Mass spectrometry assay optimization using functional programming patterns in Lua

Bennett Kalafut
1. Mass Spectrometry and Lua
   a) Mass spectrometry basics (what, why, and how?)
   b) Lua as a control language

2. Iterator pipelines and higher-order table functions
   a) Core functionals: Map, filter, and reduce
   b) Lua-specific patterns
   c) Warm-up 1: Tuning up an ion source
   d) Warm-up 2: Check some electronics

3. Automated assay optimization
   a) Why per-assay optimization?
   b) Optimizations as composable functions
   c) How to hide/handle state
   d) Putting it all together
What is mass spectrometry?

Separation and quantification of ions in a mixture, by mass.

- An electrical or electromagnetic technique. Separation by m/z, not m.
What is mass spectrometry?

Separation and quantification of ions in a mixture, by mass.

- An electrical or electromagnetic technique. Separation by m/z, not m.
- Applications: Chemical/pharmaceutical manufacture, chemical/pharmaceutical R&D, process monitoring, food safety, toxicology, forensics, materials science, environmental monitoring, athletics (anti-doping), histology (medical),…
Separation and quantification of ions in a mixture, by mass.

- An electrical or electromagnetic technique. Separation by m/z, not m.
- Applications: Chemical/pharmaceutical manufacture, chemical/pharmaceutical R&D, process monitoring, food safety, toxicology, forensics, materials science, environmental monitoring, athletics (anti-doping), histology (medical),…
- Proteomics, metabolomics, glycomics, cell biology, genomics,…

What is mass spectrometry?

![Mass spectrum diagram](image-url)
Separation and quantification of ions in a mixture, by mass.

- An electrical or electromagnetic technique. Separation by m/z, not m.
- Applications: Chemical/pharmaceutical manufacture, chemical/pharmaceutical R&D, process monitoring, food safety, toxicology, forensics, materials science, environmental monitoring, athletics (anti-doping), histology (medical),…
- Proteomics, metabolomics, glycomics, cell biology, genomics,…
- Future: Surgery, personalized medicine.
The modern standard technique: Tandem MS

- MS/MS: Separate, fragment, separate again
The modern standard technique: Tandem MS

- MS/MS: Separate, fragment, separate again
The modern standard technique: Tandem MS

- MS/MS: Separate, fragment, separate again
The modern standard technique: Tandem MS

- MS/MS: Separate, fragment, separate again
- a.k.a. MS². MSⁿ is possible with ion traps.
The modern standard technique: Tandem MS

- MS/MS: Separate, fragment, separate again
- a.k.a. MS$^2$. MS$^n$ is possible with ion traps.
- Greater specificity than single-stage MS
The modern standard technique: Tandem MS

- MS/MS: Separate, fragment, separate again
- a.k.a. MS$^2$. MS$^n$ is possible with ion traps.
- Greater specificity than single-stage MS
- Excitation by collision, electron transfer, or laser pulse
• MS/MS: Separate, fragment, separate again
• a.k.a. MS². MSⁿ is possible with ion traps.
• Greater specificity than single-stage MS
• Excitation by collision, electron transfer, or laser pulse
• MS/MS: Separate, fragment, separate again
  
  • a.k.a. MS\(^2\). MS\(^n\) is possible with ion traps.

• Greater specificity than single-stage MS

• Excitation by collision, electron transfer, or laser pulse

• Often coupled with other separation techniques:
  
  • Liquid chromatography
  
  • Gas chromatography
  
  • Ion mobility
  
  • FAIMS
Thermo Fisher Scientific Lua-Controlled Mass Spectrometers

**TSQ Endura™, TSQ Quantiva™, Endura MD™ triple-stage quadrupole mass spectrometers**

- Unit mass to 0.2 amu resolution
- Triple quadrupole, high throughput
- Quantitation or search for known targets

**Orbitrap Fusion™, Orbitrap Fusion Lumos™ hybrid mass spectrometers**

- Resolving power (m/Δm): 500,000
- Quadrupole filter, ion trap, and Orbitrap (Tribrid™ architecture)
- De novo discovery and identification
Thermo Fisher Scientific Lua-Controlled Mass Spectrometers

**TSQ Endura™, TSQ Quantiva™, Endura MD™ triple-stage quadrupole mass spectrometers**

- Unit mass to 0.2 amu resolution
- Triple quadrupole, high throughput
- Quantitation or search for known targets

**Orbitrap Fusion™, Orbitrap Fusion Lumos™ hybrid mass spectrometers**

- Resolving power \((m/Δm): 500,000\)
- Quadrupole filter, ion trap, and Orbitrap (Tribrid™ architecture)
- De novo discovery and identification
Thermo Fisher Scientific Lua-Controlled Mass Spectrometers

**TSQ Endura™, TSQ Quantiva™, Endura MD™ triple-stage quadrupole mass spectrometers**

- Unit mass to 0.2 amu resolution
- Triple quadrupole, high throughput
- Quantitation or search for known targets

**Orbitrap Fusion™, Orbitrap Fusion Lumos™ hybrid mass spectrometers**

- Resolving power (m/Δm): 500,000
- Quadrupole filter, ion trap, and Orbitrap (Tribrid™ architecture)
- De novo discovery and identification
1. Control device (on/off, system voltage, controller target) getters/setters and readback device getters implemented in device objects. Device objects are lightweight userdatas.

