ⇒ []
                    (No descending lists)
((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]
                    (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"

```

1.5 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

```

1.6 User-Defined Functions

Defining a Function

A user can define a function in the following manner:

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

where **F** is the name of the function.

Arg_i (Depth_i) defines an argument named **Arg_i** whose depth is **Depth_i**. **Depth_i** 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.



Examples:

```

f(n) := n^2 + 1;
max(a,b) := a when a>b else b;
hd(a(1)) := a[1] when size(a) > 0;
tl(a(1)) := a[ 2...size(a)] when size(a) > 1 else [];
add any(a(?)) := a; (In interpreter)

f(5)           => 26
max(5, 4)      => 5
hd([7, 3, 2, 6]) => 7
tl([7, 3, 2, 6]) => [3, 2, 6]
any(1)         => 1
any([1, 2, 3]) => [1, 2, 3]
any([[1, 2, 3], [5, 6, 7]]) => [[1, 2, 3], [5, 6, 7]]

```

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] := sum(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 ith row and jth 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 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(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
```

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

```
eigen(m(2),v(1)) :=  
  let s:=size(v); e := sum(m * v); v1 := e/e[s];  
  in  eigen(m,v1)  
  when all(abs(v[1...s-1] - v1[1...s-1]) > 0.00999999)  
  else v1;
```

Functions may also return structures:

```
f(v(1),m(2)) := (x:v, y:m);
```

and values referenced in other functions:

```
g(n(0)) := f([1,2,3],["abc","def"]).x+n;  
  
g(3)    ⇒ [4,5,6]
```

Structures can also contain functions:

```
f(n(0)) := (fact:f(n-1).fact*n when n > 1 else n);  
f(3).fact ⇒ 6
```

1.7 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.

1.8 Recursion

SequenceL supports recursion. For example:

```
fact(n(0)) := fact(n-1)*n when n>0 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(1)) :=  
  qsort(filtermin(head(x), tail(x)))  
  ++[head(x)]  
  ++qsort(filtermax(head(x), tail(x)))  
when size(x) > 1 else x;
```

1.9 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 `*/`.



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]. For more information on the computational laws the reader is referred to [TOPLAS].

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 [DAMP]. 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,
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: `member(X(0),S(1)) := true when some(X = S) else false;`

2. Subset:

Definition: $S1 \subseteq S2$ is true if all elements of $S1$ are also elements of $S2$ otherwise false

SequenceL: `subset(S1(1),S2(1)) := true when all(member(S1,S2)) else false;`

3. Even Numbers:

Definition: $\{ x \mid x \in I \ \& \ x \bmod 2 = 0 \}$

SequenceL; `even(X(0)) := X when $X \bmod 2 = 0$;`

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

4. Matrix Multiplication

Definition: $(AB)_{i,j} = \sum_{r=1}^n A_{i,r} B_{r,j}$

SequenceL; `matmul(A(2),B(2))[i,j] := 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. 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 neighbours dies, as if caused by underpopulation.
2. Any live cell with more than three live neighbours dies, as if by overcrowding.
3. Any live cell with two or three live neighbours lives on to the next generation.
4. Any dead cell with exactly three live neighbours 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:

```
gol2(Cells(2)) [I, J] :=
  let borders := sum(Cells[I-1, J-1...J+1]) +
    sum(Cells[I+1, J-1...J+1]) +
    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/=5, J/=10])
  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)) :=
  let R := gol2(Cells);
  in gol(R, N-1) when N > 0
  else R;
```



6. 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:

```
spaces(string(1))[i] := i when all(string[i] = " ");
```

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

```
" the fox jumped the stream "   X   [1,5,9,16,20,27]
if s = " the fox jumped the stream ", then
s[1+1...5-1]                     ⇒ "the"
s[5+1...9-1]                     ⇒ "fox"
s[9+1...16-1]                    ⇒ "jumped"
s[16+1...20-1]                   ⇒ "the"
s[20+1...27-1]                   ⇒ "stream"
```

to obtain

```
["the","fox","jumped","the","stream"]
```

This is accomplished using the function:

```
tokens(string(1),space(1))[i] := string[space[i]+1...space[i+1]-1]
when i <size(space);
```

A final function orchestrates the interaction:

```
tokenizer(s(1)) := tokens(s,spaces(s));
```

So altogether:

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



```
cmd:>add tokenizer(s(1)) := tokens(s,spaces(s));  
cmd:>tokenizer(s)  
["the","fox","jumped","the","stream"]  
cmd:>
```

Other examples:

```
ceiling(x(0)):=floor(x) + 1 when floor(x) /= x else x;  
abs(n(0)) := n when n >= 0 else -n;  
exp(n(0)) := e^n;
```

Vector Cross Product:

```
veccross(M(2)):=determ(deletej(M,1...size(M[1])));  
determ(M(2)):= (M[1,1] * M[2,2]) - (M[2,1] * M[1,2]);  
deletej(M(2),J(0)) [I,C] := M[I,C] when J/=C;
```



Appendix II: SequenceL Interpreter Reference

2 The SequenceL Interpreter

2.1 Using the Interpreter

To start the interpreter, type **sli** (under Windows) or **./sli** (under Linux) into the command line.

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
```