Lua for instrument control: how Thermo does it.
1. Control device (on/off, system voltage, controller target) getters/setters and readback device getters implemented in device objects. Device objects are lightweight userdatas.

2. Table of voltage settings appropriate for scans are held in a (virtual) calibration file.
1. Control device (on/off, system voltage, controller target) getters/setters and readback device getters implemented in device objects. Device objects are lightweight user datas.

2. Table of voltage settings appropriate for scans are held in a (virtual) calibration file.

3. Voltages during scan are stored in a “scan matrix” run at high speed by the DSP. Interface provided by dedicated matrix builder functions.
1. Control device (on/off, system voltage, controller target) getters/setters and readback device getters implemented in device objects. Device objects are lightweight userdatas.

2. Table of voltage settings appropriate for scans are held in a (virtual) calibration file.

3. Voltages during scan are stored in a “scan matrix” run at high speed by the DSP. Interface provided by dedicated matrix builder functions.

4. Scan matrix is defined by a smaller set of scan state variables. Scan setup functions provided in Lua. Experiment loops and scan execution function insulate most tasks from scan matrix manipulation and DSP control.
1. Control device (on/off, system voltage, controller target) getters/setters and readback device getters implemented in device objects. Device objects are lightweight userdatas.

2. Table of voltage settings appropriate for scans are held in a (virtual) calibration file.

3. Voltages during scan are stored in a “scan matrix” run at high speed by the DSP. Interface provided by dedicated matrix builder functions.

4. Scan matrix is defined by a smaller set of scan state variables. Scan setup functions provided in Lua. Experiment loops and scan execution function insulate most tasks from scan matrix manipulation and DSP control.

5. Data query primitives provide information about the buffered spectrum to Lua.
for i = 1, totalScans do
  -- acquire TIC for DRAG_1 = 100V DRAG_2 = 0V
  if Sys.Abort()==true then
    RestoreDragCellVoltages()
    print("Aborted by user.")
    error(Diag.exceptions.ABORT)
  end
  xvalues[i]=i
  CF2:SetAndUpdate(100, DRAG_1)
  CF2:SetAndUpdate(0, DRAG_2)
  Sys.TakeAScan() -- take one scan to warm up the MS
  Sys.TakeAveragedScan(5)
  arrayTICforDRAG_1at100V[i] = DQ:TIC();
  addedTICforDRAG_1at100V = addedTICforDRAG_1at100V + arrayTICforDRAG_1at100V[i];
  -- acquire TIC for DRAG_1 = 0V DRAG_2 = 100V
  CF2:SetAndUpdate(0, DRAG_1)
  CF2:SetAndUpdate(100, DRAG_2)
  Sys.TakeAScan() -- take one scan to warm up the MS
  Sys.TakeAveragedScan(5);
  arrayTICforDRAG_2at100V[i] = DQ:TIC();
  addedTICforDRAG_2at100V = addedTICforDRAG_2at100V + arrayTICforDRAG_2at100V[i];
end
Lua in the mass spectrometer: first example

```
for i = 1, totalScans do
  -- acquire TIC for DRAG_1 = 100V DRAG_2 = 0V
  if Sys.Abort()==true then
    RestoreDragCellVoltages()
    print("Aborted by user.")
    error(Diag.exceptions.ABORT)
  end
  xvalues[i]=i
  CF2:SetAndUpdate(100, DRAG_1)
  CF2:SetAndUpdate(0, DRAG_2)
  Sys.TakeAScan()  -- take one scan to warm up the MS
  Sys.TakeAveragedScan(5)
  arrayTICforDRAG_1at100V[i] = DQ:TIC();
  addedTICforDRAG_1at100V = addedTICforDRAG_1at100V + arrayTICforDRAG_1at100V[i];
  -- acquire TIC for DRAG_1 = 0V DRAG_2 = 100V
  CF2:SetAndUpdate(0, DRAG_1)
  CF2:SetAndUpdate(100, DRAG_2)
  Sys.TakeAScan()  -- take one scan to warm up the MS
  Sys.TakeAveragedScan(5);
  arrayTICforDRAG_2at100V[i] = DQ:TIC();
  addedTICforDRAG_2at100V = addedTICforDRAG_2at100V + arrayTICforDRAG_2at100V[i];
end
```

Imperative control is possible. Easy for casual programmers, but inefficient.
Lua in the mass spectrometer: first example

for i = 1, totalScans do
    -- acquire TIC for DRAG_1 = 100V DRAG_2 = 0V
    if Sys.Abort() == true then
        RestoreDragCellVoltages()
        print("Aborted by user.")
        error(Diag.exceptions.ABORT)
    end
    xvalues[i] = i
    CF2:SetAndUpdate(100, DRAG_1)
    CF2:SetAndUpdate(0, DRAG_2)
    Sys.TakeAScan() -- take one scan to warm up the MS
    Sys.TakeAveragedScan(5)
    arrayTICforDRAG_1at100V[i] = DQ:TIC();
    addedTICforDRAG_1at100V = addedTICforDRAG_1at100V + arrayTICforDRAG_1at100V[i];

    -- acquire TIC for DRAG_1 = 0V DRAG_2 = 100V
    CF2:SetAndUpdate(0, DRAG_1)
    CF2:SetAndUpdate(100, DRAG_2)
    Sys.TakeAScan() -- take one scan to warm up the MS
    Sys.TakeAveragedScan(5);
    arrayTICforDRAG_2at100V[i] = DQ:TIC();
    addedTICforDRAG_2at100V = addedTICforDRAG_2at100V + arrayTICforDRAG_2at100V[i];
end

Imperative control is possible. Easy for casual programmers, but inefficient.

Components here:
• System voltage table manipulation
for i = 1, totalScans do
    -- acquire TIC for DRAG_1 = 100V DRAG_2 = 0V
    if Sys.Abort() == true then
        RestoreDragCellVoltages()
        print("Aborted by user.")
        error(Diag.exceptions.ABORT)
    end

    xvalues[i]=i
    CF2:SetAndUpdate(100, DRAG_1)
    CF2:SetAndUpdate(0, DRAG_2)
    Sys.TakeAScan() -- take one scan to warm up the MS
    Sys.TakeAveragedScan(5)
    arrayTICforDRAG_1at100V[i] = DQ:TIC();
    addedTICforDRAG_1at100V = addedTICforDRAG_1at100V + arrayTICforDRAG_1at100V[i];

    -- acquire TIC for DRAG_1 = 0V DRAG_2 = 100V
    CF2:SetAndUpdate(0, DRAG_1)
    CF2:SetAndUpdate(100, DRAG_2)
    Sys.TakeAScan() -- take one scan to warm up the MS
    Sys.TakeAveragedScan(5);
    arrayTICforDRAG_2at100V[i] = DQ:TIC();
    addedTICforDRAG_2at100V = addedTICforDRAG_2at100V + arrayTICforDRAG_2at100V[i];
end

Imperative control is possible. Easy for casual programmers, but inefficient.

Components here:
- System voltage table manipulation
- Scan dispatch
for i = 1, totalScans do

-- acquire TIC for DRAG_1 = 100V DRAG_2 = 0V
if Sys.Abort()==true then
    RestoreDragCellVoltages()
    print("Aborted by user."")
    error(Diag.exceptions.ABORT)
end

xvalues[i]=i

CF2:SetAndUpdate(100, DRAG_1)
CF2:SetAndUpdate(0, DRAG_2)
Sys.TakeAScan() -- take one scan to warm up the MS
Sys.TakeAveragedScan(5)
arrayTICforDRAG_1at100V[i] = DQ:TIC();
addedTICforDRAG_1at100V = addedTICforDRAG_1at100V + arrayTICforDRAG_1at100V[i];

-- acquire TIC for DRAG_1 = 0V DRAG_2 = 100V
CF2:SetAndUpdate(0, DRAG_1)
CF2:SetAndUpdate(100, DRAG_2)
Sys.TakeAScan() -- take one scan to warm up the MS
Sys.TakeAveragedScan(5);
arrayTICforDRAG_2at100V[i] = DQ:TIC();
addedTICforDRAG_2at100V = addedTICforDRAG_2at100V + arrayTICforDRAG_2at100V[i];
end

Imperative control is possible. Easy for casual programmers, but inefficient.