The trace can also be outputted to a file. Enter the file name after a trace on command:

```
trace on filename
```

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

```
trace off
```



2.2 Command Line Arguments

The interpreter has several command line options.

```
-l filename or --load filename
```

Use this option to load a file of SequenceL function definitions into the interpreter. This option can be used multiple times to load multiple files. The files will be loaded in the order that they are given at the command line.

```
-c "expression" or --command "expression"
```

Use this option to execute a SequenceL expression. This option can be used multiple times to execute multiple commands. The commands will be executed in the order that they are given at the command line.

```
-x or --quit
```

Use this flag to evaluate the expressions given at the command line without starting the interpreter.

Examples:

```
sli -l examples.sl -l examples2.sl
```

Will load `examples.sl`, the load `examples2.sl`, then start the interpreter.

```
sli -x -l examples.sl -c "f(5)"
```

Will load the file, "`examples.sl`," execute the function call, "`f(5)`," print out the result, then quit.

The SequenceL compiler takes a SequenceL file as input and returns a C++ header file and a C++ source code file. These files can then be included in a C++ project and compiled using a C++ compiler.



Appendix III: SequenceL Compiler Reference

3.1 Compiling a SequenceL file

To compile a SequenceL file, call the following from the command line:

```
sl [-parallel] -c file_name -f "intended_use" -o output_name
```

Command Line Arguments

-parallel

Use this option to generate parallel code. If this option is not given, only sequential code will be generated.

-c file_name

file_name should be the name of the file to compile.

-f "intended_use"

intended_use is the name of the function that you want to call along with the input that you will be passing it. The format for intended use is:

```
function_name(type(depth), ... type(depth))
```

Example:

```
f(int(1), string)
```

The supported types are: **int**, **float**, **string**, **bool**, and **struct**.

If the user wants to specify the types for certain labels in a problem, they should enter:

```
f(int(1),char(0)) where (a:int(1), b:float(1)) -o output_name
```

output_name is the name of the C++ files outputted. No file extension is needed.

Compiler Example

```
sl -parallel -c example.sl -f "f(int(1), string)" -o example
```

This will generate two files, **example.cpp** and **example.h**.



3.2 SequenceL Data Types in C++

There are two special data types for SequenceL that can be used within a C++ program. These are **Sequence** and **SLString**.

SLString

SLString is used in the same manner as the built in string class in C++. Any time a string needs to be passed to a SequenceL function, it should be as an **SLString**.

Sequence

The **Sequence** data structure is very similar to the vector data structure in the C++ STL. It is used to represent sequences in C++. **Sequences** use C++ templates to specify the type of the **Sequence**.

Example

```
Sequence<int> a;  
Sequence< Sequence <double> > b;  
Sequence< char > haystack;
```

Note: If doing any hand coding/tuning, take care to put spaces around the <> symbols in a nested Sequence. The C++ compiler will parse "Sequence<Sequence<int>>" as ending with a right bit-shift operator.

To specify the size of a **Sequence**, use the method **setSize(int)**. A **Sequence** is indexed using the **[]** operator. The first element of a **Sequence** is at index 1, not at index 0.

Example

```
a.setSize(10);  
a[3] = 6;  
a[10] = 23;  
  
b.setSize(2);  
b[1].setSize(5);  
b[2].setSize(7);  
b[1][3] = 6.2;  
b[2][6] = 7.8;  
  
needle.setSize(3);  
needle[1] = 'a';  
needle[2] = 'a';  
needle[3] = 'l';
```



To obtain the length of a **Sequence**, use the method `size()`.

Example

```
a.size();
```

Be very careful declaring the size of a **Sequence** and accessing its elements.

In the example above, if you access `a[0]` or `a[11]` it will cause a runtime error.

3.3 Including a SequenceL call in a C++ program

To use SequenceL functions in a C++ program, include all of the generated headers.

Example:

```
#include "example.h"
```

The format for calling a SequenceL function from within a C++ program is:

```
sl_function_name(arg1, ... arg_n, num_cores, result);
```

num_cores is an integer that specifies how many cores to use to execute the function.

Example:

```
Sequence<int> a;  
a.setSize(3);  
a[1] = 4;  
a[2] = 16;  
a[3] = -7;  
  
SLString b = "test";  
  
int c;  
  
sl_f(a,b,2,c);  
  
cout<<c<<endl;
```



3.4 Building a C++ Program with SequenceL functions

The **SequenceL** directory and **libsequencel.a** must be in the same directory as the files being built.

Every source code file generated by the SequenceL compiler must be compiled by a C++ compiler.

When building the program, the **pthread** library must be included.

Example (Using gcc)

```
g++ -c example.cpp
g++ -c main.cpp
g++ -lpthread main.o example.o libsequencel.a -o example
```