Components here:
• System voltage table manipulation
• Scan dispatch
• Data query
for i = 1, totalScans do
  -- acquire TIC for DRAG_1 = 100V DRAG_2 = 0V
  if Sys.Abort() == true then
    RestoreDragCellVoltages()
    print("Aborted by user.")
    error(Diag.exceptions.ABORT)
  end
  xvalues[i] = i

  CF2:SetAndUpdate(100, DRAG_1)
  CF2:SetAndUpdate(0, DRAG_2)
  Sys.TakeAScan() -- take one scan to warm up the MS
  Sys.TakeAveragedScan(5)

  arrayTICforDRAG_1at100V[i] = DQ:TIC();
  addedTICforDRAG_1at100V = addedTICforDRAG_1at100V + arrayTICforDRAG_1at100V[i];
  -- acquire TIC for DRAG_1 = 0V DRAG_2 = 100V
  CF2:SetAndUpdate(0, DRAG_1)
  CF2:SetAndUpdate(100, DRAG_2)
  Sys.TakeAScan() -- take one scan to warm up the MS
  Sys.TakeAveragedScan(5);
  arrayTICforDRAG_2at100V[i] = DQ:TIC();
  addedTICforDRAG_2at100V = addedTICforDRAG_2at100V + arrayTICforDRAG_2at100V[i];
end

Imperative control is possible. Easy for casual programmers, but inefficient.

Components here:
• System voltage table manipulation
• Scan dispatch
• Data query

Not shown:
• Scan matrix builder
• Direct voltage control
Core higher-order functions: map, filter, and reduce

- The goal: Operate on data, don’t explicitly control execution. (More declarative, less imperative.)
- Build a program by composing operations.
Core higher-order functions: map, filter, and reduce

- The goal: Operate on data, don’t explicitly control execution. (More declarative, less imperative.)
- Build a program by composing operations.
- Core tools:
  - **Map**
    Applies a function to each element of the input. *(Maps domain points to range points.)*
    \[
    \text{Map}(f, \{a, b, c, \ldots\}) \rightarrow \{f(a), f(b), f(c), \ldots\}
    \]
The goal: Operate on data, don’t explicitly control execution. (More declarative, less imperative.)

Build a program by composing operations.

Core tools:

- **Map**
  Applies a function to each element of the input. (*Maps* domain points to range points.)
  \[
  \text{Map}(f, \{a, b, c, \ldots\}) \rightarrow \{f(a), f(b), f(c), \ldots\}
  \]

- **Filter**
  Removes input elements from the output if they do not satisfy a provided predicate.
  \[
  \text{Filter}(f, \{a, b, c, \ldots\}) \rightarrow \{f(a) \text{ and } a \text{ or } \text{nil}, f(b) \text{ and } b \text{ or } \text{nil}, \ldots\}
  \]
The goal: Operate on data, don’t explicitly control execution. (More declarative, less imperative.)

Build a program by composing operations.

Core tools:

- **Map**
  Applies a function to each element of the input. \((\text{Maps domain points to range points.})\)
  \[
  \text{Map}(f, \{a, b, c, \ldots\}) \rightarrow \{f(a), f(b), f(c), \ldots\}
  \]

- **Filter**
  Removes input elements from the output if they do not satisfy a provided predicate.
  \[
  \text{Filter}(f, \{a, b, c, \ldots\}) \rightarrow \{f(a) \text{ and } a \text{ or nil}, f(b) \text{ and } b \text{ or nil}, \ldots\}
  \]

- **Reduce (left fold)**
  Applies a function pairwise from left to right, reducing the set to a single value.
  Sums, products, projections, and multi-function composition are some examples.
  \[
  \text{Reduce}(f, \{a, b, c, \ldots\}) \rightarrow f(...) (f(f(a, b), c), \ldots)
  \]
Composable iterators (iterator pipelines)

Iterator pipelines allow the series of operations to be applied one element at a time.

- **Map:**
  
  ```lua
  function Map(fun,first,second,third)
    local itercoroutine=function ()
      for value in (function () return first,second,third end)() do
        coroutine.yield(fun(value))
      end
    end
    return coroutine.wrap(itercoroutine),nil,nil
  end
  ```
Composable iterators (iterator pipelines)

Iterator pipelines allow the series of operations to be applied one element at a time.

- **Map:**
  ```lua
  function Map(fun, first, second, third)
    local itercoroutine=function ()
      for value in (function () return first, second, third end)() do
        coroutine.yield(fun(value))
      end
    end
    return coroutine.wrap(itercoroutine), nil, nil
  end
  ```

- **Filter:**
  ```lua
  function Filter(condition, first, second, third)
    local itercoroutine=function ()
      for value in (function () return first, second, third end)() do
        if condition(value) then coroutine.yield(value) end
      end
    end
    return coroutine.wrap(itercoroutine), nil, nil
  end
  ```
Composable iterators (iterator pipelines)

Iterator pipelines allow the series of operations to be applied one element at a time.

- **Map:**
  ```lua
  function Map(fun, first, second, third)
    local itercoroutine= function ()
      for value in (function () return first, second, third end)() do
        coroutine.yield(fun(value))
      end
    end
    return coroutine.wrap(itercoroutine), nil, nil
  end
  ```

- **Filter:**
  ```lua
  function Filter(condition, first, second, third)
    local itercoroutine= function ()
      for value in (function () return first, second, third end)() do
        if condition(value) then coroutine.yield(value) end
      end
    end
    return coroutine.wrap(itercoroutine), nil, nil
  end
  ```

- **Reduce:**
  ```lua
  function Reduce(func, first, second, third)
    local initialized=false
    local accumulator
    for value in (function () return first, second, third end)() do
      if not initialized then
        accumulator=value
        initialized=true
      else
        accumulator=func(accumulator, value)
      end
    end
    return accumulator
  end
  ```
• Operations should support unordered hashtables, where applicable.
• Map and Filter iterate using pairs, keeping input table keys:

\[
\text{Map}(f, \{ [\text{foo}]=a, [\text{bar}]=b, [\text{baz}]=c \})
\rightarrow \{ [\text{foo}]=f(a), [\text{bar}]=f(b), [\text{baz}]=f(c) \}
\]
Operations should support unordered hashtables, where applicable.

Map and Filter iterate using pairs, keeping input table keys:

\[
\text{Map}(f, \{[\text{foo}]=a, [\text{bar}]=b, [\text{baz}]=c\}) \\
\rightarrow \{[\text{foo}]=f(a), [\text{bar}]=f(b), [\text{baz}]=f(c)\}
\]

Key\_Value\_Mapping operation is very useful: define a table as a function of the keys. E.g:

\[
\text{Key\_Value\_Mapping}(\text{OddOrEven}, \{2, 10, 11\}) \\
\rightarrow \{[2]=\text{“even”}, [10]=\text{“even”}, [11]=\text{“odd”}\}
\]
• Operations should support unordered hashtables, where applicable.
• Map and Filter iterate using pairs, keeping input table keys:
  \[
  \text{Map}(f, \{[\text{foo}]=a, [\text{bar}]=b, [\text{baz}]=c\}) \\
  \rightarrow \{[\text{foo}]=f(a), [\text{bar}]=f(b), [\text{baz}]=f(c)\}
  \]

• KeyValueMapping operation is very useful: define a table as a function of the keys. *E.g:*
  \[
  \text{KeyValueMapping}(\text{OddOrEven}, \{2, 10, 11\}) \\
  \rightarrow \{[2]=\text{“even”}, [10]=\text{“even”}, [11]=\text{“odd”}\}
  \]

• Keyed Zip and Unzip: Zip into keyed tables, unzip from keyed tables:
  \[
  \text{KeyedZip}(\{\text{“pants”,”size”}\}, \{\text{“corduroy”,”gabardine”}\}, \{32, 36\}) \\
  \rightarrow \{\{\text{pants=“corduroy”}, \text{size}=32\}, \{\text{pants=“gabardine”}, \text{size}=36\}\}
  \]
Operations should support unordered hashtables, where applicable.

Map and Filter iterate using pairs, keeping input table keys:

\[
\text{Map}(f, \{ [\text{foo}]=a, [\text{bar}]=b, [\text{baz}]=c \}) \\
\rightarrow \{ [\text{foo}]=f(a), [\text{bar}]=f(b), [\text{baz}]=f(c) \}
\]

Key-Value Mapping operation is very useful: define a table as a function of the keys. E.g:

\[
\text{KeyValueMapping}(\text{OddOrEven}, \{2, 10, 11\}) \\
\]

Keyed Zip and Unzip: Zip into keyed tables, unzip from keyed tables:

\[
\text{KeyedZip}(\{\"pants\","size\}\}, \{\"corduroy\",\"gabardine\\}, \{32, 36\}) \\
\rightarrow \{ \{\text{pants}=\"corduroy\", \text{size}=32\}, \{\text{pants}=\"gabardine\", \text{size}=36\} \}
\]

*(Keyedunzip is inverse of keyed zip)*
Example 1: Ion source tuning (with complications)

```javascript
local setpoints, sprayCurrents, intensities =
    table.KeyedUnzip(
        fun.IteratorToArray(
            fun.Map(AcquireResponse,
                fun.TerminateIf(APCIIsOutOfControl,
                    fun.Map(SetAndReadback,
                        fun.Values(grid))))),
        {"setting","readback","ionIntensity"})

• Motivation: current is limited by spray chemistry, and true value can lag setpoint
```
Example 1: Ion source tuning (with complications)

```python
local setpoints, sprayCurrents, intensities =
    table.KeyedUnzip(
        fun.IteratorToArray(
            fun.Map(
                AcquireResponse,
                fun.TerminateIf(APCIIsOutOfControl,
                    fun.Map(SetAndReadback,
                        fun.Values(grid)))))
    ,
    {"setting","readback","ionIntensity"})
```

- Motivation: current is limited by spray chemistry, and true value can lag setpoint
- From the inside out:
  1. Values makes an iterator from a table
Example 1: Ion source tuning (with complications)

```javascript
local setpoints, sprayCurrents, intensities =
  table.KeyedUnzip(
    fun.IteratorToArray(
      fun.Map(AcquireResponse,
        fun.TerminateIf(APCIIsOutOfControl,
          fun.Map(SetAndReadback,
            fun.Values(grid)))),
    {"setting", "readback", "ionIntensity"}))
```

- **Motivation:** current is limited by spray chemistry, and true value can lag setpoint
- **From the inside out:**
  1. Values makes an iterator from a table
  2. SetAndReadback sets the current target, waits, and takes a measurement
Example 1: Ion source tuning (with complications)

```haskell
local setpoints, sprayCurrents, intensities=
  table.KeyedUnzip(
    fun.IteratorToArray(
      fun.Map(AcquireResponse,
        fun.TerminateIf(APCIIsOutOfControl, 
          fun.Map(SetAndReadback, 
            fun.Values(grid)))), 
    {"setting","readback","ionIntensity"})
```

- **Motivation:** current is limited by spray chemistry, and true value can lag setpoint
- **From the inside out:**
  1. Values makes an iterator from a table
  2. SetAndReadback sets the current target, waits, and takes a measurement
  3. TerminateIf is an iterator controller/passthrough, breaking iteration if the current iterand satisfies a predicate
Example 1: Ion source tuning (with complications)

local setpoints, sprayCurrents, intensities =
    table.KeyedUnzip(
        fun.IteratorToAnArray(
            fun.Map(AcquireResponse,
                fun.TerminateIf(APCIIIsOutOfControl,
                    fun.Map(SetAndReadback,
                        fun.Values(grid)))))
    )

• Motivation: current is limited by spray chemistry, and true value can lag setpoint

• From the inside out:
  1. Values makes an iterator from a table
  2. SetAndReadback sets the current target, waits, and takes a measurement
  3. TerminateIf is an iterator controller/passthrough, breaking iteration if the current iterand satisfies a predicate
  4. AcquireResponse is at left

local AcquireResponse =
    function(sourceStatus)
        Sys.TakeAveragedScan(nScans)
        return {setting = sourceStatus.setting,
            readback = sourceStatus.readback,
            ionIntensity = DQ:TIC()}
    end
Example 1: Ion source tuning (with complications)

```haskell
local setpoints,sprayCurrents,intensities=
  table.KeyedUnzip(
    fun.IteratorToArray(
      fun.Map(AcquireResponse,
        fun.TerminateIf(APCIIsOutOfControl,
          fun.Map(SetAndReadback,
            fun.Values(grid))))),
    {"setting","readback","ionIntensity"})
```

- Motivation: current is limited by spray chemistry, and true value can lag setpoint
- From the inside out:
  1. Values makes an iterator from a table
  2. SetAndReadback sets the current target, waits, and takes a measurement
  3. TerminateIf is an iterator controller/passthrough, breaking iteration if the current iterand satisfies a predicate
  4. AcquireResponse is at left
  5. We pack it all into an array, then unzip for further processing.

```haskell
local AcquireResponse=  
  function (sourceStatus)
    Sys.TakeAveragedScan(nScans)
    return {setting=sourceStatus.setting,
       readback=sourceStatus.readback,
       ionIntensity=DQ:TIC()}
  end
```
Example 1: Ion source tuning (with complications)

```
local setpoints, sprayCurrents, intensities =
    table.KeyedUnzip(
        fun.IteratorToArray,
        fun.Map(AcquireResponse,
            fun.TerminateIf(APCIIsOutOfControl,
                fun.Map(SetAndReadback,
                    fun.Values(grid))))),
    {"setting", "readback", "ionIntensity"})
```

- Motivation: current is limited by spray chemistry, and true value can lag setpoint

- From the inside out:
  1. Values makes an iterator from a table
  2. SetAndReadback sets the current target, waits, and takes a measurement
  3. TerminateIf is an iterator controller/passthrough, breaking iteration if the current iterand satisfies a predicate
  4. AcquireResponse is at left
  5. We pack it all into an array, then unzip for further processing.

```
local AcquireResponse =
    function (sourceStatus)
        Sys.TakeAveragedScan(nScans)
        return {setting = sourceStatus.setting,
            readback = sourceStatus.readback,
            ionIntensity = DQ:TIC()}
    end
```
Example 1 continued: Function decoration for output and more.

```lua
function utils.AutoPlotOneParameterFunction(f, Plotter)
    return function(x)
        local y = f(x)
        Plotter(x, y)
        return y
    end
end
```

![Optimization of Discharge Current for m/z=102](image)
Example 1 continued: Function decoration for output and more.

```lua
function
    utils.AutoPlotOneParameterFunction(f, Plotter)
        return function(x)
            local y = f(x)
            Plotter(x, y)
            return y
        end
    end

AcquireResponse =
    utils.AutoPlotOneParameterFunction(
        AcquireResponse,
        function (x, y)
            graph:Plot(x.readback, y.ionIntensity, 2)
        end)
```

Optimization of Discharge Current for m/z=102

![Graph](image)
function
utils.AutoPlotOneParameterFunction(f,Plotter)
    return function(x)
        local y=f(x)
        Plotter(x,y)
    return y
end
end

AcquireResponse=
utils.AutoPlotOneParameterFunction(
    AcquireResponse,
    function (x,y)
        graph:Plot(x.readback,y.ionIntensity,2)
    end)

AcquireResponse=
CO.ErrorHandlerDecorator(
    AcquireResponse,
    Cal.CommonErrorHandler())
Example 2: Electronics ramp test

```lua
local pCallStatus, result =
    pcall(function ()
        return fun.IteratorToArray(
            fun.Map(fun.AutoPlotOneParameterFunction(ReadDevices(readbackDevices), Plotter),
                fun.Map(SetDeviceAndSleep(rampDevice, sleepTime),
                    fun.iValues(table.Range(rampStart, rampStop, stepSize))))
    end)
```
Example 2: Electronics ramp test

```lua
local pCallStatus, result =
    pcall(function ()
        return fun.IteratorToArray(
            fun.Map(utls.AutoPlotOneParameterFunction(ReadDevices(readbackDevices), Plotter),
                fun.Map(SetDeviceAndSleep(rampDevice, sleepTime),
                    fun.iValues(table.Range(rampStart, rampStop, stepSize)))))
    end)
```

Creates a function of setpoint that returns a table:
{setpoint=setpoint,
  responses=a table of device readbacks, keyed by device}
local pCallStatus, result =
    pcall(function ()
        return
            fun.IteratorToArray(
                fun.Map(lambda: lambda: lambda:
                    fun.Map(lambda: lambda: lambda:
                        fun.iValues(table.Range(rampStart, rampStop, stepSize))))))
    end)
Example 2: Electronics ramp test

```lua
local pCallStatus, result = pcall(function ()
    return fun.IteratorToArray(
        fun.Map({
            utils.AutoPlotOneParameterFunction(ReadDevices(readbackDevices), Plotter),
            fun.Map(SetDeviceAndSleep(rampDevice, sleepTime),
                fun.iValues(table.Range(rampStart, rampStop, stepSize)))
        }
    )
end)

(Restore system state, process pcall, omitted from example)

local setpoints, responses = table.KeyedUnzip(result, {
    "setpoint", "readbacks"
})
local selfResponse, crossResponses = table.KeyedUnzip(responses, {selfReadback}),
    table.KeyedTranspose(responses, otherReadDevices)
responses = table.KeyedTranspose(responses, readbackDevices)
```
Example 2: Electronics ramp test

```lua
local pCallStatus, result =
    pcall(function ()
        return fun.IteratorToArray(
            fun.Map(utils.AutoPlotOneParameterFunction(ReadDevices(readbackDevices), Plotter),
                fun.Map(SetDeviceAndSleep(rampDevice, sleepTime),
                    fun.iValues(table.Range(rampStart, rampStop, stepSize)))))
    end)

(Restore system state, process pcall, omitted from example)

local setpoints, responses = table.KeyedUnzip(result, {"setpoint", "readbacks"})
local selfResponse, crossResponses = table.KeyedUnzip(responses, {selfReadback}),
    table.KeyedTranspose(responses, otherReadDevices)
responses = table.KeyedTranspose(responses, readbackDevices)

(Evaluate device setpoint-readback correspondence, omitted from example)

--Now check the cross responses:

local impedances =
    table.KeyValueMapping(function (dev) return MutualImpedance(selfResponse, crossResponses[dev]) end, otherReadDevices)

local shortedDevices =
    fun.Filter(function (dev) return impedances[dev] < minMutualImpedance end, otherReadDevices)

(Plotting of suspected shorts, return table formatting omitted)
Compound-dependent tuning of a TSQ mass spectrometer
Compound-dependent tuning of a TSQ mass spectrometer

More ions = greater sensitivity.
Greater precision, lower LLOD and LLOQ, or Higher throughput, or lower assay cost
Compound-dependent tuning of a TSQ mass spectrometer

Ion source parameters (voltages, gas flows) are dependent on chemistry and sample delivery rate.
Compound-dependent tuning of a TSQ mass spectrometer

Ion funnel: RF tuning trades off several effects; DC offset used for declustering or fragmentation.
Compound-dependent tuning of a TSQ mass spectrometer

Ion funnel: RF tuning trades off several effects; DC offset used for declustering or fragmentation.
Compound-dependent tuning of a TSQ mass spectrometer

Fragment ions to select in Q3 for are often unknown.
Compound-dependent tuning of a TSQ mass spectrometer

Collision energy optimizing any CID transition must be determined empirically.
• We want to be able to insert or remove optimizations at (customer) will.
• Future compatibility is also desired.
• Optimizations should use previous results in a clean way.
• Design:
  • Precursor and product data structures: attributes name, mass, tunings, etc.
  • Precursor ion optimizations take a precursor as input and yield a precursor as output, with updated tunings or updated mass.
  • Product ion optimizations are similar
  • Optimizations are composed/put into sequence by function composition.
  • Optimization internals may be procedural code, but no persistent side effects or communication at a distance through instrument state allowed. (Either clean up state changes or be indifferent.)
Assay optimization: Handling state using decorators

- Decorator sets system according to previous tunings; optimizer functions concerned only with their proper optimization operation:

```lua
local function PrecursorStateSetter(precursor)
    local doWait=false
    if Sys.Polarity()~=precursor.polarity then
        DS:SetSystemPolarity(precursor.polarity)
        doWait=true
    end
    -- Set ion source devices
    for _,dev in pairs(Sys.IonSourceDevices()) do
        if precursor.tunings[dev.name] and (precursor.tunings[dev.name]~=dev.value) then
            CF2:SetAndUpdate(precursor.tunings[dev.name],dev)
            doWait=true
        end
    end
    -- Now set everything else:
    (...) if doWait then SleepSec(SOURCE_WAIT) end
    return
end

function CO.StateManipulationDecoratorPrecursor(f)
    return function (precursor)
        if precursor then PrecursorStateSetter(precursor) end
        return f(precursor)
    end
end
```
Assay optimization: Handling state using decorators

Sample delivery request and detection by sample delivery decorator:

```lua
function CO.GetSampleDecorator(f, timeout)
    return function (ion)
        local monitoredMZ = ion.precursor and ion.precursor.mz or ion.mz
        --This construction makes this precursor or product compatible
        local monitoredPolarity = ion.precursor and ion.precursor.polarity or ion.polarity
        if firstInjectionReceived then --Don't do this for first injection of a series.
            Signal(DS.SIG_REQUEST_SAMPLE)
            if not MethodControl.WaitCC() then
                error(CO.exceptions.ABORTED_WAITING)
            end
        end
        local result = Sys.GetSample({mass parameters: omitted details}, timeout, false)
        if result == 0 then
            Signal(DS.SIG_SAMPLE_NOT_RECEIVED)
            (...) error(CO.exceptions.SAMPLE_NOT_RECEIVED)
        elseif result == 1 then
            Signal(DS.SIG_RECEIVED_SAMPLE)
            (...) end
        SleepSec(0.5)
        return f(ion)
    end
end
```
Composite spectrum generated by

\[
\text{fun.Reduce}(\text{function } (a,b) \ \text{return} \ \text{CO.MergeSpectra}(a,b,\text{false}) \ \text{end}, \\
\text{fun.Map}(\text{SingleRampScan}, \text{fun.Values}(\text{CEs})))
\]

where \text{MergeSpectra} is a pointwise max intensity selector

\[
\text{function} \ \text{CO.MergeSpectra}(a,b) \\
\text{local} \ \text{retstructure}=a \\
\text{(...)} \\
\text{for} \ \text{index}=1,\#a.y \ \text{do} \\
\text{retstructure}.y[\text{index}]=\text{math.max}(a.y[\text{index}],b.y[\text{index}]) \\
\text{end} \\
\text{return} \ \text{retstructure} \\
\text{end}
\]
Optimize precursor list:

```lua
local runSucceeded, result =
  pcall(function ()
    return ProductOptimization(
      PrecursorOptimization(
        fun.Map(SourceOptimization,
          GetInjection(experiment.precursors)))))
end
```

Optimize product list:

```lua
ProductOptimization =
  function (precursor)
    if precursor then
      precursor.products =
        fun.IteratorToArray(
          fun.Map(function (x) return GraphProductPoint(x) end,
            fun.TakeFirstN(experiment.nProducts,
              CheckProductExistencesAndCountFailures(
                fun.Filter(ProductMassFilterCondition,
                  fun.Map(OptimizeProduct,
                    GetProducts(precursor).ProductsIterator)))))
        (...)
    end
    return precursor
  end
end
```

`OptimizeProduct` is a composition of optimizer functions (tuning mass, collision energy). A binary compose operation is reduced across a list to make it.